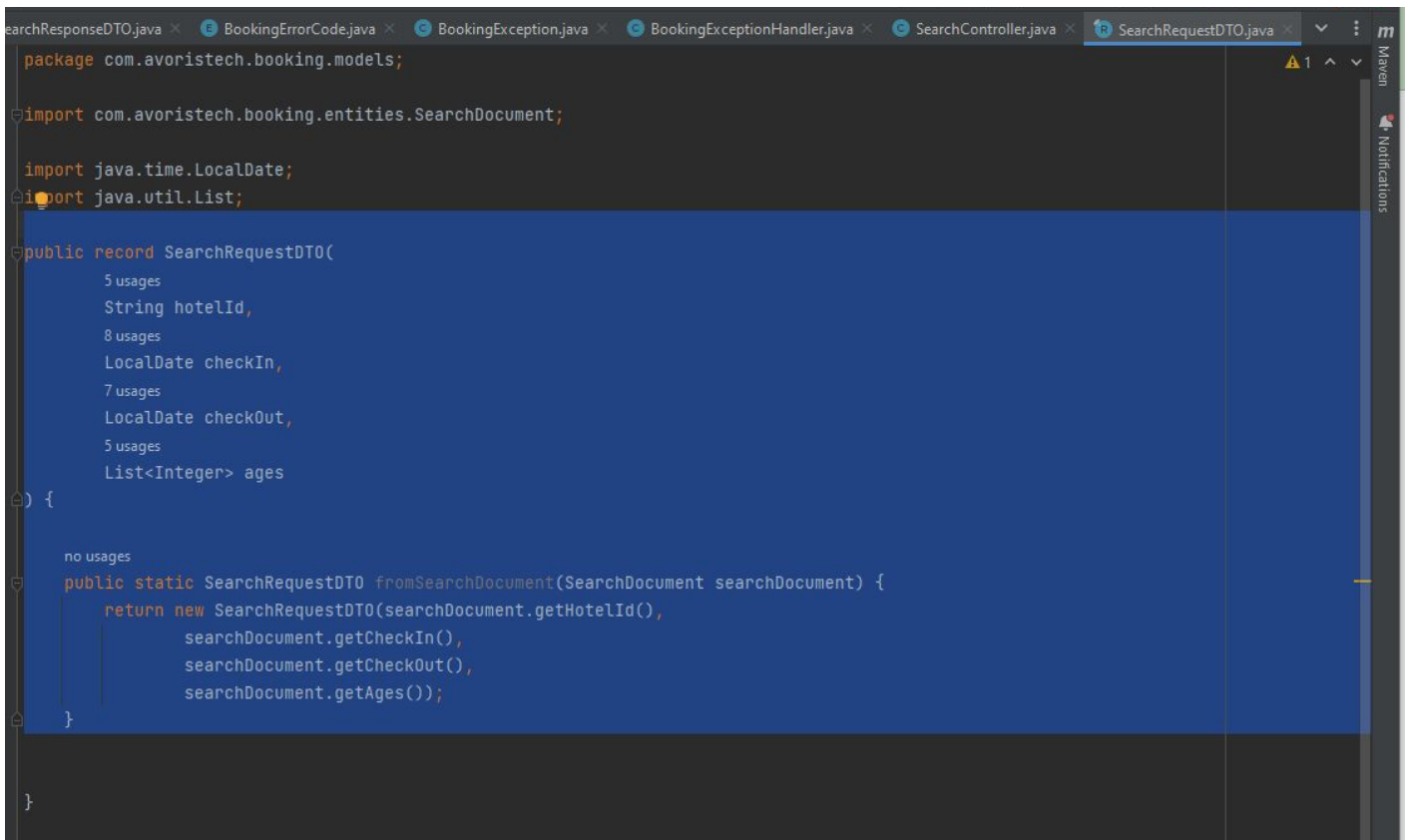


Recomendaciones

- Objeto SearchRequestDTO tiene que ser completamente inmutable.
- Evitar malas prácticas como concatenación de Strings con +
- Corroborar que las peticiones son thread safe .
- Corroborar que las peticiones no permitan la inyección SQL.

Objeto SearchRequestDTO tiene que ser completamente inmutable.



```
package com.avoristech.booking.models;

import com.avoristech.booking.entities.SearchDocument;

import java.time.LocalDate;
import java.util.List;

public record SearchRequestDTO(
    String hotelId,
    LocalDate checkIn,
    LocalDate checkOut,
    List<Integer> ages
) {

    public static SearchRequestDTO fromSearchDocument(SearchDocument searchDocument) {
        return new SearchRequestDTO(searchDocument.getHotelId(),
            searchDocument.getCheckIn(),
            searchDocument.getCheckOut(),
            searchDocument.getAges());
    }
}
```


Evitar malas prácticas como concatenación de Strings con + .

Evitar Concatenación de Strings con '+' :

- Utiliza `StringBuilder` en lugar de concatenación directa para construir cadenas largas. La concatenación de cadenas con '+' en bucles puede ser ineficiente.

Ejemplo:

java

 Copy code

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Texto1");
stringBuilder.append("Texto2");
String resultado = stringBuilder.toString();
```

Corroborar que las peticiones son thread safe .

Corroborar que las Peticiones son Thread-Safe:

- Asegúrate de que tus clases y métodos sean thread-safe si van a ser utilizados en entornos multi-hilo.
- Si tienes variables de instancia mutables, considera el uso de sincronización (`synchronized`) o utiliza estructuras de datos inmutables para evitar problemas de concurrencia.

Aquí hay algunos puntos clave relacionados con la thread-safety en Spring Boot:

Spring Framework y Thread-Safety:

- La mayoría de los componentes clave de Spring Framework, en el que se basa Spring Boot, son thread-safe. Por ejemplo, los contenedores de beans de Spring (ApplicationContext), los beans singleton y la mayoría de los objetos administrados por Spring son thread-safe.

Manejo de Solicitudes HTTP:

- Spring Boot utiliza el modelo de programación basado en servlets para manejar solicitudes HTTP. Los servlets en sí mismos son thread-safe, y Spring Boot gestiona la concurrencia al manejar las solicitudes en diferentes hilos. Cada solicitud HTTP generalmente se maneja en un hilo separado.

Beans y Singleton Scope:

- Los beans con ámbito singleton en Spring son instancias únicas compartidas por todos los hilos. Asegúrate de que los beans que defines como singleton sean realmente inmutables o thread-safe.

Corroborar que las peticiones son thread safe .

1. Evitar el Estado Compartido:

- Evita compartir estado mutable entre múltiples hilos sin la debida sincronización.
Si un componente de Spring Boot mantiene estado mutable, debes garantizar la correcta sincronización o considerar otras estrategias, como el uso de ámbitos de prototipo o el uso de servicios de concurrencia de Spring.

```
java Copy code

@Component
public class MyComponent {
    private List<String> mutableList = new ArrayList<>();

    public void addToMutableList(String value) {
        // Operaciones en mutableList deben ser sincronizadas apropiada
        synchronized (mutableList) {
            mutableList.add(value);
        }
    }
}
```

Corroborar que las peticiones son thread safe .


1. Seguridad y Transacciones:

- Spring Boot proporciona características de seguridad y manejo de transacciones que son thread-safe. Por ejemplo, las anotaciones `@Transactional` se utilizan para gestionar transacciones de manera segura.

```
java Copy code

@Service
public class MyTransactionalService {

    @Transactional
    public void performTransactionalOperation() {
        // Operación transaccional...
    }
}
```

En resumen, Spring Boot proporciona mecanismos para escribir aplicaciones thread-safe, pero siempre es importante entender y gestionar adecuadamente la concurrencia en tu código, especialmente do se trata de objetos con estado mutable compartido entre hilos.

Corroborar que las peticiones no permitan la inyección SQL.

La inyección SQL es un tipo de ataque informático que tiene como objetivo explotar vulnerabilidades en sistemas que utilizan bases de datos relacionales. Este tipo de ataque ocurre cuando un atacante inserta o "inyecta" comandos SQL maliciosos en las consultas que se envían a la base de datos. Estos comandos SQL adicionales pueden alterar el comportamiento previsto de la consulta y, en algunos casos, permitir al atacante acceder, modificar o eliminar datos en la base de datos.

El escenario típico de la inyección SQL se da cuando una aplicación web o un sistema no valida correctamente las entradas del usuario antes de incluirlas en una consulta SQL. Esto puede ocurrir en formularios web, campos de búsqueda o cualquier lugar donde la entrada del usuario se utilice en la construcción de consultas SQL.

Corroborar que las peticiones no permitan la inyección SQL.

```
sql Copy code  
  
SELECT * FROM usuarios WHERE nombre_usuario = 'usuario' AND contraseña
```

Si la aplicación no valida adecuadamente las entradas del usuario y alguien ingresa el siguiente nombre de usuario:

```
sql Copy code  
  
' OR '1'='1'; --
```

La consulta resultante sería:

```
sql Copy code  
  
SELECT * FROM usuarios WHERE nombre_usuario = '' OR '1'='1'; --' AND co
```

Para prevenir la inyección SQL, es esencial utilizar consultas parametrizadas o declaraciones preparadas, que separan los datos de la consulta y evitan que la entrada del usuario se interprete como comandos SQL maliciosos. Además, la validación adecuada de las entradas del usuario es fundamental para mitigar este tipo de ataques.

Recomendaciones

- Hacer énfasis en la inmutabilidad de clases (propiedades “private final” y “objetos mutables con referencia al valor y no al objeto”, no hacer conversión de datos en getters sino en constructor) o usar record.
- Utilizar últimas versiones y features de Spring boot 3.1+, java, junit, etc.
- Corroborar performance tanto en clases como en test.
- Crear la mínima cantidad de objetos en los test y reutilizarlos en cada caso de prueba de caso feliz y excepciones (corroborar varias condiciones a la vez con assertAll).

Inmutabilidad de clases en Java.

1. Propiedades privadas y finales:

Asegúrate de que las propiedades de tu clase sean privadas y finales para hacerlas inmutables. De esta manera, una vez que se establezca el valor en el constructor, no se puede cambiar.

```
java Copy code

public class MiClase {
    private final String nombre;
    private final int edad;

    public MiClase(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }


    // Getters...
}
```

Inmutabilidad de clases en Java.

2. Objetos mutables con referencia al valor:

Si una propiedad es un objeto mutable, asegúrate de pasar una copia del objeto en lugar de una referencia directa. Esto evita que alguien externo modifique el objeto original después de haberse usado en la construcción de la instancia.

java

 Copy code


```
public class MiClase {  
    private final List<String> lista;  
  
    public MiClase(List<String> lista) {  
        this.lista = new ArrayList<>(lista); // Pasar una copia  
    }  
  
    // Getters...  
}
```

Inmutabilidad de clases en Java.

3. No hacer conversiones en getters sino en el constructor:

Evita hacer conversiones o modificaciones en los métodos getters, realiza esas transformaciones en el constructor para garantizar que la instancia siempre tenga un estado coherente.

java

 Copy code


```
public class MiClase {  
    private final LocalDate fechaNacimiento;  
  
    public MiClase(String fechaNacimientoString) {  
        this.fechaNacimiento = LocalDate.parse(fechaNacimientoString);  
    }  
  
    public LocalDate getFechaNacimiento() {  
        return fechaNacimiento;  
    }  
}
```

Inmutabilidad de clases en Java.

4. Usar record (Java 16 y versiones posteriores):

Los records en Java son una excelente manera de crear clases inmutables con sintaxis más concisa. En un record, las propiedades son automáticamente privadas y finales, y se generan automáticamente los métodos `equals()`, `hashCode()`, y `toString()`.

java

 Copy code

```
public record MiClase(String nombre, int edad) {  
    // No se necesitan getters, se generan automáticamente.  
}
```

Corroborar performance tanto en clases como en test.

Medición de Rendimiento en Clases de la Aplicación:

1. Profiler de Java:

Utiliza un profiler de Java para analizar el rendimiento de tu aplicación en tiempo real.

Algunas herramientas populares son VisualVM, YourKit, y Java Mission Control.

2. Herramientas de Monitoreo:

Emplea herramientas de monitoreo como Prometheus o New Relic para obtener métricas en tiempo real sobre el rendimiento de tu aplicación.

3. Logging y AOP:

Agrega logging a tus clases clave para medir el tiempo de ejecución de diferentes métodos. También, puedes utilizar Aspect-Oriented Programming (AOP) para encapsular la lógica de medición de rendimiento.

Corroborar performance tanto en clases como en test.

Pruebas de Rendimiento (Performance Testing):

1. JUnit 5 y Anotaciones Específicas:

Utiliza las capacidades de JUnit 5 para pruebas de rendimiento. Puedes emplear la anotación `@RepeatedTest` para ejecutar un método de prueba un número específico de veces, y `@Timeout` para establecer un límite de tiempo para la ejecución de una prueba.

```
java Copy code  
  
@RepeatedTest(10)  
@Timeout(5) // segundos  
void testPerformance() {  
    // ... tu lógica de prueba de rendimiento  
}
```

Corroborar performance tanto en clases como en test.

2. **Herramientas de Pruebas de Rendimiento:**

Utiliza herramientas específicas de pruebas de rendimiento como Apache JMeter o Gatling para simular la carga en tu aplicación y medir su rendimiento bajo diferentes escenarios.

3. **Integración con Plataformas de CI/CD:**

Integra las pruebas de rendimiento en tus pipelines de integración continua/despliegue continuo (CI/CD) para garantizar que la mejora del rendimiento sea parte integral de tu ciclo de desarrollo.

Corroborar performance tanto en clases como en test.

Revisión de Código y Perfilado de Tests:

1. Revisión de Código:

Asegúrate de revisar el código de tus pruebas y clases en busca de posibles cuellos de botella y optimizaciones. Esto puede incluir revisar el uso eficiente de estructuras de datos, minimizar el acoplamiento, y asegurar la correcta gestión de recursos.

2. Performance Profiling en Tests:

Utiliza herramientas de perfilado de código para evaluar el rendimiento específicamente durante la ejecución de tus pruebas. Esto puede ayudarte a identificar qué partes de tus pruebas consumen más recursos y tiempo.

3. Automatización de Pruebas de Rendimiento:

Automatiza la ejecución de pruebas de rendimiento como parte de tu suite de pruebas automáticas. Esto facilita la detección temprana de posibles problemas de rendimiento.

Recuerda que la medición de rendimiento es un proceso continuo y puede ser específica para cada aplicación. Debes adaptar estas sugerencias según las necesidades y características particulares de tu proyecto.



Crear la mínima cantidad de objetos en los test y reutilizarlos en cada caso de prueba de caso feliz y excepciones (corroborar varias condiciones a la vez con `assertAll`).

La creación mínima de objetos y la reutilización en los casos de prueba es una buena práctica para mantener las pruebas eficientes y reducir la sobrecarga de recursos. Aquí tienes un ejemplo usando JUnit 5 y `assertAll` para verificar varias condiciones a la vez:

Supongamos que tienes una clase `Calculator` con un método `divide` que puede lanzar una excepción `ArithmeticException` en caso de división por cero. Vamos a crear casos de prueba para el caso feliz y el manejo de excepciones:

Crear la mínima cantidad de objetos en los test y reutilizarlos en cada caso de prueba de caso feliz y excepciones (corroborar varias condiciones a la vez con `assertAll`).

```
testing.java
1  import static org.junit.jupiter.api.Assertions.*;
2
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.Test;
5
6  public class CalculatorTest {
7
8      private Calculator calculator;
9
10     @BeforeEach
11     void setUp() {
12         // Crear un objeto Calculator reutilizable para todos los casos de prueba
13         calculator = new Calculator();
14     }
15
16     @Test
17     void testDivide_CasoFeliz() {
18         // Reutilizar el objeto calculator
19         assertAll("Division exitosa",
20             () -> assertEquals(2, calculator.divide(6, 3)),
21             () -> assertEquals(-2, calculator.divide(-6, 3)),
22             () -> assertEquals(-2, calculator.divide(6, -3)),
23             () -> assertEquals(2, calculator.divide(-6, -3))
24         );
25
26
27     @Test
28     void testDivide_PorCero() {
29         // Reutilizar el objeto calculator
30         assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0));
31     }
32 }
33
```

Crear la mínima cantidad de objetos en los test y reutilizarlos en cada caso de prueba de caso feliz y excepciones (corroborar varias condiciones a la vez con `assertAll`).

En este ejemplo:

- `@BeforeEach` garantiza que se crea un nuevo objeto `Calculator` antes de cada prueba.
- `assertAll` se utiliza para agrupar varias afirmaciones en un solo bloque, lo que facilita la lectura y la identificación de problemas.
- Los métodos `testDivide_CasoFeliz` y `testDivide_PorCero` reutilizan el objeto `calculator` creado en la fase de configuración (`setUp`).

Esta estructura ayuda a mantener las pruebas organizadas, eficientes y con un mínimo número de objetos creados, ya que se reutilizan cuando sea posible. Asegúrate de adaptar este ejemplo según tus necesidades específicas y la estructura de tu código.

Recomendaciones

- Mínimo 80% de cobertura en los test.
- No usar Date para el manejo de fechas, mejor usar LocalDate.
- Aplicar principios SOLID y ALID.
- Tener bien separadas las clases “consumidor” y “productor” de Kafka

No usar Date para el manejo de fechas, mejor usar LocalDate.

Es una buena práctica evitar el uso de la clase `Date` de Java, especialmente si estás trabajando con Java 8 o versiones posteriores. En lugar de eso, se recomienda utilizar la API `java.time` que se introdujo en Java 8, y en particular, la clase `LocalDate` para el manejo de fechas sin incluir información de zona horaria.

Aquí hay algunas razones por las cuales es preferible usar `LocalDate` en lugar de `Date`:

Inmutabilidad:

Las instancias de `LocalDate` son inmutables, lo que significa que no puedes modificarlas una vez que se han creado. Esto evita problemas potenciales de concurrencia y facilita la creación de código más seguro.

Manejo de Zonas Horarias:

`LocalDate` no almacena información de zona horaria. Si necesitas manejar zonas horarias, puedes utilizar `ZonedDateTime` o `OffsetDateTime`. Esto te permite tener un mayor control y precisión sobre las fechas y horas.

Claridad en el Código:

El uso de clases como `LocalDate` proporciona un código más claro y expresivo al realizar operaciones de fecha. Los métodos como `plusDays()`, `minusMonths()`, y otros facilitan la manipulación de fechas.

No usar Date para el manejo de fechas, mejor usar LocalDate.

```
import java.time.LocalDate;

public class EjemploFecha {

    public static void main(String[] args) {
        // Crear una instancia de LocalDate
        LocalDate hoy = LocalDate.now();
        System.out.println("Hoy es: " + hoy);

        // Operaciones con LocalDate
        LocalDate mañana = hoy.plusDays(1);
        System.out.println("Mañana será: " + mañana);
    }
}
```

Es importante recordar que si estás trabajando con fechas y horas en un contexto más amplio, también puedes explorar las clases `LocalDateTime`, `ZonedDateTime` y `OffsetDateTime` según tus necesidades específicas. Estas clases proporcionan una mayor flexibilidad para abordar distintos escenarios de manejo de fechas y horas.

Aplicar principios SOLID y ALID.



Principios SOLID:

1. **S - Principio de Responsabilidad Única (Single Responsibility Principle):**

Cada clase debe tener una única razón para cambiar. Asegúrate de que cada clase se ocupe de una sola responsabilidad.

2. **O - Principio de Abierto/Cerrado (Open/Closed Principle):**

Las clases deben estar abiertas para la extensión pero cerradas para la modificación. Esto significa que puedes agregar nuevas funcionalidades sin cambiar el código existente.

3. **L - Principio de Sustitución de Liskov (Liskov Substitution Principle):**

Las instancias de una clase base deben poder ser sustituidas por instancias de sus clases derivadas sin afectar la funcionalidad correcta del programa.

4. **I - Principio de Segregación de Interfaces (Interface Segregation Principle):**

Es mejor tener muchas interfaces específicas que una interfaz general. Esto evita que las clases implementen métodos que no necesitan.

5. **D - Principio de Inversión de Dependencia (Dependency Inversion Principle):**

Depende de abstracciones, no de implementaciones concretas. Las clases de alto nivel no deben depender de clases de bajo nivel, ambas deben depender de abstracciones.



Aplicar principios SOLID y ALID.

Principios ALID (Atomicidad, Localidad, Independencia, Duración):

1. Atomicidad:

Asegúrate de que las operaciones sean atómicas, es decir, se ejecuten como una unidad indivisible. Esto es crucial para evitar inconsistencias en el estado de la aplicación.

2. Localidad:

Mantén las operaciones y los datos locales cuando sea posible. Evitar la globalidad en las variables y operaciones contribuye a una mayor cohesión y facilidad de mantenimiento.

3. Independencia:

Las partes del sistema deben ser independientes entre sí. La modificación de una parte no debería afectar a otras, siempre que la interfaz entre ellas se mantenga constante.

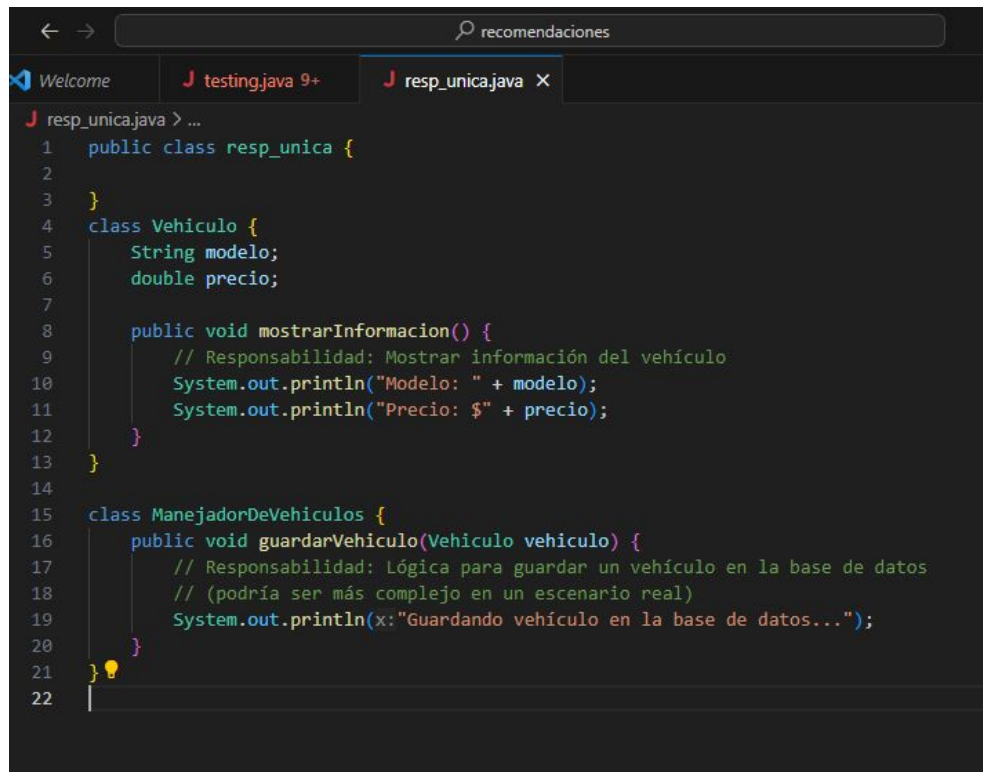
4. Duración:

Controla la duración de las operaciones para minimizar el tiempo de bloqueo y mejorar la eficiencia. Esto es especialmente relevante en operaciones de concurrencia.

Aplicar principios SOLID y ALID.

S - Principio de Responsabilidad Única (Single Responsibility Principle):

Cada clase debe tener una única razón para cambiar. Asegúrate de que cada clase se ocupe de una sola responsabilidad.

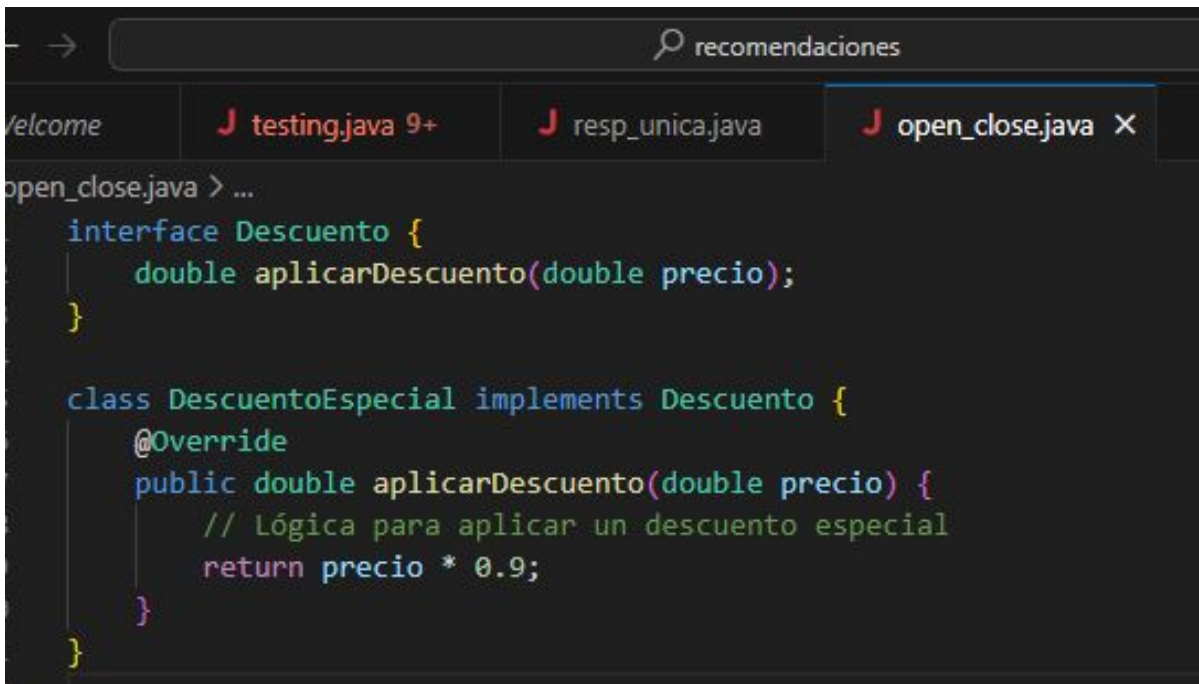
A screenshot of an IDE window with a dark theme. The title bar shows a search icon and the text 'recomendaciones'. Below the title bar, there are three tabs: 'Welcome', 'testing.java 9+', and 'resp_unica.java X'. The 'resp_unica.java' tab is active. The code editor shows the following Java code:

```
1 public class resp_unica {  
2  
3 }  
4 class Vehiculo {  
5     String modelo;  
6     double precio;  
7  
8     public void mostrarInformacion() {  
9         // Responsabilidad: Mostrar información del vehículo  
10        System.out.println("Modelo: " + modelo);  
11        System.out.println("Precio: $" + precio);  
12    }  
13 }  
14  
15 class ManejadorDeVehiculos {  
16     public void guardarVehiculo(Vehiculo vehiculo) {  
17         // Responsabilidad: Lógica para guardar un vehículo en la base de datos  
18         // (podría ser más complejo en un escenario real)  
19         System.out.println(x:"Guardando vehículo en la base de datos...");  
20     }  
21 }  
22
```

Aplicar principios SOLID y ALID.

O - Principio de Abierto/Cerrado (Open/Closed Principle):

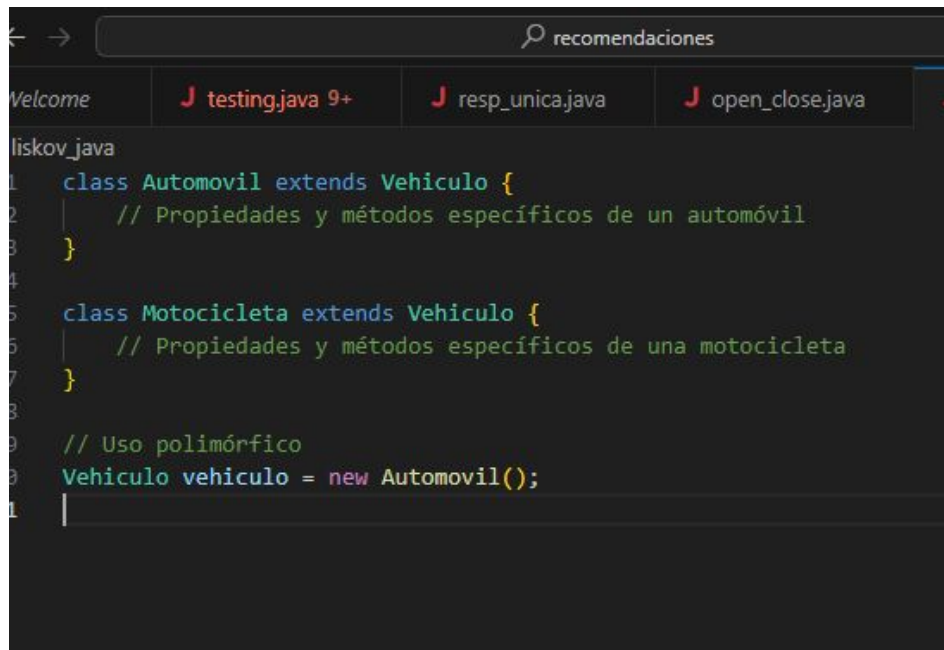
Las clases deben estar abiertas para la extensión pero cerradas para la modificación. Esto significa que puedes agregar nuevas funcionalidades sin cambiar el código existente.



```
→ [recomendaciones]
welcome | J testing.java 9+ | J resp_unica.java | J open_close.java X
open_close.java > ...
interface Descuento {
    double aplicarDescuento(double precio);
}

class DescuentoEspecial implements Descuento {
    @Override
    public double aplicarDescuento(double precio) {
        // Lógica para aplicar un descuento especial
        return precio * 0.9;
    }
}
```

Aplicar principios SOLID y ALID.



The screenshot shows a code editor with a dark theme. At the top, there's a search bar with the text "recomendaciones". Below it, there are tabs for "Welcome", "testing.java 9+", "resp_unica.java", and "open_close.java". The active tab is "testing.java 9+", which contains the following Java code:

```
liskov_java
1  class Automovil extends Vehiculo {
2      // Propiedades y métodos específicos de un automóvil
3  }
4
5  class Motocicleta extends Vehiculo {
6      // Propiedades y métodos específicos de una motocicleta
7  }
8
9  // Uso polimórfico
10 Vehiculo vehiculo = new Automovil();
11
```

L - Principio de Sustitución de Liskov (Liskov Substitution Principle):

Las instancias de una clase base deben poder ser sustituidas por instancias de sus clases derivadas sin afectar la funcionalidad correcta del programa.

Aplicar principios SOLID y ALID.

```
Welcome    J testing.java 9+    J resp_unica.java    J open_close.java    J liskov
isp.java > ...
1  public // Interfaz segregada para diferentes tipos de vehículos
2  interface Motorizado {
3      void arrancar();
4      void apagar();
5  }
6
7  class Automovil implements Motorizado {
8      // Implementación para un automóvil
9  }
10
11 class Motocicleta implements Motorizado {
12     // Implementación para una motocicleta
13 }
14 {
15
16 }
17
```

I - Principio de Segregación de Interfaces (Interface Segregation Principle):

Es mejor tener muchas interfaces específicas que una interfaz general. Esto evita que las clases implementen métodos que no necesitan.

Aplicar principios SOLID y ALID.

```
elcome  J testing.java 9+  J resp_unica.java  J open_close.java  J
nversion_dependencias.java > ...
public // Dependencia invertida a través de una interfaz
interface Almacenamiento {
    void guardar(String datos);
}

class BaseDeDatos implements Almacenamiento {
    @Override
    public void guardar(String datos) {
        // Lógica para guardar datos en una base de datos
    }
}

class Archivo implements Almacenamiento {
    @Override
    public void guardar(String datos) {
        // Lógica para guardar datos en un archivo
    }
}
}
```

D - Principio de Inversión de Dependencia (Dependency Inversion Principle):
Depende de abstracciones, no de implementaciones concretas. Las clases de alto nivel no deben depender de clases de bajo nivel, ambas deben depender de abstracciones.

Tener bien separadas las clases “consumidor” y “productor” de Kafka

Cuando trabajas con Apache Kafka, es una buena práctica tener bien separadas las responsabilidades entre los productores y consumidores para lograr un diseño modular y mantenible. A continuación, te proporcionaré ejemplos básicos de cómo estructurar clases de productor y consumidor en Java utilizando la biblioteca oficial de Kafka para Java, conocida como Apache Kafka Clients.

Recomendaciones

- No tener dependencias sin usar o innecesarias.
- Asegurarse de enviar código fuente no compilado.
- Utilizar arquitectura hexagonal.
- Dockerizar la solución completa con docker-compose. Quien levante la aplicación no debe tener nada más instalado que docker-compose. Realizar la compilación de la aplicación con docker.

Utilizar arquitectura hexagonal.

La arquitectura hexagonal, también conocida como la arquitectura de puertos y adaptadores, es un enfoque que busca separar las preocupaciones y mantener una estructura modular y fácil de mantener. En esta arquitectura, se distinguen claramente las capas internas (dominio) de las externas (puertos y adaptadores). Aquí tienes un ejemplo básico utilizando Java y Spring Boot:

Utilizar arquitectura hexagonal.

1. Capa del Dominio:

En esta capa, defines tu lógica de negocio y las entidades del dominio.



```
→ recomendaciones
va 9+  J resp_unica.java  J open_close.java  J liskov_java  J isp.java

apa_dominio.java > ...
// Dominio (Entidades y Lógica de Negocio)
public class Usuario {
    private String id;
    private String nombre;

    // Constructores, métodos, etc.
}

public interface UsuarioService {
    Usuario obtenerUsuarioPorId(String id);
    List<Usuario> obtenerTodosLosUsuarios();
}

public class UsuarioServiceImpl implements UsuarioService {
    private final UsuarioRepository usuarioRepository;

    public UsuarioServiceImpl(UsuarioRepository usuarioRepository) {
        this.usuarioRepository = usuarioRepository;
    }

    @Override
    public Usuario obtenerUsuarioPorId(String id) {
        return usuarioRepository.findById(id).orElse(null);
    }


    @Override
    public List<Usuario> obtenerTodosLosUsuarios() {
        return usuarioRepository.findAll();
    }
}
```

Utilizar arquitectura hexagonal.

2. Capa de Puertos:

Define interfaces que serán implementadas por adaptadores. Estas interfaces son puertos que definen cómo se interactuará con el dominio.

java

 Copy code

```
// Puerto para el repositorio de usuarios
public interface UsuarioRepository {
    Optional<Usuario> findById(String id);
    List<Usuario> findAll();
}
```

3. Adaptadores:


Implementa las interfaces definidas en los puertos. Estos adaptadores interactúan con el dominio y se encargan de la comunicación con el exterior.

```
// Adaptador de repositorio para MongoDB
@Repository
public class MongoUsuarioRepository implements UsuarioRepository {
    private final MongoTemplate mongoTemplate;

    public MongoUsuarioRepository(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    @Override
    public Optional<Usuario> findById(String id) {
        return Optional.ofNullable(mongoTemplate.findById(id, Usuario.class));
    }

    @Override
    public List<Usuario> findAll() {
        return mongoTemplate.findAll(Usuario.class);
    }
}
```



4. Configuración y Punto de Entrada:

En esta capa, configuras la inyección de dependencias y defines los controladores o adaptadores de entrada.

```
J isp.java 6  J inversion_dependencias.java 2  J capa_dominio.java 7  J capa_puertos.java 3  J
J capa_cofig.java > capc_cofig
1 // Configuración de Spring Boot
2 @Configuration
3 public class AppConfig {
4     @Bean
5     public UsuarioService usuarioService(UsuarioRepository usuarioRepository) {
6         return new UsuarioServiceImpl(usuarioRepository);
7     }
8 }
9
10 // Controlador de la aplicación (Adaptador de entrada)
11 @RestController
12 @RequestMapping("/api/usuarios")
13 public class UsuarioController {
14     private final UsuarioService usuarioService;
15
16     @Autowired
17     public UsuarioController(UsuarioService usuarioService) {
18         this.usuarioService = usuarioService;
19     }
20
21     @GetMapping("/{id}")
22     public ResponseEntity<Usuario> obtenerUsuarioPorId(@PathVariable String id) {
23         Usuario usuario = usuarioService.obtenerUsuarioPorId(id);
24         return ResponseEntity.ok(usuario);
25     }
26
27     @GetMapping
28     public ResponseEntity<List<Usuario>> obtenerTodosLosUsuarios() {
29         List<Usuario> usuarios = usuarioService.obtenerTodosLosUsuarios();
30         return ResponseEntity.ok(usuarios);
31     }
32 }
33
```

Recomendaciones

- Crear un README con el paso a paso para levantar el aplicativo (comandos de docker necesarios), mencionar la url de Swagger y hacer referencia a los endpoints desarrollados. Agregar en el README lo que crea relevante.
- Utilizar alguna herramienta de análisis de cobertura, como JaCoCo u otra, para obtener un reporte de cobertura de los tests, y agregar las indicaciones en el README para obtener el reporte

<https://refactoring.guru/es/design-patterns>
https://www.onlinegdb.com/online_java_compiler