Práctica 3 (con errata corregida 6-marzo)

Introducción a la Programación Orientada a Objetos con Java

Inicio: Semana del 27 de febrero.
Duración: 3 semanas.
Entrega: Semana del 20 de marzo. (¡Ojo! Lunes 20 es festivo)
Peso de la práctica: 20%

El objetivo de esta práctica es introducir al alumno en la programación orientada a objetos con el lenguaje Java, pidiéndole que *desarrolle de forma incremental varias clases en Java (incluyendo sus pruebas y documentación)* que implementen diversos componentes software para una aplicación de biblioteca.
En el desarrollo de esta práctica se utilizarán principalmente los siguientes conceptos de Java:

- *tipos de datos primitivos, String, Array y tipos referencia* (objetos) definidos por el programador,
- *clases* sencillas definidas por el programador para implementar tipos abstractos de datos mediante *variables de instancia, variables de clase, métodos de instancia, métodos de clase y constructores.*
- *herencia, sobrescritura de métodos*
- *iniciación a las colecciones*
- *buen estilo de programación y comentarios para documentación automática mediante javadoc.*

Apartado 0. Introducción

El enunciado describe progresivamente los requisitos de funcionalidad para una aplicación de servicios bibliotecarios en cuyo paquete **`es.uam.eps.ads.p3.biblioteca`** debes ir integrando las clases que desarrolles en cada apartado y al mismo tiempo debes ir **creando y actualizando el diagrama de clases** que deberás entregar al final.

Apartado 1. Creación de objetos y cálculos básicos (3 puntos)

En esta aplicación de biblioteca los objetos más básicos que se manipulan son los usuarios de la misma (ya sean empleados o público general) y las obras de que dispone para su préstamo, que pueden ser libros o películas. Nótese, que de una misma obra ("Don Quijote de la Mancha") pueden existir numerosos ejemplares. En esta práctica también se utiliza la clase auxiliar denominada `DatosPruebas`, de la que no se crearán instancias pero en ella se crean varios objetos que servirán para simplificar las pruebas de tu código y para ilustrar la siguiente descripción de los objetos. Como se ve en el listado, `DatosPruebas` sencillamente genera un conjunto de obras, usuarios y ejemplares.

Datos para pruebas (disponible en Moodle):

```
package es.uam.eps.ads.p3.tester;
import es.uam.eps.ads.p3.biblioteca.*;

public abstract class DatosPruebas {
            public static final Obra obra1 = new Libro("La Caverna", "Saramago", 2000, "Alfaguara", 9);
            public static final Obra obra2 = new Pelicula("Atame", "Almodóvar", 1990, Genero.DRAMA);
            public static final Obra obra3 = new Libro("Crepúsculo", "Meyer", 2008, "Alfaguara", 1);
            public static final Obra obra4 = new Libro("Crepúsculo", "Meyer", 2011); // misma obra

            public static final Usuario u1 = new Publico("Ramón");
            public static final Usuario u2 = new Publico("Ana");
            public static final Usuario u3 = new Publico("Ana");  // no es el mismo usuario que u2
            public static final Usuario u4 = new Empleado("Marta");
            public static final Usuario u5 = u4;


            public static final Ejemplar e11 = new Ejemplar(obra1);
            public static final Ejemplar e21 = new Ejemplar(obra1);   // e21, ejemplar 2 de la obra 1
            public static final Ejemplar e31 = new Ejemplar(obra1);
            // el archivo en moodle contiene algunos ejemplares más ...
}
```

Todas las obras tienen título, autor y año. Los libros son obras que además tienen editorial y número de edición, pero en algunos casos se pueden tener obras de las que se desconozcan editorial y edición (ver `obra4`). Las películas son obras (con título, autor y año, aunque se entiende que el autor es su director) que además se describen como pertenecientes a un género: DRAMA, COMEDIA, TERROR, etc.

Los atributos propios de libros y películas deben poder cambiarse, pero no así los atributos comunes a las obras que no debe permitirse su modificación. Cada obra tendrá definido un *plazo de préstamo* que se calculará de forma distinta para libros y películas. Para los libros el plazo inicial de préstamo es de 25 días, pero si es un libro muy popular (se presta mucho) se reduce su plazo de préstamo a razón de 1 día por cada 10 préstamos (evitando que se reduzca por debajo de 7

días). La películas tiene un plazo de préstamo fijo de 2 días (aunque en futuras versiones se podría flexibilizar).

Todos los usuarios de la biblioteca se crean con su nombre, que no se podrá cambiar, y según sean empleados de la biblioteca o público general tendrán un comportamiento distinto en cuanto a las reglas de préstamo: el número máximo de préstamos simultáneos y las sanciones por retraso en la devolución.

Los empleados tienen ventajas (hasta 4 días de retraso sin sanción, y hasta 20 préstamos simultáneos) pero sus sanciones por retraso son económicas (2.50€ por día de retraso, los cuatro primeros también se cobran si el retraso es superior a 4 días). El total de la sanción sigue acumulándose hasta que se abone.

En cambio, el público general tiene un *número inicial de préstamos simultáneos* muy bajo, 2, que irá creciendo según vayan devolviendo más y más préstamos sin retraso, y sus sanciones por retraso no serán económicas sino que reducirán el límite máximo de préstamos simultáneos que ese usuario público haya ido acumulando. Concretamente, cuando un usuario público acumule, a lo largo del tiempo, un número de préstamos superior a su límite de préstamos simultáneos permitidos, entonces se le bonificará sumando `1` a dicho límite y se restablecerá a uno el contador que acumula el número de préstamos para la próxima bonificación. Cuando se le sancione por un retraso de *n* días, se disminuirá en *n* el número de préstamos actualmente acumulados para la siguiente bonificación y se restablecerá su límite de préstamos simultáneos al valor inicial mencionado arriba.

Tester apartado 1 (disponible en Moodle):

```
package es.uam.eps.ads.p3.tester;

import es.uam.eps.ads.p3.biblioteca.*;
import static es.uam.eps.ads.p3.tester.DatosPruebas.*;

public class TesterApartado1 {
  public static void main(String[] args) {
     // veamos el contenido de algunos objetos
          System.out.println( u1 );    System.out.println( u4 );
          System.out.println( obra1 ); System.out.println( obra2 );        // con plazos de préstamo iniciales

    // en el constructor Prestamo se debe usar  FechaSimulada.getHoy() para obtener fecha "simulada" de hoy
    Prestamo pf = new Prestamo(e11, u1); // préstamo ficticio reutilizado para simular múltiples préstamos

    // veamos cómo un usuario Publico gana privilegios según se le hace préstamos (ver enunciado)
    System.out.println(u1);
    for (int i = 1; i <= 4; i++) {
          u1.anyadirPrestamo(pf);  u1.eliminarPrestamo(pf);  System.out.println(u1);
    }
    // el plazo de préstamo de la obra 1 sigue siendo el inicial, pero BAJA tras 20 préstamos más
    System.out.println(obra1);
    for (int i = 1; i <= 20; i++) {
          u1.anyadirPrestamo(pf);  u1.eliminarPrestamo(pf);
    }
    System.out.println(obra1);

    // sanciones por retraso a un usuario Publico
    u1.sancionarPorRetraso(2);  // pierde parte de los privilegios ganados arriba
    System.out.println(u1);

    // sanciones por retraso a un Empleado
    System.out.println(u4);
    u4.sancionarPorRetraso(10);     System.out.println(u4); // suma 25
    u4.sancionarPorRetraso(3);      System.out.println(u4); // perdonado, menos de 4 días
    u4.sancionarPorRetraso(5);      System.out.println(u4); // suma 5 x 2.5 = 12.75
    u4.eliminarSancion();           System.out.println(u4); // paga sus sancion, a cero.
  }
}
```

Aunque lo anterior es la parte fundamental de este apartado 1, para poder ejecutar el primer tester tendrás que definir clases para otros objetos que se detallan en apartados posteriores. Debes tener una clase para describir cada ejemplar de una obra, y otra para describir cada préstamo. En este apartado es suficiente con que la clase `Ejemplar` tenga un constructor con la obra del ejemplar como argumento, y la clase `Prestamo` tenga un constructor con el ejemplar prestado y el usuario que se lo lleva. Además cada préstamo debe almacenar su fecha de vencimiento (importando y usando el tipo `java.time.LocalDate`) calculándola a partir de la fecha de hoy y los plazos de préstamo de cada usuario descritos arriba. *Nota*: para facilitar las pruebas, incluye en tu código la clase `FechaSimulada` (disponible en moodle) y utiliza `FechaSimulada.getHoy()` como fecha de hoy en el constructor de préstamos, y de esta forma podrás hacer pruebas con resultados reproducibles en distintos días.

Según vayas desarrollando tus clases ve construyendo un diagrama de clases UML que, al final de la práctica, deberás incluir con el resto de material entregable.

No te olvides de seguir la guía de estilo de programación Java disponible en Moodle, incluyendo comentarios, especialmente los usados para javadoc, en el código de todas las clases que escribas.

Salida esperada del tester del apartado 1:

```
[P: Ramón,ppb:0,ps:2]                    //    ppb -> prestamos para bonificación     ps -> prestamos simultáneos permitidos
[E: Marta]
[L:La Caverna, de Saramago (2000) plazo:25]
[P:Atame, de Almodóvar (1990)plazo:2]
[P: Ramón,ppb:0,ps:2]
[P: Ramón,ppb:1,ps:2]
[P: Ramón,ppb:2,ps:2]
[P: Ramón,ppb:1,ps:3]
[P: Ramón,ppb:2,ps:3]
[L:La Caverna, de Saramago (2000) plazo:25]
[L:La Caverna, de Saramago (2000) plazo:23]
[P: Ramón,ppb:2,ps:2]           Errata corregida  6-marzo:  antes decía erróneamente ppb:1, ps:2
[E: Marta]
[E: Marta $25.0]
[E: Marta $25.0]
[E: Marta $37.5]
[E: Marta]
```

Apartado 2. Definición de igualdad de objetos (con `equals`) según sea su clase (1 punto)

Antes de enfrentarnos al Apartado 3 donde leeremos requisitos como que *se debe impedir que un mismo usuario tenga prestadas simultáneamente dos ejemplares de obras iguales*, debemos aclarar (y programar) qué significa un mismo usuario y obras iguales. En los datos de prueba se comenta que los usuarios *u2* y *u3* no deben ser considerados el mismo usuario por el mero hecho de que tengan el mismo nombre. En cambio, *u4* y *u5* sí que son el mismo usuario por ser ambos referencia al mismo objeto (único objeto de clase Empleado usado en los datos de prueba). Este comportamiento básico se obtiene usando el operador **==** de Java, o el método **`equals`** heredado de la clase Object.

Sin embargo, en esta aplicación se establece que dos obras deben ser consideradas iguales siempre que tengan el mismo título y el mismo autor, independientemente de que coincidan o no en los demás atributos. Así, en los datos de prueba se comenta que *obra3* y *obra4* son obras iguales. Esta definición de igualdad debes implementarla sobrescribiendo correctamente en tu clase `Obra` el método **`equals`** de la clase Object. Esto hará más fácil que puedas programar después el requisito mencionado al comienzo de este apartado, según el cual ningún usuario puede tener prestados los ejemplares *e23* y *e14* porque ambos son ejemplares de obras iguales (la 3 y la 4). En cambio, esto no significa que los ejemplares sean iguales. Por otro lado, estos dos ejemplares se podrían prestar respectivamente a los usuarios *u2* y *u3*, que como vimos arriba no son el mismo usuario.

Comprueba el siguiente tester con su salida esperada, y añade código a tu clase `Obra` para obtener la misma salida.

Tester apartado 2 (disponible en Moodle)**:**

```java
package es.uam.eps.ads.p3.tester;

import static es.uam.eps.ads.p3.tester.DatosPruebas.*;

public class TesterApartado2 {
  public static void main(String[] args) {
     // u2 y u3 son estudiantes, ambas de nombre "Ana" pero debe considerarse distintas
     System.out.println("u2.equals(u3): " + u2.equals(u3) ); // debe imprimir false
     // u4 y u5 son el mismo objeto, comparten estructura de datos y contenido
     System.out.println("u4.equals(u5): " + u4.equals(u5) ); // debe imprimir true
     // obra3 y obra4 deben considerarse como la misma obra (mismo titulo y autor)
     System.out.println("o3.equals(o4): " + obra3.equals(obra4) ); // debe imprimir true
     // pero sus ejemplares no son iguales
     System.out.println("e23.equals(e14): " + e23.equals(e14) ); // debe imprimir false
  }
}
```

Salida esperada apartado 2:

```
u2.equals(u3): false
u4.equals(u5): true
obra3.equals(obra4): true
e23.equals(e14): false
```

Apartado 3. El objetivo principal de la biblioteca: prestar ejemplares a usuarios (3 puntos)

Apoyándose en los anteriores apartados, éste se centra en el principal objetivo: prestar ejemplares a usuarios, y consecuentemente también habrá que prever la devolución de los ejemplares prestados (pero por ahora sin preocuparnos de préstamos retrasados y sus consiguientes sanciones a los usuarios). Aunque puede que no sea el único diseño razonable, aquí se propone que añadas a tu clase `Ejemplar` los métodos para prestar el ejemplar a un usuario y para devolver el ejemplar, y que crees una clase `Prestamo` para describir el objeto que se crea al sacar un ejemplar prestado y desaparecer cuando se devuelve.

Al crear un nuevo ejemplar tan solo es necesario indicar la obra a la que corresponde, pero en esta creación la propia clase Ejemplar debe asignar al nuevo ejemplar un identificador numérico único. El tester de este apartado con su salida esperada muestra en primer lugar que los ejemplares *e11* y *e14* tienen, respectivamente, identificadores 1 y 10 (que se corresponden con su orden de creación en la clase `DatosPruebas`). Después, realiza una serie de invocaciones al método prestar para verificar su correcto funcionamiento con los requisitos siguientes:

El préstamo no se puede realizar si el usuario tiene prestados en ese momento un número ejemplares igual a su máximo de préstamos simultáneos, o si el ejemplar está en ese momento prestado a otro usuario. Tampoco se puede realizar el préstamo si el usuario ya tiene prestado otro ejemplar de la misma obra. Para ello será necesario almacenar en cada usuario los préstamos que tiene sin devolver en cada momento, que deberá poder consultarse mediante el getter `getPrestamos()` como se muestra en el tester y con las precauciones que se describen abajo.

Si el préstamo se puede realizar se debe crear una instancia de la clase `Prestamo` con el ejemplar prestado y el usuario que lo tiene en préstamo, y devolver dicha instancia como resultado del método `prestar()`. En caso de no poderse realizar el préstamo, el método `prestar()` retornará **`null`**.

Tester apartado 3 (disponible en Moodle)**:**

```
package es.uam.eps.ads.p3.tester;

import es.uam.eps.ads.p3.fechasimulada.FechaSimulada;
import static es.uam.eps.ads.p3.tester.DatosPruebas.*;
import es.uam.eps.ads.p3.biblioteca.*;

public class TesterApartado3 {
  public static void main(String[] args) {
     System.out.println( e11 );   System.out.println( e14 );

     System.out.println( e11.prestar(u1) );  // OK, prestamo realizado
     System.out.println( e12.prestar(u1) );  // OK, prestamo realizado
     System.out.println( e13.prestar(u1) );  // rechazar por exceso de préstamos, ya tiene 2 prestados

     System.out.println( e13.prestar(u4) );  // OK, porque no se prestó en linea anterior
     System.out.println( e21.prestar(u4) );  // OK
     System.out.println( e22.prestar(u4) );  // OK porque el límite de préstamos de empleados es superior a 2

     System.out.println( e13.prestar(u2) );  // rechazar por ya prestado (a cualquier usuario)
     System.out.println( e23.prestar(u2) );  // OK ejemplar 2 de obra 3 está disponible
     System.out.println( e33.prestar(u2) );  // rechazar  u2 ya tiene otro ejemplar (el 2) de la "misma" obra (la 3)

     System.out.println( e14.prestar(u2) );  // rechazar  u2 ya tiene otro ejemplar (el 2) de la "misma" obra (ver enunciado)
     System.out.println( e33.prestar(u3) );  // OK, prestamo realizado  porque u3 NO ES  el mismo usuario que u2

     e22.devolver();  // sin sanción
     e21.devolver();  // sin sanción
     e12.devolver();  // sin sanción

     FechaSimulada.avanzar( 3 ); // adelantar la fecha simulada 3 dias
     System.out.println( "\nFecha simulada: " + FechaSimulada.getHoy() );

     System.out.println( e22.prestar(u1) ); // OK, e22 se devolvió, y u1 no supera límite de préstamos simultáneos
     System.out.println( "\nPréstamos actuales de " + u1 + "\n" + u1.getPrestamos() );

     System.out.println( "Num. prestamos realizados: " + Prestamo.numPrestamosHistoricos() );   // 8 segun mis cuentas
     System.out.println( "Num. prestamos pendientes: " + Prestamo.numPrestamosPendientes() );  // 5 segun mis cuentas
  }
 }
```

Revisa la secuencia de préstamos que se intentan realizar en el tester de este apartado, comparándola con la salida esperada. Verifica que el cambio de fecha simulada se ve reflejado en el vencimiento de los préstamos posteriores a ese cambio. Al final del tester se incluyen algunas devoluciones de préstamos con el método `devolver()` sin contemplar sanciones por ahora. Dichas devoluciones permiten, por ejemplo, que un ejemplar devuelto esté ahora disponible para préstamo o que un usuario que había llegado a su límite de préstamos simultáneos pueda ahora solicitar otro préstamo.

En este apartado debes implementar con cuidado toda la gestión de los préstamos y devoluciones con métodos que permitan saber qué préstamos tiene un usuario, como se ve al final del tester. Igualmente debes implementar los métodos que se usan al final del tester para mostrar el número total de préstamos históricos y el número de préstamos pendientes de devolución. En el siguiente apartado, te pediremos que guardes no solo el número de préstamos sino también todos sus datos para poder saber, por ejemplo, cuáles vencen hoy.

Atención. El párrafo anterior menciona un método `getPrestamos()` en la clase `Usuario` que a primera vista parece un simple *getter*, pero debe implementarse de forma distinta a los demás. Dado que su valor de retorno será una estructura o colección como, por ejemplo, `List<Prestamo>`, no podemos devolverlo tal cual porque eso implicaría que desde fuera de la clase `Usuario` se podría modificar el contenido de esa lista, violándose así el principio básico de la abstracción de datos. Este método debe devolver algo así como `Collections.unmodifiableList(misPrestamos)` para evitar esas modificación no deseadas.

Salida esperada apartado 3:

```
{1[L:La Caverna, de Saramago (2000) plazo:25](disponible)}
{10[L:Crepúsculo, de Meyer (2011) plazo:25](disponible)}
{1[L:La Caverna, de Saramago (2000) plazo:25](prestado)} prestado a [P: Ramón,ppb:1,ps:2] hasta 2017-03-26
{4[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [P: Ramón,ppb:2,ps:2] hasta 2017-03-03
null
{7[L:Crepúsculo, de Meyer (2008) plazo:25](prestado)} prestado a [E: Marta] hasta 2017-03-26
{2[L:La Caverna, de Saramago (2000) plazo:25](prestado)} prestado a [E: Marta] hasta 2017-03-26
{5[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [E: Marta] hasta 2017-03-03
null
{8[L:Crepúsculo, de Meyer (2008) plazo:25](prestado)} prestado a [P: Ana,ppb:1,ps:2] hasta 2017-03-26
null
null
{9[L:Crepúsculo, de Meyer (2008) plazo:25](prestado)} prestado a [P: Ana,ppb:1,ps:2] hasta 2017-03-26

Fecha simulada: 2017-03-04
{5[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [P: Ramón,ppb:1,ps:3] hasta 2017-03-06

Préstamos actuales de [P: Ramón,ppb:1,ps:3]
[{1[L:La Caverna, de Saramago (2000) plazo:25](prestado)} prestado a [P: Ramón,ppb:1,ps:3] hasta 2017-03-26,
{5[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [P: Ramón,ppb:1,ps:3] hasta 2017-03-06]
Num. prestamos realizados: 8
Num. prestamos pendientes: 5
```

Apartado 4. Añadir sanciones y métodos para seguimiento y control de préstamos vencidos (3 puntos)

En este apartado incluirás en tus clases el mecanismo de sanción por devolución de una obra una vez vencido el plazo de devolución, así como otros métodos para conocer qué préstamos vencen hoy y cuáles están ya pasados de vencimiento. Recuerda que el cálculo concreto de sanciones para cada tipo de usuario se describió y lo debiste programar y probar en el apartado 1. Eso simplificará mucho este apartado. Además en este apartado usamos la clase `FechaSimulada` (disponible en moodle) para cambiar la fecha que debes usar en tu código y forzar así los vencimientos sin esperar días.

Al devolver un préstamo debes comprobar si su fecha de vencimiento ha pasado, y calcular el número de días de retraso:

```
long numDiasRetraso  = Period.between(fechaVencimiento, FechaSimulada.getHoy()).getDays();
```

Recuerda que en el apartado 1 ya programaste el cálculo de sanciones dado el número de días de retraso.

En el tester de este apartado primero creamos tres préstamos (de ejemplares de dos libros y una película), y después avanzamos la fecha simulada exactamente lo justo para forzar que uno de esos préstamos (e12, ejemplar de la película obra2) tenga vencimiento hoy (según fecha simulada). Se comprueba que sea así ejecutando el método `Prestamo.conVencimientoHoy()` y verificando su salida. Ahora añadimos otro préstamo (de libro) y avanzamos la fecha simulada para forzar que hoy sea el vencimiento de los dos libros prestados al principio, pero no del recién prestado. Nótese que el préstamo de la película está vencido, pero no tiene su vencimiento precisamente hoy. Más abajo en el tester se usa el método `Prestamo.pasadosDeVencimiento()` para ver todos los prestamos de vencimiento pasado y no devueltos todavía. El último paso del tester consiste en devolver todos los préstamos pasados de vencimiento, y como se puede ver en la salida, esa devolución cambia el estadio del ejemplar a disponible, aplica sanciones a los usuarios y añade la fecha de devolución al préstamo devuelto.

La característica principal común a los dos métodos mencionados en el párrafo anterior es que requieren que la propia clase `Prestamo` guarde en variables de clase o estáticas (**`static`**) una colección (una lista, por ejemplo) de todos los préstamos aún no devueltos. En el apartado anterior tuviste que almacenar el número de préstamos pendientes de devolución, pero ahora debes almacenar cada préstamo y no solo el número total de los mismos. Por ello, ya no es necesario memorizar el número, es mejor usar el tamaño de la colección o lista que los almacena.
(*Nota*: en aplicaciones completas probablemente no usarías variables estáticas para este objetivo sino una clase nueva `Biblioteca`, singleton, que almacenaría las colecciones de préstamos, usuarios, obras, …)

Tester apartado 4 (disponible en Moodle)**:**

```
package es.uam.eps.ads.p3.tester;

import es.uam.eps.ads.p3.fechasimulada.FechaSimulada;
import static es.uam.eps.ads.p3.tester.DatosPruebas.*;
import es.uam.eps.ads.p3.biblioteca.*;

public class TesterApartado4 {
  public static void main(String[] args) {
          e11.prestar(u1);      e12.prestar(u4);    e13.prestar(u1);

          FechaSimulada.avanzar( obra2.plazoPrestamo() );  // forzamos el vencimiento de la obra2 pelicula
          System.out.println( "Vencen hoy " + FechaSimulada.getHoy()  + "\n" + Prestamo.conVencimientoHoy() +"\n");

          e21.prestar(u4);

          FechaSimulada.avanzar(  obra1.plazoPrestamo() - obra2.plazoPrestamo() ); // forzamos vencimiento de libros e11 y e13
          System.out.println( "Vencen hoy " + FechaSimulada.getHoy()  + "\n" + Prestamo.conVencimientoHoy());

          // De momento solo hay uno pasado de vencimiento
          System.out.println( "\nPasados de vencimiento " + FechaSimulada.getHoy()  + "\n" + Prestamo.pasadosDeVencimiento());

          // Avanzamos un día más, los dos de vencimiento ayer, también estan vencidos hoy, total 3
          FechaSimulada.avanzar( 1 );
          System.out.println( "\nPasados de vencimiento " + FechaSimulada.getHoy()  );
          for (Prestamo p :  Prestamo.pasadosDeVencimiento() ) {
                    System.out.println( p );
          }
          System.out.println( "-----");

          // Avanzamos unos días más, y los 4 préstamos estarán vencidos, y se van devolviendo
          FechaSimulada.avanzar( 3 );
          System.out.println( "\nPasados de vencimiento " + FechaSimulada.getHoy()  );
          for (Prestamo p :  Prestamo.pasadosDeVencimiento() ) {
             if (p.devolver())  System.out.println(p);
          }
          System.out.println( "\nYa no debe haber préstamos pasados de vencimiento");
          System.out.println( Prestamo.pasadosDeVencimiento() );
  }
}
```

Salida esperada apartado 4:

```
Vencen hoy 2017-03-03
[{4[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [E: Marta] hasta 2017-03-03]

Vencen hoy 2017-03-26
[{1[L:La Caverna, de Saramago (2000) plazo:25](prestado)} prestado a [P: Ramón,ppb:2,ps:2] hasta 2017-03-26, {7[L:Crepúsculo,
de Meyer (2008) plazo:25](prestado)} prestado a [P: Ramón,ppb:2,ps:2] hasta 2017-03-26]

Pasados de vencimiento 2017-03-26
[{4[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [E: Marta] hasta 2017-03-03]

Pasados de vencimiento 2017-03-27
{1[L:La Caverna, de Saramago (2000) plazo:25](prestado)} prestado a [P: Ramón,ppb:2,ps:2] hasta 2017-03-26
{4[P:Atame, de Almodóvar (1990)plazo:2](prestado)} prestado a [E: Marta] hasta 2017-03-03
{7[L:Crepúsculo, de Meyer (2008) plazo:25](prestado)} prestado a [P: Ramón,ppb:2,ps:2] hasta 2017-03-26
-----

Pasados de vencimiento 2017-03-30
{1[L:La Caverna, de Saramago (2000) plazo:25](disponible)} prestado a [P: Ramón,ppb:0,ps:2] hasta 2017-03-26 devuelto 2017-03-
30
{4[P:Atame, de Almodóvar (1990)plazo:2](disponible)} prestado a [E: Marta $67.5] hasta 2017-03-03 devuelto 2017-03-30
{7[L:Crepúsculo, de Meyer (2008) plazo:25](disponible)} prestado a [P: Ramón,ppb:0,ps:2] hasta 2017-03-26 devuelto 2017-03-30
{2[L:La Caverna, de Saramago (2000) plazo:25](disponible)} prestado a [E: Marta $67.5] hasta 2017-03-28 devuelto 2017-03-30

Ya no debe haber préstamos pasados de vencimiento
[]
```

Apartado 5. Recordatorio final: diagrama de clases, javadoc y pruebas (penalización máxima 3 puntos)

Aunque lo normal es desarrollar todo el diagrama de clases a partir de los requisitos (y así lo hiciste en la práctica 2), en esta práctica te hemos ido guiando por la implementación mediante varios testers, y desde el principio te dijimos que fueses actualizando tu diagrama de clases. Así pues, ahora antes de entregar recuerda: (1) generar el javadoc de todas las clases del proyecto, (2) verificar que tu diagrama de clases está completo y es correcto conforme al estándar UML, y (3) incluir tus resultados para cada uno de los testers, verificando que coinciden con el esperado. Errores o deficiencias en esta parte de la entrega conllevarán penalizaciones de hasta 3 puntos.

Apartado 6 (opcional). (1 punto)

Sin cambiar la implementación con variables de clase para los préstamos actuales (no devueltos), añádase una clase `Biblioteca` Singleton (como se sugiere en la nota del apartado 4) que almacene colecciones de préstamos históricos (ya devueltos), ejemplares, obras y usuarios. Añadirle métodos para las siguientes funcionalidades:

- Eliminar de circulación un ejemplar (que no esté prestado) pero con cuidado de no invalidar la información que hay en los préstamos históricos que haya tenido.
- Eliminar una obra y retirando de circulación todos sus ejemplares.

Normas de Entrega:

- Se deberá entregar
  - un directorio **src** con todo el código Java en su **versión final del apartado 5 (recuerda guardar esta versión antes de empezar la parte opcional)**, y en caso de haber realizado la parte opcional, **otra versión final del apartado 6**, incluidos los datos de prueba y testers adicionales que hayas desarrollado en los apartados que lo requieren,
  - un directorio **doc** con la documentación generada
  - un directorio **txt** con todos los archivos utilizados y generados en las pruebas
  - un archivo PDF el **diagrama de clases** y una breve justificación de las decisiones que se hayan tomado en el desarrollo de la práctica, los problemas principales que se han abordado y cómo se han resuelto, así como los problemas pendientes de resolver.
- Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2213, entregarían el fichero: GR2213_MarisaPedro.zip.