Examen Final de Desarrollo de software

Notas:

- Duración de la prueba: 3 horas
- Entrega: construye una carpeta dentro de tu repositorio personal en Github que se llame ExamenFinal-3S2 donde se incluya todas tus respuestas cada una dividida por carpetas. No olvides colocar un Readme para indicar el orden de tus respuestas y procedimientos.
- Construye un proyecto de IntelliJ Idea para todos tus códigos.
- **No se admiten imágenes sin explicaciones.**
- Evita copiar y pegar cualquier información de internet. Cualquier acto de plagio anula la evaluación.

Pregunta 1 TDD (6 puntos)

En este ejercicio deberás construir una carpeta llamada Pregunta1, y dentro de ellas debes construir 6 pequeños proyectos que se llamen Antes, Fase1, Fase2, Fase3, Fase4, Fase5 y debes coloca un archivo readme.md general que indique claramentes las diferencias y los avances de cada carpeta. No es necesario colocar imágenes.

La aplicación de gestión de vuelos

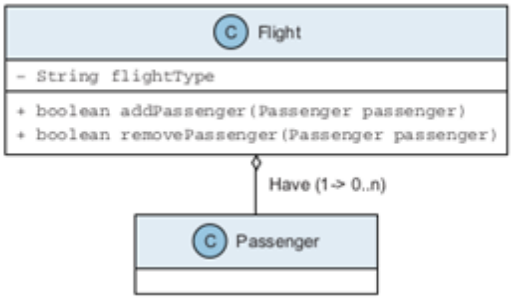
Tested Data Systems está desarrollando una aplicación de gestión de vuelos para uno de sus clientes. Actualmente, la aplicación puede crear y configurar vuelos, agregar pasajeros y eliminarlos de los vuelos.

El desarrollo de la aplicación de gestión de vuelos, es una aplicación Java creada con la ayuda de Maven. El software debe mantener una política con respecto a agregar pasajeros y eliminarlos de los vuelos. Los vuelos pueden ser de diferentes tipos: actualmente, hay vuelos económicos y de negocios, pero es posible que se agreguen otros tipos más adelante, según los requisitos del cliente. Tanto los pasajeros VIP como los clientes regulares pueden agregarse a los vuelos económicos, pero solo los pasajeros VIP pueden agregarse a los vuelos de negocios.

Antes

También existe una política para la eliminación de pasajeros de los vuelos: un pasajero regular puede ser eliminado de un vuelo, pero un pasajero VIP no puede ser eliminado.

Veamos el diseño inicial de esta aplicación. Tiene un campo llamado flightType en la clase Flight. Su valor determina el comportamiento de los métodos addPassenger y removePassenger. Los desarrolladores deben centrarse en la toma de decisiones a nivel del código para estos dos métodos.



Ver el listado de la clase Passenger, Flight

Revisa la clase Airport, que incluye un método main que actúa como cliente de las clases Flight y Passenger y trabaja con los diferentes tipos de vuelos y pasajeros.

Fase 1

Para crear una aplicación confiable y poder comprender e implementar la lógica comercial de manera fácil y segura, se considera cambiar la aplicación al enfoque TDD.

Pregunta:

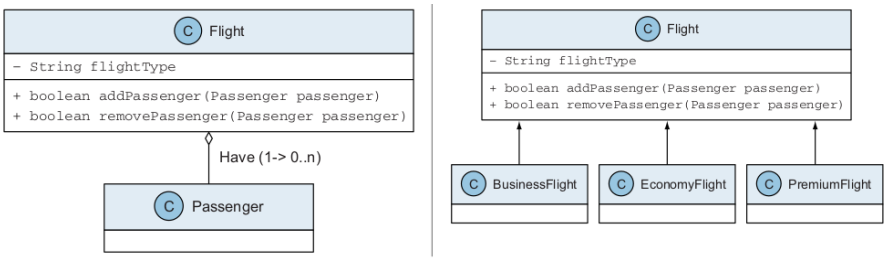
Sigue la lógica comercial para un vuelo comercial y traduce eso escribiendo una prueba llamada AirportTest. ¿Cuáles son los resultados de las pruebas con cobertura obtenidad?
¿Puedes indicar algunas conclusiones de lo anterior, necesitamos refactorizar?

Fase 2

La clave para la refactorización es mover el diseño para usar polimorfismo en lugar de código condicional de estilo procedimental. Con el polimorfismo el método al que está llamando no se determina en tiempo de compilación, sino en tiempo de ejecución, según el tipo de objeto efectivo.

Para el ejercicio tratamos la refactorización de la aplicación de gestión de vuelos reemplazando el condicional con polimorfismo: se elimina el campo flightType y se introduce una jerarquía de clases.

El principio en acción aquí se llama principio abierto/cerrado.



Preguntas:

- La refactorización se logrará manteniendo la clase Flight base y para cada tipo condicional, agregando una clase separada para extender Flight. Debes cambiar addPassenger y

removePassenger a métodos abstractos y delegar su implementación a subclases. El campo flightType ya no es significativo y se eliminará.
- Implementa una clase EconomyFlight que amplía Flight e implementa los métodos abstractos heredados addPassenger y removePassenger.
- Implementa una clase BusinessFlight que amplía Flight e implementa los métodos abstractos heredados addPassenger y removePassenger.
- La refactorización y los cambios de la API se propagan a la implementacion de las pruebas. ¿Cómo?.

¿Cuál es el código de cobertura ahora?. ¿Ayudó la refactorización a la mejor calidad de código?

Fase 3

Implementa nuevas funciones requeridas por el cliente que amplían las políticas de la aplicación.

Las primeras características nuevas son un nuevo tipo de vuelo, premium, y políticas relacionadas con este tipo de vuelo. Existe una política para agregar un pasajero: si el pasajero es VIP, el pasajero debe agregarse al vuelo premium; de lo contrario, la solicitud debe ser rechazada.

También existe una política para la eliminación de un pasajero: si se requiere, un pasajero puede ser eliminado de un vuelo.

Considera que, después de recibir el requisito para la implementación de este tercer tipo de vuelo, es hora de agrupar más las pruebas existentes utilizando la anotación JUnit 5 @Nested y luego implementar el requisito de vuelo premium de manera similar.

Pregunta

Utiliza la clase AirportTest refactorizada antes de pasar al trabajo para el vuelo premium en el código desarrollado como mejora a tus resultados. Ver el codigo entregado en la evaluación.

Ejecuta las pruebas.

Fase 4

Pregunta

Realiza la implementación de la clase PremiumFlight y su lógica. Debes crear PremiumFlight como una subclase de Flight y sobreescribir los métodos addPassenger y removePassenger, pero actúan como stubs: no hacen nada y simplemente devuelven false. El estilo TDD de trabajo implica crear primero las pruebas y luego la lógica de negocios.

Ahora implementa las pruebas de acuerdo con la lógica comercial de vuelos premium:

- Se agrega un pasajero a un vuelo premium, solo si es pasajeros VIP
- Cualquier tipo de pasajero puede ser eliminado de un vuelo premium.

Después de escribir las pruebas, realiza la ejecución.

Recuerda, trabajar con el estilo TDD significa ser impulsado por las pruebas, por lo que primero creamos la prueba para fallar y luego escribimos la pieza de código que hará que la prueba pase. Pero aquí hay otra cosa destacable: **la prueba para un vuelo premium y un pasajero regular ya está verde**. Esto significa que la lógica comercial existente (los métodos addPassenger y removePassenger que devuelven false) es suficiente para este caso.

Regresa a la clase PremiumFlight y agrega la lógica comercial solo para pasajeros VIP. Impulsado por las pruebas,

Comprueba que la cobertura del código es del 100%.

Fase 5

Pregunta

Ocasionalmente, a propósito o por error, el mismo pasajero se ha agregado a un vuelo más de una vez. Esto ha causado problemas con la gestión de asientos y estas situaciones deben evitarse. Tu como desarrollado necesitas asegurarte de que cada vez que alguien intente agregar un pasajero, si el pasajero se ha agregado previamente al vuelo, la solicitud debe ser rechazada.

Esta es una nueva lógica comercial y debes implementarla al estilo TDD.

Para garantizar la unicidad de los pasajeros en un vuelo, cambia la estructura de la lista de pasajeros a un conjunto. Entonces, hace una refactorización de código que también se propagará a través de las pruebas.

¿Cómo cambia la clase de vuelo en este contexto?.

Luego crea una nueva prueba para verificar que un pasajero solo se puede agregar una vez a un vuelo

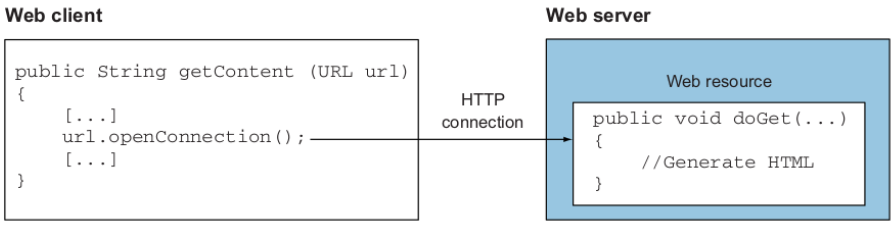
¿Consigues una mejor cobertura de código?

Escala de puntuaciónes

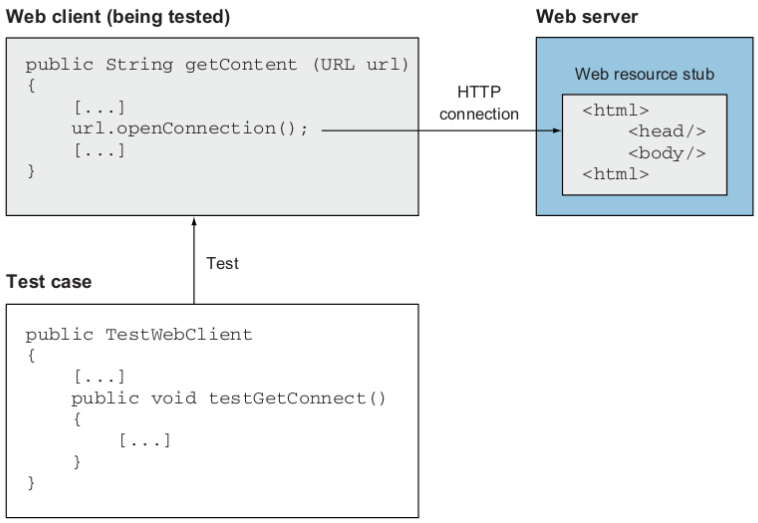
- **Menos de la fase 3 :0**
- **Hasta las fase 3 : 1**
- **Hasta las fase 4: 2**
- **Todas las fase: 6**

Pregunta 2 Stubs (5 puntos)

Los ingenieros de una empresa construyen stubs para una aplicación que abre una conexión HTTP a una URL y lee su contenido. La aplicación de muestra (limitada a un método WebClient.getContent) abre una conexión HTTP a un recurso web remoto. El recurso web remoto es un servlet, que genera una respuesta HTML.



El objetivo de los ingenieros de esta empresa es realizar una prueba unitaria del método getContent mediante el uso de stubs del recurso web remoto, como se muestra.



El recurso web remoto aún no está disponible y los ingenieros deben avanzar con su parte del trabajo en ausencia de ese recurso. Reemplazan el recurso web de servlet con el código stub, una página HTML simple que devuelve todo lo que necesitan para el caso de prueba TestWebClient.

Este enfoque va a permitir probar el método getContent independientemente de la implementación del recurso web (que a su vez podría llamar a varios otros objetos en la cadena de ejecución, posiblemente hasta una base de datos).

Revisa el ejemplo de un código que muestra un stub en acción, utilizando el ejemplo de conexión HTTP simple. Ver WebClient.java.

La aplicación de ejemplo tiene dos escenarios posibles: el servidor web remoto o podría ser parte de la plataforma donde se implementará la aplicación. En ambos casos, necesitamos introducir un servidor en la plataforma de desarrollo para poder realizar pruebas unitarias de la clase WebClient. Una solución relativamente fácil sería instalar un servidor de prueba Apache y colocar algunas páginas web de prueba en tu raíz de documentos.

Para este ejercicio puedes usar apache si estás más familiarizado o cualquier otra herramienta que conozcas.

Pregunta

Utiliza el código Jetty.java y realiza experimentos de cómo iniciar desde Java y cómo definir una raíz de documento (/) desde la cual comenzar a servir archivos. Si inicias el programa y escribes en el navegador http://localhost:8081, deberías poder ver el contenido del archivo pom.xml

Obtiene un efecto similar cambiando el código para establecer la base como root.setResourceBase ("."), reiniciando el servidor y luego navegando a http://localhost:8081/.

Los desarrolladores de la empresa escriben pruebas para verificar que pueden llamar a una URL válida y obtener su contenido.
Estas pruebas son los primeros pasos para verificar la funcionalidad de las aplicaciones web que interactúan con clientes externos. Ver TestWebClientInicial.java.

Pregunta

Explica los resultados de ejecutar este código.
Alternativamente, podemos configurar Jetty para usar un controlador personalizado que devuelva la cadena "Esto trabaja" en lugar de obtenerla de un archivo. Esta técnica es mucho más poderosa porque nos permite realizar pruebas unitarias cuando el servidor HTTP remoto devuelve un código de error a la aplicación cliente WebClient.

```
private static class TestGetContentOkHandler extends AbstractHandler {
      public void handle(String target, HttpServletRequest request, HttpServletResponse response, int dispatch)
throws IOException {

         OutputStream out = response.getOutputStream();

         ByteArrayISO8859Writer writer = new ByteArrayISO8859Writer ();

         writer.write("Esto funciona");

         writer.flush();

         response.setIntHeader(HttpHeaders.CONTENT_LENGTH, writer.size());

         writer.writeTo(out);

         out.flush();

      }

   }
```

Pregunta

Ahora que este controlador está escrito, podemos decirle a Jetty que lo use llamando a context.set-Handler(new TestGetContentOkHandler()). Escribe una prueba llamada Testwebclient.java y explica los resultados.

Pregunta

Analiza el uso de stubs en conexión HTTP. Se quiere probar el código de forma aislada. Las pruebas funcionales o de integración probarán la conexión en una etapa posterior.

Sugerencia: cuando se trata de usar stub en la conexión sin cambiar el código, nos beneficiamos de las clases Java URL y HttpURLConnection, que nos permiten conectar controladores de protocolo personalizados para procesar cualquier tipo de protocolo de comunicación. Podemos hacer que cualquier llamada a la clase HttpURLConnection sea redirigida a la propia clase, que devolverá lo que necesitemos para la prueba.

Pregunta

Para implementar un controlador de protocolo de URL personalizado, se llama al método estático de URL setURLStreamHandlerFactory y se le pada un URLStreamHandlerFactory personalizado. Realiza la implementación de código auxiliar del controlador de flujo de URL de manera que cada vez que se llama al método URL openConnection, se llama a la clase URLStreamHandlerFactory para devolver un URLStream-Handler.

Sugerencia:

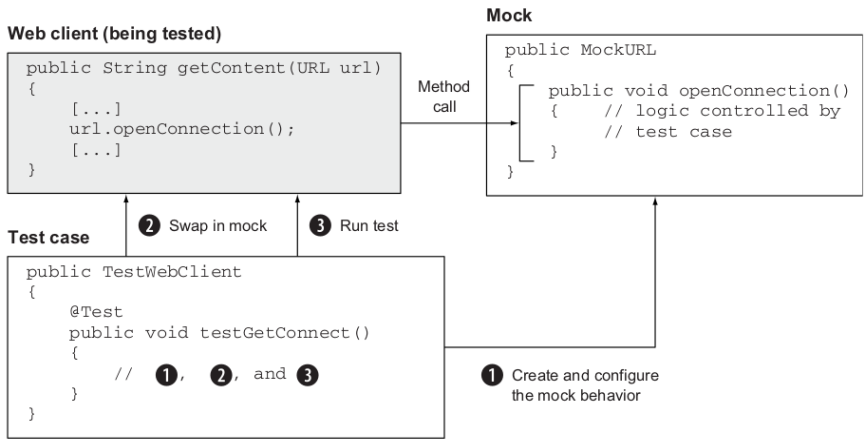
```
private static class StubStreamHandlerFactory implements
   URLStreamHandlerFactory {
   @Override
     public URLStreamHandler createURLStreamHandler(String protocol) {
      return new StubHttpURLStreamHandler();
    }
}
private static class StubHttpURLStreamHandler
  extends URLStreamHandler {
    @Override
    protected URLConnection openConnection(URL url)
              throws IOException {
       return new StubHttpURLConnection(url);
   }
}
```

Pregunta

Realiza una implementación stub de la clase HttpURLConnection para que podamos devolver cualquier valor que queramos para la prueba. Esta implementación simple devuelve la cadena "Esto funciona" . ¿Se pasa la prueba?.

Pregunta 3 Mocks (5 puntos)

La figura ilustra un objeto mock. La clase MockURL representa la clase de URL real y todas las llamadas a la clase de URL en getContent se dirigen a la clase MockURL. La prueba es el controlador: crea y configura el comportamiento que debe tener el mock para esta prueba, (de alguna manera) reemplaza la clase URL real con la clase MockURL y ejecuta la prueba.



Pregunta

Implementa código que abre una conexión HTTP a una URL determinada y lea el contenido de esa URL. Suponga que este código es un método de una aplicación más grande que queremos probar unitariamente (mira la clase WebClient).

Pregunta

Prueba el método getContent independientemente de una conexión HTTP real a un servidor web. Esto significa escribir un mock URL en el que el método url.openConnection devuelve un mock HttpURLConnection . La clase MockHttpURLConnection proporcionaría una implementación que permite que la prueba decida qué devuelve el método getInputStream.

```
@Test
public void testGetContentOk() throws Exception {
      MockHttpURLConnection mockConnection = new MockHttpURLConnection();
      mockConnection.setupGetInputStream(
                              new ByteArrayInputStream("It works".getBytes()));
       MockURL mockURL = new MockURL();
       mockURL.setupOpenConnection(mockConnection);
      WebClient client = new WebClient();
      String workingContent = client.getContent(mockURL);
       assertEquals("Esto funciona", workingContent);
}
```

¿Funciona o no funciona?. Explica los detalles de tu respuesta

Pregunta

Refactorizar el método get-Content es una técnica que usa objetos mocks. Este método hace dos cosas: obtiene un objeto HttpURLConnection y luego lee su contenido. Utiliza esta refactorización y modifica la clase WebCliente por WebCliente1. Indica cual es la parte que recuperó el objeto HttpURLConnection.

Pregunta

Un enfoque de refactorización común llamado Method Factory es especialmente útil cuando la clase a usar un mock no tiene interfaz. La estrategia es extender esa clase, agregar algunos métodos setter para controlarla y sobreescribir algunos de sus métodos getter para devolver lo que queremos para la prueba. En la prueba, podemos llamar al método setHttpURLConnection y pasarle el objeto mock HttpURLConnection. Indica como se cambia la prueba dada hasta este momento.

Los desarrolladores quieren probar otro enfoque de refactorización que dice que cualquier recurso que usemos debe pasarse al método getContent o a la clase WebClient. El único recurso que usamos es el objeto HttpURLConnection.

Podríamos cambiar la firma WebClient.getContent a :

public String getContent (URL url, conexión HttpURLConnection)

Este cambio significa que estamos enviando la creación del objeto HttpURLConnection al llamador de WebClient. Sin embargo, la URL se recupera de la clase HttpURLConnection y la firma no se ve muy bien.

Una mejor solución consiste en crear una interfaz ConnectionFactory, como se muestra en los siguientes códigos. El papel de las clases que implementan la interfaz ConnectionFactory es devolver un InputStream desde una a conexión, cualquiera que sea la conexión (HTTP, TCP/IP, etc.).

```
[...]
import java.io.InputStream;
public interface ConnectionFactory {
     InputStream getData() throws Exception;
}
```

Pregunta

Cambia el código de WebClient se cambia a otra clase llamada WebClient2. Indica las diferencias y si es una mejor solución.
La siguiente lista muestra la implementación de ConnectionFactory para el protocolo HTTP. Ver HttpURLConnectionFactory.java

```
[...]
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class HttpURLConnectionFactory implements ConnectionFactory {
  private URL url;
public HttpURLConnectionFactory(URL url) {
     this.url = url;
}
public InputStream getData() throws Exception {
    HttpURLConnection connection =
       (HttpURLConnection) this.url.openConnection();
       return connection.getInputStream();
   }
}
```

Pregunta

Prueba el método getContent escribiendo un mock para ConnectionFactory. Rescribe la prueba para usar MockConnectionFactory (uso del mock).

Una **expectativa** es una función integrada en el mock que verifica si la clase externa que llama a este mock tiene el comportamiento correcto.

Para demostrar las expectativas de que el recurso se ha cerrado y evitamos una fuga de recursos, eche un vistazo a la siguiente lista.

```
import java.io.IOException;
import java.io.InputStream;
public class MockInputStream extends InputStream {
```

```
  private String buffer;
   private int position = 0;
   private int closeCount = 0;
   public void setBuffer(String buffer) {
      this.buffer = buffer;
}
public int read() throws IOException {
 if (position == this.buffer.length()) {
      return -1;
}
       return this.buffer.charAt(this.position++);
}
public void close() throws IOException {
     closeCount++;
      super.close();
}
public void verify() throws java.lang.AssertionError {
   if (closeCount != 1) {
     throw new AssertionError ("close()...");
      }
 }
}
```

Pregunta
En el caso de la clase MockInputStream, la expectativa de cierre es simple: siempre queremos que se llame una vez. Sin embargo, la mayoría de las veces, la expectativa de closeCount depende del código bajo prueba. Un mock generalmente tiene un método como setExpectedCloseCalls para que la prueba pueda decirle al mock qué esperar.

Modifique el método de prueba testGetContentOk de la siguiente manera para usar el nuevo MockInputStream:

```
[...]
public class TestWebClientFail {

  @Test
    public void testGetContentOk() throws Exception {
     MockConnectionFactory mockConnectionFactory =
                new MockConnectionFactory();
      MockInputStream mockStream = new MockInputStream();
       mockStream.setBuffer("Esto funciona!");
       mockConnectionFactory.setData(mockStream);
      WebClient2 client = new WebClient2();
      String workingContent = client.getContent(mockConnectionFactory);

      assertEquals("Esto funciona", workingContent);
      mockStream.verify();
    }
}
```

¿Cuál es el resultado de la prueba?.

Pregunta

Utiliza Mockito (https://site.mockito.org), para la prueba de WebClient y analiza los resultados.

Pregunta 4 Microservicios (4 puntos)

Completa la actividad dada en https://github.com/kapumota/Actividades/blob/main/Microservicios-cooperativos.md desarrollada en clase.

La clase de interfaz de Java, ProductCompositeService.java, sigue el mismo patrón que utilizan los servicios core .

La clase de modelo, ProductAggregate.java, es un poco más compleja que los modelos core ya que contiene campos para listas de recomendaciones y reseñas.

Pregunta

Para evitar codificar la información de la dirección de los servicios core en el código fuente del microservicio compuesto, este último utiliza un archivo de propiedades donde se almacena la información sobre cómo encontrar los servicios core. Escribe el archivo de propiedades, application.yml de la carpeta resources que tiene el siguiente aspecto para esa funcionalidad.

El componente de integración, ProductCompositeIntegration.java. Se declara como Spring Bean utilizando la anotación @Component e implementa las interfaces API de los tres servicios principales:

```
 @Component
public class ProductCompositeIntegration implements ProductService,
    RecommendationService, ReviewService {
```

El componente de integración utiliza una clase auxiliar en Spring Framework, RestTemplate, para realizar las solicitudes HTTP reales a los microservicios principales. Antes de inyectarlo en el componente de integración, se debe configurar. Se hace en la clase de aplicación main, ProductCompositeServiceApplication.java, de la siguiente manera:

```
@Bean
RestTemplate restTemplate() {
  return new RestTemplate();
}
```

Ahora se pueden inyectar RestTemplate, junto con un mapper JSON que se usa para acceder a los mensajes de error en caso de errores, y los valores de configuración que hemos establecido en el archivo de propiedades.

1. Los objetos y los valores de configuración se inyectan en el constructor.

2. El cuerpo del constructor almacena los objetos inyectados y crea las URL en función de los valores inyectados, de la siguiente manera:

3. Finalmente, el componente de integración implementa los métodos API para los tres servicios core mediante el uso de RestTemplate para realizar las llamadas salientes reales:

Finalmente, veamos a clase de implementación de la API ProductCompositeServiceImpl.java.

1. De la misma manera que con los servicios core, el servicio compuesto implementa su interfaz API, ProductCompositeService, y se anota con @RestController para marcarlo como un servicio REST:

```
@RestController
public class ProductCompositeServiceImpl implements
ProductCompositeService {
```

2. La clase de implementación requiere el bean ServiceUtil y su propio componente de integración, por lo que se inyectan en su constructor.

3. Finalmente, el método API se implementa de la siguiente manera:

```
@Override
public ProductAggregate getProduct(int productId) {
    Product product = integration.getProduct(productId);
   List<Recommendation> recommendations = integration.getRecommendations(productId);
    List<Review> reviews = integration.getReviews(productId);
     return createProductAggregate(product, recommendations,
    reviews, serviceUtil.getServiceAddress());
}
```

El manejo de errores de una manera estructurada y bien pensada es esencial en un panorama de microservicios donde una gran cantidad de microservicios se comunican entre sí mediante API sincrónicas, por ejemplo, mediante HTTP y JSON. También es importante separar el manejo de errores específico del protocolo, como los códigos de estado HTTP, de la lógica empresarial.

En este caso los microservicios implementan su lógica comercial directamente en los componentes @RestController.

Las implementaciones de API usan las excepciones en el proyecto util para señalar errores. Se informarán al cliente REST como códigos de estado HTTPS que indican qué salió mal. Por ejemplo, la clase de implementación de microservicio Product, ProductServiceImpl.java, usa la excepción InvalidInputException para devolver un error que indica una entrada no válida, así como la excepción NotFoundException para decirnos que el producto que se solicitó no existe.

Preguntas

- Dado que actualmente no estamos usando una base de datos, tenemos que simular cuándo lanzar NotFoundException. ¿Muestra eso?
- ¿Qué hace el cliente API, es decir, el componente de integración del microservicio Composite?

- Explica el manejo de errores para getRecommendations() y getReviews() en el componente de integración,

Prueba todo el proceso dado de la evaluación realizando los siguientes pasos:

1. Crea e inicia los microservicios como procesos en segundo plano.
2. Usa curl para llamar a la Composite API .
curl http://localhost:7000/product-composite/1
curl http://localhost:7000/product-composite/1 -s | jq .
3. Detenlos.