TEMA 3: Reutilización y polimorfismo

Ejercicios de las lecciones 3.1 y 3.2

1.Dada la siguiente clase en Java:

```
public class Vertebrado
{...
public ArrayList<String> partesDelAbdomen();
public void desplazarse();
public String comunicarse(Vertebrado vertebrado);
}
```

Indica con una cruz en la casilla correspondiente según si la declaración de los siguientes métodos de la clase Mamifero, subclase de Vertebrado, sobrecargan (overloading) o redefinen (overriding) a los de la clase Vertebrado:

	Sobrecarga / Overloading	**Redefinición / Overriding**
public ArrayList<String> partesDelAbdomen();		
public void desplazarse(Modo m);		
public String comunicarse(Vertebrado vertebrado);		
public String comunicarse(Mamifero mamifero);		

2. A partir de las siguientes clases, implementa la clase *Alumno-Informatica* que hereda de Alumno. Debe tener una nueva variable de instancia que es una colección de strings con los *dispositivos* que utiliza: como por ejemplo {PC, tablet, smartphone}. Esa variable debe poder consultarse y modificarse. También debe heredar el método *estudiar*(), pero modificando su implementación para que además de estudiar como los otros alumnos, diga que estudia con el último dispositivo de su colección. No olvides implementar su constructor. Implementa este mismo problema en Ruby.

```
class Persona {
   protected String nombre ;
   public Persona(String nom) {  this.setNombre(nom); }
   protected String getNombre() { return this.nombre; }
   protected void setNombre(String nom) { this.nombre=nom; }
   public String hablar(){ return 'bla bla bla';}
}
class Alumno extends Persona {
        public String  carrera;
        public int curso;

public Alumno(String nom, String carr,int cur)
{       super(nom);
        carrera=carr;
        curso=cur;}
        public void estudiar() { System.out.println('estudiando'); }
}
```

3. Dada la siguiente clase abstracta en Java,

```
abstract class Transporte {
   protected String marca;
   protected String getMarca() { return this.marca; }
   protected void setMarca(String marc) { this.marca=marc; }
   public abstract String hacerRuta(String origen, String destino);
}
```

Crea dos nuevas clases, por ejemplo Bicicleta y NaveEspacial, que hereden de ella y que implementen el método *hacerRuta* de diferente forma cada una. Dibuja el diagrama de clases UML correspondiente a este ejercicio.

4. Haz el ejercicio anterior en Ruby.

5. Implementa en Java y en Ruby una jerarquía de herencia en la que estén representados los distintos tipos de instrumentos musicales: viento, percusión y cuerda. Además, los de viento pueden ser de madera o de metal. Todas las clases deben tener implementados los siguientes métodos:

6. ***tocar(String nota)****:* muestra la nota que se le pasa (do, re, mi, fa, so, la si).
7. ***ajustar()****:* muestra "soy el instrumento fulanito y me estoy ajustando" .
8. ***queSoy()****:* muestra el tipo de instrumento que es.

Escribe un pequeño main() en el que podamos oír a la orquesta.

6. Representa mediante un diagrama de clases las relaciones entre los distintos tipos de clases que podemos encontrarnos en Java: Clase, ClaseAbstracta, ClaseConcreta, ClaseHoja y ClaseNoHoja, siendo ClaseHoja aquella que está al final de una jerarquía de herencia (aquella que ya no tiene subclases).

7. Dado el siguiente supuesto:

> *"Queremos diseñar un sistema software para el control por voz de los siguientes dispositivos de regulación de temperatura de una habitación: persianas, ventanas, radiadores de calefacción, dispositivos combo A/C - bomba de calor y cortinas. Debe existir un dispositivo de control general para cada habitación (DCGHabitacion) que dispone de un reconocedor de voz para reconocer las órdenes verbales: "maximizarConfort", "maximizarAhorro", "optimizarUso" de forma que se activen/desactiven los distintos dispositivos de regulación de temperatura de la forma más apropiada para conseguir el máximo confort (25 ° C), el máximo ahorro (no se gasta energía) y la máxima relación confort/ahorro (21° C) respectivamente teniendo en cuenta que todos tienen un termómetro incorporado que les da la temperatura ambiente"*

Completa la tabla de la página siguiente, en la que la primera columna se indican descripciones adicionales del supuesto anterior.

En la segunda columna indica el mecanismo de reutilización usado para resolver ese problema.

Los mecanismos de reutilización (MR) pueden ser:

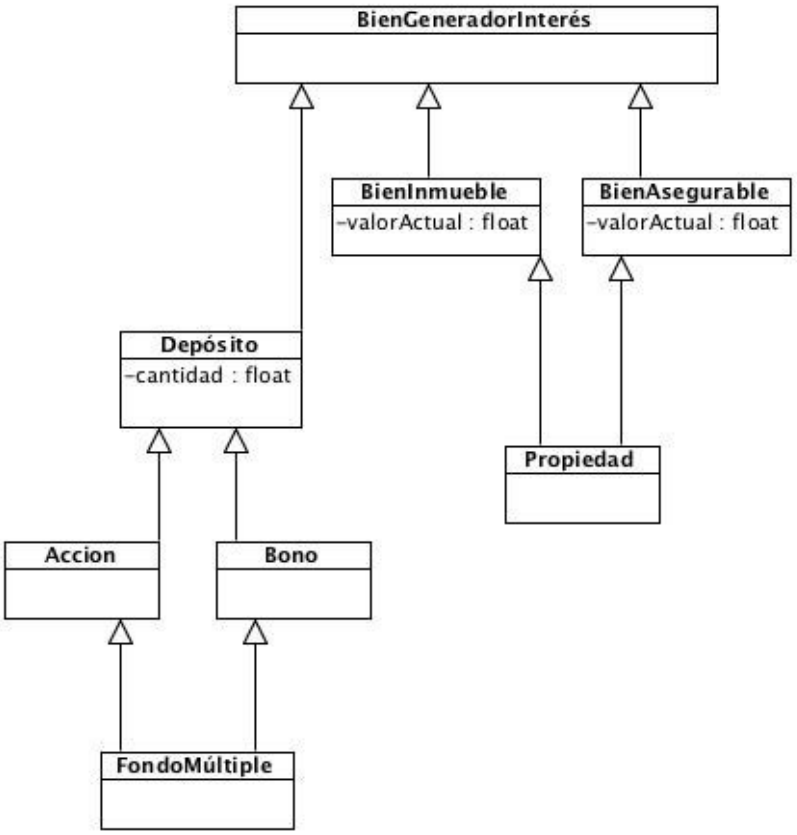
a. Herencia simple de clases, con una clase padre concreta.
b. Herencia simple de clases, con una clase padre abstracta.
c. Herencia múltiple.
d. Implementación de interfaces.
e. Clase parametrizable.
f. Composición de clases.

g. Ninguna reutilización: sobrecarga de operadores (java) / método con un número variable de argumentos (ruby)

En la tercera columna escribe la cabecera de las clases o métodos que resuman la aplicación del mecanismo elegido.

Descripción adicional del problema	MR	Código en Java o Ruby para implementarlo
1. *Todos los dispositivos de regulación de temperatura tienen que activarse y desactivarse*		
2. *Hay dispositivos de regulación térmica que consumen energía para su funcionamiento y otros que no la necesitan*		
3. *Existe un tipo de cortina llamado "store" pero se activa/desactiva bajando/subiendo como las persianas.*		
4. *Cuando un dispositivo se activa hay que indicar el grado de activación, siendo éstos los siguientes (1,2,3,4 y 5), siendo el valor más alto el de mayor activación. Si no se especifica nada, se asume la máxima activación.*		
5. *Los dispositivos combo A/C - bomba de calor estan compuestos de A/C y bomba de calor que deben reconocer la orden "calentar", "enfriar" que permita ponerlos en funcionamiento como A/C o bomba de calor respectivamente.*		
6. *Existe un tipo de radiador que incorpora un termostato y que acepta la orden "regular" con una temperatura como argumento.*		
7. *También existe un Dispositivo de Control Específico en la habitación (DCE) el cuál debe poder traducir las órdenes "maximizarConfort", "maximizarAhorro", "optimizarUso" por niveles concretos de activación o desactivación, para cada tipo de dispositivo de regulación térmica*		

8. Considerando el diagrama de clases con herencia múltiple que se presenta a continuación, identifica todos los conflictos que se darían y propón distintas alternativas para solventarlos indicando sus pros y contras.



9. Considerando el diagrama anterior:

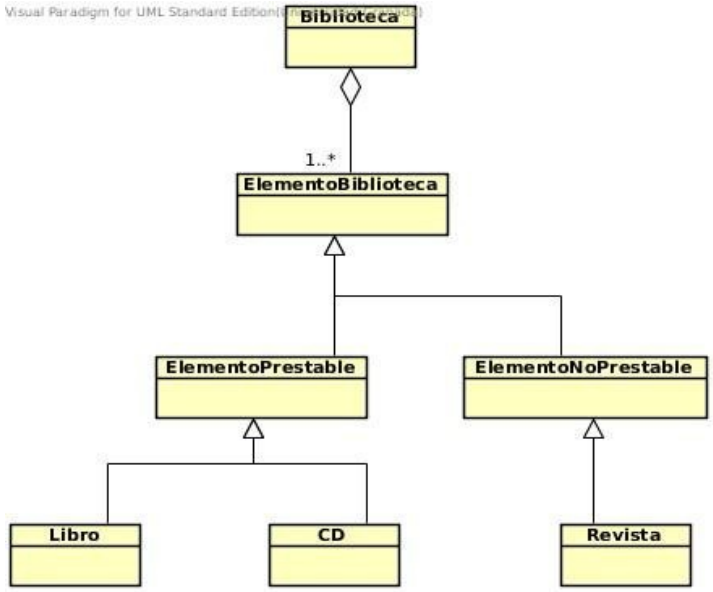
•Implementa una solución en Java, simulando la herencia múltiple mediante interfaces.
•Implementa una solución en Ruby, simulando la herencia múltiple mediante módulos.

•¿Qué inconvenientes encuentras en los mecanismos de simulación vistos en clase?

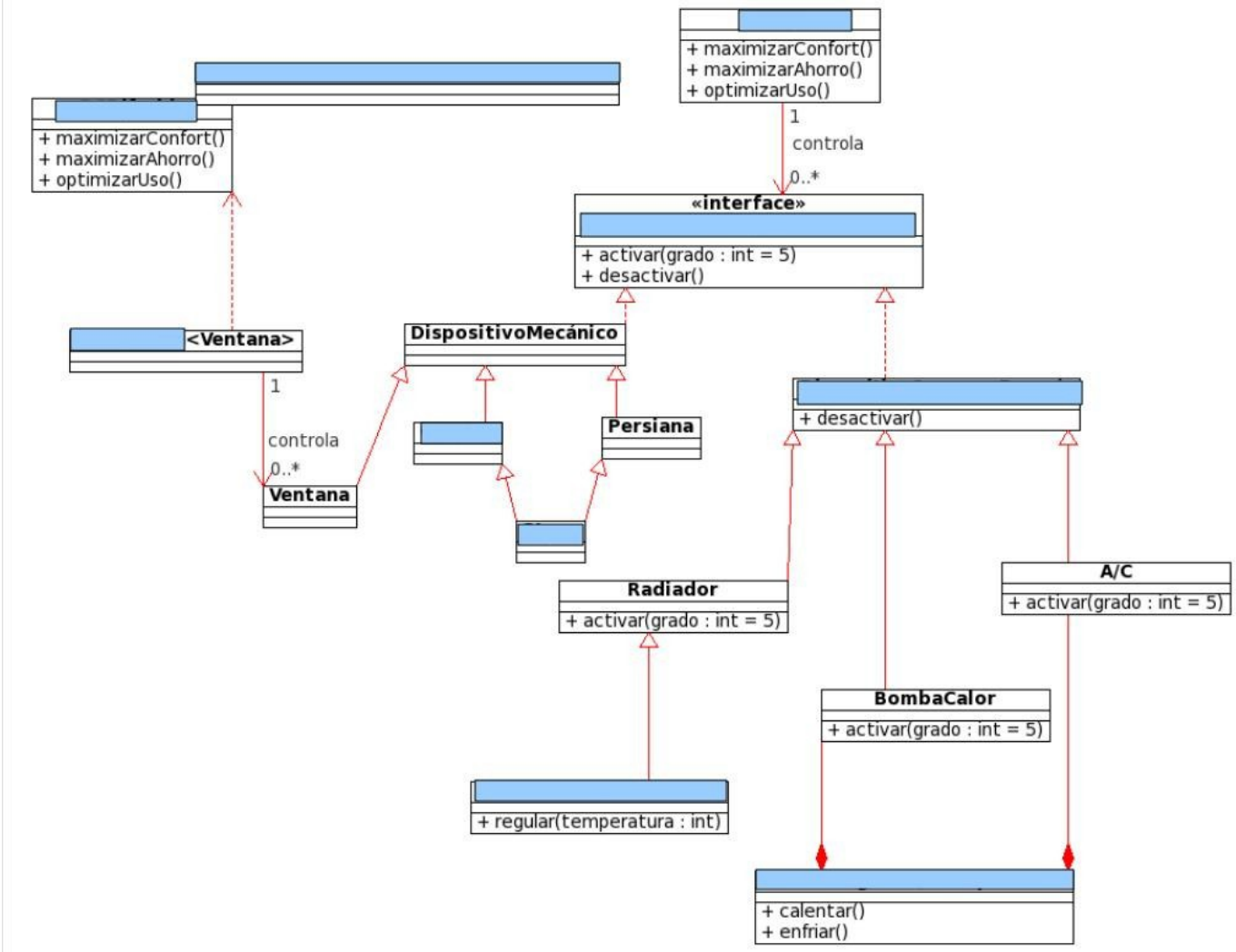
10. Implementa en Java una clase paramétrica a partir de la cual se puedan definir clases de grupos de personas con un líder, como por ejemplo grupos de música con un cantante, o empresas con un jefe. Implementa también alguna de esas clases a partir de la clase paramétrica definida.

11. Representa mediante un diagrama de clases todas las clases e interfaces que parten o están relacionadas con la clase HashMap de Java, así como sus relaciones. La clase *HashMap* está parametrizada con dos parámetros <K,V>. Representa en este mismo diagrama de clases un nuevo *HashMap* donde K es un objeto Point y V es una colección de objetos Point. Para ello, se recomienda utilizar la documentación de Java (javadoc).

12. Partiendo del siguiente diagrama de clases, modifícalo añadiendo una interfaz *Prestable* implementada por la clase *ElementoPrestable* y sus subclases. Añade a este diagrama de clases los atributos y operaciones de las distintas clases y de la interfaz *Prestable*.

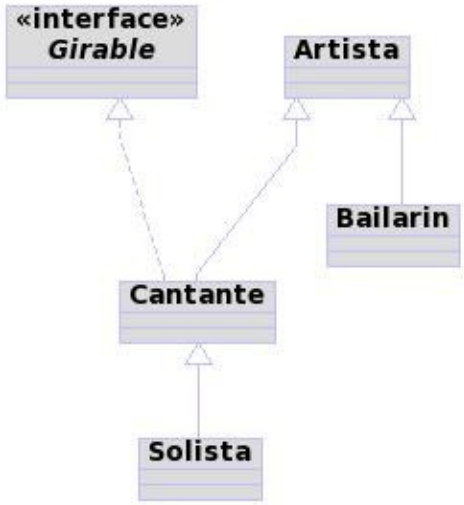


13. Escribe los nombres de clases que faltan en el diagrama de clases siguiente para cumplir con los requisitos funcionales especificados en el ejercicio 7 de esta relación.



Ejercicios de la lección 3.3

14. Teniendo el cuenta el siguiente diagrama de clases, señala en la segunda columna de la tabla aquellas líneas de código que contienen un error de compilación por incompatibilidad de tipos (ECIT) y escribe en la tercera columna la corrección necesaria para evitarlos.



Código	ECIT	Corrección
Girable arti=new Artista();		
Cantante cant1=new Cantante();		
Cantante sol=new Solista();		
Solista cant2=new Cantante();		
Bailarin bail= new Artista();		

15. Teniendo en cuenta:

- el diagrama de clases del ejercicio anterior,
- que se han resuelto todos los errores del ejercicio,
- que en Java la primera posición de los contenedores es la 0,
- que todos los artistas actúan, pero sólo los cantantes cantan y sólo los solistas cantan solos,

marca las líneas donde se produce un error, indicando en la segunda columna si es de compilación (C) o de ejecución (E) y en la tercera el código correcto.

Código	Error(C/E)	Corrección
List<Artista> lista=new ArrayList();		

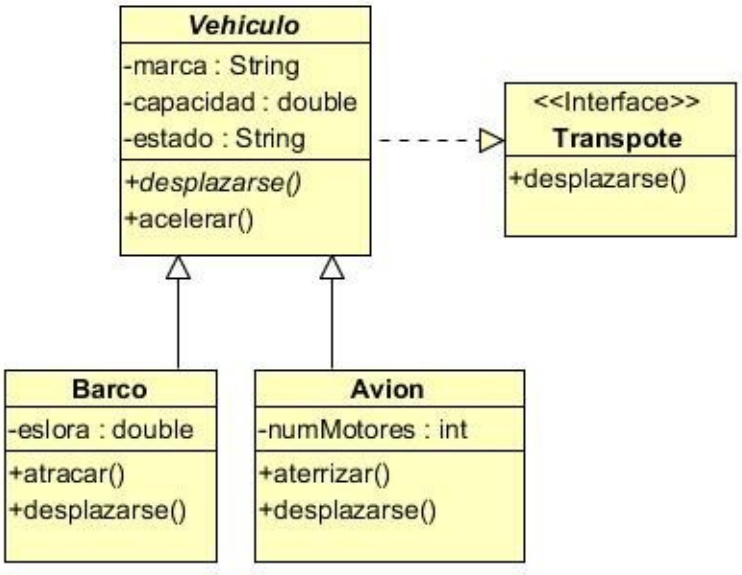
lista.add(arti);
lista.add(cant1);
lista.add(sol);
lista.add(cant2);
lista.add(bail);		

lista.get(1).canta();		
lista.get(0).actua();		
(Solista) lista.get(3).cantaSolo();		

16. Pon tres ejemplos en Java de polimorfismo basándote en el diagrama de clases del ejercicio 12, indicando un ejemplo correcto, un ejemplo que presente errores de compilación y otro que presente un error de ejecución.

17. Usando la clase *HashMap* de Java, vista en el ejercicio 10, pon dos ejemplos de polimorfismo que sean correctos y dos que no lo sean.

18. Dado el siguiente diagrama de UML, impleméntalo en Java. Presta atención a que el nombre de la clase Vehículo y su método desplazarse() aparecen en cursiva.



19. Teniendo presente el diagrama anterior, indica cuáles de los siguientes trozos de código son correctos y cuáles darían error de compilación o de ejecución. En caso de que den error, indica por qué y cómo lo solucionarías.

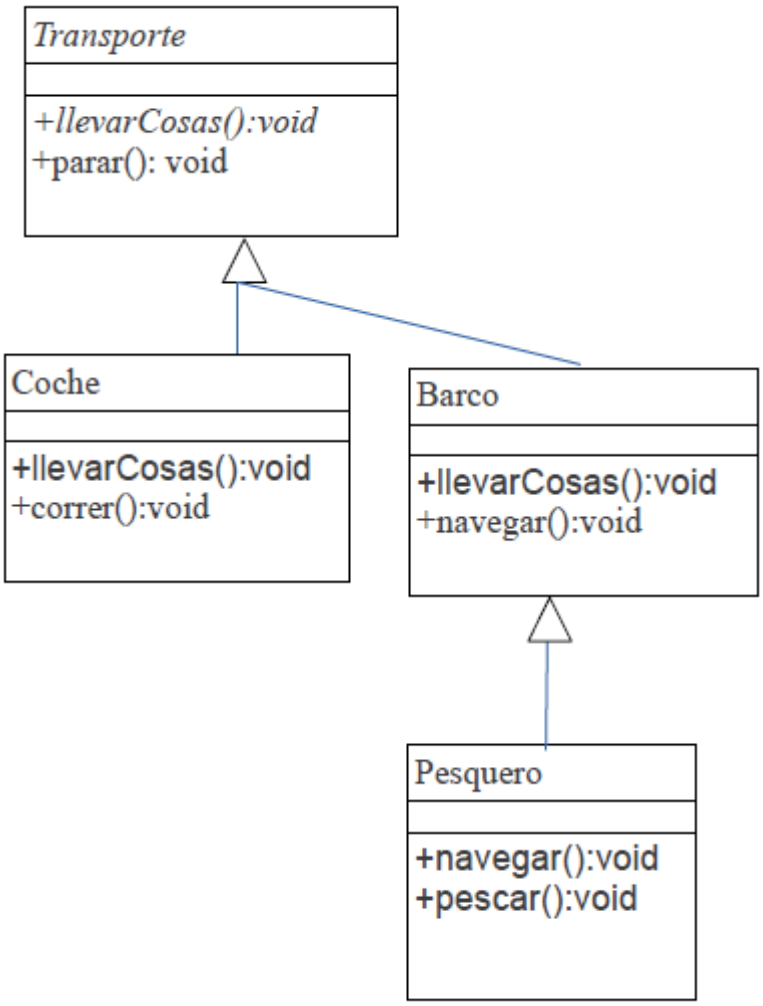
Código	Error y si procede corrección
Transporte x = new Barco(); x.atracar();	
Avion av = new Avion(); av.acelerar();	
Vehiculo v = new Vehiculo(); v.desplazarse();	
Vehiculo v2 = new Vehiculo(); v2.acelerar();	
Transporte t = new Avion(): Barco b= new Barco(); t = b;	
Avion a = new Avion(); String est = a.estado;	
Vehiculo v3 = new Barco(); ((Barco) v3).atracar();	

List<Transporte> listaTransportes = new ArrayList();
listaTransportes.add(new Barco());
listaTransportes.add(new Avion());
listaTransportes.add(new Barco());
for(Transporte tr: listaTransportes){
tr.acelerar();
tr.atracar();}	

List<Transporte> otraLista = new ArrayList();
otraLista.add(new Barco());
otraLista.add(new Avion());
otraLista.add(new Barco());
for(Transporte tr: otraLista){
((Barco) tr).acelerar();
((Barco) tr).atracar();}	

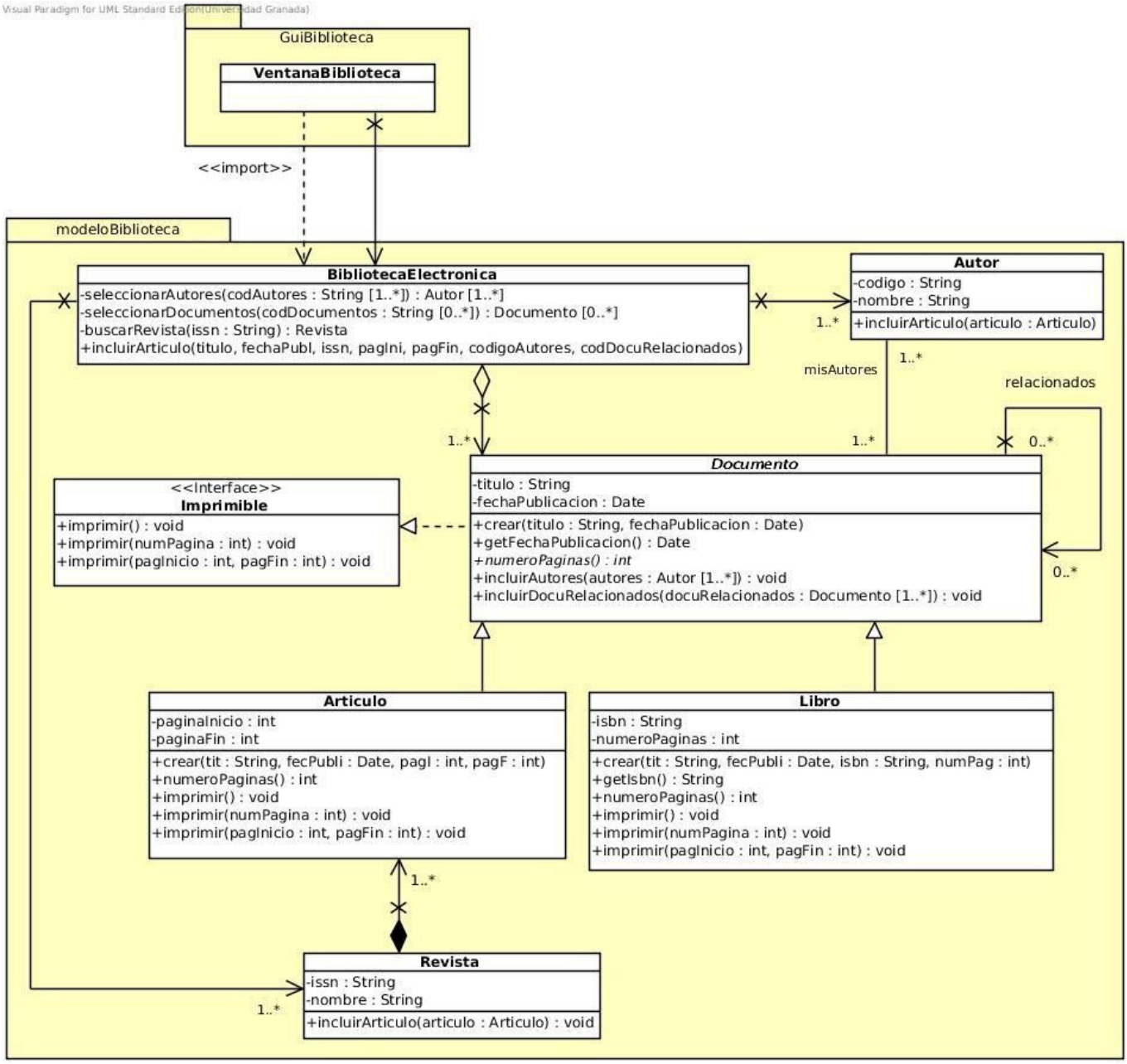
20. A partir del siguiente diagrama de clases y del código proporcionado, detecta errores de compilación y de ejecución. Corrígelos para que funcione correctamente.

```
class PruebaLigadura {
  public static void main ( String args[]){

        Coche c1 = new Coche();
        c1. llevarCosas();
        c1.correr();

        Barco b = new Barco();
        b.llevarCosas();
        b.navegar();
        b.correr();
        b=c1;

        Pesquero p = new Pesquero();
        p.navegar();
        p.pescar();
        p.llevarCosas();
        p=b;
        b=p;
        b.navegar();
        b.pescar();

        ArrayList v= new ArrayList();
        v.add(c1);
        v.add(p);
        (v.get(1)).navegar();
        (v.get(1)).llevarCosas();
 }}
```



Ejercicios de examen

21. A partir de los siguientes diagramas de clases, resuelve las cuestiones que se plantea (siempre que sean de codificación hazlo en Java y en Ruby):



- Define la clase *Documento* (cabecera, atributos y cabeceras de los métodos).
- Define la clase *Articulo* (cabecera, atributos y cabeceras de los métodos).
- Implementa el constructor que se indica en la clase *Artículo*.
- Define la interfaz *Imprimible*.
- Indica los atributos que definen el estado de la clase *BibliotecaElectronica.*
- La clase *Documento* figura en cursiva, lo que indica que es abstracta. Indica los dos motivos por los que lo es.

- En este modelo, ¿puede haber artículos que no estén ligados a una revista?
- Indica sobre las relaciones del diagrama de clases con cuál de los siguientes tipos se corresponden: asociación (AS), agregación (AG), composición (C), realización (R), herencia (H), dependencia (D).
- Escribe el contenido del fichero *VentanaBiblioteca* completo (pero sin añadir nada que no aparezca en el diagrama).
- En la clase *Articulo*, indica cuáles de sus métodos están sobrecargados o redefinidos. Justifica tu respuesta.
- Corrige el código:

```
Imprimible docu = new Documento("Intemperie", fecha); // suponiendo que fecha está inicializada
docu.imprimir();
```

- Corrige el código:

```
Documento docu = new Libro("ISBN10102030");
String codigo = docu.getIsbn();
```

- Rellena la siguiente tabla indicando el tipo estático y dinámico de la variable *docu* en las siguientes líneas de código:

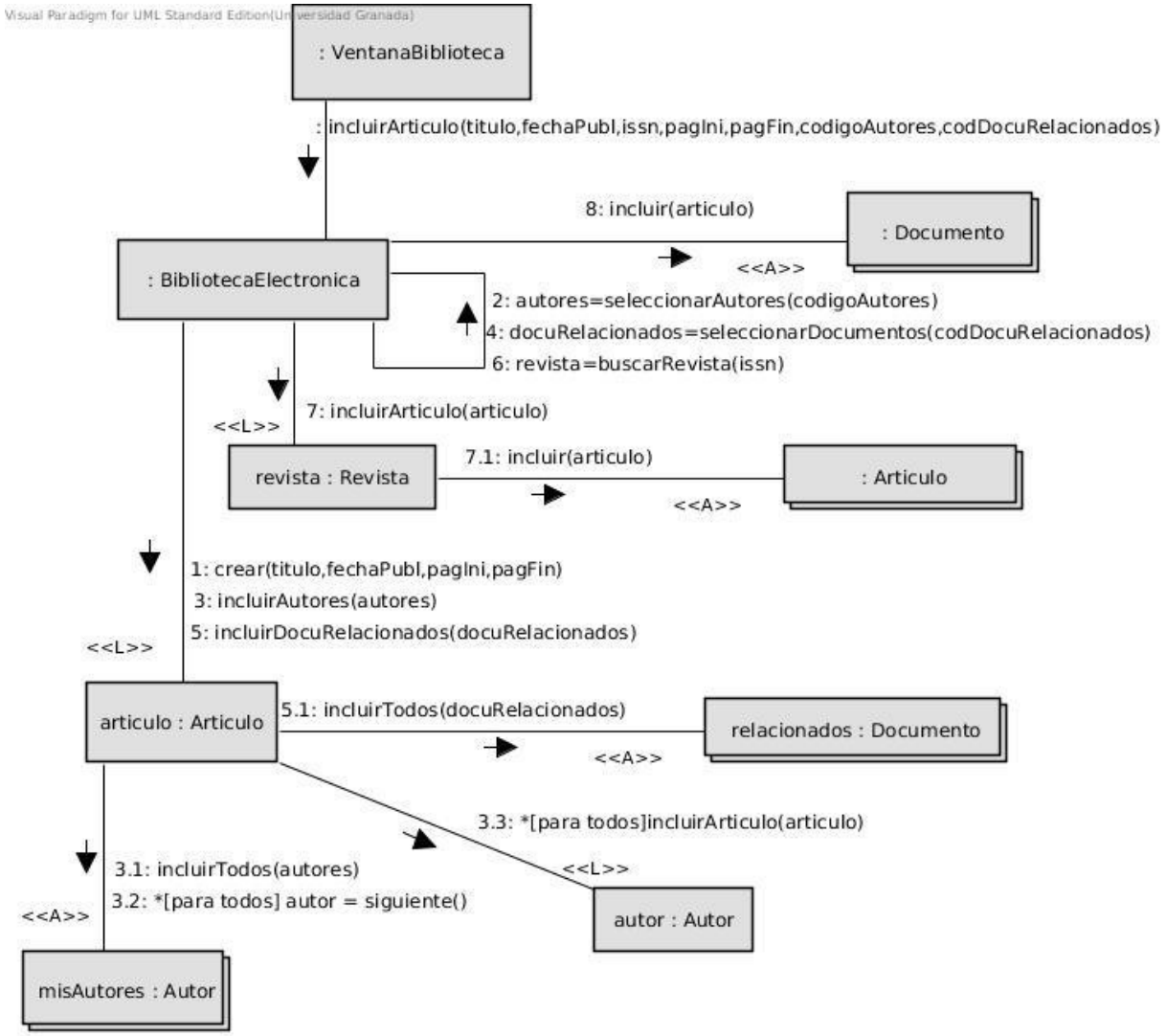
	Tipo estático	**Tipo dinámico**
Documento docu = new Articulo(titulo, fecha, pagIni, pagFin);		
docu.imprimir(pagIni, pagFin);		
docu = new Libro(titulo, fecha, isbn, pags);		
docu.imprimir();		

- Indica si hay errores de compilación o ejecución en el código anterior (suponiendo que las variables *titulo, fecha, pagIni, pagFin, isbn* y *pags* han sido declaradas e inicializadas convenientemente con anterioridad). Justifica tu respuesta.
- Rellena la siguiente tabla marcando con una "X" la situación que corresponda a cada una de las líneas del siguiente bloque de código (suponiendo que las variables *titulo, fecha, pagIni* y *pagFin* han sido declaradas e inicializadas convenientemente con anterioridad).

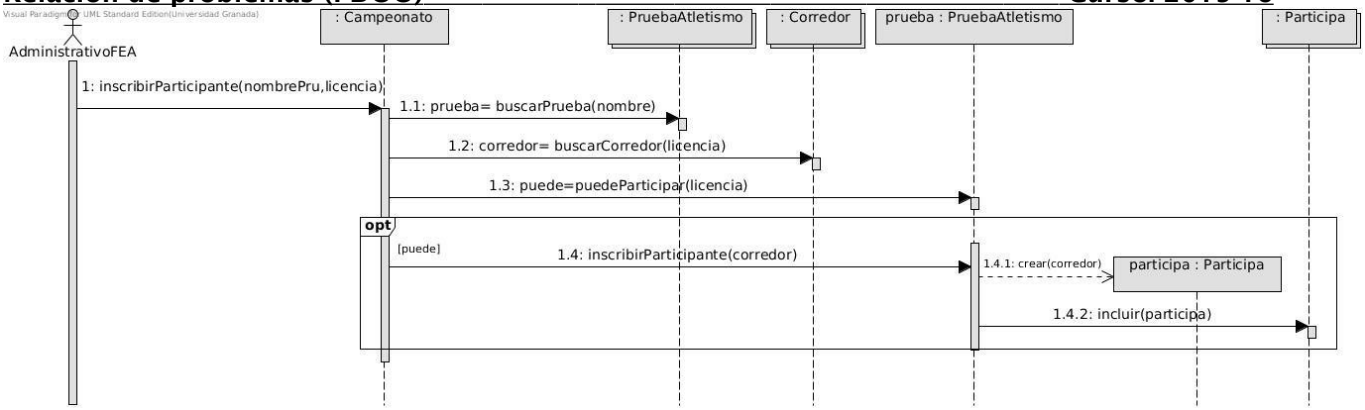
	Ningún error	Sólo error de compilación	Sólo error de ejecución
Imprimible imp = new Articulo(titulo, fecha, pagIni, pagFin) ;			
Documento docu = imp;			
imp.numeroPaginas();			
((Libro) docu).getIsbn();			

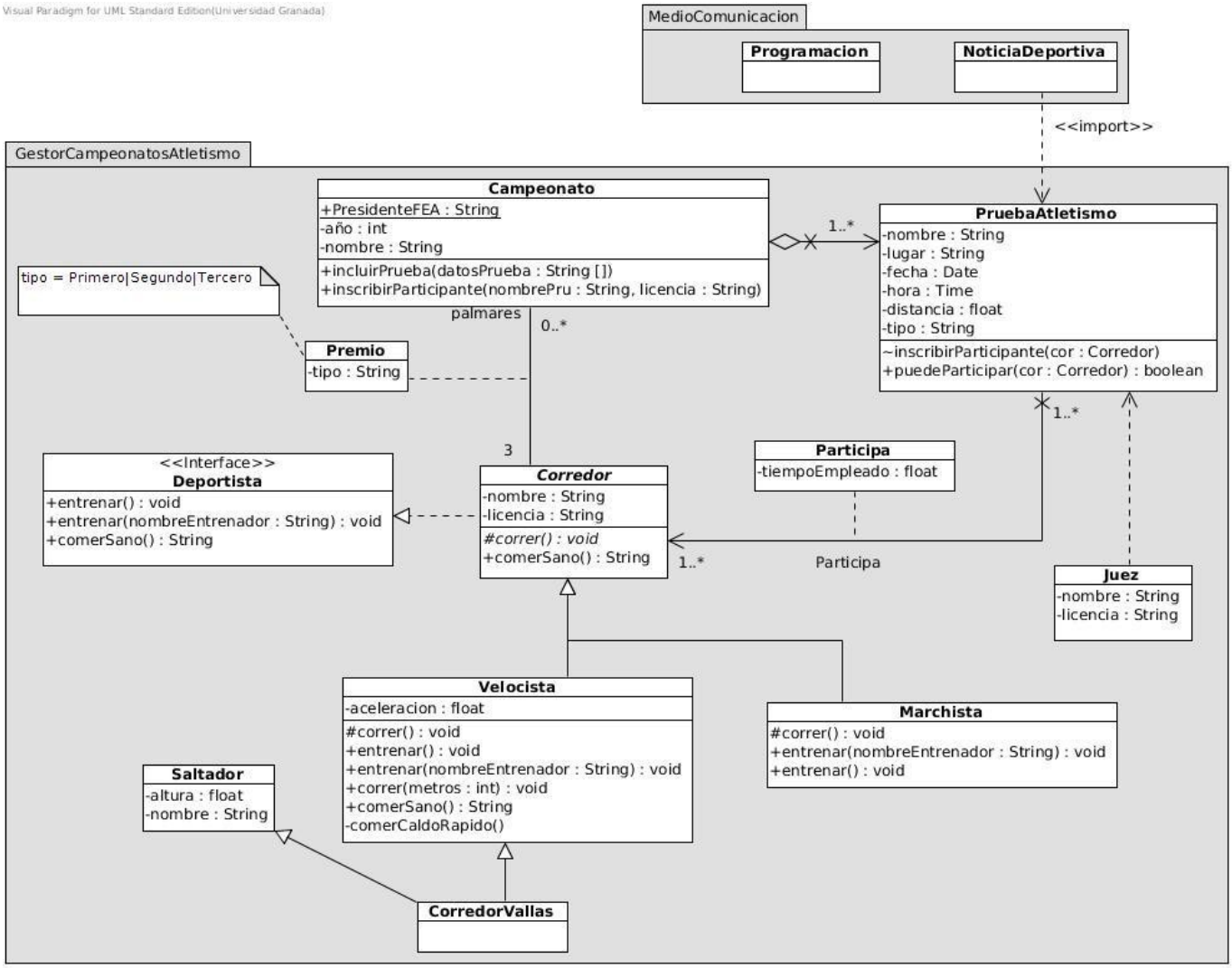
22. Siguiendo el diagrama de comunicación que se presenta a continuación, correspondiente al método *incluirArticulo* de la clase *BibliotecaElectronica*:

- Implementa el diagrama en Java y en Ruby.
- Dibuja el diagrama de secuencia equivalente.



23. Partiendo de los siguientes diagramas de clases y de secuencia:


AdministrativoFEA
: Campeonato
: PruebaAtletismo
: Corredor
prueba : PruebaAtletismo
: Participa
1: inscribirParticipante(nombrePru,licencia)
1.1: prueba= buscarPrueba(nombre)
1.2: corredor= buscarCorredor(licencia)
1.3: puede=puedeParticipar(licencia)
opt
[puede]
1.4: inscribirParticipante(corredor)
1.4.1: crear(corredor)
participa : Participa
1.4.2: incluir(participa)


MedioComunicacion
Programacion
NoticiaDeportiva
<<import>>
GestorCampeonatosAtletismo
Campeonato
+PresidenteFEA : String
-año : int
-nombre : String
+incluirPrueba(datosPrueba : String [])
+inscribirParticipante(nombrePru : String, licencia : String)
1..*
PruebaAtletismo
-nombre : String
-lugar : String
-fecha : Date
-hora : Time
-distancia : float
-tipo : String
~inscribirParticipante(cor : Corredor)
+puedeParticipar(cor : Corredor) : boolean
tipo = Primero|Segundo|Tercero
palmares
0..*
Premio
-tipo : String
3
1..*
<<Interface>>
Deportista
+entrenar() : void
+entrenar(nombreEntrenador : String) : void
+comerSano() : String
Corredor
-nombre : String
-licencia : String
#correr() : void
+comerSano() : String
Participa
-tiempoEmpleado : float
1..*
Participa
Juez
-nombre : String
-licencia : String
Velocista
-aceleracion : float
#correr() : void
+entrenar() : void
+entrenar(nombreEntrenador : String) : void
+correr(metros : int) : void
+comerSano() : String
-comerCaldoRapido()
Marchista
#correr() : void
+entrenar(nombreEntrenador : String) : void
+entrenar() : void
Saltador
-altura : float
-nombre : String
CorredorVallas

- Responde V (Verdarero) o F (Falso) a las siguientes cuestiones:

Desde la clase NoticiaDeportiva se puede acceder a todos los elementos públicos del paquete GestorCampeonatosAtletismo directamente	
Un CorredorVallas es un Velocista y Saltador	
El estado de un objeto de la clase Juez no viene determinado por el estado de un objeto de la clase PruebaAtletismo	
Una PruebaAtlettismo puede existir sin estar asociada a la clase Campeonato	
Un Velocista es un Corredor	
Un Deportista es un Corredor	
La clase Corredor tiene 4 métodos, 3 abstractos y 1 concreto	
La clase Marchista está mal representada en el diagrama, debe ser abstracta	
La clase CorredorVallas presenta un conflicto de nombres	
Todas las instancias de la clase Campeonato tienen una copia de la variable PresidenteFEA	
La condición del fragmento combinado opt está mal, tendría que estar negada	
En el envío de mensaje 1.3 tendría que entrar como parámetro el objeto corredor y no su licencia	
El envío de mensaje 1.4.2 está mal, el objeto receptor no es un multiobjeto	
El objeto participa:Participa se crea si (puede == true)	
El argumento del envío de mensaje 1.1 está mal debe ser nombrePru	
El envío de mensaje 1.4.1 se corresponde con la instaciación de un objeto de la clase Participa	
El objeto receptor del envío de mensaje 1.3 es corredor	
Los envíos de mensaje 1.4.1 y 1.4.2 deberían estar fuera del fragmento combinado opt	

- Usando la siguiente nomenclatura: AS = Asociación, CO = Composición, AG = Agregación, DE = Dependencia, CA = Clase Asociación, RE = Realización, HE = Generalización o Herencia y HM = Herencia múltiple, etiqueta los elementos correspondientes en el propio diagrama de clases.
- Implementa en Java y Ruby los atributos de Campeonato.
- Implementa en Java y en Ruby la clase NoticiaDeportiva.
- Implementa en Java y en Ruby la cabecera de Corredor, Saltador y CorredorVallas.
- Obtén el diagrama de comunicación de la operación inscribirParticipante de la clase PruebaAtletismo, partiendo del diagrama de secuencia proporcionado.
- Implementa en Java y el Ruby el método inscribirParticipante en la clase PruebaAtletismo.
- Implementa en Java el método comerSano() de Velocista teniendo en cuenta que los velocistas tienen que comerCaldoRapido() antes de comerSano() como Corredor.
- Marca de los siguientes cuáles son métodos abstractos en Marchista:

correr()	
comerSano()	
entrenar()	
entrenar(nombreEntrenador:String)	

- Marca qué métodos están redefinidos y/o sobrecargados en la clase Velocista.

	redefinido	sobrecargado
correr()		
comerSano()		
entrenar()		

- Proporciona el tipo estático y dinámico de la variable que se indica en las siguientes sentencias Java.

	Variable	Tipo Estático	Tipo Dinámico
Deportista c= new Velocista();	c		
Corredor d= new Marchista();	d		
c=new Marchista();	c		
d = c;	d		

- En el siguiente código Java:
  - Corrige los posibles errores de compilación.
  - Una vez corregidos los errores de compilación, indica las líneas de código en las que habría error de ejecución.

	Corrección del error en Compilación	Error en ejecución
Corredor c= new Velocista();		
Deportista d= new Marchista();		
d.comerSano();		
Marchista m= d;		
d=c;		
d.correr(150);		
Velocista v = d;		
ArrayList<Corredor> corredores = new ArrayList();		
corredores.add(c);		
corredores.add(m);		
corredores.get(0).correr(10);		
corredores.get(1).correr(10);		
c= new Corredor();		

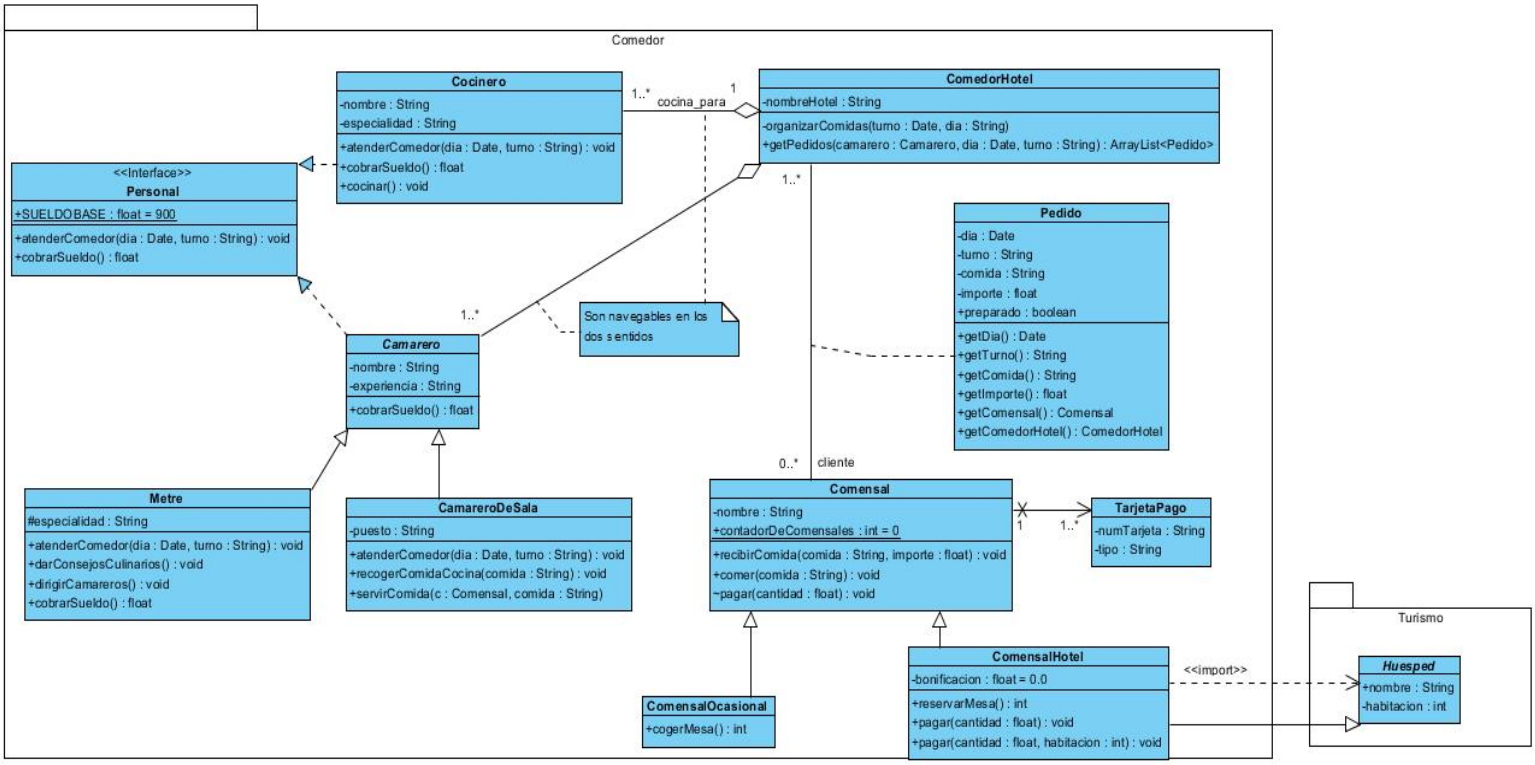
- Completa el código de la siguiente clase parametrizada en Java, la cual representa a un club de

corredores de cualquier tipo: velocistas o marchistas.

```
public class Club<.....> {
    private Map<String, .....> miembros; // La key es el  número de licencia
    private ….  lider;

        // Constructor....
public Club(..... lider) {

        }
// Consultor de un miembro del club a partir del numero de licencia
public ….. getMiembro(String numeroLicencia){

}
// Incluir un nuevo miembro en el club
        public void incluirMiembro(.... miembro){

        }
        // cambiar el líder
public void setLider(..... lider) {

}
// Obtener el lider
public …..  getLider(){

}
}
```

- Escribe un pequeño main en el que se use la clase Club<T> como club de Velocistas
- ¿Cómo lo harías en Ruby?

24. Ejercicio de Examen. Resuelto. Dado el siguiente diagrama de clases:



- Indica en cada casilla si existe o no error de compilación o ejecución, y en caso afirmativo indica cómo solucionarlo. Se asume que las clases tienen constructores que inicializan todos sus atributos.

	Línea de código	Error de compilación	Error de ejecución	Solución
1	`Personal p;`			
2	`Metre me;`			
3	`p = new CamareroDeSala("Braulio","media", "zona3");`			
4	`me= p`	No se puede asignar a un *Metre* un objeto de la clase *Personal* (regla de compatibilidad de tipos)		Un casting arreglaría error de compilación pero se daría luego en ejecución
5	`p.cobrarSueldo();`			
6	`p.recogerComidaCocina("sopa");`	El método *recoger.ComidaCocina* no está en Personal		`(CamareroDeSala)p.recogerComidaCocina`
7	`ArrayList<Camarero> camareros = new ArrayList();`			
8	`Camarero c1, c2, c3;`			
9	`c1 = new Camarero("Juan", "alta");`	No se puede crear una instancia de una clase abstracta		Eliminarlo o meterle dentro un objeto de una clase concreta que herede de

				Camarero
10	`c2 = new Metre("Ana","alta", "pasta");`			
11	`c3 = new CamareroDeSala("Pedro", "media", "zona1");`			
12	`camareros.add(c2);` `camareros.add(c3);`			
	`for (Camarero c: camareros){`			
13	`c.recogerComidaCocina("macarrones");` `}`	Falla porque no todos los objetos *Camarero* tienen ese método	El casting no resuelve	`if (c instanceof CamareroDSala) ((CamareroSala)c).recogerComidaCocina`
14	`((Metre)c3).dirigirCamareros();`		c3 no es un *Metre* y no tiene ese método	
15	`camareros.get(0).dirigirCamareros();`	el receptor es de la clase *Camarero* y no conoce ese método		`((Metre)camareros.at(1)).dirigirCamareros();`

- Escribe en JAVA la cabecera de la declaración y el constructor de las clases *Camarero y Metre.*

```
package Comedor

abstract class Camarero implements Personal{
	private String nombre;
	private String experiencia;
	private ComedorHotel comedor;

	public Camarero (String nom, String exp, ComedorHotel com) {
		this.nombre = nom;
		this.experiencia = exp;
		this.comedor = com;
	}

	@Override
	public float cobrarSueldo() {// IMPLEMENTACIÓN PARA CAMARERO}
}

-----------------------------------------------

package Comedor

class Metre extends Camarero {

    protected String especialidad;

    public Metre (String esp, String nom, String exp, ComedorHotel com) {
		super(nom, exp, com);
		this.especialidad = esp;
    }

    public void atenderComedor(Date dia, String turno) {...}
    public void darConsejosCulinarios() {...}
    public void dirigirCamareros() {...}

    @Override
    public float cobrarSueldo() {// IMPLEMENTACIÓN PARA METRE}
}
```

- Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F). Cada respuesta incorrecta resta una correcta.

La clase *Huesped* importa la clase *ComensalHotel*	F
Puede haber polimorfismo sin ligadura dinámica	F
El atributo *SUELDOBASE* de *Personal* puede ser consultado y modificado por las clases que implementan la interfaz	F
Un objeto de la clase *TarjetaPago* puede consultar *nombre* del objeto *Comensal* con el que está relacionado.	F
Un *Metre* no hereda el atributo *nombre* de *Camarero* porque es privado	F
El método *pagar* está sobrecargado y redefinido en *ComensalHotel*	V
Una clase encapsula atributos y métodos solo si éstos se declaran con visibilidad privada	F