10 | Refactorización

Meta

Que el alumno modifique su programa para aplicar el principio de *acoplamiento mínimo*, separando componentes que realizan trabajos independientes.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Implementar una lista que almacene cualquier tipo de objetos.
2. Utilizar dicha lista para una aplicación particular.
3. Adaptar un programa diseñado originalmente para un objetivo particular, de tal modo que sus componentes se vuelvan más independientes y, por lo tanto, más reutilizables.

Antecedentes

Contención

Las relaciones de *contención*[1] en esta práctica son particularmente interesantes.

Agregación Un `RegistroAgenda` puede existir, con los datos de la persona registrada, aunque ya no tengamos ni la lista ni la base de datos; pero comunmente está contenido en la lista que lo almacena.

Composición La Figura 10.1 muestra el caso de `ManejadorDeLista` y `ListaAgenda`. `ListaAgenda` no puede existir fuera de su manejador, nadie más la ve ni la usa.

[1] Figura 8.5.

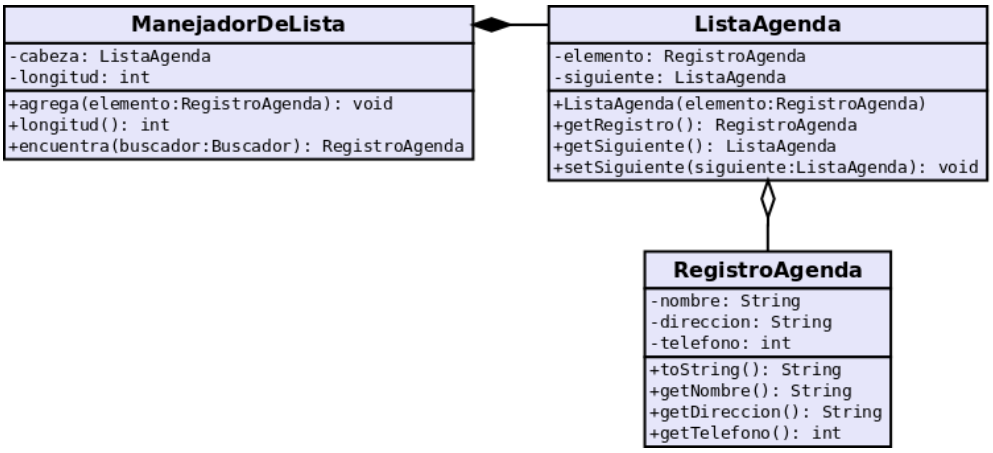

Figura 10.1 Ejemplos de contención UML. Agregación entre `ListaAgenda` y `RegistroAgenda`. Composición entre `ManejadorDeLista` y `ListaAgenda`.

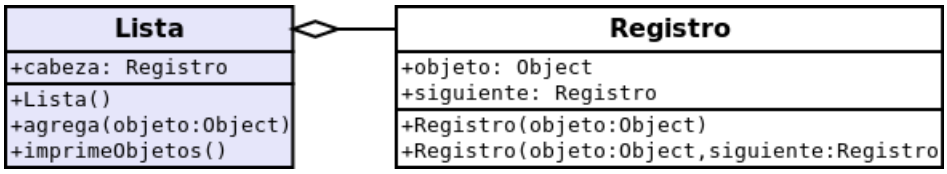

Figura 10.2 Diagrama UML que muestra la relación entre la clase `Planta`, su `Especie` y las clases correspondientes a tipos concretos de plantas.

Listas de Objetos

Al programar tu lista de contactos ya aprendiste a programas listas. Tienes dos clases: el registro, que contiene los datos y la dirección del siguiente registro en la lista, y una clase que se encarga de manipular a toda la lista. Ésta última encapsula la forma en que almacenaste los datos en la memoria de la computadora y permite que el usuario de la lista se imagine que los datos se encuentran almacenados secuencialmente.

Para esta práctica repetirás la misma estructura, pero aprovecharás la herencia para que, a partir de ahora, puedas almacenar cualquier objeto en la lista que programes. La idea es que no tengas que repetir el mismo código cada vez que quieras guardar algo diferente en los registros. Para ello, en lugar de nombre, dirección y teléfono, lo que irá dentro de un registro será un `Object`. Dado que todas las clases en `Java` heredan de `Object`, podrás guardar objetos de cualquier tipo en la lista. La Figura 10.2 muestra esta estructura.

A continuación se incluye un extracto de código que muestra cómo se deben utilizar estas listas:

```java
public class UsoListaRegistro {
  public static void main(String[] args) {
    Lista rosas = new Lista();
    rosas.agrega(new Rosa()); // Agrega el objeto al final
    rosas.agrega(new Rosa());
    rosas.imprimeObjetos();
  }
}
```

Desarrollo

Arquitectura del programa

La Figura 10.3 muestra la estructura de las clases tal y como quedaron en la práctica anterior e ilustra el uso de los componentes descritos anteriormente.

Diseño orientado a objetos

El diagrama anterior muestra algo muy extraño en nuestro sistema:

1. La lista de los registros agenda y sus clases relacionadas se encuentran en otro paquete. ¡Pero `RegistroAgenda` esta en `icc.agenda`!

Actividad 10.1

Escribe, antes de continuar leyendo, tu hipótesis sobre porqué tiene esta estructura el sistema de la práctica. (No, no fue un accidente.) Toma en cuenta que ahora sabes de herencia.

Ejercicios

Para esta práctica utilizarás el trabajo que realizaste en la práctica anterior, por lo que necesitarás los archivos siguientes:

- `icc.agenda`
  - ⋆ `UsoBaseDeDatosAgenda.java`
  - ⋆ `BaseDeDatosAgenda.java`
  - ⋆ `RegistroAgenda.java`
  - ⋆ `BuscadorPorNombre.java`

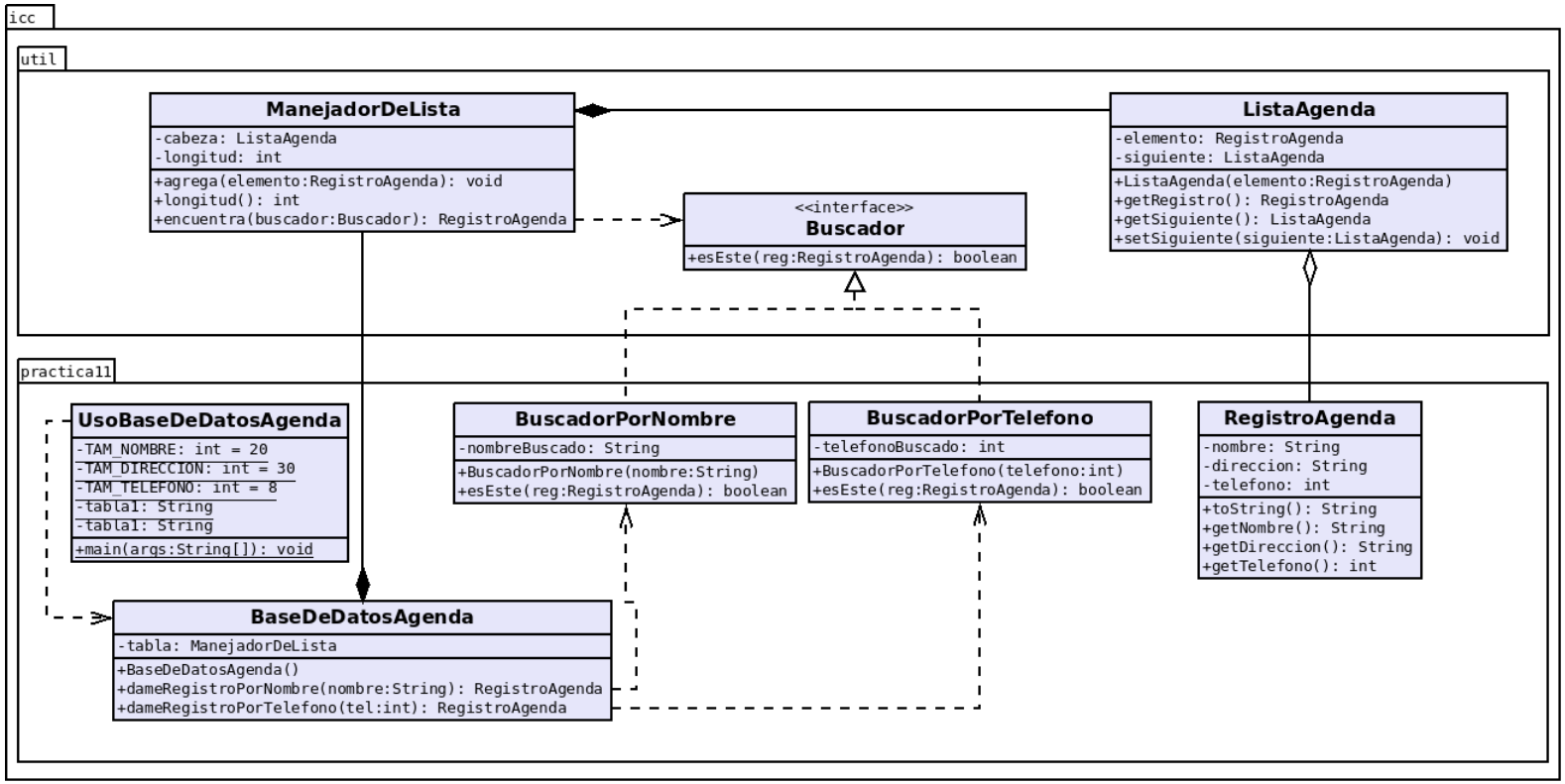

Figura 10.3 Diagrama de clases UML de un pequeño sistema.

10. Refactorización

    - ⋆ BuscadorPorTelefono.java
- icc.util
    - ⋆ ManejadorDeLista.java
    - ⋆ ListaAgenda.java
    - ⋆ Buscador.java

En esta práctica vamos a *refactorizar* el código de la práctica anterior. Este es un proceso por el cual pasan los ingenieros de software conforme su sistema crece y su código debe cumplir nuevas funciones, aún más generales que las originales. ¿Recuerdas lo que significa *factorizar* en álgebra? Cito un ejemplo a continuación para refrescar la memoria:

$$ab^3x^2 + 2ab^3xy + ab^3y^2 = ab^3(x^2 + 2xy + y^2) \tag{10.1}$$
$$= ab^3(x + y)^2 \tag{10.2}$$

Como recordarás, para factorizar un polinomio, debes buscar los elementos comúnes a todos los monomios y *extraerlos*, para luego realizar otras simplificaciones. Pues bien, con el código se hace lo mismo, pero a nivel conceptual.

El código se encuentra a la mitad de uno de estos procesos. Las clases de la base de datos están divididas en dos paquetes para dar una pista. En este sistema hay dos funcionalidades completamente diferentes:

1. Crear y manipular una lista, que se usa para contener y consultar secuencias de datos.

2. Manejar la agenda.

Ambas cosas son distintas: la lista no necesita a la agenda (conceptualmente), y la agenda podría preferir guardar sus datos en algún otro lado, que no sea la lista. Así que ambas cosas deberían ser separables manteniendo únicamente la comunicación necesaria para que la agenda guarde sus datos en la lista y la use para hacer consultas.

Bueno, esto no se podía hacer mientras no tuviéramos herencia. Para que una lista pueda almacenar cualquier cosa, haremos uso de un hecho básico en `Java`: todas las clases heredan de la clase `Object`. Así que, donde podamos poner un objeto, podemos poner un `RegistroAgenda`. Observa que las clases que se encuentran en el paquete `icc.util` no necesitan ninguno de los métodos de `RegistroAgenda` y podrían hacer exactamente el mismo trabajo si recibieran `Object` en lugar de `RegistroAgenda`. Por ello es que fueron puestas dentro de un paquete diferente; las podemos separa de las demás.

Por otro lado, si las listas guardan objetos, la base de datos tendrá que lidiar con algunos `castings`. Cuando la lista devuelva un `Object`, las clases en `agenda` deberán recordarle al compilador que se trataba de un `RegistroAgenda`. Esto se vería como:

```
1  RegistroAgenda rg = (RegistroAgenda)(tabla.encuentra(buscador));
```

De ahí en fuera, el código es el mismo.

1. Tu tarea para esta práctica es modificar las clases para que cumplan ahora con lo especificado en la Figura 10.4.
2. Arregla primero lo que esté en `icc.util`. Sólo debes cambiar tipos, no debes necesitar cambiar otro código.
3. Ahora ajusta `icc.agenda` para que funcione con las modificaciones, cambia los nombres de los constructures, agrega castings y cualquier otro pequeño detalle que impida que funcione tu código. Verifica que tu programa vuelva a funcionar como antes.

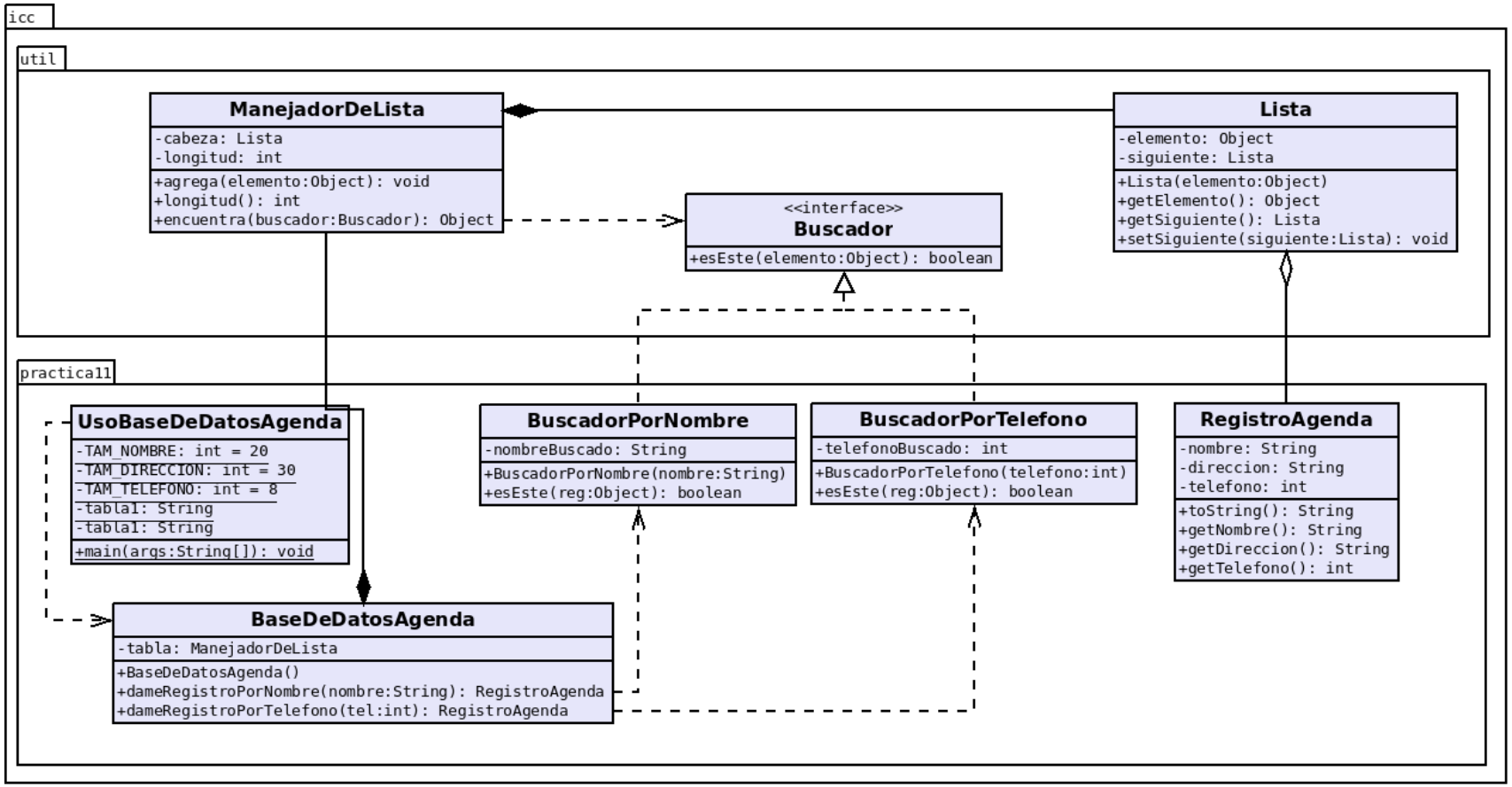

Figura 10.4 Diagrama de clases UML de un pequeño sistema.