**Definiciones:**

**Ingeniería de software:** es la aplicación de un enfoque sistemático, disciplinado y cuantificable al diseño, operación y mantenimiento de software.

**Modelo:** una representación o especificación desde un determinado punto de vista y con un objetivo concreto.

**Diseño:** conjunto de planes y decisiones para definir un producto con los suficientes detalles como para su realización física de acuerdo a unos requisitos.

(EXCEPTO DISEÑO EL RESTO ME LOS SÉ YA OH YEAH)

**Patrón de diseño:** solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo software.

**Método Fábrica y Factoría Abstracta**

El método fábrica delega la creación del tipo concreto a crear a las subclases mientras que la factoría abstracta es un patrón para facilitar la creación de familias de clases relacionadas.

Ambos se asemejan en que son patrones de creación y en que el sistema tiene que ser independiente de cuál es el tipo de los objetos a crear. Sin embargo, la Factoría Abstracta crea familias de objetos, mientras que el Método Fábrica crea símplemente un tipo determinado de objeto a la vez.

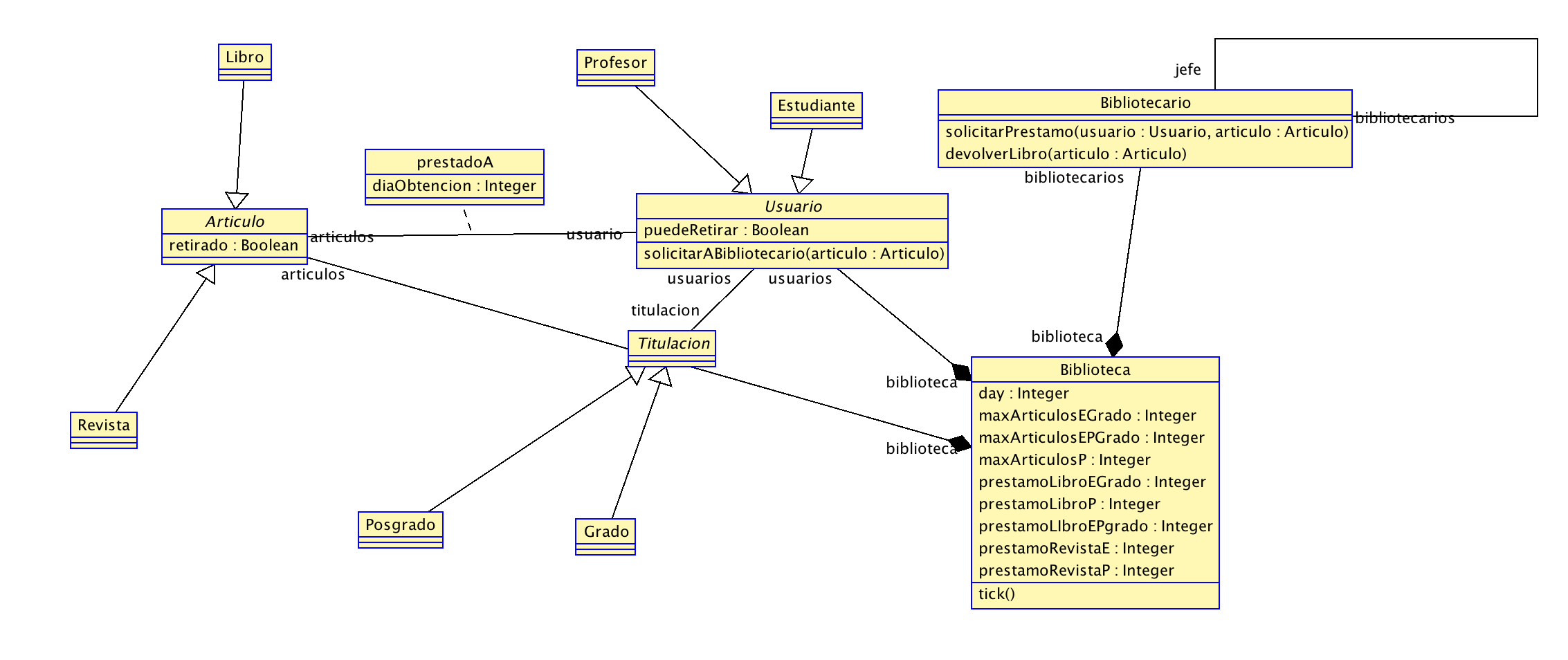
**Estilo Arquitectónico y Patrón de Diseño**

Ambos son maneras de solucionar un tipo determinado de problemasa la hora de crear un proyecto, sin embargo, el estilo arquitectónico es a muy alto nivel, de como se tienen que estructurar las clases, mientras que el patrón de diseño, es más bajo nivel para problemas en concreto, que no involucran gran cantidad de clases a la hora de resolver un problema. El estilo arquitectónico es la manera de organizar las clases para que se comuniquen entre ellas.

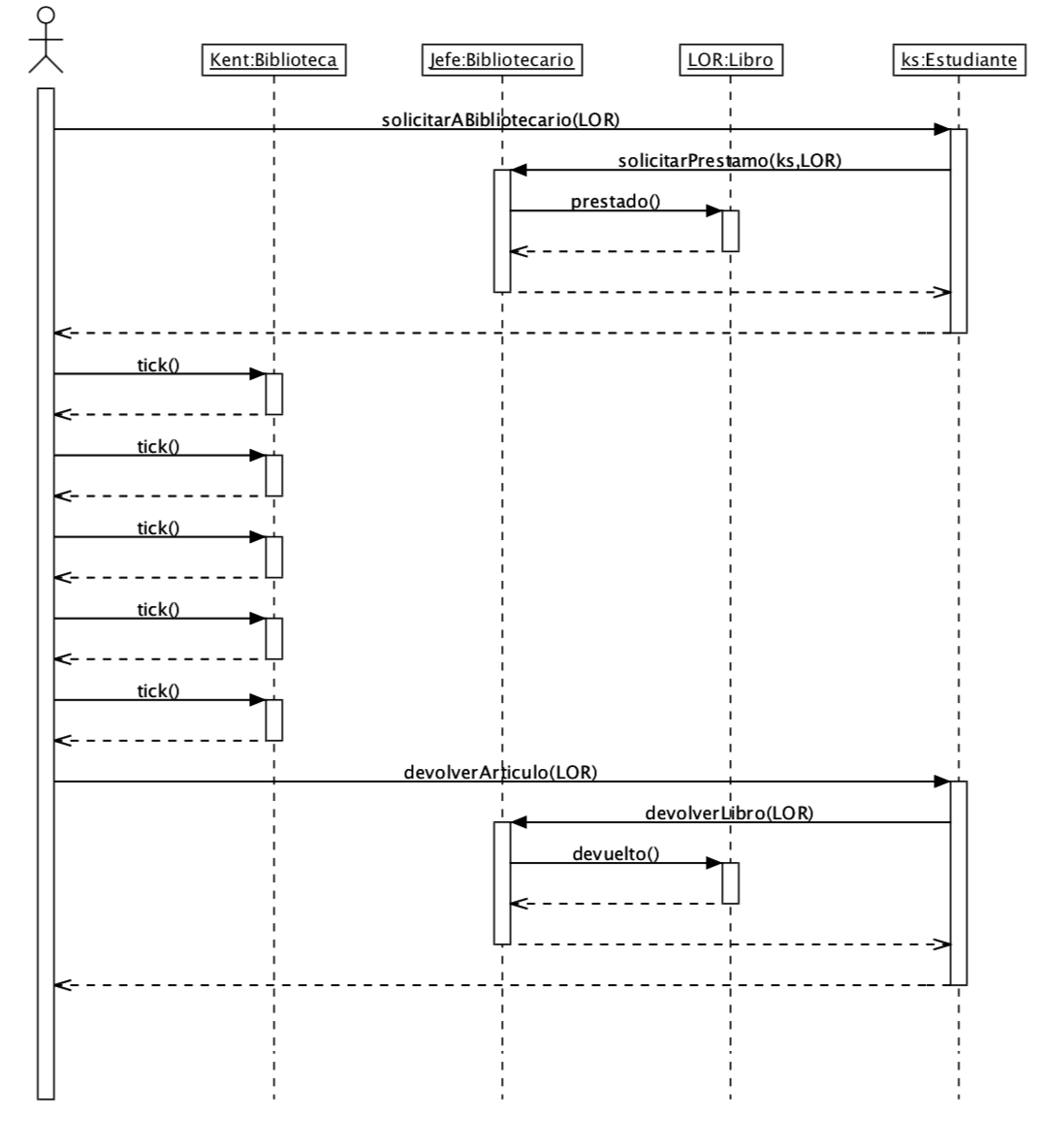
**Ejercicio 1:**

1. Para el ejercicio vamos a tener que modelar el paso del tiempo de algún modo, para ello vamos a crear una clase Sistema en el que tenga el día actual, y una operación tick() que aumenta el día en 1, así se podrá tener un control de los préstamos de manera temporal. Para la clase Artículo hemos añadido un atributo para saber si el libro está o no retirado temporalmente (o permanentemente) del sistema por un bibliotecario. También hemos añadido un atributo booleano para saber si un usuario puede o no retirar libros, ya que los bibliotecarios pueden “banear” a los usuarios que no devuelvan el libro a tiempo y no paguen la sanción económica. Como nos dicen que los límites pueden ser establecidos por el bibliotecario jefe, tendremos que almacenar en algún lugar dicha información, hemos optado por ponerlo en la clase Biblioteca. En la clase bibliotecario añadiremos unas operaciones para modificar dichos valores, comprobando antes que sólo el bibliotecario jefe puede hacer modificaciones a esas variables. Para modelar el comportamiento de préstamo de artículos vamos a añadir varias operaciones. Para empezar, añadiremos una operación que es solicitarPrestamo en la clase bibliotecario, que recibirá el libro a ser prestado y la persona que pide dicho préstamo. Además, vamos a añadir operaciones para que los usuarios paguen multas, hasta que un usuario no pague la multa, no podrá volver a sacar libros. Hemos añadido también, para el diagrama de secuencia, varias operaciones para modelar correctamente el comportamiento del sistema.

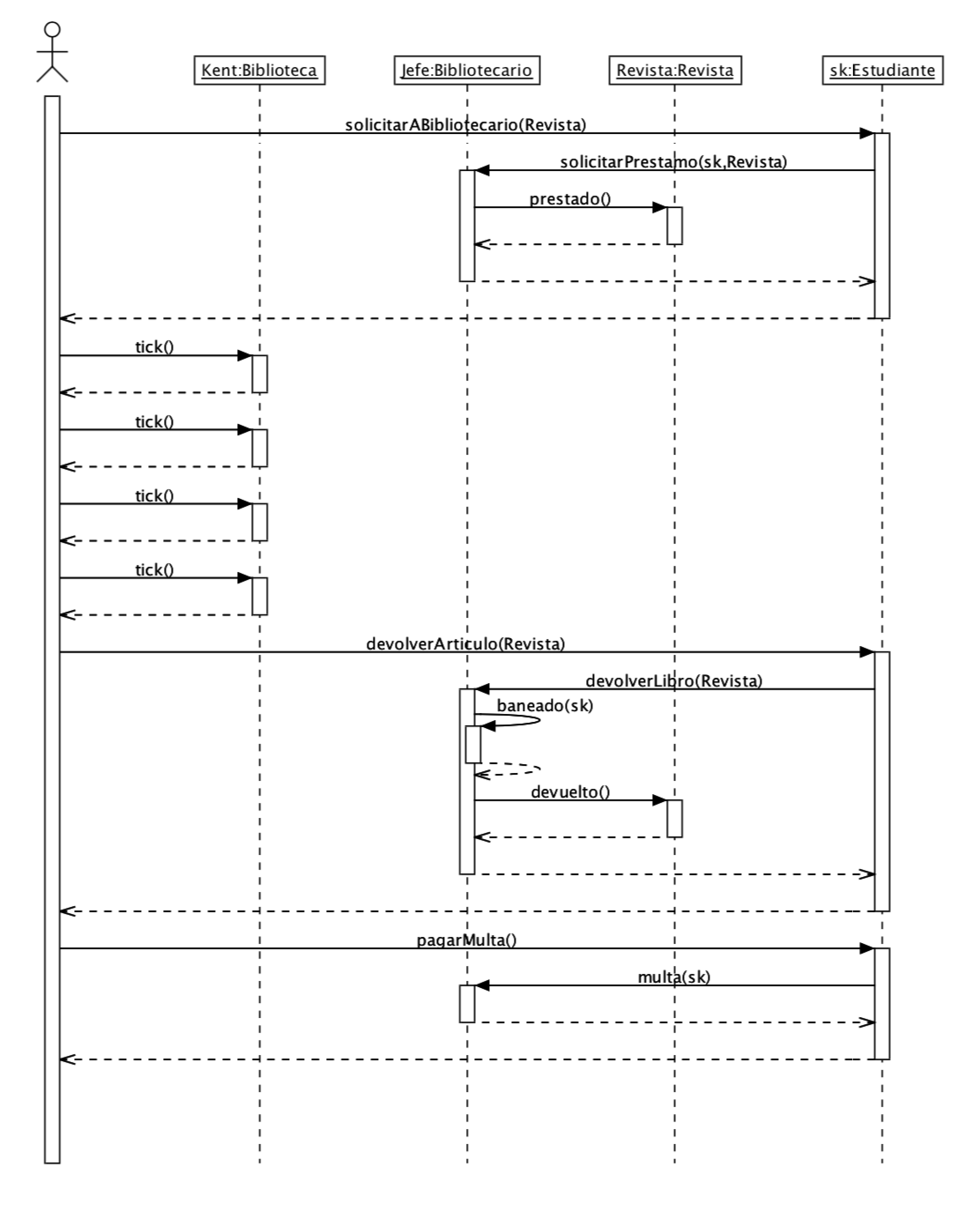
UML:



1. Vamos a definir un diagrama de estados para artículo. Tendrá 3 estados, Libre, Alquilado y NoDisponible, que representa cuando un jefe de biblioteca da de baja un artículo por estar en mal estado, o porque ha decidido retirarlo temporalmente. Para poder realizar las transiciones entre estados vamos a necesitar operaciones adicionales en la clase de Artículo, empezando por ejemplo por la operación alquilar. Además necesitaremos una operación para dar de baja un libro y devolverlo.
2. Diagrama de secuencia de un préstamo devuelto a tiempo:



1. Diagrama de secuencia de un préstamo fuera de plazo y pago de la multa:



USE:

model KentLibrary

class Biblioteca

attributes

day: Integer

maxArticulosEGrado: Integer init: 3

maxArticulosEPGrado: Integer init: 8

maxArticulosP: Integer init: 16

prestamoLibroEGrado: Integer init: 7

prestamoLibroP: Integer init: 56

prestamoLIbroEPgrado: Integer init: 14

prestamoRevistaE: Integer init: 3

prestamoRevistaP: Integer init: 14

operations

tick()

begin

self.day := self.day + 1

end

post: self.day@pre + 1 = self.day

end

class Bibliotecario

operations

solicitarPrestamo(usuario: Usuario, articulo: Articulo)

begin

insert(articulo, usuario) into prestadoA;

articulo.prestado();

articulo.prestadoA.diaObtencion := self.biblioteca.day;

end

pre usuarioNotBanned: usuario.puedeRetirar

pre estudianteGradoNoLibro: (usuario.oclIsTypeOf(Estudiante) and usuario.titulacion.oclIsTypeOf(Grado) implies articulo.oclIsTypeOf(Libro))

pre noSuperaLimitePrestamos: ((usuario.oclIsTypeOf(Estudiante) and usuario.titulacion.oclIsTypeOf(Grado)) implies usuario.articulos->size() < self.biblioteca.maxArticulosEGrado)

and ((usuario.oclIsTypeOf(Estudiante) and usuario.titulacion.oclIsTypeOf(Posgrado)) implies usuario.articulos->size() < self.biblioteca.maxArticulosEPGrado)

and (usuario.oclIsTypeOf(Profesor) implies usuario.articulos->size() < self.biblioteca.maxArticulosP)

post articuloPrestado: usuario.articulos@pre->size() + 1 = usuario.articulos->size()

post mismoUsuario: usuario = articulo.usuario

post diaObtencionToday: articulo.prestadoA.diaObtencion = articulo.biblioteca.day

devolverLibro(articulo: Articulo)

begin

declare u: Usuario;

u := articulo.usuario;

if not (articulo.oclIsTypeOf(Libro) implies ((u.oclIsTypeOf(Profesor) implies (self.biblioteca.day - articulo.prestadoA.diaObtencion < self.biblioteca.prestamoLibroP))) or

((u.titulacion.oclIsTypeOf(Grado) and u.oclIsTypeOf(Estudiante)) implies (self.biblioteca.day - articulo.prestadoA.diaObtencion < self.biblioteca.prestamoLibroEGrado)) or

((u.titulacion.oclIsTypeOf(Posgrado) and u.oclIsTypeOf(Estudiante)) implies (self.biblioteca.day - articulo.prestadoA.diaObtencion < self.biblioteca.prestamoLIbroEPgrado))) or

(articulo.oclIsTypeOf(Revista) implies ((u.oclIsTypeOf(Profesor) implies (self.biblioteca.day - articulo.prestadoA.diaObtencion < self.biblioteca.prestamoRevistaP))) or

((u.titulacion.oclIsTypeOf(Grado) and u.oclIsTypeOf(Estudiante)) implies (self.biblioteca.day - articulo.prestadoA.diaObtencion < self.biblioteca.prestamoRevistaE))) then

self.baneado(u);

delete(articulo, articulo.usuario) from prestadoA;

articulo.devuelto();

else

delete(articulo, articulo.usuario) from prestadoA;

articulo.devuelto();

end

end

post libroDevuelto: articulo.usuario->size() = 0

baneado(u: Usuario)

begin

u.puedeRetirar := false

end

cambiarValores(articulosEGrado: Integer, articulosPGrado: Integer, articulosProf: Integer, prestamoLibGrado: Integer, prestamoLibPGrado: Integer,

prestamoRevE: Integer, prestamoRevP: Integer, prestamoLibP: Integer)

begin

self.biblioteca.maxArticulosEGrado := articulosEGrado;

self.biblioteca.maxArticulosEPGrado := articulosPGrado;

self.biblioteca.maxArticulosP := articulosProf;

self.biblioteca.prestamoLibroEGrado := prestamoLibGrado;

self.biblioteca.prestamoLibroP := prestamoLibP;

self.biblioteca.prestamoLIbroEPgrado := prestamoLibPGrado;

self.biblioteca.prestamoRevistaE := prestamoRevE;

self.biblioteca.prestamoRevistaP := prestamoRevP;

end

pre esJefeBiblioteca: self.jefe.oclIsUndefined()

multa(u: Usuario)

begin

u.puedeRetirar := true;

end

post: u.puedeRetirar

retirarEjemplar(articulo: Articulo)

begin

articulo.retirado := true;

articulo.darDeBaja();

end

pre esJefeBiblioteca: self.jefe.oclIsUndefined()

end

abstract class Articulo

attributes

retirado: Boolean init: false

operations

prestado()

begin

end

devuelto()

begin

end

darDeBaja()

begin

end

pre: not retirado

statemachines

psm Estado

states

s: initial

Disponible

Prestado

NoDisponible

transitions

s -> Disponible

Disponible -> NoDisponible {darDeBaja()}

Prestado -> Disponible {devuelto()}

Disponible -> Prestado {prestado()}

end

end

class Libro < Articulo

end

class Revista < Articulo

end

abstract class Usuario

attributes

puedeRetirar: Boolean init: true

operations

solicitarABibliotecario(articulo: Articulo)

begin

self.biblioteca.bibliotecarios->asSequence()->first().solicitarPrestamo(self, articulo);

end

devolverArticulo(articulo: Articulo)

begin

self.biblioteca.bibliotecarios->asSequence()->first().devolverLibro(articulo);

end

pre articuloPrestadoASelf: self.articulos->exists(a | a = articulo)

pagarMulta()

begin

self.biblioteca.bibliotecarios->asSequence()->first().multa(self);

end

end

class Profesor < Usuario

end

class Estudiante < Usuario

end

abstract class Titulacion

end

class Grado < Titulacion

end

class Posgrado < Titulacion

end

association usuarioTitulacion between

Usuario [\*] role usuarios

Titulacion [1] role titulacion

end

associationclass prestadoA between

Articulo [\*] role articulos

Usuario [0..1] role usuario

attributes

diaObtencion: Integer

end

composition perteneceUsuario between

Biblioteca [1] role biblioteca

Usuario [\*] role usuarios

end

composition trabaja between

Biblioteca [1] role biblioteca

Bibliotecario [\*] role bibliotecarios

end

composition pertenecen between

Biblioteca [1] role biblioteca

Articulo [\*] role articulos

end

association jefeBiblioteca between

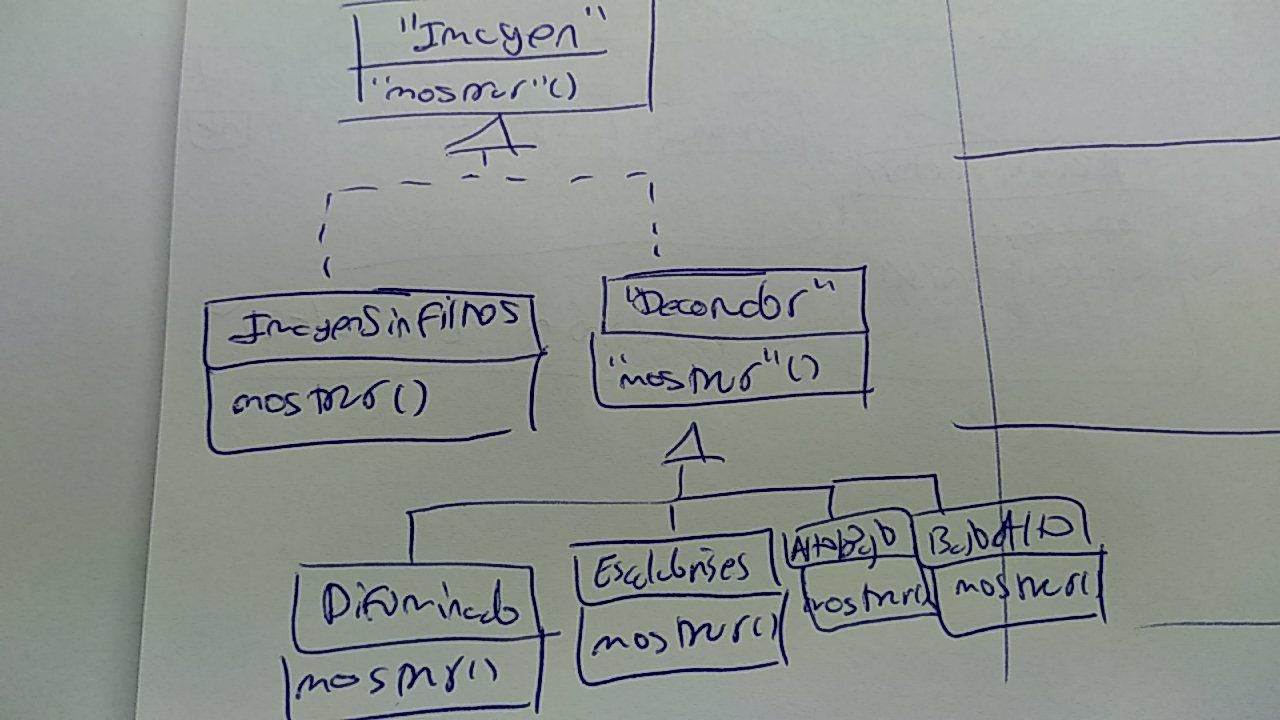
Bibliotecario [0..1] role jefe

Bibliotecario [\*] role bibliotecarios

end

**Ejercicio 2:**

1. Para poder resolver este problema utilizaremos el patrón decorador, que nos permite de manera eficiente y sencilla añadir comportamientos adicionales a una clase, en este caso, más que comportamientos adicionales, queremos añadirle filtros a nuestra imagen, para ello crearemos una subclase por cada tipo de filtrado. Para ello vamos a crear una interfaz que defina la cabecera de una operación mostrar, y de ella extenderán 2 clases, una de ellas será la ImagenSinFiltros y la otra una clase abstracta que será nuestro decorador, que nos permitirá añadir filtros a nuestra imagen.
2. Para poder aplicar correctamente al modelo el patrón decorador vamos a necesitar realizar un par de modificaciones a dicho modelo. Para empezar, creando una interfaz que represente la Imagen, y nos permita añadirle filtros. Dicha interfaz tendrá simplemente la definición de la cabecera de nuestro método mostrar. Luego, 2 clases que implementan la interfaz, una de ellas es la Imagen sin filtros, por otro lado tendremos el decorador, que tendrá también el método mostrar. Esta es una clase abstracta. Finalmente, tendremos 4 clases que hereden del decorador, cada una de ellas representa un filtrado y añade comportamiento adicional al mostrar de la clase abstracta. El diagrama UML queda de la siguiente forma:



1. [Código en Java:](https://github.com/Alkesst/ModeladoStuff/tree/master/Practica3/src/examen2018v2)