MEDAC
Instituto Oficial de Formación Profesional

Ejercicio Tema 10

Alejandro Cifuentes Sanchez

Álvaro Olmos Sierra

1.- (1p) AWS Lambda, Microsoft Azure y Google Cloud Platform son los servicios más usados del mundo para las arquitecturas Serverless y Arquitectura Event-Drive.

• Investiga estos 3 servicios y describe qué ofrece el servicio gratuito y de pago.

• El problema de estos servicios es que piden demasiada información para usar la versión gratuita (datos personales, dirección e incluso método de pago para evitar bots) ¿Qué otros servicios podríamos usar de código abierto sin dar nuestra información? Busca información de al menos 2 servicios de este tipo.

- **AWS Lambda:**

  El servicio gratuito proporciona 1 millón de solicitudes y 400,000 GB-seconds al mes.

  Y en el servicio de pago proporciona una estructura de precios basada en el uso real.

- **Microsoft Azure (Azure Functions):**

  La versión gratuita da al igual que el anterior 1 millón de solicitudes y 400,000 GB-segundos al mes.

  Y en el de pago se basan en el consumo real y varían según factores como el tiempo de ejecución y la cantidad de recursos utilizados.

- **Google Cloud Platform (Cloud Functions):**

  En el gratuito 2 millones de solicitudes y 400,000 GB-seconds al mes.

  En el de pago se basa en el tiempo de ejecución y memoria utilizada.

- **Alternativas de Código Abierto:**

  Yo he seleccionado estos dos servicio: OpenFaaS (Function as a Service) y Apache OpenWhisk:

3.- (1p) El siguiente código viola el principio de responsabilidad única.

Propón una solución.

Recuerda:Cada clase ha de tener un único propósito,responsabilida y motivopara el cambio.

```
clas  Emplead  {
s    void calcularSalari () {
    // lógica para calcular salari
    o
}   void generarReport () {
    // lógica para generar report
    e
}
}
```

El código proporcionado viola el principio de responsabilidad única, ya que la clase Empleado tiene dos responsabilidades distintas: calcular el salario y generar un informe. Para mejorar esto, podrías dividir estas responsabilidades en dos clases diferentes, cada una con su propósito específico. Aquí hay una posible solución utilizando dos clases: CalculadoraSalario y GeneradorReporte.

```
class CalculadoraSalario {
    double calcularSalario(Empleado empleado) {
        // lógica para calcular salario
    }
}

class GeneradorReporte {
    void generarReporte(Empleado empleado) {
        // lógica para generar reporte
    }
}

class Empleado {
    // Atributos y métodos específicos para la gestión de empleados
}
```

Aquí lo que hemos realizado es que a la clase CalculadoraSalario tiene que solo calcular el salario, mientras que la clase GeneradorReporte se encarga solo de generar informes. La clase Empleado se mantiene con responsabilidades específicas para la gestión de empleados.

4.- (1p) El siguiente código viola el principio de Abierto/Cerrado. Propón una solución.

Recuerda: Una clase ha de estar abierta para añadir más funcionalidades, pero cerrada para ser modificada.

```
clas  Descuent  {
s   doubl  aplicarDescuent (doubl  preci ) {
    é/ lógioa para aplicar descuento direct
    o
}
}
```

Solución

```
1
2   interface Descuento {
3       double aplicarDescuento(double precio);
4   }
5
6   class DescuentoDirecto implements Descuento {
7       @Override
8       double aplicarDescuento(double precio) {
9           // lógica para aplicar descuento directo
10      }
11  }
```

Con esta estructura, puedes agregar nuevos tipos de descuentos implementando la interfaz Descuento sin tener que modificar la clase existente. Esto respeta el principio de Abierto/Cerrado, ya que la clase Descuento está cerrada para modificaciones, pero puedes extender su funcionalidad agregando nuevas clases que implementen la interfaz Descuento.

5.- (1p) El siguiente código viola el principio de sustitución de Liskov. Propón una solución. Recuerda: se debe aplicar correctamente la herencia

```
clas  Ave {
s    void vola () {
    // lógica para vola
    r
}
}   clas  Pinguin  extend  Ave {
    void vola () { s
    // ¡Un pinguino no puede vola
    r!
}
}
```

En este caso, el método volar de la clase Pinguino contradice el comportamiento esperado de la clase base Ave. Para solucionar esto, podríamos redefinir el diseño de las clases para evitar la inconsistencia.

Solucion:

```
class Ave {
    void volar() {
        // lógica común para volar
    }
}

class Pinguino extends Ave {
    @Override
    void volar() {
        throw new UnsupportedOperationException("Los pingüinos no pueden volar.");
    }
}

```

Ya cambiado la clase Pinguino aún hereda de Ave, pero el método volar en Pinguino lanza una excepción para indicar que los pingüinos no pueden volar. Esta solución respeta el principio de sustitución de Liskov, ya que un objeto de la clase derivada (Pinguino) puede ser usado en lugar de un objeto de la clase base (Ave) sin introducir comportamientos inesperados.

6.- (1p) El siguiente código viola el principio de inversión de dependencias. Propón una solución.

Recuerda: las clases deben ser modulares para probarlas con facilidad.

El código proporcionado viola el principio de inversión de dependencias, ya que la clase `Switch` depende directamente de la implementación concreta de `Bombilla`

```
clas  Switc  {
s    Bombill  bombill ;
     Switc (Bombill  bombill ) {
     this.bombill  = bombill ;
          a          a
}    void presionarBoto () {
     if (bombill .estaEncendid ())
     bombill .apaga ();        {
     } else { r
     bombill .encende ();
     a         r
}
}
}    clas  bombill  {
     booleaa estaEncendid () {
     // lógica para verificar si la bombilla está encendid
     a
}    void encende () {
     // lógica para encender la bombill
     a
}    void apaga () {
     // lógica para apagar la bombill
     a
}
}
```

Solucion:

```
interfac  Bombill  {
e   booleaa estaEncendid ();
    void encende ();
    void apaga ();
}       r

// Implementación concreta de la bombill
clas  BombillaImp  implement  Bombill  {
s   privat  boolea sencendid ;a
    e       n       a
     Overrid
@   publi  boolea  estaEncendid () {
    c   retur  enceadid ;
        n      a
}
     Overrid
@   publi  void encende () {
    c   encendidr = true;
        á// Lógica para encender la bombill
        a
}
     Overrid
@   publi  void apaga () {
    c   encendidr = fals ;
        á// Lógica paea apagar la bombill
        a
}

// Clase Switch que ahora depende de la interfaz Bombill
clas  Switc  {
s   Bombill  bombill ;
    a        a
    Switc (Bombill  bombill ) {
    h   thia.bombilla = bombill ;
            a          a
}
    void presionarBoto () {
        if (bombill .estaEncendid ())
            bombill .apaga ();    {
        } else {     r
            bombill .encende ();
            a        r
}
}
```