EJERCICIO 1

En primer lugar, destacaremos el modelo de diseño implementado en la herramienta Visual Paradigm.

Diagrama

Descripción generada automáticamente

Como podemos observar, hemos plasmado fielmente el diagrama proporcionado en el enunciado del ejercicio, por lo que no haremos ningún comentario adicional.

En cuanto al código de andamiaje, hemos decidido crear una clase java para cada una de las clases del diagrama (Profesional, Acceso, Expediente, Paciente), de las que especificaremos los detalles.

Clase Acceso

En primer lugar, cabe destacar que hemos implementado la clase de asociación “Acceso” como una clase java intermedia entre “Profesional” y “Expediente”. Posee, además, los siguientes atributos privados:

Texto

Descripción generada automáticamente

Sus atributos constan de los atributos “fecha” y “tipo”, junto a “profesional” y “expediente”, que hacen referencia a la relación que mantiene con Profesional y Expediente.

En el constructor de la clase podemos observar la siguiente restricción:



Que, como bien determina el comentario adicional, indica que los campos fecha, tipo, profesional y expediente no pueden ser nulos.

Hemos añadido los siguientes métodos en la clase:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente con confianza media

En los que establecemos los *setters* y *getters* de cada atributo. Cabe destacar que tanto getProfesional(), setProfesional() como getExpediente() y setExpediente() se refieren a UN ÚNICO elemento tanto de la clase Profesional como Expediente. Esto se debe a que la relación entre Acceso y Profesional, y Acceso y Expediente, son de 1..\*. Es decir, debe existir un solo Profesional que accede a un solo expediente en un momento dado, puesto que puede haber múltiples accesos.

Clase TipoAcceso

Esta clase de tipo *enum* simplemente enumera los tipos de acceso existentes, que son *consulta, modificacion, creacion* y a*rchivo.*

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Clase Profesional

La clase Profesional consta del atributo privado “accesos”.



Que hace referencia a la relación que mantiene la clase Profesional con la clase Acceso. Reflejamos la multiplicidad de la relación (\*) haciendo que dicho atributo sea una lista (puede existir más de un acceso a un expediente por parte de un profesional).

Por otro lado, contamos con los siguientes métodos para manipular la clase Acceso.

***getAccesos()***

Texto

Descripción generada automáticamente

Devuelve los accesos haciendo uso de Collections, puesto que los getters no deben permitir la actualización en las listas.

***addAcceso (Acceso a)***

Texto

Descripción generada automáticamente con confianza media

Añade el objeto de tipo Acceso que se pasa por parámetro a la lista de accesos.

***rmvAcceso (Acceso a)***

Texto

Descripción generada automáticamente con confianza baja

Borra el objeto Acceso pasado por parámetro de la lista de accesos.

Clase Expediente

Esta clase posee los siguientes atributos privados:

Texto

Descripción generada automáticamente

El atributo “accesos” es equivalente al atributo con el mismo nombre que hemos descrito en la clase Profesional. Por otro lado, el atributo “paciente” hace referencia a la relación que tiene la clase Expediente con la clase Paciente. Dicha relación posee la multiplicidad de 1, por lo que esto se ve reflejado al declarar el atributo como un solo objeto de tipo Paciente.

En el constructor de Expediente, encontramos las siguientes líneas de código:

Escala de tiempo

Descripción generada automáticamente

La primera de ellas se refiere a una restricción que nos indica, como bien dice el comentario asociado, que el paciente no puede ser nulo.

Por otro lado, dada la relación que mantiene Expediente con la clase Paciente, se debe poder actualizar dicha clase. De este modo, se permite que pueda añadirse un expediente asociado a un paciente.

A su vez, observamos los siguientes métodos:

Texto

Descripción generada automáticamente

getAccesos(), addAcceso(Acceso a) y rmvAcceso(Acceso a) poseen la misma funcionalidad que en la clase Profesional, por lo que no nos detendremos a describir dichos métodos.

Los métodos getPaciente() y setPaciente() hacen referencia a la relación que mantiene Expediente con la clase Paciente, con multiplicidad 1, puesto que un expediente solo puede pertenecer a un único paciente.

Clase Paciente

La clase Paciente posee los siguientes atributos privados:

Texto

Descripción generada automáticamente

Como bien podemos observar en el diagrama, las clases Expediente y Paciente poseen dos relaciones distintas:

Diagrama

Descripción generada automáticamente

El primero de los atributos, expedienteAbierto, hace referencia a la segunda relación que mantienen Expediente y Paciente. Un paciente solo tiene un único expediente abierto, por lo que el atributo expedienteAbierto se corresponde con un solo objeto de tipo Expediente.

Por otro lado, el atributo expedientes, descrito con una lista, hace referencia a la primera relación entre Expediente y Pacientes. Un paciente tiene más de un expediente, por lo que acumulamos los expedientes dentro de la lista implementada.

Cabe destacar que, dentro del constructor de Paciente, hemos implementado las siguientes líneas de código:



Entre otras cosas, podemos observar que expedienteAbierto no se inicializa. Esto se debe a que dicho atributo puede ser nulo, por lo que no haría falta.

Además, la lista de expedientes debe contener, como mínimo, un expediente. Esto lo conseguimos con la segunda línea de código que vemos en la imagen superior.

En la clase Expediente implementamos los siguientes métodos:

***getExpedienteAbierto()***

Texto

Descripción generada automáticamente

Gracias a este método, obtenemos el expediente abierto del paciente.

No hemos implementado un setExpedienteAbierto(Expediente e), puesto que no podemos abrir un expediente que no se encuentre en la lista de expedientes.

***abrirExpediente (Expediente e)***

Texto

Descripción generada automáticamente con confianza baja

El método abrirExpediente se encarga de abrir un expediente que se encuentre dentro de la lista de expedientes de un paciente. El expediente pasado por parámetro debe ser distinto de null y debe estar contenido en la lista de expedientes para poder abrirse.

***cerrarExpediente ()***

Texto

Descripción generada automáticamente con confianza media

Este método pone a *null* el expediente abierto para cerrarlo.

addExpediente (Expediente e), rmvExpediente (Expediente e), getExpedientes()

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Los siguientes métodos hacen referencia a la lista de expedientes. El método addExpediente añade el expediente pasado por parámetro a la lista de expedientes, rmvExpediente borra el expediente pasado por parámetro de dicha lista y getExpedientes() devuelve la lista con todos los expedientes asociados a un paciente.

Ejercicio 2

1. **Justificar por qué las clases descritas no pueden ser implementadas directamente en Java.**

El atributo salario al que quiere acceder la clase MedioPensionista es un atributo que pertenece a la clase Activo y Pensionista. En nuestro diagrama, vemos que la clase MedioPensionista hereda de Activo y Pensionista, accediendo de esta manera a ambos atributos salario. El problema es que en java existe una restricción de herencia única, por lo que la clase MedioPensionista solo puede heredar de una de las dos clases y por lo tanto, acceder solo a uno de los dos atributos salario.

1. **Discutir y desarrollar una solución que permita resolver la situación descrita. Como es lógico, la solución propuesta debe mantener la funcionalidad actualmente existente en las tres subclases de Trabajadores y asegurar la consistencia de los atributos y la reutilización de los métodos de las clases Trabajador, Activo y Pensionista.**

**1ª opción:** Sabemos que las clases Activo y Pensionista son clases abstractas porque sus métodos, incrementar(), lo son. Para solucionar el problema, podríamos cambiar las clases abstractas a interfaces, ya que una clase puede implementar más de una interfaz. Por tanto, el único cambio sería cambiar la herencia de clases abstractas por la de una interfaz.

**2ª opción**: Vemos en el diagrama que en el caso del trabajador activo, su nuevo salario se calcula a partir de la fórmula salario@pre\*1.02, mientras que en el caso del pensionista, el nuevo salario se calcula como salario@pre\*1.04. Esto quiere decir que el salario nuevo de un medio pensionista es, como dice en el enunciado la suma del salario como activo y pensionista, es decir, salario@pre\*1.02 + salario@pre\*1.04. Esta nueva fórmula nos permitiría que MedioPensionista herede directamente de la clase Trabajador, evitando la doble herencia. Resumiendo, los cambios serían los siguientes:

Diagrama

Descripción generada automáticamente

Teniendo el método incrementar de MedioPensionista como post condición que salario = salario@pre\*1.02 + [salario@pre\*1.04](mailto:salario@pre*1.04).

1. **Implementación en java**

Para solventar el problema presentado en el ejercicio, procederemos a implementar lo descrito en el apartado anterior. Para ello, las clases Activo, MedioPensionista y Pensionista tienen que heredar el método incrementar() de Trabajador, e implementar cada uno esta función. Para ello, observaremos dichas clases:

*Clase Trabajador*

Texto

Descripción generada automáticamente

Esta clase nos permite definir el constructor y los métodos que queramos. Sin embargo, incrementar() deberá ser de tipo abstracto, para que el resto de clases hagan la implementación deseada.

*Clase Activo*

Captura de pantalla de computadora

Descripción generada automáticamente

La clase Activo hereda de la clase Trabajador, creando el mismo constructor, e implementa el método incrementar() (de ahí el @Override).

*Clase Pensionista*

Texto

Descripción generada automáticamente

Sigue el mismo procedimiento que el anterior.

*Clase MedioPensionista*

Texto

Descripción generada automáticamente

Esta clase, al igual que las otras, hereda de trabajador, y a su vez puede implementar cada clase por separado. En este caso, el constructor lo hereda igual, pero crea dos objetos de las clases Pensionista y Trabajador con los mismos atributos que hereda.

El método incrementar() es igual a la suma de ambos salarios. Por ello, debemos aplicar el incremento de cada salario por separado, y finalmente el salario de MedioPensionista (this.salario) será la suma de las nóminas ya incrementadas de ambas clases.

Ejercicio 3

Este ejercicio podía solventarse de tres formas distintas:

* Mediante comportamiento condicional, cuya implementación se caracteriza por el uso de if-else-switch (estructuras condicionales) para efectuar las transiciones entre un estado y otro.
* Mediante tablas de estado, que se usan más como herramienta visual para mostrar los distintos estados.
* Mediante patrón diseño estado, que es la estrategia por la que nos hemos decantado, dado a la facilidad que nos proporciona de crear nuevos estados sin modificar el código existente. Tal y como veremos posteriormente en el código, cada estado se ve representado en una clase distinta, e implementa los métodos según las propias necesidades de cada estado. Por ejemplo, el método put(…) en EstadoFull no hace nada, puesto que no se pueden poner más piezas en una bandeja si ésta está llena, mientras que en EstadoEmpty o EstadoNormal sí se implementa el método con normalidad.

A continuación, comentaremos cada clase para dejar los conceptos explicados más claros.

*Clase Bandeja*

Texto

Descripción generada automáticamente

Como podemos observar en la imagen, la clase Bandeja implementa las funcionalidades básicas basándose en el objeto de tipo EstadoBandeja, que representa cualquier estado que pueda tomar la bandeja (en este caso, Empty, Full, Normal), y delega las operaciones a cada estado.

*Interfaz EstadoBandeja*

Texto

Descripción generada automáticamente

Es la interfaz de la que heredan los distintos estados implementados.

*Clases EstadoEmpty, EstadoFull y EstadoNormal*

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Como ya hemos explicado en el inicio del ejercicio, cada una de estas clases representa un estado, y modela el comportamiento que tendrá el programa en función al estado en el que se encuentre. Como podemos observar en las imágenes, los métodos put(…) y get(…) se comportan de forma diferente dependiendo en el estado en el que nos encontremos, siguiendo las condiciones dadas por el enunciado.

*Diagrama de diseño*

Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente