Ejercicios

1- Refactorizar el siguiente fragmento de código

```
…SomeClass…{
     …
     bool wasInitialized(){
          return…
     }

     someFunction… {
          if ((platform.toUpperCase().indexOf("MAC") > -1)
               && (platform.toUpperCase().indexOf("IE") >
          -1)
               && wasInitialized()
               && resize > 0) {
            someCode();
          }
          otherCode();
     }
}
```

Respuesta:

```
…SomeClass…{
     final String MAC = "MAC";
     final String IE = "IE";
     …
     bool wasInitialized(){
          return…
     }

     bool shouldExecute… {
          return platform.toUpperCase().indexOf(MAC) > -1)
               && (platform.toUpperCase().indexOf(IE) > -1)
               && wasInitialized()
               && resize > 0
     }

     someFunction… {
          if (shouldEecute()) {
            someCode();
          }
          otherCode();
     }
}
```

2- Nombre las ceremonias de Scrum y detalle brevemente cada una.

Spring Planning: Se planifica el trabajo a realizar durante el spring.
Daily Scrum: Es una reunión diaria de unos 15 min para poder discutir qué estuvo haciendo cada uno y que va a hacer ese día.
Sprint Review: Se inspecciona el incremento del producto y se da una retroalimentación.
Spring Retrospective: Se reflexiona sobre el spring con el fin de avanzar como equipo. Qué cosas fueron bien, cuáles mal y cuáles a mejorar.

3- Explique las ventajas del uso del modelo de vistas de 4+1 para una arquitectura de software, haga un ejemplo de una posible vista de despliegue.

Este modelo nos permite analizar el problema desde distintas perspectivas a través de 4 vistas, haciendo que sea más fácil entender la arquitectura del sistema. Estas vistas son:
Vista lógica: Se centra principalmente en los requisitos funcionales, lo que se debe brindar al usuario.
Vista de procesos: Hace énfasis en los requisitos no funcionales, como el rendimiento y la disponibilidad.
Vista de desarrollo: Es la organización de los módulos de software, las distintas partes que pueden ser bibliotecas o subsistemas utilizados durante el desarrollo.
Vista física: similar a la vista de procesos, también se centra en requisitos no funcionales como la disponibilidad, la tolerancia a fallas, el rendimiento y la escalabilidad.

Un ejemplo de la vista de despliegue podría ser un usuario que se conecta desde un navegador a una página web, dicha página está en un servidor web que a su vez está conectado con una base de datos que vive en otro servidor.

4- ¿Qué es refactoring? Como se asegura de lograrlo. De un ejemplo simple con código.

Es una técnica de reestructuración de código que modifica la estructura interna de este sin cambiar su comportamiento.
El refactoring es una forma de reestructurar el código fuente sin afectar a lo externo. Pudiendo lograr un código más legible, simple y eficiente.
Podemos asegurar un refactoring exitoso:
- Teniendo pruebas unitarias que puedan comprobar que una vez hecho el refactoring sigan pasando sin haber alternado el comportamiento (sigue siendo el mismo).
- Realizando pequeños cambios en vez de grandes cambios para asegurarnos de no cometer errores.
- Haciendo revisiones de código ya sea involucrando a otros miembros del equipo para que puedan detectar posibles errores o mejoras.

Código antes del refactoring:

```
def calcular_area(cuadrado):
        return cuadrado[0] * cuadrado[1]
```

Con refactoring:

```
class Cuadrado:
    def __init__ (self, alto, base):
        self.alto = alto
        self.base = base

    def calcular_area(self):
        return self.alto * self.base
```

Con el refactoring se ve una mejora a la hora de la lectura del código y además con la clase Cuadrado se encapsula la idea de Área.
Se mejora la estructura interna de la misma pero sin afectar el comportamiento (nos sigue calculando el área).

5- ¿Qué es un code smell? Nombre dos, y ejemplifíquelos con código

Un code smell es un indicio cuando el código sugiere un problema potencial aunque no sea necesariamente un problema un error técnico o de compilación.

- Long class
- Duplicated code
- Long method
- Long parameter list

Ej long parameter list:

```
public miFuncion(int edad, string nombre, int altura, string genero, string nacionalidad){}
```

[faltan ejemplos]

6- Qué diferencia un token tradicional de un token JWT, comente las ventajas y desventajas.

Tokens tradicionales: Es un identificador, que no tiene un formato legible y carece de significado para los usuarios y aplicaciones. Un ejemplo podrían ser las cookies.
Suelen ser referencias a datos almacenados en el servidor.
Para la validez el servidor consulta en la base de datos o almacén de sesión.
Puede renovarse fácilmente.

Tokens JWT: Es un token auto-contenido, que sigue el formato de **JSON** encriptado o codificado en base64. Tiene 3 partes: **Header, Payload y la firma.**
Contiene los datos dentro del token por lo que el servidor no debe de almacenar nada.
Para la validez el servidor puede verificarlo sin chequear en la base de datos, utilizando la clave secreta.
Es más complicado para renovar.

Ventajas y desventajas:

- El servidor tiene el control total sobre el token tradicional y almacenarlo en el servidor suele ser más seguro (no exponemos la info crítica al cliente).

- Puede ser menos escalable al requerir verificación y almacenamiento constante en el servidor. En un ambiente distribuido sin una base de datos centralizada mantener una sesión o estado compartido puede ser complicado.
- Los tokens JWT tienen escalabilidad porque no necesitamos de la base de datos, sino que el token se valida directamente en el cliente. No tiene estado por lo que el servidor no debe almacenar esa información. Y es autocontenido, es decir, toda la información relevante está contenida dentro de sí mismo.
- Es difícil de revocar, no hay una forma sencilla de invalidar un JWT antes de que expire. Puede ser inseguro si se usa de forma incorrecta ya que el mismo token contiene toda la información.

7- ¿Qué significa autenticar y que autorizar. De 3 ejemplos diferentes de cómo podría autenticarse a un usuario en un sistema.
Autenticar: Es la verificación de la identidad de un usuario
Autorizar: Es qué permisos tiene un usuario

8- ¿Cuáles son los síntomas de un mal diseño? Detalle brevemente cada uno.

- Rigidez: para realizar un cambio debo tocar varias partes del código
- Fragilidad: uno toca una parte del sistema y deja de andar otra parte
- Inmovilidad: incapacidad de reutilizar partes del código en otros sistemas o proyectos debido a una fuerte dependencia con otros componentes o una falta de modularización.

9- ¿Qué es la inversión de dependencia? Explique y dé un ejemplo de código.

Es un principio de programación que se encuentra entre los (SOLID).

1. Los módulos de alto nivel **no deberían depender de módulos de bajo nivel**. Ambos deberían depender de abstracciones.
2. **Las abstracciones no deberían depender de los detalles**. Los detalles deberían depender de las abstracciones.

El objetivo del **Dependency Inversion Principle (DIP)** consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

Ejemplo de código:

En este caso, incumpliendo DIP, UserService dependería de la clase **concreta** userRepository.

```
class ConcreteUserRepository {
    public User findUserBy(String username) {
        // Lógica para buscar un usuario en la base de datos
        return new User(username, "secret"); // Ejemplo simple
    }
}
```

```
class UserService {
    private ConcreteUserRepository userRepository = new ConcreteUserRepository();

    public boolean authenticateUser(String username, String password) {
        User existingUser = userRepository.findUserBy(username);
        if (existingUser != null) {
            // Lógica para autenticar al usuario
            return existingUser.getPassword().equals(password);
        }
        return false;
    }
}
```

Refactorizando para cumplir **DIP** agregaríamos una interfaz userRepository de manera tal que UserService pase a depender de una interfaz en lugar de una clase concreta y usaríamos inyección de dependencias para pasar la implementación correspondiente.

```
interface UserRepository {
    User findUserBy(String username);
}

// Implementación concreta del repositorio (ejemplo de conexión a la base de datos)
class ConcreteUserRepository implements UserRepository {
    @Override
    public User findUserBy(String username) {
        // Lógica para buscar un usuario en la base de datos
        return new User(username, "secret"); // Ejemplo simple
    }
}

// Servicio que ahora depende de la abstracción (UserRepository)
class UserService {
    private UserRepository userRepository;

    // Inyección de dependencias mediante el constructor
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public boolean authenticateUser(String username, String password) {
        User existingUser = userRepository.findUserBy(username);
        if (existingUser != null) {
            // Lógica para autenticar al usuario
            return existingUser.getPassword().equals(password);
        }
        return false;
    }
}
```

10- Una tienda de múltiples sucursales vende ropa, manteniendo estadísticas de venta por región y producto.
La tienda permite cambios de productos, el cambio de un producto se realiza personalmente en la tienda donde se compró originalmente.
Cada producto puede cambiarse una única vez, ya sea por otros productos diferentes cuyo precio total sea de mayor valor, o por el mismo en otro color o talla.
Cualquier compra realizada de más de 10 productos de un mismo tipo, aplica un 10% de descuento.

a) Realice un modelo de dominio

11- ¿Se viola algún/os principio SOLID? En caso afirmativo indicar cuál/es, y cómo lo resolvería.

a)
```java
public class MileageCalculator {
    private List<Car> cars;
    public MileageCalculator(List<Car> cars) {
        this.cars = cars;
    }
    public void CalculateMileage() {
        for (Car car : cars) {
            if (car.name.equals("Audi")) {
                System.out.println("Mileage of the car " + car.name +
" is 10M");
            } else if (car.name.equals("Mercedes")) {
                System.out.println("Mileage of the car" + car.name +
"is 20M");
            }
        }
    }
}
```

Violación del Principio de Responsabilidad Única (SRP):

i) La clase MileageCalculator tiene la responsabilidad de calcular el kilometraje **y** de gestionar las particularidades de los diferentes modelos de coches (Audi, Mercedes). Esto viola el SRP, ya que una clase debería tener una única responsabilidad.

Violación del Principio Abierto/Cerrado (OCP):

ii) La clase MileageCalculator no está abierta para la extensión y cerrada para la modificación. Cada vez que se quiera añadir un nuevo tipo de coche, habría que modificar el método CalculateMileage(), lo que viola el principio OCP. Debería ser posible añadir nuevos tipos de coches sin modificar el código existente.

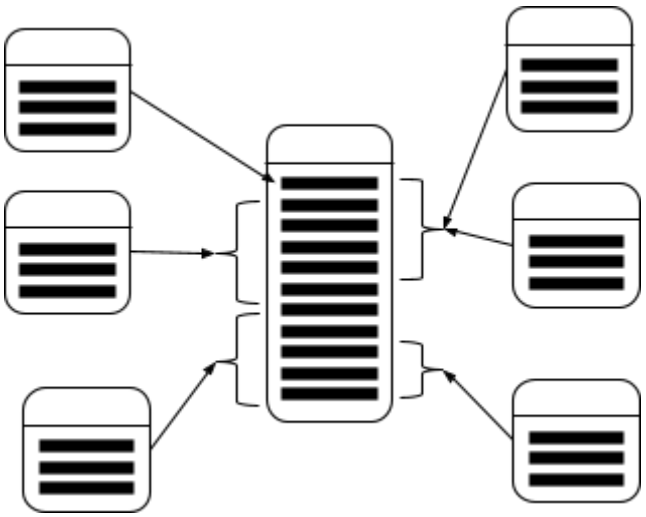
Solución:

1. **Aplicación del SRP**: Cada tipo de coche debería encargarse de su propio cálculo de kilometraje. Esto puede lograrse utilizando una abstracción como una interfaz o clase base.
2. **Aplicación del OCP**: En lugar de usar condicionales if-else, podemos usar **polimorfismo** para que cada coche implemente su propia lógica de kilometraje, de modo que cuando se agregue un nuevo tipo de coche no sea necesario modificar el código existente.

```
// Interface para definir el comportamiento de cálculo de kilometraje
public interface Car {
    String getName();
    void calculateMileage();
}

// Implementación concreta para Audi
public class Audi implements Car {
    @Override
    public String getName() {
        return "Audi";
    }

    @Override
    public void calculateMileage() {
        System.out.println("Mileage of the car Audi is 10M");
    }
}

// Implementación concreta para Mercedes
```

```
public class Mercedes implements Car {
    @Override
    public String getName() {
        return "Mercedes";
    }

    @Override
    public void calculateMileage() {
        System.out.println("Mileage of the car Mercedes is 20M");
    }
}

// Clase que gestiona los coches sin necesidad de conocer las
implementaciones concretas
public class MileageCalculator {
    private List<Car> cars;

    public MileageCalculator(List<Car> cars) {
        this.cars = cars;
    }

    public void calculateMileage() {
        for (Car car : cars) {
            car.calculateMileage();
        }
    }
}
```

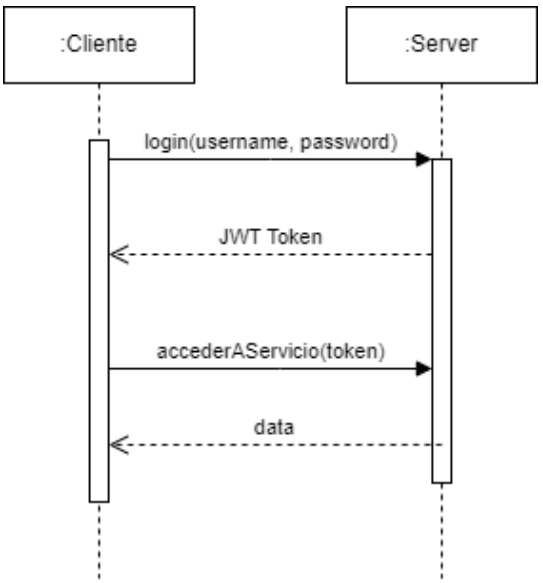
b)



Si alguien puede hacer este :) se lo agradecería je

12- Explique el mecanismo de Autenticación Basic Authentication, y como lo utiliza en una API Rest
Este mecanismo combina el nombre de usuario y su contraseña, y codifica esto en base 64. Al ser fácilmente decodificable, es recomendable que se utilice en conjunto con otro mecanismo de seguridad como HTTPS/SSL.

13- Haga un diagrama de secuencia explicando cómo se realiza un login que retorna un token JWT y como luego se accede a un servicio que requiere autenticación.



14- A qué nos referimos por Calidad Semántica de las historias de usuario.
Se refiere a que estas deben ser conceptualmente acertadas, sin ambigüedades, orientadas al problema y sin conflictos.

15- ¿Qué son los criterios de aceptación? ¿Qué características se desea que tengan? Escriba una historia de usuario y defina los criterios de aceptación para la misma

Los criterios de aceptación son condiciones específicas que deben cumplirse para aceptar trabajo realizado.
Estos deben ser: Claros, Concisos, verificables, independientes, orientados al problema.
Historia de usuario:
Como usuario registrado en una app de películas,
quiero poder seguir a otros usuarios
para recibir actualizaciones sobre sus opiniones y puntuaciones.
Criterios de aceptación:

- **Dado** que el usuario está autenticado, **Cuando** visualiza el perfil de otro usuario **entonces** debe haber un botón visible para seguir o dejar de seguirlo.
- **Dado** que el usuario está en su perfil **cuando** accede a la sección de usuarios seguidos **entonces** debe poder ver una lista con los usuarios que este sigue.
- **Dado** que el usuario sigue a otro usuario **cuando** este segundo califica o comenta sobre una película **entonces** el usuario seguidor recibirá una notificación sobre su actividad.
- **Dado** que un usuario ingresa a su propio perfil **cuando** intenta seguirse a sí mismo **entonces** no debe aparecer la opción de seguir.
- **Dado** que un usuario ya ha sido seguido **cuando** el usuario intenta seguirlo nuevamente **entonces** el botón debe cambiar a “dejar de seguir” y no permitir el seguimiento duplicado.