Examen DS

Deseño Software (Universidade da Coruña)

UNIVERSIDADE DA CORUÑA

Diseño Software, Curso 2019-2020

1ª Oportunidad

Apellidos	Nombre	Calificación
MAESTRO		

Instrucciones:

- Antes de comenzar cubrir los datos referentes a los **apellidos y nombre**.
- Solo hay una respuesta válida por pregunta, que debe ponerse en la **tabla de respuestas**, no se tienen en cuenta las anotaciones hechas sobre la propia pregunta.
- Todas las preguntas valen lo mismo y las respuestas incorrectas **restan 1/4** del valor de la pregunta.
- Las respuestas nulas (en blanco, dos contestaciones, respuesta no clara o no legible, etc.) no suman ni restan.
- La duración del examen es de **dos horas y media**.

Respuestas:

1	2	3	4	5	6	7	8	9	10
D	C	D	B	D	D	A	B	D	C

11	12	13	14	15	16	17	18	19	20
D	A	B	B	F	B	B	D	A	A

21	22	23	24	25	26	27	28	29	30
B	C	D	D	E	C	C	C	C	B

UNIVERSIDADE DA CORU

2019-2020

Diseño Software

1ª Or

1. **Este trozo de código intenta formatear "HELLO WORLD" como H!E!L!L!O! !W!O!R!L!D!. ¿Qué está mal en esta implementación?**

```
String result;
StringBuilder sb = new StringBuilder("HELLO WORLD");
for (int i = 0; i <= sb.length(); i++) {
    result = result + sb.charAt(i) + "!";
}
```

A. La variable `result` no ha sido inicializada.

B. Sería más eficiente declarar `result` como un `StringBuilder`.

C. La condición del `for` es errónea.

D. Todas las anteriores.

Solución:

Todas los anteriores. No tiene sentido declarar `result` como un `String` inmutable que es constantemente modificado. Si no se inicializa la variable `result`, obtenemos un error de compilación, ya que no hay nada a lo que concatenar. La condición de permanencia del `for` debería ser estrictamente "`<`", ya que los índices van de `0` a `longitud-1`.

2. **Dados las siguientes ficheros Java:**

```
// Foo.java
package pkgA;

public class Foo {
    int a = 5;
    protected int b = 6;
    public int c = 7;
}
```

```
// Bar.java
package pkgB;
import pkgA.*;

public class Bar {
    public static void main(String[] args) {
        Foo f = new Foo();
        System.out.println(" " + f.a); // Línea A
        System.out.println(" " + f.b); // Línea B
        System.out.println(" " + f.c); // Línea C
    }
}
```

¿Cuál es el resultado de compilar ambos ficheros y ejecutar la clase `Bar`?

A. Imprime por consola: `5 6 7`

B. Error de compilación en las líneas A, B y C.

C. Error de compilación en las líneas A y B.

D. Error de compilación únicamente en la línea A.

Solución:

El atributo `a` tiene visibilidad por defecto (`package`) lo que significa que no es visible en otro paquete. El atributo `b` tiene visibilidad (`protected`) y tampoco es visible en otro paquete. El atributo `c` tiene visibilidad pública (`public`) lo que significa que es visible en otro paquete. Por lo tanto el error de compilación se da solo en `A` y `B`.

3. **¿Tiene sentido definir en Java constructores con visibilidad por defecto (package) ?**

   A. No, en Java los constructores no pueden llevar especificadores de visibilidad.

   B. No, en Java todos los constructores tienen que ser públicos.

   C. Sí, es lo que hacen el patrón *Type Safe Enum* y el patrón *Singleton*.

   D. Sí, si queremos que solo clases del mismo paquete que la clase del constructor puedan crear instancias de la misma.

Solución:
Los métodos constructores en Java sí llevan especificadores de visibilidad y no tienen porqué ser públicos. Los patrones *Type Safe Enum* o *Singleton* lo que hacen es declarar a los constructores privados para solo permitir la creación de instancias desde dentro de la propia clase. Si declaramos el constructor con visibilidad por defecto (package) las clases del mismo paquete que la clase del constructor podrán crear instancias de la misma.

4. **¿Cuál es el resultado de la compilación y ejecución del siguiente código?**

```
class MiClase {
    @Override
    public boolean equals(MiClase obj) {
        return false;
    }
}
```

   A. Un error de compilación, ya que la etiqueta o *tag* `@Override` es una etiqueta del *javadoc* y como tal debe situarse en un comentario *javadoc*.

   B. Un error de compilación al no sobrescribir correctamente el método `equals`.

   C. Un error de ejecución al llamar a `equals` ya que la clase `MiClase` entrará en conflicto al tener dos versiones incompatibles de `equals` (la definida y la heredada de `Object`).

   D. El código compila y se ejecuta correctamente.

Solución:
El código da un error al compilar debido a que la anotación `@Override` indica que estamos haciendo una sobrescritura, cosa que no es cierta porque el método `equals` debería usar como parámetro un objeto de tipo `Object` para así sobrescribir correctamente al método `equals` de la clase `Object`.

5. **La función `allRowSums`, dado un *array* bidimensional, suma el contenido de cada fila de dicho *array* y retorna dicho resultado en un *array* donde cada posición corresponde a una fila. ¿Lo hemos hecho correctamente?**

```
public class MatrixFunctions {
    // Sums the value of each row and returns the results in an array .
    public static int[] allRowSums(int[][] a) {
        int rows = a.length;
        int cols = a[0].length;

        int[] result = new int[rows];

        for (int i = 0; i < rows; i++) {
            result[i] = 0;
```

UNIVE

Diseño Software, Curso 2019-2020

1ª Oportunidad

Calificación

Apellidos

MARCOS ROLDAN

Instrucciones:

- Antes de comenzar cu
- Solo hay una respue tienen en cuenta las
- Todas las preguntas
- Las respuestas mal ni restan.
- La duración del

Respuestas:

1

D. Error de ejecución.

Solución:
El código es correcto e imprime `Subclase - Sub2 - Sub2` ya que la ligadura dinámica usa el tipo actual del objeto creado y no el tipo declarado de la variable a la hora de llamar a los correspondientes métodos

14. **En UML hay una relación de realización entre `A` y `B` cuando...**

A. `A` hereda de la clase `B`.

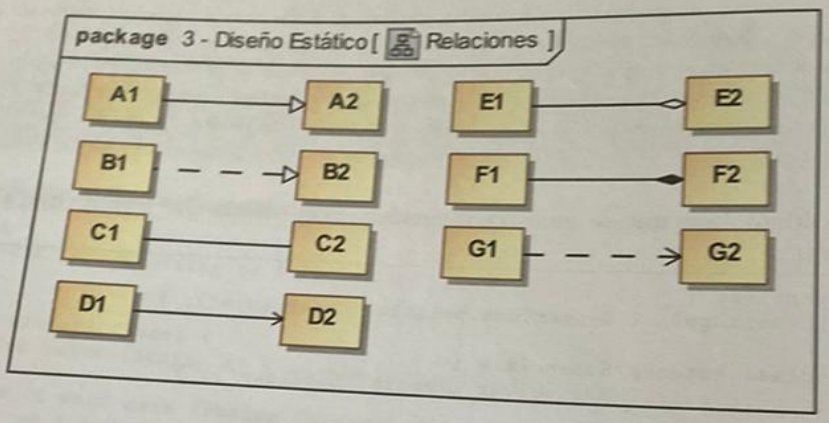
B. `A` **implementa el interfaz** `B`.

C. `A` tiene un atributo de tipo `B` en su estado.

D. `A` tiene un método en el que se le pasa como parámetro un objeto de tipo `B`.

Solución:
La relación de realización consiste en la implementación de interfaces.

15. **Cuál de las siguientes relaciones representa a una Composición UML.**



A. La relación entre A1 y A2. *B.* La relación entre B1 y B2. *C.* La relación entre C1 y C2.
D. La relación entre D1 y D2. *E.* La relación entre E1 y E2. **F. La relación entre F1 y F2.**
G. La relación entre G1 y G2.

Solución:

La composición se representa como una línea continua que comienza con un rombo negro y que va de la clase que representa un *todo* a la clase que representa las *partes*. Las *partes* sólo pueden pertenecer a un *todo* y no pueden tener una existencia propia independiente.

16. **Dado el siguiente código indica cuál es la relación UML existente entre las clases `A` y `B`.**

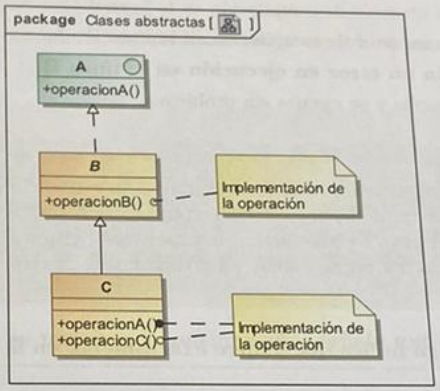
B. Error de compilación en la línea A.
C. Error de compilación en la línea B.
D. Error de compilación en la línea C.

Solución:

El código es correcto porque:

- La línea A es correcta siempre y cuando se incluya el constructor de la línea B.
- La línea B es correcta ya que los enumerados en Java pueden incluir constructores que inicialicen atributos.
- La línea C es correcta porque `sound` tiene visibilidad por defecto (*package*) y por lo tanto es visible desde clases del mismo paquete como `TestEnum`.

8. **Dado el siguiente diagrama de clases, indica cuál de las siguientes sentencias es falsa.**



A. La clase abstracta `B` no puede ser nunca instanciada usando el operador `new`.

B. El diagrama es incorrecto ya que la clase abstracta `B` debe contener al menos un método abstracto.

C. La clase abstracta `B` no está obligada a implementar el método `operacionA()` tal y como se muestra en el diagrama.

D. La clase abstracta `B` puede contener un método NO abstracto como `operacionB()` tal y como se muestra en el diagrama.

Solución:

B es falsa ya que una clase abstracta no tiene por qué tener métodos abstractos.

9. **¿Cuál de las siguientes sentencias acerca de los interfaces `Comparable` y `Comparator` es falsa?**

A. `Comparable<T>` incluye un método `compareTo(T o)` que compara un objeto con otro usando el *orden natural* especificado en el propio objeto `T`.

B. `Comparator<T>` incluye un método `compare(T o1, T o2)` para comparar dos objetos `T`.

C. El método `Collections.sort` tiene dos versiones, una que ordena una lista por su *orden natural* y otra que ordena una lista a partir de un `Comparator` dado.

12. Dado el siguiente código. ¿Cómo deberíamos modificar el parámetro `le` que le pasamos al método `agregarEmpleados` para cumplir con el principio *Get y Put*?

```
class Empleado { /* ... */ }
class Dependiente extends Empleado { /* ... */ }
class Reponedor extends Empleado { /* ... */ }
class Mercado {
    private final List<Empleado> empleados = new ArrayList<>();

    public void agregarEmpleados(List<Empleado> le) {
        empleados.addAll(le);
    }
}
```

A. Añadiendo el comodín `<? extends Empleado>`, al solo leer valores de la colección `le`.

B. Añadiendo el comodín `<? super Empleado>`, ya que usamos los elementos de `le` para escribir en la colección `empleados`.

C. No utilizando comodines, ya que no se pueden usar cuando queremos leer y escribir al mismo tiempo valores de una colección.

D. Utilizando ambos comodines, ya queremos leer y escribir al mismo tiempo valores de una colección.

Solución:

Lo importante aquí es qué hacemos con la colección `le`. En el método `agregarEmpleados` lo único que hacemos es leer de esa colección a través del método `addAll`, que básicamente es un bucle que recorre todos los elementos de la colección. Recorrer los elementos significa leerlos, así que el principio *Get y Put* nos dice que podemos flexibilizar el método poniendo un comodín `extends`. Es cierto que luego se escribe, pero se escribe en una colección distinta (`empleados`) y eso no afecta como se usa `le`.

13. Dado el código Java que se muestra a continuación, señala qué se imprime al ejecutar el método `main`.

```
class Superclase {
    public void go() { System.out.println("Superclase"); }
}
class Subclase extends Superclase {
    public void go() { System.out.println("Subclase"); }
}
class Sub2 extends Subclase {
    public void go() { System.out.println("Sub2"); }
}
class Test {
    public static void main(String[] args)   {
        Superclase superclase = new Subclase();
        Subclase subclase = new Sub2();
        Sub2 sub2 = (Sub2) subclase;

        superclase.go();
        subclase.go();
        sub2.go();
    }
}
```

A. Superclase - Subclase - Sub2

B. Subclase - Sub2 - Sub2

C. Error de compilación.

UNIVERSIDADE DA C

Diseño Software, Curso 2019-2020
1ª Oportunidad

Calificación
2,33

pellidos
NARIÑO BOLAÑO

Instrucciones:
- Antes de comenzar cubrir
- Solo hay una respuesta
tienen en cuenta las ar
- Todas las preguntas
- Las respuestas nul
ni restan.
- La duración d

Respu

```
            for (int j = 0; j < cols; j++) {
                result[i] += a[i][j];
            }
        }
        return result;
    }
}
```

A. *Sí, el código es correcto.*

B. *No, el código falla porque un método estático solo puede acceder a valores estáticos y* `rows`, `cols` *y* `result` *no lo son.*

C. *No, el código falla al inicializar el array* `result` *porque se hace de forma incorrecta.*

D. No, el código falla si la matriz a es irregular (ragged).

Solución:

El código no es correcto. Está asumiendo que todas las filas van a tener el mismo número de columnas que la primera fila, pero esto no es cierto si el *array* es irregular (ragged). Por ejemplo: `int[][] nonsquare = new int[][]{1, 2, 3, 4, 5, 6, 7, 8, 9};` `rows`, `cols` y `result` son variables locales que no pueden declararse estáticas y la inicialización del *array* `result` se hace de forma correcta.

6. **Cuál de las siguientes afirmaciones sobre la sentencia `import` es falsa.**

A. El formato de la sentencia es `import paquete.NombreClase;`

B. Un asterisco significa importar todas las clases del paquete: `import paquete.*;`

C. En un archivo Java las sentencias `import` aparecen inmediatamente después de la instrucción `package` (si existe) y antes de las definiciones de clases.

D. La sentencia `import` es obligatoria siempre que queramos acceder al contenido de una clase que está en otro paquete distinto al actual.

Solución:

La sentencia import no es obligatoria. Si la clase es accesible (está en el `CLASSPATH`) entonces se puede acceder a ella usando el nombre completo de la misma (`paquete.NombreClase`) sin necesidad de hacer un `import`.

7. **Si declaramos las siguientes clases dentro del fichero `Animals.java` ¿Cuál es el resultado de compilar y ejecutar el siguiente código?**

```
package myenums;
public enum Animals {
    DOG("woof"), CAT("meow"), FISH("burble"); // line A
    String sound;
    Animals(String s) { sound = s; } // line B
}

ass TestEnum {
    public static void main(String[] args) {
        System.out.println(Animals.DOG.sound + " " + Animals.FISH.sound); // line C
    }
```

```
woof burble
```

B. Error de compilació
C. Error de compila
D. Error de compi

Solución:
El código es corr
- La líne
- La

A. El diagrama no es correcto porque el mensaje síncrono `calcular` se ha representado con una flecha incorrecta, debería ser una flecha abierta →.

B. El diagrama no es correcto porque el operador del fragmento `opt` es incorrecto. Debería ser el fragmento `alt` que representa alternativas.

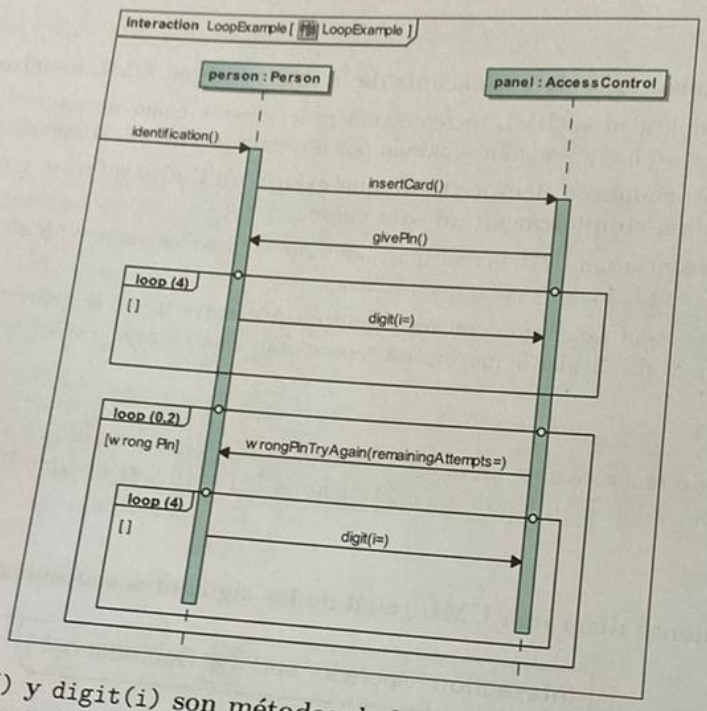
C. El diagrama no es correcto porque falta el mensaje de retorno del mensaje `calcular()`.

D. El diagrama es correcto.

Solución:

El diagrama es correcto. La flecha representa fielmente a un mensaje síncrono. El fragmento `alt` es para varias alternativas, si solo hay un mensaje opcional se usa `opt`. Los tipos de retornos no tienen por qué representarse, sobre todo con mensajes síncronos.

19. **Dado el siguiente diagrama UML ¿cuál de las siguientes sentencias es falsa?**



A. `insertCard()` y `digit(i)` son métodos de la clase `Person` mientras que `givePin()` y `wrongPinTryAgain()` son métodos de la clase `AccessControl`.

B. Los bucles `loop(4)` se corresponden a la siguiente implementación en Java: `for (int i = 0; i<4; i++) { ... }`.

C. El bucle `loop(0, 2) [wrongPin]` se corresponde a la siguiente implementación en Java: `int i = 0; while (wrongPin() && i<2) { ... i++ ... }`.

D. `identification()` es el método que inicia el proceso y es llamado por una clase distinta a `Person` y `AccessControl`, por eso su origen está en el borde del diagrama.

lución:

`ertCard()` y `digit(i)` son métodos de la clase `AccessControl` mientras que `givePin()` y `ngPinTryAgain()` son métodos de la clase `Person`.

24. **Los principios KISS y YAGNI... señala la falsa**

A. Son principios propios de las metodologías ágiles.

B. Intentan que no caigamos en la sobreingeniería.

C. KISS es un acrónimo de "*Keep It Simple, Stupid*".

D. YAGNI es un acrónimo de "*You Are Going Nowhere, Idiot*".

Solución:

YAGNI es un acrónimo de "You Aren't Gonna Need It""

25. **Una agencia de noticias se encarga de reunir noticias de distintas fuentes y luego publicarlas a los medios subscriptores de sus servicios. La agencia avisa, tan pronto como un evento sucede, enviando un titular y un texto asociado a la noticia. Los subscriptores se dan de alta o de baja de nuestro servicio de noticias constantemente (según contraten o dejen de contratar nuestros servicios) ¿Qué patrón es el más adecuado para representar este tipo de comunicación?**

A. Inmutable B. Instancia Única C. Estrategia D. Estado **E. Observador** F. Adaptador G. Fachada H. Composición I. Iterador J. Método factoría K. Constructor L. Método plantilla

Solución:

El patrón más adecuado es el Observador ya que permite definir una dependencia "uno a muchos" entre objetos de tal forma que, cuando el objeto observado cambie de estado (aparece una nueva noticia), todos sus objetos dependientes (los suscriptores) sean notificados y actualizados automáticamente.

26. **Estamos creando una clase que represente un color RGB y queremos hacerla inmutable ¿Hemos implementado bien la inmutabilidad de esta clase?**

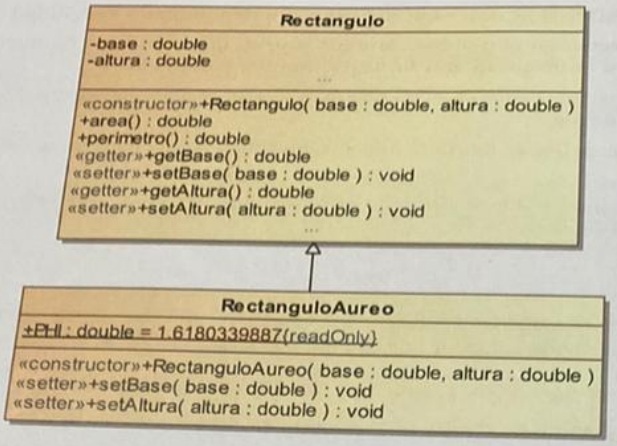
```
public final class RGBColor {
    private final int[] rgb = new int[3]; // 0-Red, 1-Green, 2-Blue

    public RGBColor(int red, int green, int blue) {
        // we assume correct values between 0 and 255.
        rgb[0] = red;
        rgb[1] = green;
        rgb[2] = blue;
    }
    public int[] getRGB() { return rgb; }
    public RGBColor setRed(int red) { return new RGBColor(red,rgb[1],rgb[2]); }
    public RGBColor setGreen(int green){ return new RGBColor(rgb[0],green,rgb[2]);}
    public RGBColor setBlue(int blue) { return new RGBColor(rgb[0],rgb[1],blue); }
}
```

A. Sí, ya que hemos hecho la clase final y sus atributos son privados y finales.

B. No, el constructor es público, por lo que cualquiera puede crear objetos de esta clase.

C. No, el método `getRGB()` devuelve una referencia a un elemento interno que es mutable (el *array*).

D. No, hemos incluido métodos de modificación o *setters* que están prohibidos en una clase inmutable.



A. Principio de Responsabilidad Única.

B. Principio Abierto-Cerrado.

C. Principio de Sustitución de Liskov.

D. Principio de Mínimo Conocimiento.

Solución:

Incumple el **Principio de Sustitución de Liskov - LSP**. La clase `RectanguloAureo` no es un subtipo de `Rectangulo` ya que altera las postcondiciones de los métodos `setBase` y `setAltura` introduciendo efectos laterales indeseados. Sería más conveniente buscar una superclase común a ámbas clases (como puede ser `ParalelogramoRectangulo`) y generalizar ahí los aspectos comunes de ambas clases que pasarán ahora a ser hermanas en el árbol de herencia (sino también podría servir `Poligono`)

23. **Cuando encontramos un código como el siguiente, donde `ObjectB` y `ObjectC` no tienen ninguna relación con `ObjectA`, sabemos que estamos incumpliendo el Principio...**

```
objectA.getObjectB().getObjectC().doSomething();
```

A. Principio de Mínima Sorpresa.

B. Principio de Sustitución de Liskov.

C. Principio de Inversión de la Dependencia.

D. Principio de Mínimo Conocimiento.

Solución:

El principio de mínimo conocimiento conocido también como la "Ley de Deméter" o "no hables con extraños" busca un bajo acoplamiento entre los objetos. Para ello indica que dentro de un método de un determinado objeto sólo se pueden mandar mensajes a: (1) El propio objeto (`this`), (2) A un parámetro del método, (3) A un atributo del propio objeto, (4) A un elemento de una colección que es un atributo del propio objeto, y (5) A un objeto creado dentro del método. Como podemos observar la línea de código está obteniendo distintos objetos y llamando a métodos de los mismos, objetos que son extraños al objeto original y por lo tanto estamos incumpliendo el principio.

síncrono `calcular` se ha representado con una
→.
fragmento `opt` es incorrecto. Debería ser
etorno del mensaje `calcular()`.
o. El fragmento `alt` es
de retornos no tienen

20. **Cuando nos referimos al *mal olor* del software denominado Fragilidad hablamos de...**
    A. **Un sistema en el que los cambios causan que se rompa en lugares que no tienen relación conceptual con la parte que fue cambiada.**
    B. Un sistema en el que es difícil separar sus componentes para que puedan ser reutilizados en otros sistemas.
    C. Un sistema que es difícil de leer y comprender ya que no expresa correctamente sus intenciones.
    D. Un sistema en el cual el diseño contiene infraestructuras complejas que no añaden ningún beneficio directo.

Solución:
La Fragilidad provoca que los cambios en el sistema causan que el mismo se rompa en lugares que no tienen relación conceptual con la parte que fue cambiada.

21. **En el Principio Abierto-Cerrado... (señala la falsa)**
    A. Se fomenta el uso del polimorfismo y la ligadura dinámica, y se desaconseja el uso de condicionales basadas en `instanceof`.
    B. Se fomenta que el código de la superclase contenga referencias a las subclases.
    C. Un módulo abierto está disponible para ser extendido; por ejemplo, para añadir nuevos métodos.
    D. Un módulo cerrado tiene una interfaz estable, de forma que sus antiguas clases cliente sigan pudiendo utilizarlo.

Solución:
El código de la superclase no debería tener referencias a las subclases, ya que si se añadiera una nueva subclase sería necesario modificar dicho código, incumpliendo por tanto en principio al tener que modificar código existente al añadir nuevo código al sistema.

22. **Dado el diagrama UML que vemos a continuación, en el que una clase `RectanguloAureo` hereda de una clase `Rectangulo` y sobrescribe sus métodos `setBase(...)` y `setAltura(...)`, de forma que siempre que se modifique la base se modificará la altura (y viceversa), para mantener la proporción áurea (número *phi* ≈ 1.618) entre dichos valores... ¿Qué principio de diseño estamos incumpliendo?**

UNIVERSIDADE DA

o Software, Curso 2019-2020

1ª Oportunidad

Calificación

2,33

D. El método `compare(T o1, T o2)` devuelve +1, cero o −1 si `o1` es menor, igual o mayor que `o2`, respectivamente.

Solución:

La D es falsa porque, en primer lugar, es exactamente al revés. En segundo lugar, el método `compare(T o1, T o2)` devuelve un valor negativo (no necesariamente -1), cero o un valor positivo (no necesariamente +1) si `o1` es menor, igual o mayor que `o2`, respectivamente.

10. **Dado el código Java que se muestra más abajo, señala cuál es la opción correcta.**

```
class Base {}
class Sub1 extends Base {}
class Sub2 extends Base {}
public class Ejemplo {
  public static void main(String[] args) {
    Base b = new Sub2(); // Linea A
    Sub1 s = (Sub1) b;   // Linea B
  }
}
```

A. El código da un error de compilación en la línea A.

B. El código da un error de compilación en la línea B.

C. El código da un error en ejecución en la línea B.

D. El código compila y se ejecuta sin problemas.

Solución:

El código da un error de ejecución en la línea B porque la variable `b` contiene una instancia de `Sub2`. En la línea B estamos intentando convertir esa instancia a una instancia de `Sub1` (una clase hermana), lo cual es claramente erróneo. El compilador no protestará porque al ser una conversión explícita de tipos confía en el criterio del programador (el código no daría error si `b` almacenara efectivamente una instancia de `Sub1`).

11. **Dado el siguiente código indica qué ocurre exactamente en la línea identificada como "A"**

```
class Clase1 {
    public void metodo(String s) { System.out.println ("Clase 1 " + s);   }
}

class Clase2 extends Clase1 {
    public void metodo(String s) { System.out.println ("Clase 2 " + s);  }

    public static void main (String [] args) {
        Clase1 c1 = new Clase2();
        c1.metodo("Hola");  // LINEA A
    }
}
```

A. Polimorfismo de inclusión

B. Sobrescritura

C. Sobrecarga

D. Ligadura dinámica

Solución:

igadura dinámica ya que el compilador debe decidir en tiempo de ejecución qué método ejecutar, el definido en la `Clase1` o el definido en la `Clase2`, en base al objeto dinámico que alberga la erencia `c1`

12. **Dado el siguiente código. ¿Cómo deberíamos al método `agregarEmpleados` para cumplir con**

```
class Empleado { /* ... */ }
class Dependiente extends Empleado { /*
class Reponedor extends Empleado { /*
class Mercado {
    private final List<Empleado> e
    public void agregarEmpleado
        empleados.addAll(le);
    }
}
```

A. **Añadiendo el c**
le.

B. Añadiendo e
en la colec

C. No utili
tiem

D. U

So

```
class B extends RuntimeException {
    public B (String message)  { System.out.println(message); }
}
class A {
    public static void main(String [] args) { throw new B("Exception B"); }
}
```

A. No existe ninguna relación.

B. Dependencia.

C. Asociación.

D. Composición.

Solución:

A crea una instancia de B pero no la usa como atributo para definir su estado, por lo tanto es sólo una dependencia

17. **¿Cuál de las siguientes sentencias acerca de la multiplicidad UML es falsa?**

A. La multiplicidad en UML se representa generalmente como un intervalo entre un límite inferior y un límite superior separado por dos puntos: `limInf..limSup`.

B. Obligatoriamente siempre tiene que existir un límite inferior y otro superior, no se pueden simplificar en un solo valor.

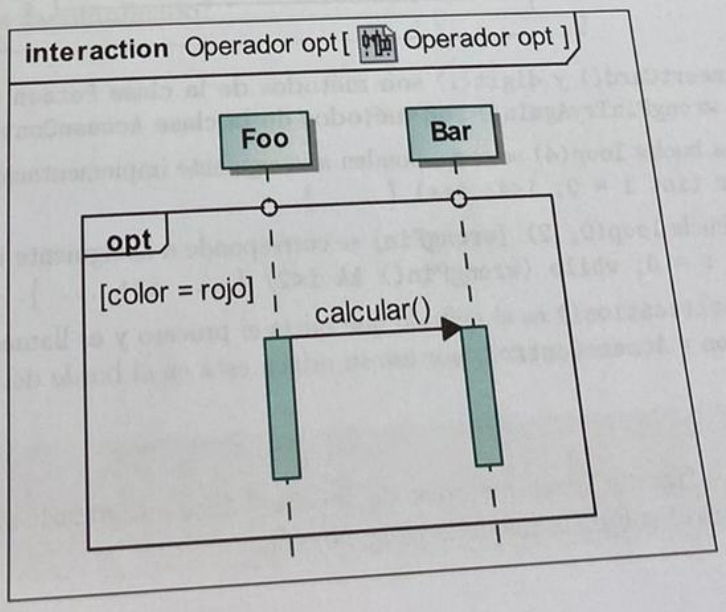
C. En una asociación UML la multiplicidad sigue el estilo *look-across*, donde las multiplicidades se sitúan al lado de la clase que califican.

D. En Java, una asociación con una multiplicidad entre 0 y 1 se representa con un atributo simple. Multiplicidades mayores se representan como arrays o colecciones de objetos.

Solución:

No es obligatorio tener siempre un límite inferior y otro superior. Algunos casos pueden ser simplificados en un solo valor. P.ej. `[5..5]` es equivalente a `[5]` y `[0..*]` es equivalente a `[*]`.

18. **Dado el siguiente diagrama UML ¿cuál de las siguientes sentencias es cierta?**



Solución:

No, no hemos creado bien la inmutabilidad. Los *arrays* son mutables por su propia naturaleza. Si en un método como `getRGB()` devolvemos una referencia a un elemento interno mutable (el array), cualquiera puede cambiar el estado de nuestra clase desde fuera de la misma. La inmutabilidad no tiene nada que ver con impedir crear instancias sino con impedir que las instancias cambien su estado una vez creadas. Por lo que es normal que tengan constructores públicos. Aunque hemos incluido métodos *setters*, estos no afecta a la inmutabilidad ya que se encargan de crear nuevas instancias, no modificar el estado de la actual.

27. **Cuál de las siguientes sentencias acerca del patrón Estado es falsa...**

A. El patrón Estado tiene la misma estructura de clases que el patrón Estrategia, aunque se consideran patrones distintos ya que resuelven problemas diferentes.

B. La clase que juega el rol de "Contexto" delega en las clases que representan a los estados la ejecución de las operaciones cuya ejecución es dependiente del estado.

C. Las clases que representan a los estados es obligatorio que se implementen siempre como *singletons* para así poder compartir estados entre distintos Contextos.

D. Es una solución más compleja y menos compacta (aunque más extensible) que simplemente incluir sentencias condicionales en las operaciones cuya ejecución es dependiente del estado.

Solución:

Los estados no tienen por qué ser *singletons*, sólo tiene sentido cuando se intenta evitar el tener que crear y destruir muchos objetos, o cuando estos se quieren compartir, algo que no es posible si los estados almacenan información en variables de instancia.

28. **En una colección de objetos ¿Qué es un iterador *fail-fast*?**

A. Un iterador de prueba que siempre lanza una excepción al intentar usarlo para recorrer la colección.

B. Un iterador que lanza una excepción si se intenta leer la colección con más de un iterador de forma simultánea.

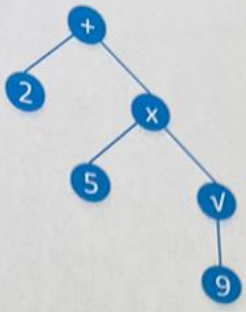
C. Un iterador que lanza una excepción si en medio de la iteración se modifica el contenido de la colección por otra clase.

D. Un iterador que crea una copia de la colección que está recorriendo para evitar lanzar excepciones si la colección es modificada en medio de la iteración.

Solución:

Los iteradores *fail-fast* son aquellos que lanzan una excepción si la colección que están recorriendo es modificada, incluso si esa modificación no afectara a la iteración en marcha. Solo están permitidas las modificaciones hechas por el propio iterador. Un iterador que crea una copia de la colección que está recorriendo para evitar lanzar excepciones si la colección es modificada en medio de la iteración es un iterador *fail-safe*, que es lo contrario que *fail-fast*.

29. **Las expresiones aritméticas son expresiones matemáticas que suelen representarse a través de grafos jerárquicos como el que aparece en la siguiente figura y que corresponde a la expresión aritmética:** $2 + (5 \times \sqrt{9})$



Como vemos las constantes ocupan las hojas del grafo mientras que los operadores (tanto unarios como binarios) ocupan los nodos del mismo. Sobre esta expresión puede aplicarse la operación `evaluar()` que consiste en calcular el valor representado por dicha expresión. Evaluar una hoja es devolver el valor de la misma mientras que evaluar un nodo es realizar la operación de dicho nodo sobre el resultado de evaluar sus hijos. En nuestro caso sería: $2 + (5 \times \sqrt{9}) = 2 + (5 \times 3) = 2 + 15 = 17$

¿Qué patrón es el más adecuado para representar expresiones aritméticas de manera que sea fácil añadir nuevos operadores y nuevas operaciones a realizar sobre dicha expresión?

A. Estrategia

B. Estado

C. Composición

D. Método Plantilla

Solución:

El grafo que representa a la expresión aritmética es un árbol. El patrón más adecuado para representar estructuras de árbol y aplicar operaciones que tienen un comportamiento distinto en las hojas que en los nodos es el patrón Composición

30. **¿Cuál de estos patrones de diseño usa elementos estáticos?**

A. Inmutable. El método `getInmutable()` de la clase mutable es `static`.

B. Instancia Única. El método `getInstancia()` es `static`.

C. Composición. El método recursivo `operacion()` es `static`.

D. Método Plantilla. El `metodoPlantilla()` es `static`.

Solución:

Instancia Única (Singleton). El atributo `instanciaUnica` y su *getter* son estáticos precisamente porque solo hay una instancia que nunca es instanciada desde fuera. En todos los demás patrones, los métodos mencionados son no estáticos. El `getInmutable` copia los valores de una instancia mutable específica. El patrón composición está pensado precisamente para trabajar con colecciones de múltiples objetos, siendo cada uno de ellos una instancia con valores distintos, que son leídos por la operación. El método plantilla es final, no estático, y la clase plantilla suele ser abstracta y existe solo para ser extendida por subclases concretas.