Práctica 3

En esta práctica, se nos pide implementar distintos diagramas en código java. Para hacerlo más fácil, pondré punto a punto qué es lo que se pide en cada ejercicio.

Ejercicio 1

En este ejercicio, el profesor nos proporciona un diagrama sobre la gestión de expedientes de pacientes en el ámbito hospitalario. Nos pide que diseñemos un esquema del código de andamiaje Java.

Un código de andamiaje se realiza creando clases con sus respectivos atributos y operaciones. Básicamente, tratando de modelar en código java lo que nos refleja en el diagrama. En este caso, por ejemplo, crearíamos cuatro clases (Profesional, Expediente, Acceso y Paciente), y diseñaríamos en código los que nos pide cada una.

Importante: un código de andamiaje NO es una aplicación terminada, sino más bien un esqueleto que permite centrar la atención en el modelo en sí. Entre otras cosas, dentro del andamiaje destacan las siguientes características dentro de la implementación:

- **Atributos** → deben de tener visibilidad privada, en principio.
- **Operaciones** → Las básicas set(), get(), add(). Definen la interfaz de la asociación. De nuevo, en las diapositivas pide que se restringa la visibilidad al máximo, así que supongo que deberán ser privadas o protected como mucho.
- **CÓMO PONER LOS NOMBRES A CADA COSA** →
  - Para el nombre del atributo : hay que ponerle el nombre de los roles del extremo opuesto.
  - Para el tipo de atributo : nombre de la clase del extremo opuesto.
  - Para la multiplicidad del atributo : Tengo que mirar cómo se implementaría, pero de igual manera, la multiplicidad será la del rol del extremo opuesto.

MULTIPLICIDADES DENTRO DEL ANDAMIAJE

Para reflejar las multiplicidades, las diapositivas nos indican las siguientes pautas a seguir dentro de las clases.

MULTIPLICIDAD 0..1

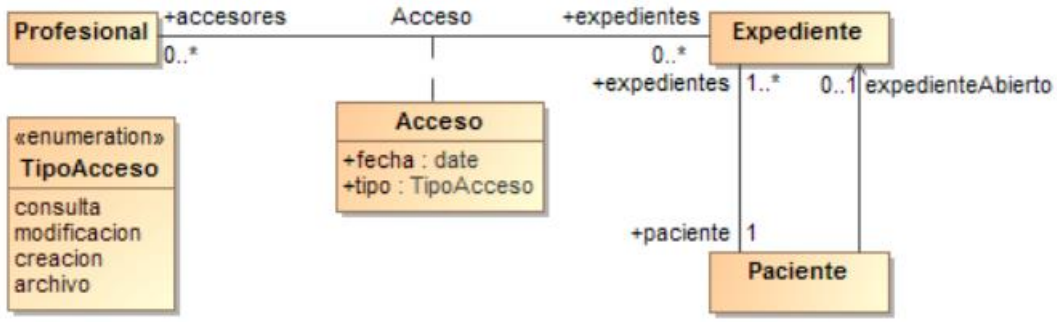

```
public class Alumno {
    private String nombre;
```

```
    private Direccion domicilio;

    public Direccion getDomicilio()
    { return domicilio; }

    public void setDomicilio(Direccion domicilio)
    { this.domicilio = domicilio; }
    }
```

Como podemos observar, los atributos de Alumno son “nombre” y “domicilio”, que es de tipo DIRECCIÓN. Esto quiere decir que las relaciones con otras clases, a su vez, serán **ATRIBUTOS**. En nuestro caso, por ejemplo, sabiendo que este es el diagrama que nos proporciona el profesor:



La clase Expediente deberá tener un atributo de tipo Paciente que se llame paciente, quedando del siguiente modo:

```
public class Expediente {
    private Paciente paciente;
}
```

Después de esta introducción para aclarar algunas cosas, procedo a explicar lo de la multiplicidad.

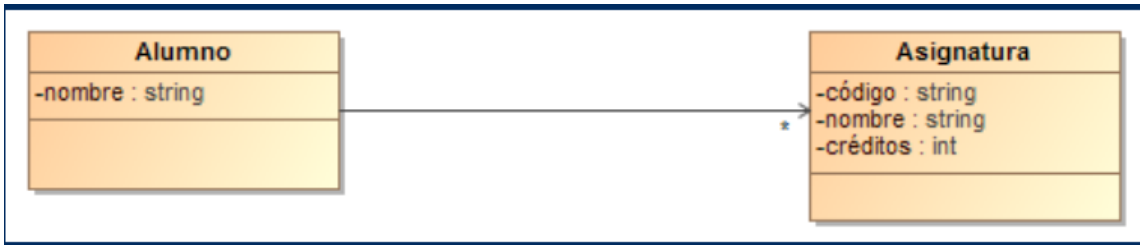
Volviendo al ejemplo del Alumno y Dirección (primer diagrama), para dejar constancia de que la relación entre ambas clases tiene multiplicidad de 0..1, sabemos que cuando le pidamos a Alumno que devuelva su dirección, sólo nos va a devolver UNA Y SOLO UNA dirección. Es decir, NO un conjunto de direcciones. Por tanto:

Public Direccion getDomicilio() { return domicilio }

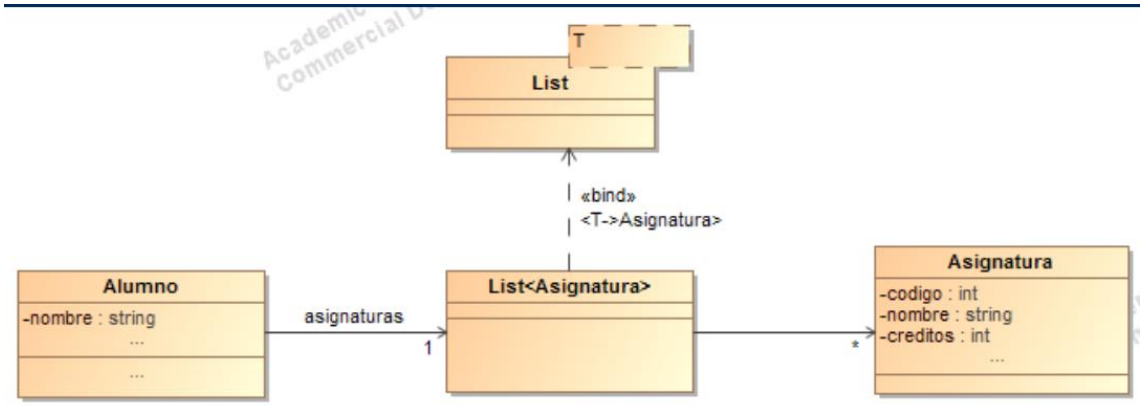
Es decir, no devuelve una colección. En el caso de muchas multiplicidades (lo que veremos en el siguiente punto), podría devolver un array de direcciones.

ASOCIACIONES UNIDIRECCIONALES CON EXTREMO M > 1

Como ejemplo, en las diapositivas viene el siguiente diagrama.

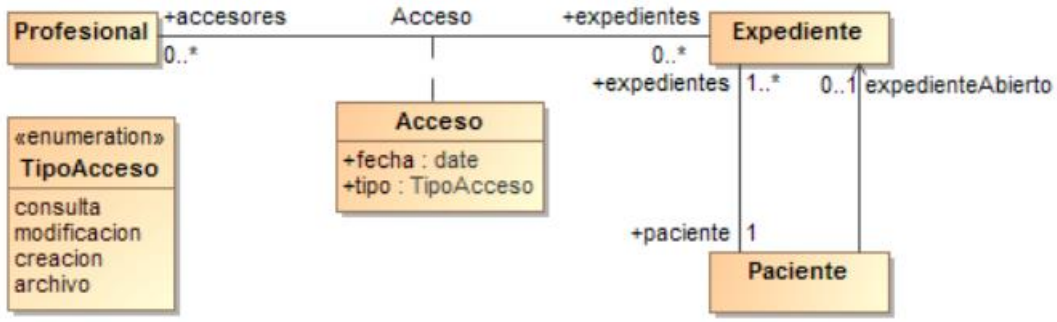


Tal y como he explicado en el punto anterior, esta vez cuando hagamos un llamamiento al atributo asignaturas de alumno, nos debe devolver una colección de asignaturas. Esto podemos implementarlo con listas, arrays, o lo que sea que nos devuelva una colección.




```
private List<Asignatura> asignaturas = new LinkedList<Asignatura>();
public Enumeration<Asignatura> getAsignaturas() {...}
```

así quedaría en el código en el ejemplo. En nuestro caso sería así:



Aquí en la clase Paciente, podríamos hacer una **lista de expedientes** y cuando llamemos al get, que nos devuelva dicha lista. Debemos poner la condición de que la lista no esté vacía, pues mínimo un paciente tiene un expediente para estar registrado (if !lista.isEmpty …)

A su vez, deberemos implementar un método que nos devuelva **UN ÚNICO EXPEDIENTE**, es decir, le pasamos por parámetro el índice del expediente que queremos obtener y el programa nos lo tiene que devolver.

Quedaría algo así, poniendo de nuevo

```
public expediente getExpediente(int i) { ... }
```

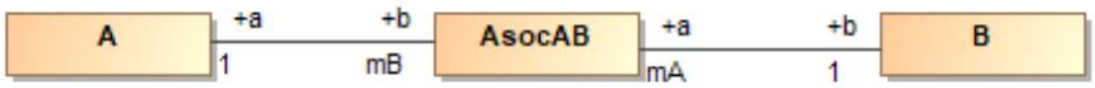
PARA AÑADIR CLAVES A LOS EXPEDIENTES, DEBEREMOS IMPLEMENTARLOS CON TABLAS HASH O CON DICCIONARIOS

```
public expediente getExpediente(int i) { ... }
private HashMap<int, Expediente> expedientes = new
HashMap<int,Expediente>();
public void addExpediente (int Codigo, Expediente e) {...}
public void removeExpediente (int Codigo, Expediente e) {...}
```

ASOCIACIONES BIDIRECCIONALES

Se implementan como dos asociaciones unidireccionales.

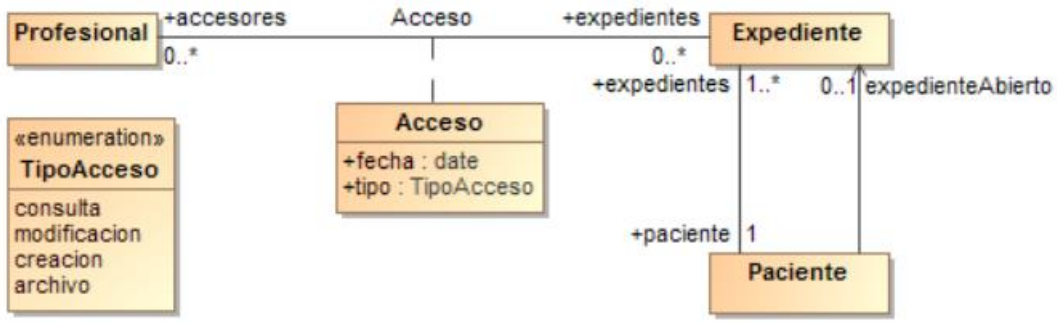
Para establecer estas asociaciones, se debe crear una clase **ASOCIACIÓN**. Es decir, viéndolo como un diagrama, quedaría como algo así:



aunque no venga explícitamente en el diagrama, en JAVA hay que implementar una clase que sea, por ejemplo, AsocAB.

ASOCIACIONES CON ATRIBUTOS Y N-ARIAS

Sería el caso de, por ejemplo, la clase Acceso entre Profesional y Expediente.



Se aplica la reificación de nuevo (lo que hemos visto en el punto anterior básicamente), la clase Acceso debe encargarse de mantener la consistencia.

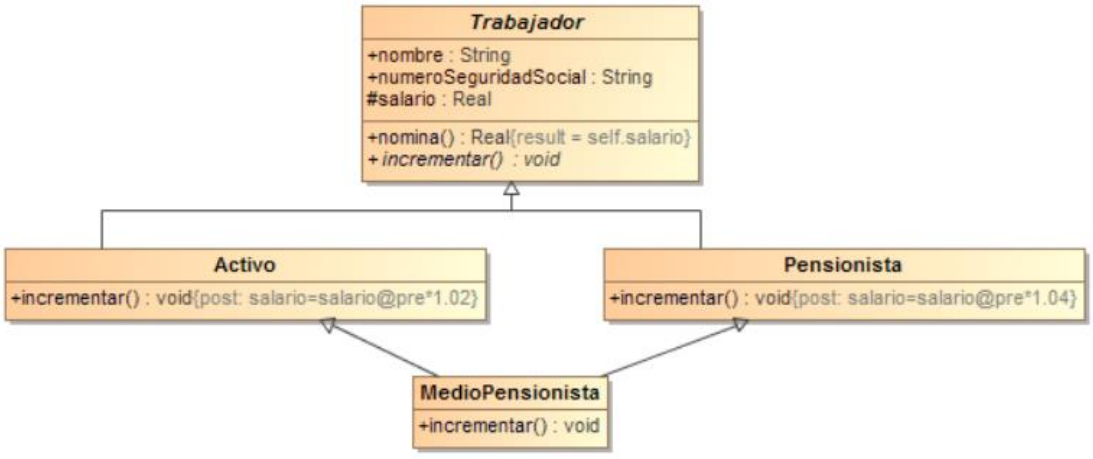
COSAS IMPORTANTES DEL ANDAMIAJE

Esto es muy importante porque son las pautas que nos pone el profesor para que el ejercicio esté bien hecho.

- Los atributos **SIEMPRE EN PRIVATE**. Y las operaciones igual. Bajo ningún concepto deben permitirse setters públicos. Las operaciones de las asociaciones solo deben ser invocadas por la clase que implementa dicha asociación.
- Los getters **NO DEBEN PERMITIR LA ACTUALIZACIÓN**, sobre todo en las listas. Para esto, usar java.util.Enumerator() o Collections.unmodifiableCollection(). El java.util.Iterator permite modificar la colección, por lo que el profesor recomienda NO USARLO.
- A partir de la diapositiva 33, el profesor pone varios ejemplos de cómo implementarlo todo en java.

EJERCICIO 2 – CLASES ABSTRACTAS

En el ejercicio 2, el profesor nos proporciona el siguiente diagrama:



Pero esto no se puede implementar directamente en java. ¿Por qué? Pues porque java no admite la herencia múltiple (esto en el caso de MedioPensionista con Activo y Pensionista, porque MedioPensionista no puede tener dos clases padre). Creo que habría que usar, en este caso, interfaces.

Yo lo haría del siguiente modo; crearía dos interfaces que fuesen Activo y Pensionista, que ambas heredasen de Trabajador. Y después, crearía la interfaz interfazMedioPensionista:

```
public interface interfazMedioPensionista extends Activo, Pensionista {
```

```
    ...
}
```

Y finalmente, que la clase MedioPensionista heredase de interfazMedioPensionista:

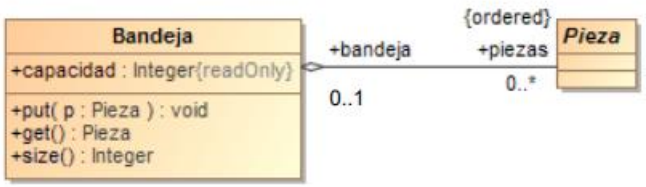
```
public class MedioPensionista implements interfazMedioPensionista{
    ...
}
```

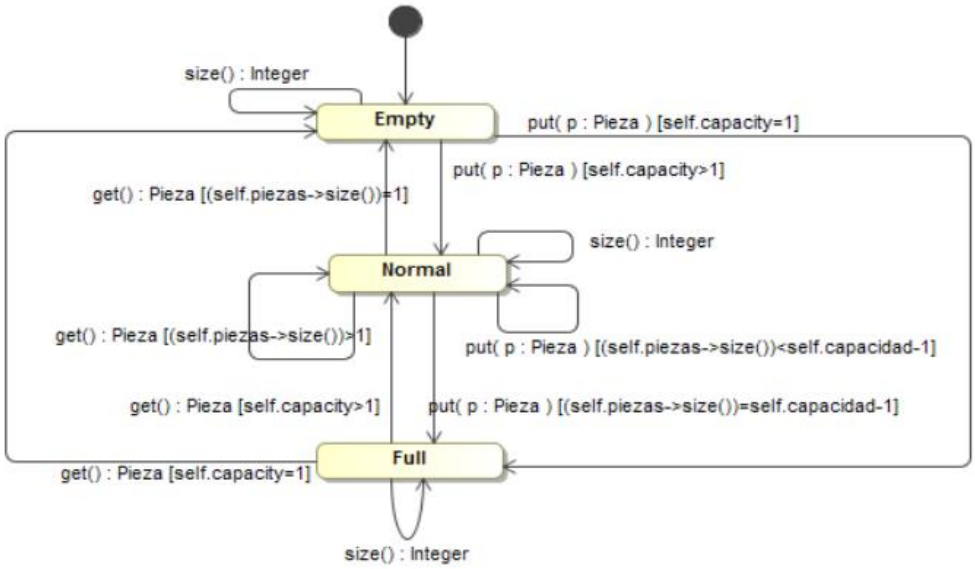
Esto es UNA IDEA. Podríamos explorar otros modos de hacerlo.

Creo que esto responde las preguntas a y b. Faltaría solo implementarlo.

EJERCICIO 3 – DIAGRAMAS DE ESTADO

Estos diagramas de estado indican las distintas formas en las que un objeto responde a un evento. El ejercicio nos propone los siguientes diagramas:





Las tres posibles estrategias de implementación son las siguientes:

- Comportamiento condicional.
- Tablas de estado.
- Patrón de diseño estado.

Este último es el que nos recomienda el profesor que usemos. Para ello, creo que deberíamos crear una interfaz llamada Bandeja que implemente las operaciones que vienen descritas en el diagrama. También habrá que definir los estados Empty, Normal, Full. Dentro de la clase pieza, que implementa la interfaz Bandeja, habría que cambiar el estado según las condiciones que nos da el diagrama.

Creo que State debe ser un atributo de Pieza que sea de los tipos que hemos enumerado antes, y que vaya cambiando según las características que se dan en el diagrama.