PROGRAMACIÓN Y DISEÑO ORIENTADO A OBJETOS 11/12 / Parcial 12/1/12

Alumno:

Profesor Teoría:

1. (3 puntos) Escribe V o F junto a cada afirmación según consideres que es verdadera o falsa. Cada respuesta correcta suma 0,3 puntos. Cada respuesta errónea resta 0,3 puntos hasta llegar como mínimo a cero puntos.

1.	Considerando el siguiente código, cuando se ejecuta el método modificaAspecto, el emisor del mensaje cambiaColor es un objeto de la clase A.
	`class B{`
	`  private int x,`
	`  public void modificaAspecto(A a){`
	`  a.cambiaColor(“rojo”);`
	`  }`
	`}`
	
	Falso: el emisor es un objeto de la clase B (this). El objeto de la clase A (a) es el receptor del mensaje.	

2.	Para que exista ocultamiento de información es necesario que exista encapsulamiento, pero si existe encapsulamiento no está garantizado que exista ocultamiento de información. Verdadero.	
3.	El uso de tarjetas CRC garantiza que se hace correctamente la asignación de responsabilidades a clases. Falso: el uso de tarjetas CRC ayuda a realizar una primera aproximación al reparto de responsabilidades que no garantiza el que la asignación de responsabilidades no cambie durante el diseño del software.	

4.	El siguiente código Java es correcto:
	
	`Integer unEntero1 = new Integer(3);`
	`Integer unEntero2 = new Integer(6);`
	`Integer sol = unEntero1 + unEntero2;`
	
	Falso: el operador “+” sólo funciona para sumar variables del tipo básico “int”, y no objetos de la clase Integer.	

5.	El siguiente código Smalltalk es correcto:
	
	`|a|`
	`a := Array new:4.`
	`a add:67.`
	
	Falso: el método “add:” no funciona con Arrays, sino con clases correspondientes a colecciones de tamaño dinámico.	

6.	El siguiente código Java es correcto:
	
	`Map m = new Map<Empleado>;`
	`m.containsKey(‘valor’);`	

	Falso: al crear el mapa es necesario indicar dos datos: el tipo de la clave y el tipo de los elementos que va a contener. En este caso sólo se indica la clase de los elementos (Empleado).	
7.	Si la clase Empleado implementa la interfaz Comparable, debe implementar el método compareTo con la cabecera que aparece a continuación. `public int compareTo(Empleado unEmpleado)` Falso: los métodos declarados en una interfaz deben definirse con exactamente la misma cabecera, que en el caso del método compareTo requiere un Object como argumento.	
8.	El código que aparece a continuación crea y lanza una excepción en Java. `throw e;` Falso: el código lanza la excepción, pero no la crea. Para crearla hay que invocar al constructor usando el comando new.	
9.	Si un método *m* en Java invoca a otro que lanza una excepción, ésta siempre debe tratarse en *m* con try{} catch{} Falso: siempre existe la posibilidad de indicar en la cabecera que *m* lanza la excepción en lugar de gestionarla, a no ser que se trate del último método que pudiese gestionarla (p.ej. el método main).	
10.	Estos tres códigos Smalltalk producen el mismo resultado final para la variable “a”:	
	Verdadero: a es igual a 3 en todos los casos	

	`|a|`
	`a:= 1.`
	`2 timesRepeat:[a := a +1]`
		`|a|`
		`a := 1.`
		`#(1 1) do: [:b |a:=a+b]`
		
		`|a|`
		`a:= 1.`
		`unBloque:= [:b :c | a := a+b+c].`
		`unBloque value:1 value:1`

2. (3,5 puntos) Rodea con un círculo todas las respuestas correctas en cada caso.
Cada respuesta completamente correcta suma 0,7 puntos. Las preguntas donde hay rodeada al menos una respuesta incorrecta o faltan respuestas correctas por rodear suman 0 puntos.

1.	Considera el siguiente orden cronológico (de más antiguo a más moderno) :	
	Lenguajes de programación	Métodos de diseño
	1.Simula 2.Smalltalk 3.Java	1.Booch 2.UML 3.OMT
	a) Ambos órdenes cronológicos son correctos. b) El orden cronológico de los lenguajes es correcto, pero no el de los métodos de diseño. c) Ambos órdenes cronológicos son incorrectos. d) El orden cronológico de los métodos es correcto, pero no el de los lenguajes. b) Pues los lenguajes están ordenados, y no así los métodos de diseño ya que UML es el más moderno de los tres.	
2.	Considera el siguiente código Java y Smalltalk .	
	Java	Smalltalk
	`Package empresa;` `class Empleado {`	`//Creación de la clase Empleado` `Object subclass:#Empleado`

<table>
<tr><td></td><td>

```
  protected String nombre;
  private float sueldo;
  // Construtor
  public Empleado(String n,
         float s)
  {
     nombre = n;
     sueldo = s;
  }
}
```

</td><td>

```
instanceVariableNames:'nombre
sueldo'
classVariableNames: ''
poolDictionaries:''
category:'Empresa'
//Como método de clase:constructor
nombre: n sueldo:s
     nombre := n.
     sueldo := s.
```

</td></tr>
<tr><td></td><td colspan="2">a) La clase Empleado tiene el mismo número de variables de instancia, tanto el Java como en Smalltalk.
b) La visibilidad de las variables de instancia en ambos lenguajes es la misma.
c) El constructor de Smalltalk no es correcto.
d) Ambos constructores son métodos de clase.
e) En Java, si se crea una subclase EmpleadoAsalariado de Empleado, dentro de sus métodos no se puede acceder directamente a la variable nombre.

a) y c)
Ya que ambos tienen dos variables de instancia, que tienen visibilidad privada y protegida (protegida indica que nombre será visible en las subclases) en Java y privada en ambas en Smalltalk. El constructor Smalltalk no es correcto ya que ni siquiera llega a crear la instancia.</td></tr>
<tr><td>3.</td><td colspan="2">Considera el siguiente código Java y Smalltalk.</td></tr>
<tr><td></td><td>Java</td><td>Smalltalk</td></tr>
<tr><td></td><td>

```
Empleado e1, e2;
e1 = new Empleado("juan", 1670);
e2 = e1.clone();
```

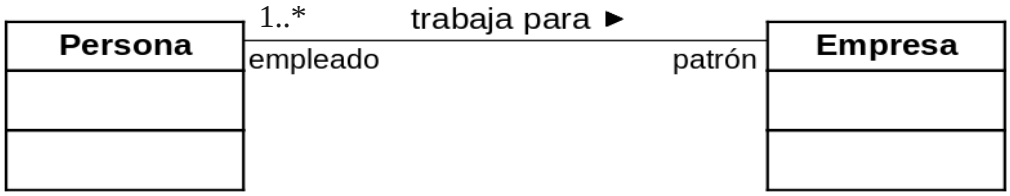
</td><td>

```
|e1 e2|
e1:= Empleado new.
e2 := e1.
```

</td></tr>
<tr><td></td><td colspan="2">a) Ambos códigos son equivalentes.
b) e2 == e1 es cierto sólo en el caso de Smalltalk.
c) e2 == e1 es cierto sólo en el caso de Java.
d) e1.equals(e1) es cierto en ambos casos.
e) En Java se construye un objeto inicializado, pero en Smalltalk no.
f) e1.equals(e2) es cierto sólo en el caso de Java.

b) d) y e) ó b) e) y f)
El método clone crea una copia del objeto, con el mismo estado pero distinta identidad, por eso para java "==" devuelve falso. En el caso de Smalltalk, al hacer e2:=e1, e2 hace referencia al mismo objeto que e1 (por eso son idénticos y tanto comparar identidad como estado devuelven true). Las respuestas correctas son b) d) y e) considerando que la sintaxis no es importante (es decir, e1.equals(e1) se sobreentiende que para Smalltalk se escribiría como e1 = e1, y e1.equals(e2) como e1=e2). Si se considera que la sintaxis era importante para resolver la pregunta, las respuestas correctas son b) e) y f).</td></tr>
<tr><td>4.</td><td colspan="2">Señala los métodos que tienen capacidad para hacer introspección (reflexión por consulta).</td></tr>
<tr><td></td><td colspan="2">a) instanceof
b) class
c) getClass()
d) equals()
e) deepCopy
f) instanceVariableNames:

a) b) y c)
equals compara pero no hace introspección</td></tr>
</table>

5.	En el siguiente diagrama de clases: 
	a) La navegabilidad es en ambos sentidos. b) La navegabilidad es de Persona a Empresa. c) Existe una relación de **dependencia** entre Persona y Empresa. d) Un empleado puede consultar quién es su patrón pero no al revés. e) empleado y patrón son respectivamente los roles de las clases Persona y Empresa en la relación. f) Un objeto Persona se puede relacionar con uno o ningún objeto Empresa. g) Un objeto Empresa se puede relacionar con uno o mucho objetos Persona. h) Hay una relación de **asociación** entre Persona y Empresa. a) e) g) h) Se trata de una asociación y no de una relación de dependencia (pues está marcada con línea continua). La navegabilidad es en ambos sentidos (la flecha de arriba indica el sentido de lectura del nombre de la asociación, si fuese una restricción de navegabilidad iría al final de la línea). Al no indicarse nada, por defecto la multiplicidad en el extremo de Empresa es de 1. “Empleado” y “patrón” son los roles que juegan las clases en la asociación.

3. (3,5 puntos) A partir de los siguientes diagramas de clases y el diagrama de comunicación en el que se representa la operación que indica si hay plazas libres para realizar un viaje de una ciudad origen hasta una ciudad destino, para un día y hora determinada:

a) Codifica en Java el diagrama de clases y el diagrama de comunicación.

b) Codifica en Smalltalk la clase Asiento.

a) Java

```
class Taquilla {

	private ArrayList <Ruta> rutas;

	public boolean obtenerDisponibilidad(String origen, String destino, Fecha diaHora){
		Ruta unaRuta = this.obtenerRuta(origen, destino);
		boolean dis = unaRuta.obtenerDisponibilidad(diaHora);
		return dis;
	}

	private Ruta obtenerRuta(String origen, String destino){}
}

class Ruta{

	private String origen;
	private String destino;
```

```
	private ArrayList <Viaje> viajes;

	public boolean obtenerDisponibilidad(Fecha diaHora){
		Viaje unViaje = this.obtenerViaje(diaHora);
		boolean dis = unViaje.obtenerDisponibilidad();
		return dis;
	}

	private Viaje obtenerViaje(Fecha diaHora){}
}
class Viaje{

	private Fecha diaHora;
	private ArrayList<Asiento> asientos;

	public boolean obtenerDisponibilidad(){
		Iterator <Asiento> iterador = asientos.iterator();
		Asiento unAsiento;
		boolean dis=false;

		while(!dis && iterador.hasNext())
		{
			unAsiento = iterador.next();
			dis = unAsiento.estaLibre();
		}

		return dis;
	}
}
class Asiento {

	private int numero;
	private boolean libre;

	public boolean estaLibre(){}
}
```

b) Smalltalk

```
Object subclass: #Asiento
instanceVariableNames: ' numero libre '
classVariableNames: ' '
poolDictionaries: ' '
category: 'ejercicio3'
```

De manera opcional se pedía también la implementación de algunos métodos:

```
 “Métodos de clase”
nuevoAsiento: unNumero
  ^self new numero: unNumero libre: true

“Métodos de instancia”
libre: valor
  libre:=valor

numero: unNumero
   numero:=unNumero

numero
  ^numero

estalibre
  ^libre
```

Diagrama de clases:

Diagrama de comunicación: