



Diseño Software, Curso 2020-2021

1ª Oportunidad

Apellidos	Nombre	Firma

Instrucciones:

- Antes de comenzar cubrid los datos referentes a los **apellidos y nombre y firmad el examen**.
 - 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**.
-

Respuestas:

1	2	3	4	5	6	7	8	9	10

11	12	13	14	15	16	17	18	19	20

21	22	23	24	25	Calificación

1. La función `isRectangular`, dado un *array* bidimensional, devuelve `true` si éste es rectangular (todas las filas tienen la misma longitud, todas las columnas tienen la misma longitud y la longitud de filas y columnas puede diferir). ¿Lo hemos hecho correctamente?

```
private static boolean isRectangular(char[][] keypad) {  
    for (char[] row : keypad) {  
        if (row.length != keypad[0].length) {  
            return false;  
        }  
    }  
    return true;  
}
```

- A. Sí, el código es correcto.
- B. No, el código da un error de compilación en la línea del `for` porque no está permitido usar bucles *for-each* con *arrays*.
- ☒ C. No, el código es incorrecto porque la condición del `if` es insuficiente, debe comprobar también si `row` es nulo (`row == null`) antes de mirar su tamaño para evitar un `RuntimeException`.
- D. No, el código falla porque estamos comprobando solo la longitud de las filas, habría que añadir otro bucle para comprobar la longitud de las columnas.

2. ¿Cuál es el resultado de compilar y ejecutar el siguiente código?

```
String s = "java string quiz";  
s.toUpperCase();  
System.out.println(s);
```

- ☒ A. Imprime "java string quiz". No se guarda en ningún momento el `s.toUpperCase()`, así que la modificación que se le hizo a `s` desaparece
- B. Imprime "Java String Quiz"
- ☒ C. Imprime "JAVA STRING QUIZ"
- D. Error de compilación en la segunda línea por no recoger o valor de retorno de `s.toUpperCase()`; (en Java es obligatorio recoger el valor de retorno de los métodos).

3. Cuando decimos que, en Java, los métodos `equals` y `hashCode` tienen que funcionar de forma coordinada nos referimos a que...

equals hace comparación de igualdad

- ☒ A. ... en su implementación en `Object`, `equals` hace una comparación de identidad (`==`) mientras que `hashCode` devuelve una representación entera de la dirección de memoria del objeto. Las sobrescrituras de ambos métodos no pueden cambiar nunca este funcionamiento básico.

puede pasar pero no tiene por que

- ☒ B. ... si dos objetos son iguales, de acuerdo con el método `equals`, entonces llamar al método `hashCode` sobre ambos objetos debe producir el mismo resultado.

- ☒ C. ... si dos objetos producen el mismo resultado al llamar a `hashCode` entonces ambos objetos tienen que ser iguales de acuerdo con el método `equals`.

- ☒ D. ... el algoritmo `hashCode` debe usar siempre el algoritmo *hash 31* sobre los atributos definidos en el objeto y que usamos en el `equals`, cualquier otra implementación del `hashCode` no es válida.

Es una optima, pero no es la unica valida

4. ¿Cuál de las siguientes sentencias sobre los especificadores de visibilidad Java es falsa?

- ☒ A. La visibilidad *protected* es más restrictiva que la visibilidad *package*, ya que un atributo *protected* sólo es visible en la propia clase y en sus subclases
- ☒ B. Para establecer un atributo con visibilidad *package* simplemente no hay que poner ningún especificador de visibilidad
- ☒ C. Los especificadores de visibilidad se establecen a nivel de clase, de tal forma que una instancia de una clase podrá ver los atributos privados de otra instancia de la misma clase (siempre que se haga dentro de la propia clase)
- ☒ D. Los métodos constructores con visibilidad *private* evitan la creación de instancias desde fuera de la clase

La *protected* permite visibilidad entre el mismo paquete y para las subclases, nunca podría ser mas restrictiva que solamente el paquete

5. Dado el siguiente código, señala la opción correcta.

```
enum Calificacion {  
    MATRICULA(10), SOBRESALIENTE(9), NOTABLE(7),  
    APROBADO(5), SUSPENSO(0), NO_PRESENTADO(0);  
  
    private int valor;  
  
    public int getValor() { return valor; }  
  
    Calificacion(int valor) { this.valor = valor; }  
}
```

- A. La definición no es correcta porque los literales del enumerado (MATRICULA, SOBRESALIENTE, etc.) no son creados por un operador *new*.
- B. La definición no es correcta porque en un enumerado no se pueden definir constructores.
- C. La definición no es correcta porque en un enumerado se pueden definir constructores pero no otros métodos.

☒ D. La definición es correcta.

6. Dadas las siguientes clases e interfaces:

```
interface Gadget { void doStuff(); }  
  
abstract class Electronic { public abstract void getPower(); }  
  
class Tablet extends Electronic implements Gadget {  
    public void doStuff() { System.out.print("show book"); }  
    public static void main(String[] args) { new Tablet().doStuff(); }  
}
```

¿Cuál es el resultado de compilar y ejecutar el código?

- ☒ A. Las clases compilan correctamente y el *main* de la clase *Tablet* imprime *show book* por consola.
- ☒ B. La clase *Electronic* no compila porque el método *getPower()* se ha declarado sin una implementación en *Electronic*.
- ☒ C. La clase *Tablet* no compila porque no le da una implementación al método *getPower()* definido en *Electronic*.
- ☒ D. La clase *Tablet* no compila porque no puede, al mismo tiempo, extender a una clase e implementar un interfaz.

7. Cuando una subclase crea un método con el mismo nombre que un método de la superclase pero distinto número de parámetros decimos que se produce...

- A. Ligadura dinámica.
- ☒ B. Sobrecarga.
- C. Sobrescritura.
- D. Polimorfismo paramétrico.

8. Dada la siguiente clase Employee:

```
package employees;
public abstract class Employee {
    String name;
    LocalDate hireDate;

    public Employee(String name, LocalDate hireDate) {
        this.name = name;
        this.hireDate = hireDate;
    }
}
```

Y la siguiente subclase SalariedEmployee que reside en el mismo paquete que Employee:

```
package employees;
public class SalariedEmployee extends Employee {
    private int salary;

    public SalariedEmployee(String name, LocalDate hireDate, int salary) {
        this.name = name;
        this.hireDate = hireDate;
        this.salary = salary;
    }
}
```

Qué ocurrirá al compilar ambas clases y ejecutar el siguiente código:

```
public static void main(String[] args) {
    Employee e = new SalariedEmployee("John", LocalDate.now(), 20000);
}
```

- ? ☒ A. Un error de compilación porque el constructor de **SalariedEmployee** no llama correctamente al constructor de **Employee**.
- ☒ B. Un error de compilación porque la clase **SalariedEmployee** no puede acceder a los atributos **name** y **hireDate** de la superclase por cuestiones de visibilidad.
- C. Un error de compilación al definir una variable de tipo **Employee** cuando es una clase abstracta que impide la declaración de variables de dicho tipo.
- ☒ D. El código es correcto y compila y se ejecuta sin problemas.

9. Dadas las siguientes declaraciones de clases:

```
public class Student {  
    public String getFood() { return "Pizza"; }  
    public String getInfo() { return this.getFood(); }  
}  
  
public class GradStudent extends Student {  
    public String getFood() { return "Taco"; }  
}
```

¿Cuál es la salida del siguiente código?

```
Student s1 = new GradStudent();  
s1.getInfo();
```

- A. No compila ya que GradStudent no tiene un método getInfo.
- B. No compila ya que GradStudent no puede ser asignado a una variable Student.
- ☒ C. Taco.
- D. Pizza.

La clase GradStudent ha redefinido el metodo getFood pero sigue compartiendo el metodo getInfo, que llama al nuevo getFood, es decir, devuelve "Taco"

10. Dada la clase Caja que se muestra a continuación, y dada la clase Animal, indica cuál de las siguientes es la forma correcta de crear un objeto de dicha clase Caja que contenga un Animal como elemento (teniendo en cuenta la versión 7 de Java o superior).

```
public class Caja<E> {  
    private E elemento;  
  
    public void setElemento(E elemento) { this.elemento = elemento; }  
    public E getElemento() { return elemento; }  
}
```

- ~~A. Caja<Animal> c = new Caja();~~
- ~~B. Caja c = new Caja<Animal>();~~
- ☒ C. Caja<Animal> c = new Caja<>();
- ~~D. Todas las anteriores son incorrectas~~

Deben tener el simbolo "<>" a los dos lados

11. Dada una clase Stack<E>implements Stackable<E> que define una pila genérica que incluye los siguientes métodos: "void push (E e)" y "E pop()". Y dada la clase StackInteger implements Stackable<Integer> que define una pila de enteros que incluye los métodos: "void push (Integer e)" y "Integer pop()". ¿Cuál sería la forma correcta de declarar el interfaz Stackable?

A.

```
interface Stackable {  
    void push(Object e);  
    Object pop();  
}
```

☒ B.

```
interface Stackable<?> {  
    void push(Object e);  
    Object pop();  
}
```

C.

```
interface Stackable<E> {  
    void push(E e);  
    E pop();  
}
```

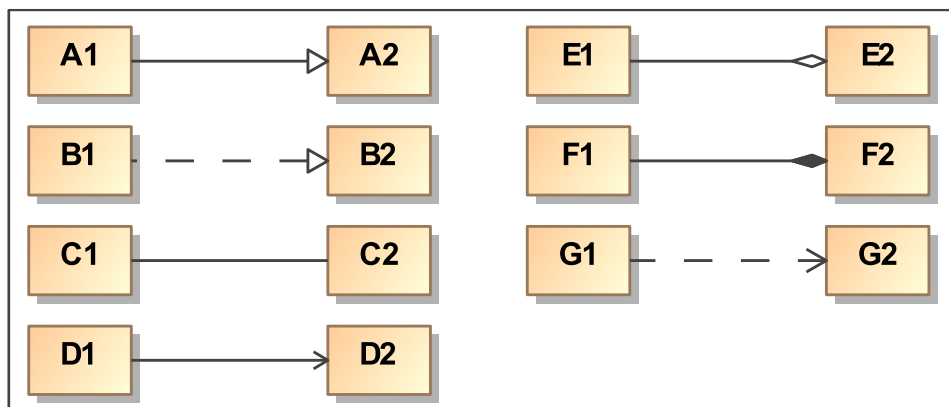
D.

```
interface Stackable<Integer> {  
    void push(Integer e);  
    Integer pop();  
}
```

12. ¿Cuál de las siguientes afirmaciones sobre la relación de generalización en UML es falsa?

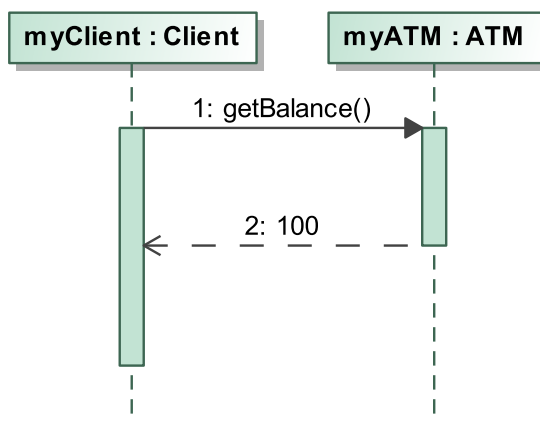
- A. Se corresponde con la palabra clave de Java `extends`.
- B. Se representa como una línea continua con una punta de flecha en forma de triángulo hueco que apunta a la superclase.
- C. Si una clase tiene varias subclases, las puntas de flecha se pueden combinar en una sola.
- ☒ D. Cuando la superclase es abstracta, añadimos el estereotipo «create».

13. ¿Cuál de las siguientes relaciones representa a una Agregación?



- 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

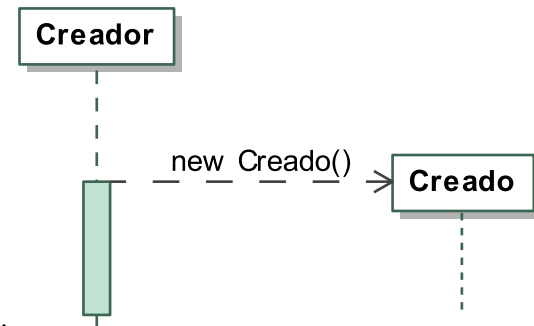
14. Dado el siguiente diagrama de secuencia, indica la sentencia correcta.



- ☒ A. El método `getBalance()` está en la clase ATM. myClient pide `getBalance` a ATM, así que el método debe estar ahí
- ☒ B. El método `getBalance()` está en la clase Client.
- ☒ C. El objeto `myATM` es destruido inmediatamente después de que 100 es devuelto. no muere porque la línea discontinua continua
- D. El objeto `myATM` es instanciado por el objeto `myClient`.

15. Dadas las siguientes afirmaciones sobre diagramas de secuencia, indica cuál es falsa.

- ☒ A. El operador del fragmento para un bucle es opt.
- B. La línea de vida de un objeto es una línea discontinua que indica que el objeto está creado, es accesible y puede recibir mensajes.
- C. Los mensajes de retorno de los mensajes síncronos son opcionales.



D. El mensaje para la creación de un objeto se representa así:

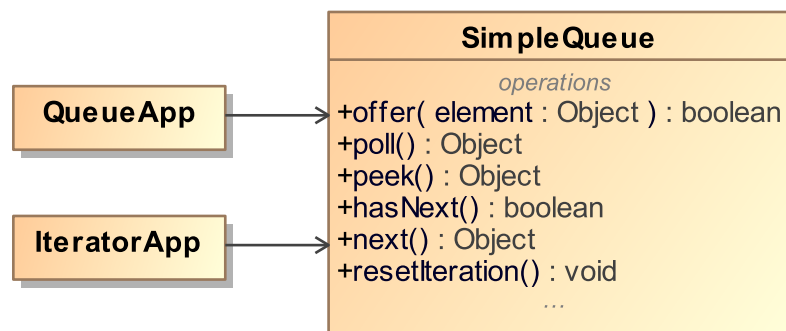
16. El siguiente código, en donde Square, Rectangle y Circle son subclases de Shape, ¿qué principio de diseño SOLID incumple?

```
public void drawShape(Shape s) {
    if (s instanceof Square)
        drawSquare((Square)s);
    else if (s instanceof Rectangle)
        drawRectangle((Rectangle)s);
    else if (s instanceof Circle)
        drawCircle((Circle)s);
}
```

- ☒ A. Principio Abierto-Cerrado.
- B. Principio de Sustitución de Liskov.
- C. Principio de Inversión de la Dependencia.
- D. Principio de Segregación de Interfaces.

Abierto - cerrado dice que se deberían poder añadir mas clases sin necesidad de reescribir el codigo ya existente, si quisieramos añadir triangulo, tendríamos que reescribir la funcion drawShape

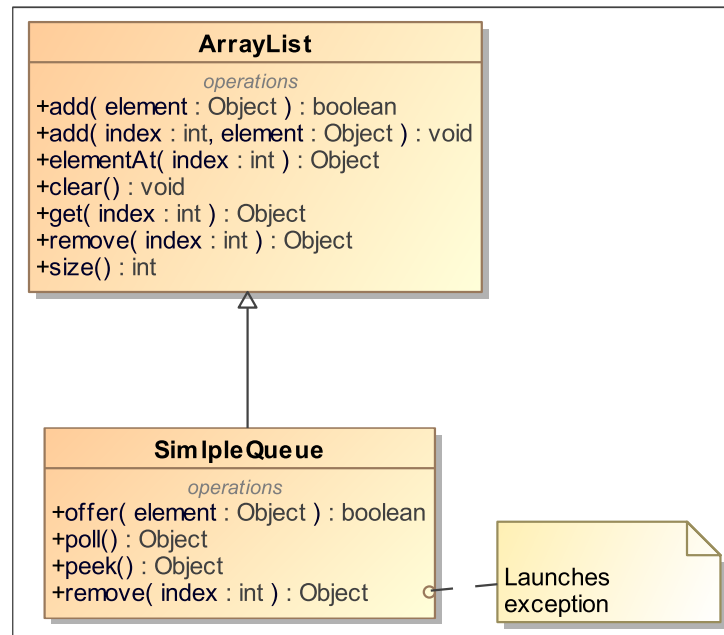
17. El siguiente diagrama UML en el que vemos una clase SimpleQueue que define el comportamiento de una cola, una clase QueueApp que solo está interesada en gestionar la cola (insertar y eliminar valores) y una clase IteratorApp que solo está interesada en recorrer la cola ¿Qué principio de diseño incumple SimpleQueue?



SimpleQueue tiene varias responsabilidades, si cambiasemos metodos de insertar y eliminar valores, no tendria sentido que afectasen a IteratorApp, que solo quiere recorrer la cola

- ☒ A. Principio de Responsabilidad Única.
- B. Principio de Sustitución de Liskov.
- C. Principio de Subcontratación.
- D. Principio de Minimo Conocimiento.

18. En la siguiente figura tenemos las siguientes clases:



- ArrayList. Clase del API de Java que representa una lista con los típicos métodos para gestionarla: add, remove, size, etc.
- SimpleQueue. Se trata de una clase que representa a una cola simple. Hereda de la clase ArrayList para aprovecharse de su código y tiene los siguientes métodos para gestionar la cola y poder iterarla:
 - offer: Inserta un elemento en la cola.
 - poll: Extrae el primer elemento de la cola.
 - peek: Devuelve (sin extraer) el primer elemento de la cola.
 - remove: Sobrescrito para lanzar una excepción para evitar eliminar elementos que estén en la mitad de la cola.

¿Qué principio de diseño SOLID estamos incumpliendo con este diseño?

- A. Principio de Responsabilidad Única.
- B. Principio Abierto-Cerrado.
- ☒ C. Principio de Sustitución de Liskov.
- D. Principio de Segregación de Interfaces.

No siempre podemos pasar una SimpleQueue alla donde nos piden un ArrayList

19. ¿Cuál de las siguientes sentencias acerca del Principio de Mínimo Conocimiento es falsa?

- A. Otro nombre por el que se lo conoce es *Ley de Démetre*.
- B. Una frase que podría resumirlo sería “No hables con extraños... habla solo con tus amigos inmediatos”.
- C. El principio dice que dentro de un método de un determinado objeto solo se pueden mandar mensajes a: (1) El propio objeto (**this**), (2) un parámetro del método, (3) un atributo del propio objeto, (4) un elemento de una colección que es un atributo del propio objeto y (5) un objeto creado dentro del método.
- ☒ D. El principio prohíbe expresamente el encadenamiento de métodos, por lo tanto la siguiente sentencia, obtenida del Patrón Constructor (*Builder*) incumple dicho principio

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240,8).
    calories(100).sodium(35).carbohydrate(27).build();
```

Está bien porque cada metodo solo va hablando con su vecino inmediato, NutritionFacts no llama

20. Dadas las clases Collaborator y Client que vemos a continuación. Qué principio de diseño está incumpliendo Client en el método doSomethingElse()

```
public class Collaborator {  
    public int property1;  
    public int property2;  
    public int property3;  
}
```

```
public class Client {  
    private final Collaborator collaborator;  
    public Client(Collaborator collaborator) {  
        this.collaborator = collaborator;  
    }  
  
    public int doSomethingElse() {  
        return (collaborator.property1 * collaborator.property2)  
            + collaborator.property3;  
    }  
}
```

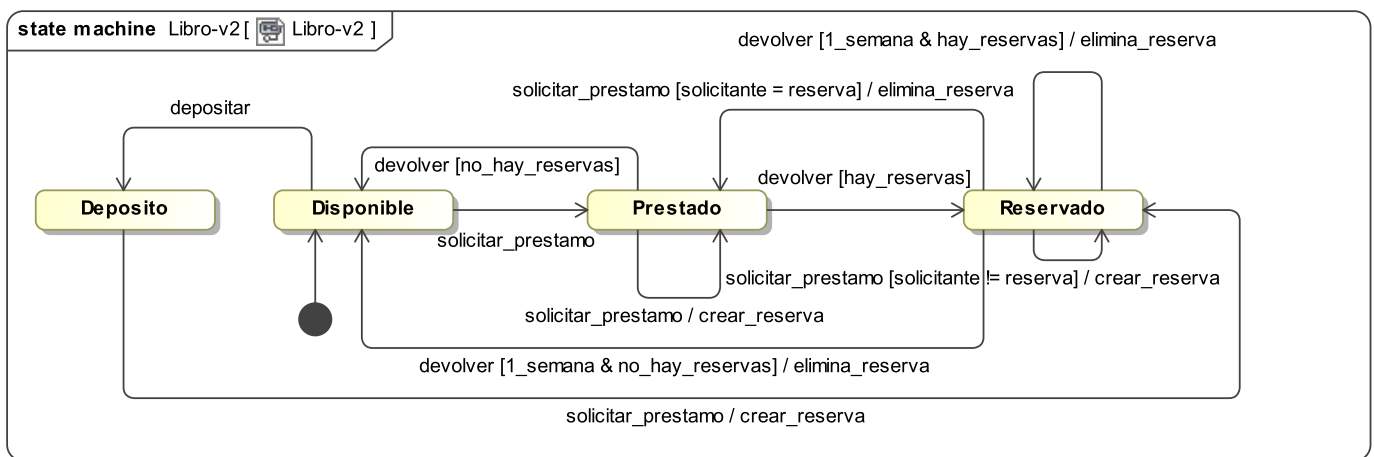
- A. Principio de Sustitución de Liskov
- B. Principio Abierto-Cerrado
- C. Principio de Hollywood
- ☒ D. Principio *Tell, Don't Ask*

21. Dada la siguiente clase Carta que hace uso de las clases Numero y Palo y que hemos definido siguiendo el patrón inmutable ¿lo hemos hecho bien?

```
public enum Numero { AS, DOS, TRES, CUATRO, CINCO, SEIS, SIETE, SOTA, CABALLO, REY }  
public enum Palo { ESPADAS, COPAS, BASTOS, OROS }  
  
public class Carta {  
    private final Numero numero;  
    private final Palo palo;  
  
    public Numero getNumero() { return numero; }  
    public Palo getPalo() { return palo; }  
  
    public Carta(Numero numero, Palo palo) {  
        this.numero = numero;  
        this.palo = palo;  
    }  
}
```

- ☒ A. La definición de inmutabilidad no es correcta porque la clase Carta no es final.
- B. La definición de inmutabilidad no es correcta porque devuelve directamente los objetos internos y privados numero y palo cuando debería devolver una copia.
- C. La definición de inmutabilidad no es correcta porque define un constructor público, cuando debería ser privado.
- D. La clase sigue correctamente las reglas para hacerla inmutable, eliminando los setters y definiendo los atributos como privados y finales.

22. Dado el siguiente Diagrama de Estados, que representa el préstamo de un libro a través de los estados Deposito, Disponible, Prestado y Reservado, y los eventos solicitar_prestamo, devolver y depositar.



A la hora de implementarlo como el Patrón Estado en Java, cuál de las siguientes acciones es incorrecta.

- A. Deberemos crear una clase abstracta Estado con cuatro subclases Deposito, Disponible, Prestado y Reservado.
- B. La clase abstracta Estado incluirá los siguientes tres métodos abstractos: depositar(), devolver() y solicitar_prestamo()
- ☒ C. El método depositar() solo se implementará en la clase Deposito en el resto tendrá una implementación por defecto ya que no tiene sentido su aplicación.
- D. En la clase Libro (el contexto) tendremos que definir tres métodos: solicitar_prestamo, devolver y depositar, que deleguen su funcionamiento en el estado actual del Libro.

23. Un restaurante tiene distintos tipos de menús (desayunos, cenas, etc.). Cada entrada de un menú se compone de un nombre, una descripción y un precio. Imprimir una entrada significa imprimir su información relevante. Imprimir un menú significa imprimir todas sus entradas. También un menú puede tener dentro otros menús. Por ejemplo, el menú de la cena puede tener dentro el menú de entrantes, el menú de platos y el menú de postres. A su vez, el menú de postres se puede dividir en el menú de helados el menú de tartas, etc. Imprimir la carta del restaurante significa imprimir todos sus menús.

¿Cuál es el patrón de diseño más apropiado para representar este problema?

- A. Inmutable B. Instancia Única C. Estrategia D. Estado ☒ E. Composición F. Iterador
G. Observador H. Adaptador I. Fachada J. Método Factoría K. Constructor L. Método Plantilla

24. ¿Cuál de los siguientes patrones busca permitir establecer una comunicación entre dos interfaces incompatibles?

- A. Estrategia.
- ☒ B. Adaptador.
- C. Composición.
- D. Método Factoría.

25. Dada la siguiente clase que representa a un corredor:

```
public class Runner {  
    private final String name;  
    public Runner(String name) { this.name = name; }  
    public String getName() { return name; }  
    public double run() { return Math.random() * 120; }  
}
```

Y la siguiente clase `Competition` que representa a una competición y que tiene un método `compete()` en el que pone a correr a los corredores.


```
public class Competition {  
    private List<Runner> listOfRunners;  
    public Competition(List<Runner> listOfRunners) {  
        this.listOfRunners = listOfRunners;  
    }  
    public void compete() {  
        double t;  
        for (Runner r : listOfRunners)  
            t = r.run();  
    }  
}
```

Existen las siguiente dos clases interesadas en el resultado de la carrera:

- La clase `StatisticsBestTime`. Que almacena el tiempo récord hasta el momento en la modalidad de la carrera y al corredor que ostenta dicho récord.
- La clase `StatisticsWinners`. Que registra la estadística del número de veces que un corredor ha ganado una carrera.

Si queremos modificar `Competition` para comunicar sus resultados a través del patrón Observador. ¿Cual de las siguientes sentencias es falsa?

- A. Un modelo *pull* en el que pasemos simplemente una notificación de que se ha celebrado una nueva carrera (y los observadores tenga que averiguar su resultado consultando a `Competition`) es la estrategia que más facilitaría incluir otros observadores con requisitos diferentes.
- B. Un modelo *push* en el que pasemos únicamente el nombre del ganador de la carrera y su tiempo presentaría la comunicación más eficiente (y suficiente para `StatisticsBestTime` y `StatisticsWinners`), pero dificultaría la inclusión de nuevos observadores con necesidades diferentes.
- C. En el modelo *push*, `Competition` tiene un conocimiento de las necesidades de los observadores, por lo tanto estos no son tan independientes de `Competition` como en el modelo *pull*.

 D. La responsabilidad de `Competition` no acaba con la notificación, tiene que comprobar qué acciones llevan a cabo `StatisticsBestTime` y `StatisticsWinners` para evitar una “cascada de actualizaciones”.