Principios básicos a tener presentes al hacer el diseño de cualquier aplicación software:

- ***Modularidad y encapsulamiento***: el programa debe dividirse en módulos que agrupen tanto la información que se manipulará en el módulo como la funcionalidad. que permita dicha manipulación. En la Programación Orientada a Objetos (POO) el módulo más sencillo es la **clase**. Una clase es **más** que un *registro clásico de datos*, ya que **encapsula atributos** (datos con sus tipos) y **métodos** (subprogramas que establecen las operaciones que se pueden hacer con los atributos de la clase).
- ***Ocultación***: es necesario preservar la privacidad tanto de los datos de los subprogramas definidos en un módulo, para evitar accesos no deseados a los elementos del módulo (evitar que algún otro módulo pueda manipular a su antojo información ajena).
- ***Abstracción***: al definir un módulo, siempre hay que tener presente el objetivo para el que se creando e incluir en él los elementos (atributos y métodos) necesarios para conseguir este objetivo.
- ***Abierto/cerrado***: un módulo debe estar **cerrado**, es decir, contener todos los elementos necesarios para alcanzar los objetivos para el que fue creado, y funcionar perfectamente, sin errores sintácticos ni semánticos. Esto no impide que, en algún momento posterior a su creación se desee ampliar el módulo para incluir nueva información y funcionalidad, siempre garantizando que no habrá efectos laterales que afecten a las implementaciones realizadas previamente, que seguirán funcionando. Desde esta perspectiva, se puede decir que el módulo está **abierto** a modificaciones posteriores a su creación.

Partiendo de estos principios fundamentales, la siguiente tabla establece una lista de buenas y malas prácticas que deben tenerse en cuenta al hacer el diseño de cualquier aplicación. Como ejemplo, la tabla incluye una buena práctica que ya se ha comentado en clase y lo que en la asignatura se considerará una mala práctica si se aplica en su lugar. Tras la tabla se muestran ejemplos que ponen de manifiesto las ventajas del uso de las buenas prácticas y los inconvenientes de seguir malas prácticas

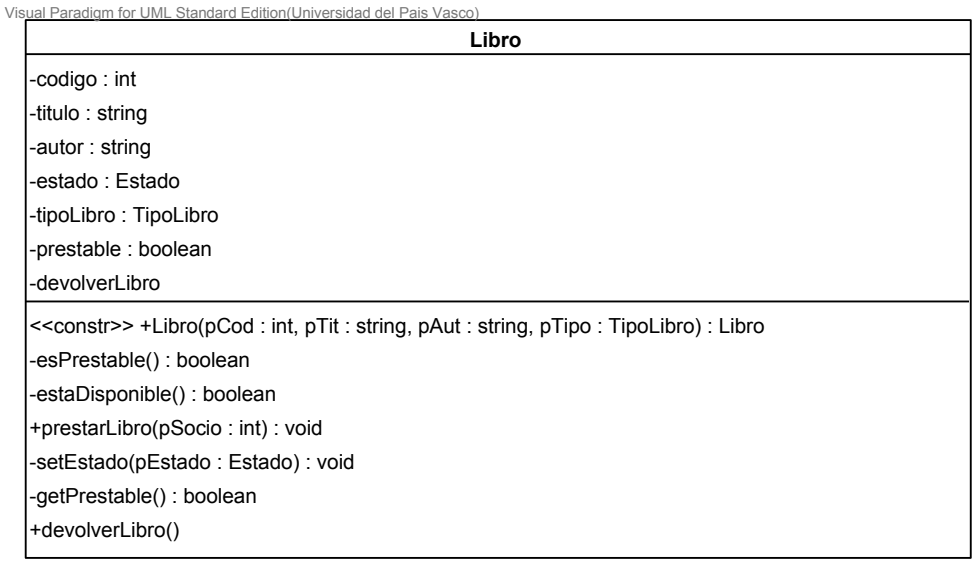
	BUENAS PRÁCTICAS	**MALAS PRÁCTICAS**
1	Representar las entidades con el nivel de abstracción adecuado a los objetivos planteados en la aplicación. Para ello: - Seguir la guía de Booch para identificar las clases y atributos necesarios para construir la aplicación. - Seguir la guía de Ellis para incluir en las clases los métodos necesarios para cubrir los objetivos de la aplicación.	Incluir clases innecesarias, cuyo uso no se justifica desde ninguna perspectiva. Duplicar código innecesariamente. No definir las clases necesarias. No definir las clases con el nivel de abstracción adecuado a los objetivos planteados en la aplicación. No definir operaciones privadas que representan pequeñas partes de otras operaciones públicas de la misma clase. Esta mala práctica da lugar a que en los diagramas

		de secuencia aparezcan numerosas flechas seguidas de entrada y salida entre los mismos bloques de activación, en vez de sólo una que represente toda esa funcionalidad encapsulada. **Ejemplo 1**.
2	Dar nombres significativos a las clases, atributos y métodos. Utilizar algún criterio de nomenclatura, preferiblemente estándar. En la asignatura se exigirá utilizar el estilo CamelCase.	Utilizar nombres que no significan nada y dificultan la comprensión del diseño.
3	Definir siempre los atributos privados	Definir atributos públicos o protected por decreto.
4	Definir los métodos privados hasta que resulte obvio que otra clase necesita usarlos	Definir métodos públicos cuando no hay evidencias de que otra clase los necesite
5		Definir métodos públicos para acceder a los atributos privados de una clase, permitiendo que otras entidades manipulen a su antojo información privada de la clase. Esto es muy **PELIGROSO** ya que compromete la ocultación de la información de la clase
6	Incluir en cada clase los métodos necesarios para gestionar sus atributos. Habitualmente, la implementación de estos métodos requiere delegar en otras clases parte del proceso, para lo que se les enviaran mensajes pidiéndoles que cada una realice su parte, gestionando, cada una, sus propios atributos. En la topología del diagrama de secuencia puede apreciarse de forma clara este proceso de delegación, al identificar flechas que salen de un bloque de activación en una clase para pedir a otra entidad que realice la parte del proceso en la que se debe manipular su información privada. **Ejemplo 2.**	Manipular de forma directa los atributos privados de una clase desde otras entidades. Esta mala práctica es una consecuencia de definir indiscriminadamente métodos públicos de acceso a los atributos privados de las clases, lo que permite que otras entidades puedan manipularlos directamente en vez de invocar a los métodos propios de cada clase. Esta mala práctica se detecta fácilmente observando la topología de los diagramas de secuencia, en los que se aprecian cantidad de flechas de ida vuelta, para llevar los contenidos de los atributos a alguna entidad que se encargará de manipularlos a su antojo. Normalmente esa clase incluye un método que implementará todo el proceso, saltándose todos los principios de modularidad, encapsulamiento y ocultación. **Ejemplo 3.**
7	Utilizar patrones de diseño, siempre que sea posible.	Hacer un diseño dependiente del lenguaje de programación, incluyendo en él referencias a instrucciones específicas del lenguaje, en vez de seguir los patrones de diseño independientes del lenguaje de programación
8	En los diseños de esta asignatura sólo se diseña el proceso correspondiente a la manipulación de los objetos del modelo del dominio, por lo que nuestros diseños siempre deben comenzar pasando un mensaje a alguna entidad Singleton.	Arrancar el proceso, pasando un mensaje a una instancia de un TAD. Un TAD puede tener muchas instancias, así que ¿cómo indicar a cuál de ellas se quiere enviar el mensaje?
9	Cuando una clase incluye varios atributos para representar las características propias de las instancias de esa clase y uno de esos atributos es	Definir un atributo de tipo Colección<*Objeto*>, cuando lo que realmente está definiendo el enunciado una entidad con significado

	de tipo Colección, conviene definir una nueva clase Lista*Objetos*, para indicar concretamente (y NO de forma genérica) el tipo de la Colección. De esta forma la clase Lista*Objetos*, contendría las operaciones concretas para el manejo de los elementos de la lista y podría utilizarse también para que la referencien desde los atributos y métodos de otras clases. **Ejemplo 4**	específico. Realmente al incluir un atributo Colección en la clase, si se desea preservar la ocultación del contenido de la Colección se está forzando a que la clase que contiene ese atributo incluya los métodos necesarios para gestionar la Colección. En ocasiones esto da lugar a implementar el mimo código en distintas clases. **Ejemplo 4**
10	En la constructora de una clase que contiene atributos de tipo Colección, esos atributos deben inicializarse con la Colección vacía y rellenarlos posteriormente. Con esto, se está reservado un espacio de memoria propio para ese atributo, que evitará efectos laterales posteriores. **Ejemplo 5**	Pasar una Colección como parámetro de la constructora de una clase. Si en la implementación de la constructora se asigna directamente dicho parámetro de tipo Colección al atributo, realmente el atributo de la clase contendrá la dirección de la Colección pasada como parámetro y, probablemente, haya más clases con acceso a esa dirección, por lo que podrían modificar la Colección sin que la clase creada se entere ➔ posibles INCOHERENCIAS debidas a la pérdida de ocultación . **Ejemplo 5**
11	La implementación básica de los métodos para incluir o borrar elementos de una lista se hace en las clases que contienen el atributo de tipo Colección. El resto de clases que incluyan atributos del tipo Lista*Objetos* se limitarán a ir delegando la ejecución de estas instrucciones. **Ejemplo 6**	Implementar la eliminación o borrados de elementos de una Lista*Objetos* en clases que no Contienen directamente la Colección de Objetos. **Ejemplo 6**

Ejemplo 1.
BUENA PRÁCTICA:

En el ejercicio de la Biblioteca se establece que es posible prestar un libro si es prestable (no se trata de una enciclopedia, diccionario, …) y está disponible en la Biblioteca (no lo tiene ningún otro Socio). En la clase Libro se define el método esPrestable() para determinar si es posible prestar un determinado libro.



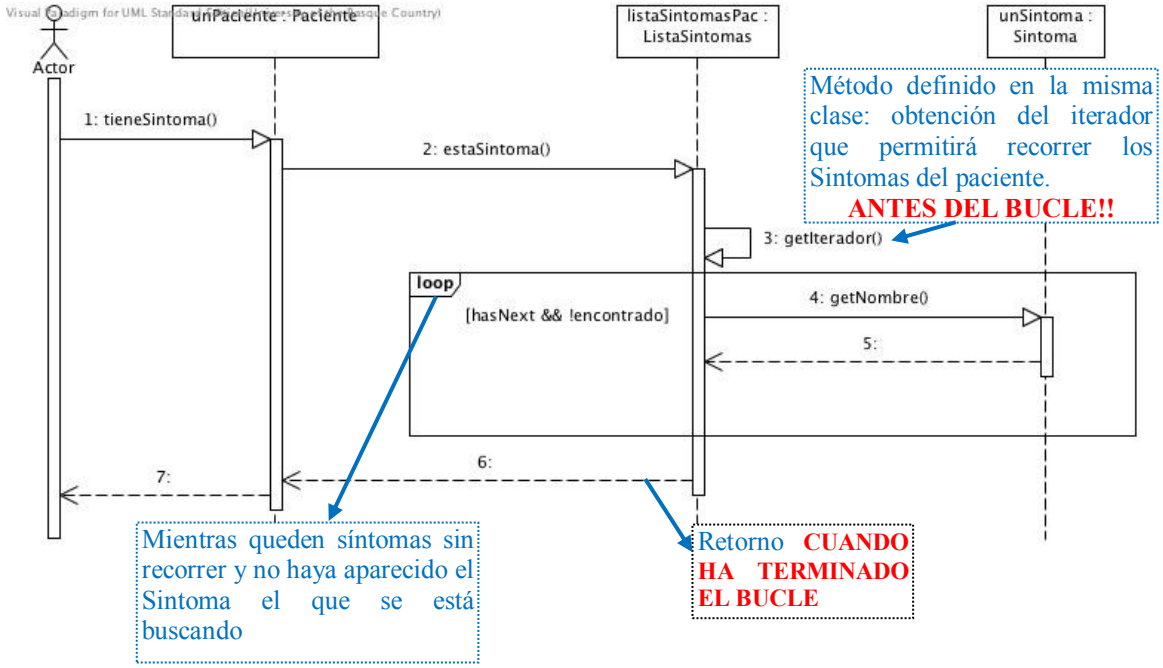
En la implementación del método esPrestable se invoca a otros dos métodos privados definidos en la misma clase:

```
private boolean esPrestable(){
        return getPrestable() && estaDisponible();
}
```

La definición de este método para encapsular la ejecución de getPrestable() y estaDisponible(), simplifica la interpretación del diagrama de secuencia del método prestarLibro(), que sólo incluirá una flecha para comprobar si es posible prestar el libro, en vez de las dos para representar cada condición

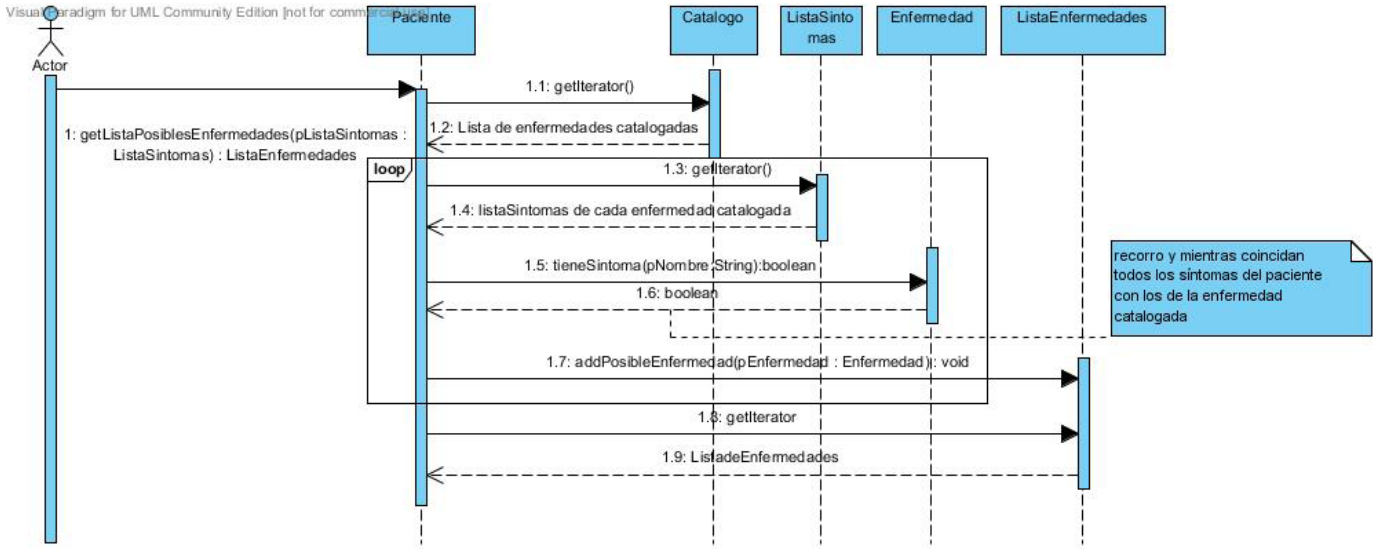
Ejemplo 2
BUENA PRÁCTICA:

Para determinar si un Paciente tiene un síntoma, se envía el mensaje a la ListaSintomas del paciente en vez de llevar la lista de síntomas hasta la clase Paciente. De esta forma será la propia clase ListaSintomas la que acceda a los síntomas del paciente, evitando exponer sus atributos privados y que la clase Paciente pueda modificarlos.

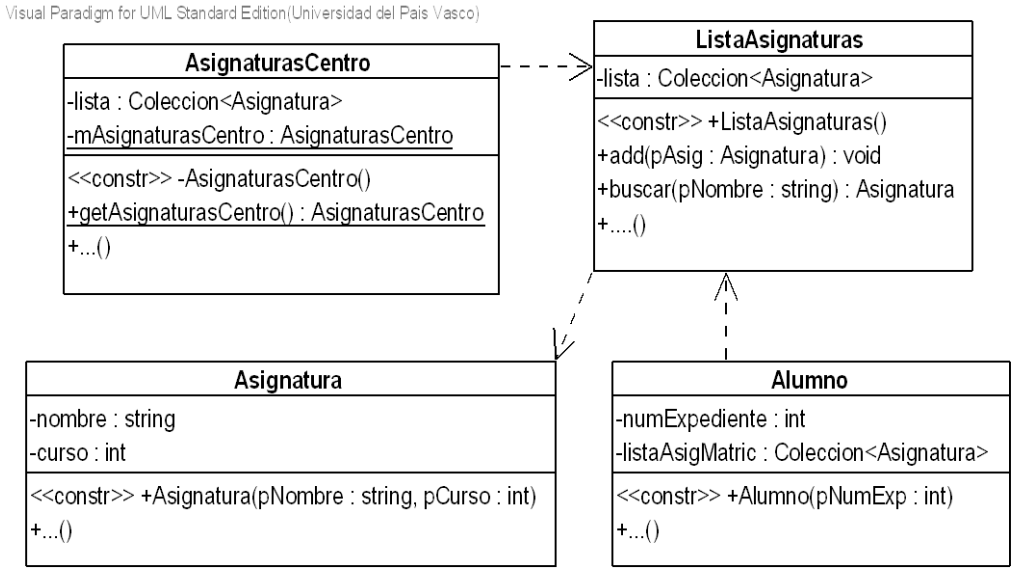


Ejemplo 3
MALA PRÁCTICA:

En este diseño, la clase Paciente, está manipulando los atributos privados de las clases Catálogo, ListaSintomas y Lista Enfermedades ➔No respeta el principio de ocultación.

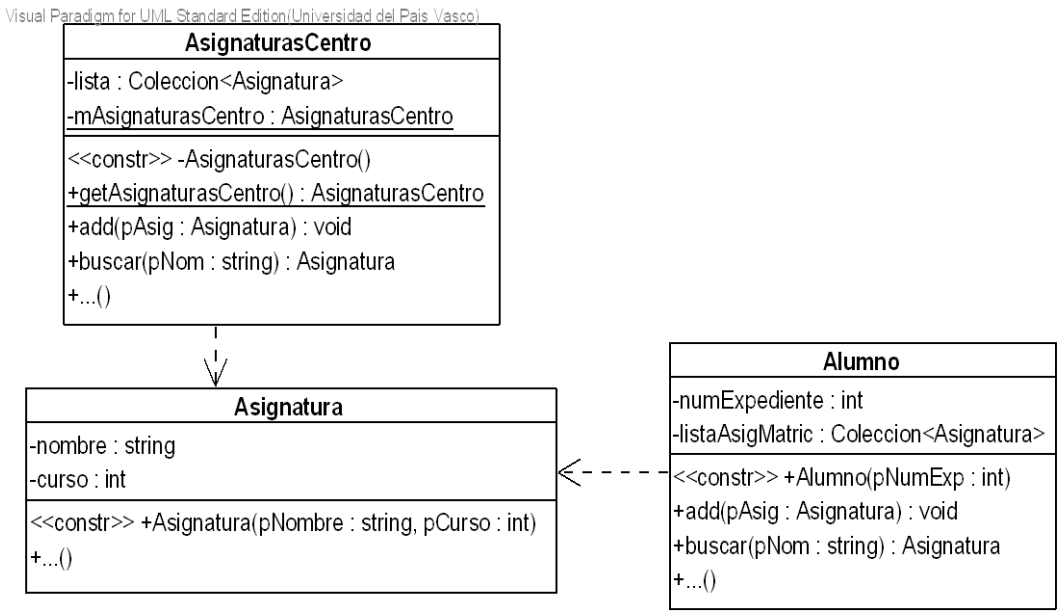


Ejemplo 4
BUENA PRÁCTICA:



Definiendo la clase ListaAsignaturas, los métodos para gestionar los elementos de la lista, se definen dentro de la misma clase, una sóla vez, independientemente de que otras clases contengan atributos de este tipo.
VENTAJAS: favorece el encapsulamiento y la ocultación, evita duplicar código.

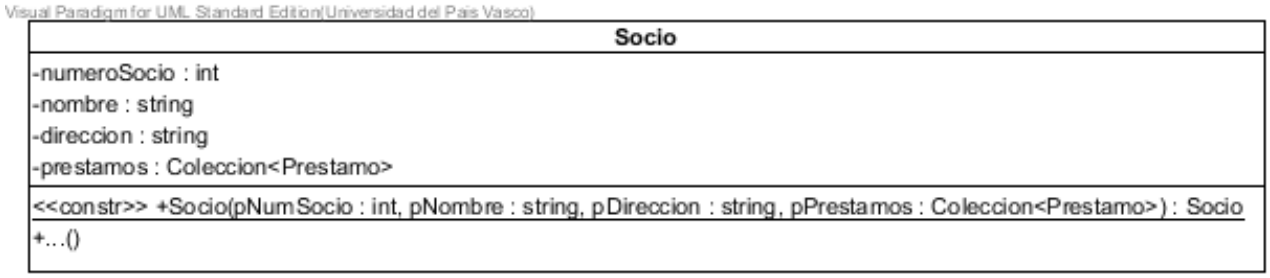
MALA PRÁCTICA



La primera incoherencia aparece si se tiene en cuenta que ***el diseño debe intentar ser una representación del mundo real***. Desde este punto de partida, carece de lógica que el Alumno disponga de métodos para gestionar la lista de asignaturas en las que está matriculado. Los alumnos siempre deben ir a secretaría si desean hacer cualquier cambio en su matrícula, no pueden acceder por su cuenta a GAUR y cambiarla a su antojo.

DESVENTAJAS: aparece código duplicado. Tanto la clase AsignaturasCentro como Alumno deben implementar los métodos que gestionan la colección de asignaturas. La duplicación de código es INTOLERABLE!!
Hay dos clases que están gestionando colecciones de asignaturas, por lo que ante cualquier fallo en una de estas colecciones, no se tendrá certeza de la clase en la que se ha producido el error ➔ BAJA ENCAPSULACIÓN Y OCULTACIÓN (muchas entidades con acceso a la misma información)

Ejemplo 5
Supongamos que tenemos una aplicación que incluye la clase Socio que se especifica a continuación



BUENA PRÁCTICA:

No incluir como parámetro de la constructora el ArrayList con la lista de préstamos del Socio. En su lugar crearemos esta lista vacía a la que se podrán ir añadiendo las instancias de préstamos del Socio. VENTAJA: si dos socios tuvieran una lista de préstamos con los mismos préstamos, lo que una haga con su lista, no afecta para nada a la lista de la otra.

```
public class Socio{
        prívate int socio;
        prívate String nombre;
        prívate String direccion;
        private ArrayList<Prestamo> prestamos;

    public Socio (int pNumSocio, String pNombre, String pDireccion, ArryList<Prestamo> pPrestamos){
        socio = pSocio;
        nombre = pNombre;
        dirección = pDireccion;
        prestamos = new ArrayList<Prestamo>; /* Reserva de una zona de memoria diferente
                                                a la del parámetro */
        Prestamo unPrestamo:
        Iterator<Prestamo> it =prestamos.getIterador();
        while (it.hasNext()){
            unPrestamo=it.next();
            prestamos.add(unPrestamo);
    }
    ...
}
```

/*__NOTA!!!__ Con esta solución garantizamos que cada socio tiene protegido el acceso a su lista de préstamos, ya que tiene una zona de memoria propia en la que almacena los PUNTEROS a sus préstamos, NO COPIAS de las instancias de Prestamo correspondientes. */

MALA PRÁCTICA:

```
public class Socio{
        prívate int socio;
        prívate String nombre;
        prívate String direccion;
        private ArrayList<Prestamo> prestamos;

    public Socio (int pNumSocio, String pNombre, String pDireccion, ArryList<Prestamo> pPrestamos){
        socio = pSocio;
        nombre = pNombre;
        dirección = pDireccion;
        prestamos = pPrestamos;
    }
    …
}
```

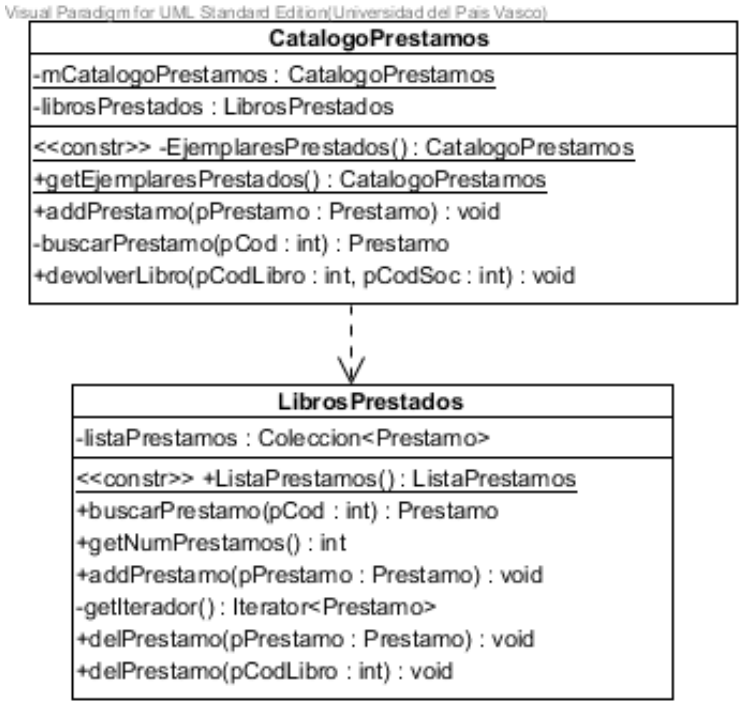
Supongamos que otra clase Biblioteca, encargada de gestionar todos los Socios, darlos de alta, modificar sus préstamos, etc…

```
public class Biblioteca{
        …

    public void gestionarSocios()
        int numSocio;
        String nombre;
        String dirección
        ArrayList<Prestamo> listaPrestamos = new ArrayList<Prestamo>();
/* Si un socio quiere darse de alta en la Biblioteca será necesario:
   - Pedirle los datos personales ➔ se guardan en las variables locales
   - Generar los Prestamos de los libros que ha cogido */
        Prestamo p1 = new Prestamo(…);
        listaPrestamos.add(p1);
        Prestamo p2 = new Prestamo(…);
        listaPrestamos.add(p2);
// Se genera la instancia del Socio
        Socio unSocio = new Socio(pNAumSocio, pNombre, pDireccion, listaPrestamos);
    …
/*CUIDADO!!! Cualquier modificación de la variable listaPrestamos afectará a los libros que tiene que Socio ➔ se está exponiendo información privada del Socio */
    }
    …
}
```

Ejemplo 6

En el ejercicio de la Biblioteca, se dispone de la clase CatalogoPrestamos para representar todos los préstamos que tiene la biblioteca y de la clase ListaPrestamos para representar cualquier Colección de préstamos.



BUENA PRÁCTICA:

public class LibrosPrestados{

```
    …
    public void addPrestamo(Prestamo pPrestamo){
        listaPrestamos.add(pPrestamo); //Método propio del lenguaje que se esté usando
    }
    …
}
```

public class CatalogoPrestamos{

```
    …
    public void addPrestamo(Prestamo pPrestamo){
        librosPrestados.addPrestamo(pPrestamo); /*  Delegación en la clase a la que
                                                    pertenece el atributo */
    }
    …
}
```

MALA PRÁCTICA:

```
public class CatalogoPrestamos{
    …
    public void addPrestamo(Prestamo pPrestamo){
        librosPrestados.getListaPrestamos().addPrestamo(pPrestamo); /*CUIDADO!!! Acceso
                                                                    al atributo privado de otra clase para moficicarlo*/
    }
    …
}
```