

Examen Parcial 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 ExamenParcial-3S2 donde se incluya todas tus respuestas. No olvides colocar un Readme para indicar el orden de tus respuestas y procedimientos.
- Construye un proyecto de IntelliJ Idea para todos tus códigos.
- Sube un documento PDF de tus respuestas como respaldo a la plataforma,
- **No se admiten imágenes sin explicaciones.**
- Evita copiar y pegar cualquier información de internet. Cualquier acto de plagio anula la evaluación.

Pregunta 1 (3 puntos)

Antes de resolver estos ejercicios revisa el siguiente enlace <https://dev.java/learn/lambda-expressions/>

¿Cuál es el resultado de la siguiente clase?

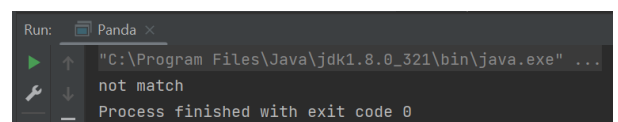
```
1: import java.util.function.*;
2:
3: public class Panda {
4: int age;
5: public static void main(String[] args) {
6: Panda p1 = new Panda();
7: p1.age = 1;
8: check(p1, p -> p.age < 5);
9: }
10: private static void check(Panda panda,
11: Predicate<Panda> pred) {
12: String result =
13: pred.test(panda)?"match":"not match";
14: System.out.print(result);
15: }}
```



El resultado es match, como se puede ver en la imagen.

La sintaxis de la expresión lambda `p -> p.age < 5` es correcta y esta se encarga de comprobar que la edad de `p1` -objeto de la clase `Panda`- sea menor que 5 años. Enfocándonos en los parámetros del método `check`, primero recibe un objeto de la clase `Panda` y de segundo argumento, por ser la expresión del tipo `Predicate`, esta retornará a un booleano, efectivamente, y como la interfaz contiene un método `test` que evalúa el `Predicate` de un argumento dado - en este caso del objeto `panda` - al ser `true` la expresión lambda se imprimirá el `result` que es "match" pues `p1.age=1`.

Si cambiáramos `p -> p.age < 1` o `p1.age=6` se imprimirá "not match"

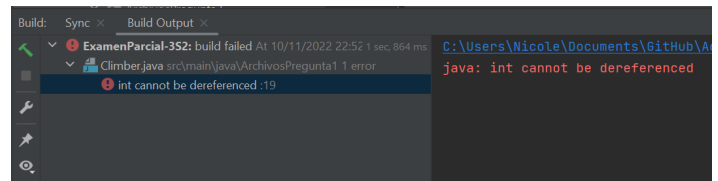


=====

¿Cuál es el resultado del siguiente código?

```
1: interface Climb {
2:   boolean isTooHigh(int height, int limit);
3: }
4:
5: public class Climber {
6:   public static void main(String[] args) {
7:     check((h, m) -> h.append(m).isEmpty(), 5);
8:   }
9:   private static void check(Climb climb, int height) {
10:    if (climb.isTooHigh(height, 10))
11:      System.out.println("too high");
12:    else
13:      System.out.println("ok");
14:   }
15: }
```

Al inicio



Después de agregar modificaciones



Sobre StringBuilder append



Se produce un error de compilación en la línea 7 - índice según el código anterior, ya que el código presenta modificaciones para hacerlo funcional - debido a que los parámetros h y m son del tipo int e intenta utilizar a uno de ellos como si fuera un String.

El método append() es usado en los tipos de StringBuilder y h es int, en caso h fuera un StringBuilder el método isEmpty() se usa en Strings y también habría un error, para que el método sea funcional implementamos una función - isTooHigh de Climb - para que check() pueda ser probada de manera simple, esto se ve reflejado en el código.

=====

¿Qué lambda puede reemplazar la clase Secret1 para devolver el mismo valor?

```
interface Secret {
    String magic(double d);
}

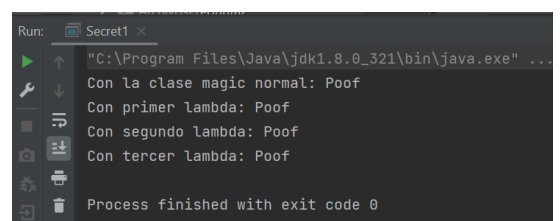
class Secret1 implements Secret {
    public String magic(double d) {
        return "Poof";
    }
}
```

La expresión lambda considerada como “más simple” según mi criterio es **d → "Poof";**

Pero también se pueden considerar ciertas variaciones que funcionan bien para reemplazar aquella clase, que funcionan de manera semejante, tales como:

```
(d) → {return "Poof";};
(d) → { String f = ""; return "Poof"; };
```

En el main se dispone a mostrar las salidas con la clase definida inicialmente y luego con las tres expresiones lambdas consideradas.



Completa sin causar un error de compilación

```
public void remove(List<Character> chars){
    char end = 'z';
    chars.removeIf(c -> {
        char start = 'a'; return start <= c && c <= end; });
    // Inserta código
    //end='y'; //esto si generará un error de compilación

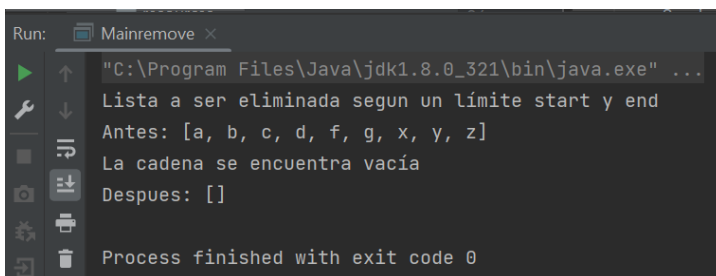
    //el código insertado a continuación evalúa si la cadena quedó vacía para
    lanzar un mensaje de que así fue
    //y en caso contrario, también lo señala
    //no genera ningún error de compilación

    String aclaracion = (chars.isEmpty())?"La cadena se encuentra vacía
    ":"La cadena no se encuentra vacía";
    System.out.println(aclaracion);
}
```

En general, es bastante ambiguo lo que podríamos agregar sin causar un error de compilación, como por ejemplo, podríamos declarar ciertas variables que no sean tan relevantes o incluso modificar las que tenemos, mientras no sea el char end, ya que este debe ser final, pues las lambdas sólo pueden hacer referencia a variables finales y esto se puede probar, esto es lo que saldría si agregáramos un `end='y';`

```
C:\Users\Nicole\Documents\GitHub\Actividades-CC3S2-\ExamenParcial-3S2\src\main\java\ArchivosPrep
java: local variables referenced from a lambda expression must be final or effectively final
```

Lo que agregamos es una “aclaración” que evalúa si la lista está vacía o no y también se implementó una función main que muestra el funcionamiento de la función remove con una lista de prueba



```
Run: Mainremove x
"C:\Program Files\Java\jdk1.8.0_321\bin\java.exe" ...
Lista a ser eliminada segun un límite start y end
Antes: [a, b, c, d, f, g, x, y, z]
La cadena se encuentra vacía
Despues: []
Process finished with exit code 0
```

¿Qué puedes decir del siguiente código?

```
int length = 3;
for (int i = 0; i < 3; i++) {
    if (i % 2 == 0) {
        Supplier<Integer> supplier = () -> length; // A
        System.out.println(supplier.get()); // B
    } else {
        int j = i;
        Supplier<Integer> supplier = () -> j; // C
        System.out.println(supplier.get()); // D
    }
}
```

```

n: MainSupplier x
"C:\Program Files\Java\jdk1.8.0_321\bin\java.exe" ...
3
1
3
Process finished with exit code 0

```

Consideremos inicialmente que la expresión Supplier devolverá un objeto o valor sin la necesidad de tener un parámetro, en este caso devolverá un entero en sus dos usos. Ahora, fijándonos en el bucle, el i tomará los valores de 0,1,2. Entonces, las instrucciones que están dentro del if serán ejecutadas dos veces e imprimirán el valor de length mediante el método get() de

supplier que retorna a ese valor, mientras que las del else solo una vez y se imprimirá el valor de j que será igual a i. Por lo visto, en la ejecución, el programa funciona correctamente con la lógica propuesta y no hay errores de compilación. Si analizamos las expresiones lambda, tal como se mencionó en el anterior apartado, estas sólo pueden hacer referencia a variables finales y justamente es lo que se evidencia acá. Tanto length como j pueden ser consideradas como final, si surgen dudas por la variable j, esta se vuelve a declarar cada vez que se ejecuta la sentencia else, y sale fuera del ámbito, no llega a ser reasignada.

=====

Inserta código sin causar un error de compilación

```

public void remove(List<Character> chars){
    char end = 'z';
    // Insertar código
    //el siguiente código hace uso también de una expresión lambda para mostrar los
    valores de la lista
    //tampoco genera un error de compilación
    System.out.println("Código agregado antes: ");
    chars.forEach(i -> System.out.print(i+" "));
    //si inicializariamos c o start, ahí sí tendríamos un error

    chars.removeIf(c -> {
        char start = 'a'; return start <= c && c <= end; });
}

```

Tanto el código añadido antes y después, basado en este mismo método, se encuentran en la misma clase MainRemove y así como en el caso del código añadido después, en general, es bastante ambiguo lo que podríamos agregar sin causar un error de compilación, como por ejemplo, podríamos declarar ciertas variables que no sean tan relevantes o imprimir cualquier cosa que se ocurra, eso sí, no podemos inicializar las variables c o start, las expresiones lambda no pueden declarar variables locales de nuevo y, como se conoce, debe trabajar con variables finales, lo que impide reasignar la variable end. Ahora, conocemos lo que no debemos agregar, para que tenga relación con el tema, se ha añadido otro uso de las expresiones lambda para mostrar los valores que la lista tiene e imprimirlos sin generar ningún error de compilación.

=====

Observación: cada pregunta vale medio punto.

Pregunta 2 (12 puntos)

Este ejercicio tiene como objetivo, desarrollar aplicaciones seguras y flexibles mediante el desarrollo basado en pruebas (TDD): una técnica que puede aumentar considerablemente la velocidad de desarrollo y eliminar gran parte de la pesadilla de la depuración, todo con la ayuda de JUnit 5 y sus funciones.

Conceptos principales de TDD

El desarrollo basado en pruebas es una práctica de programación que utiliza un ciclo de desarrollo breve y repetitivo en el que los requisitos se convierten en casos de prueba y luego el programa se modifica para que pasen las pruebas:

- 1 Se escribe una prueba fallida antes de escribir código nuevo.
- 2 Se escribe el fragmento de código más pequeño que hará que pase la nueva prueba.

Si el ciclo de desarrollo convencional es más o menos así:

[código, prueba, (repetir)]

TDD utiliza una variación sorprendente:

[prueba, código, (repetir)]

La prueba impulsa el diseño y se convierte en el primer cliente del método. Dijimos que TDD usa este ciclo de desarrollo:

[prueba, código, (repetir)]

De hecho, se ve así: [probar, codificar, refactorizar, (repetir)] en general

La **refactorización** es el proceso de modificar un sistema de software de manera que no afecte su comportamiento externo pero sí mejore su estructura interna. Para asegurarnos de que el comportamiento externo no se vea afectado, debemos confiar en las pruebas.

Aplicación: Gestión de vuelos

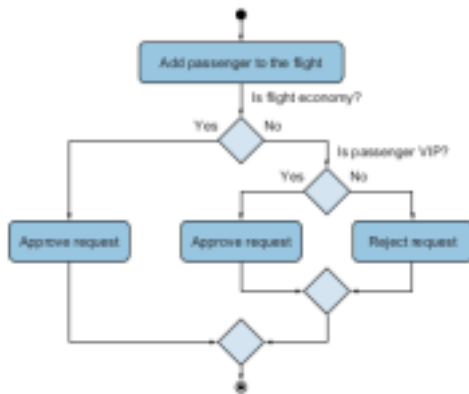
Como hemos comentado a lo largo de las actividades, Tested Data Systems (empresa de ejemplo) está desarrollando una aplicación de gestión de vuelos para uno de sus clientes. Actualmente, la aplicación puede crear y configurar vuelos, agregar pasajeros y eliminarlos de los vuelos. En esta evaluación, veremos escenarios que siguen el trabajo diario de los desarrolladores.

Comenzaremos con la aplicación que no es TDD, que se supone que debe hacer varias cosas, como seguir las políticas de la empresa para pasajeros regulares y VIP. Necesitamos comprender la aplicación y asegurarnos de que realmente está implementando las operaciones esperadas. Entonces, tenemos que cubrir el código existente con pruebas unitarias.

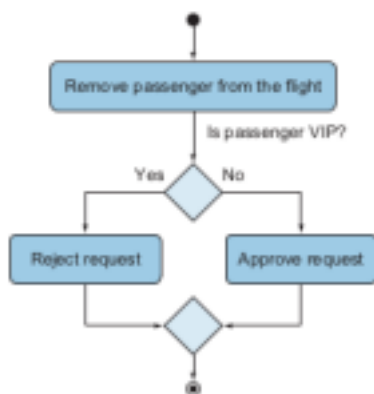
Una vez que hayamos hecho eso, abordaremos otro desafío: agregar nuevas funciones al comprender primero lo que se debe hacer; luego, escribir pruebas que fallan; y luego, escribir el código que corrige las pruebas.

Caso : John se suma al desarrollo de la aplicación de gestión de vuelos, que es una aplicación Java creada con

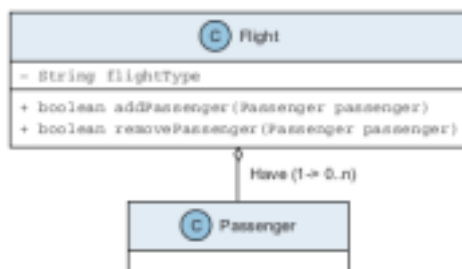
la ayuda de Maven. El software debe mantener una política con respecto a agregar pasajeros y eliminarlos de los vuelos. Los vuelos pueden ser de diferentes tipos: actualmente, hay vuelos económicos y de negocios, pero es posible que se agreguen otros tipos más adelante, según los requisitos del cliente. 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.



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. Como podemos ver en estos dos diagramas de actividades, la lógica empresarial inicial se centra en la toma de decisiones.



Veamos el diseño inicial de esta aplicación.



El diseño 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. La

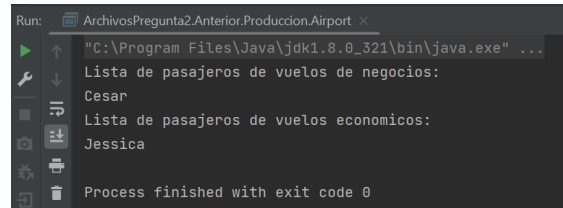
carpeta **Anterior** de la evaluación muestra la clase `Passenger`, la clase `Flight`.

La aplicación aún no tiene pruebas. En cambio, los desarrolladores iniciales escribieron un código en el que simplemente siguieron la ejecución y la compararon con sus expectativas. Por ejemplo, existe una

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.

Pregunta 1 (0.5 puntos): Ejecuta el programa y presenta los resultados y explica qué sucede.

Considere que los códigos de cada clase, según el avance de las Fases de producción han sido comentados explicando un poco su funcionamiento.



```
Run: ArchivosPregunta2.Anterior.Produccion.Airport x
"C:\Program Files\Java\jdk1.8.0_321\bin\java.exe" ...
Lista de pasajeros de vuelos de negocios:
Cesar
Lista de pasajeros de vuelos economicos:
Jessica
Process finished with exit code 0
```

En la clase **Airport**, tras la creación de dos vuelos - uno de negocio y otro económico - y dos pasajeros - Cesar vip y Jessica regular - se usan los métodos que existen en la clase **Flight** para añadir y retirar pasajeros.

Primero, se añade a Cesar en el vuelo de negocios y es aceptado por ser vip, luego se trata de retirar a Cesar del vuelo de negocio, pero esto no es permitido por la política de eliminación.

También, se trata de añadir a Jessica a un vuelo de negocio, pero tampoco se le permite por cliente ser regular y , finalmente, se añade a Jessica a un vuelo económico

Lo que se logra ver en la ejecución son las listas de pasajeros registrados en cada vuelo.

=====

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

Preparación de la aplicación de gestión de vuelos para TDD

Para trasladar la aplicación de gestión de vuelos a TDD, John primero debe cubrir la lógica comercial existente con pruebas JUnit 5. Agrega las dependencias de JUnit 5 con las que ya estamos familiarizados (junit-jupiter-api y junit-jupiter-engine) al archivo Maven pom.xml.

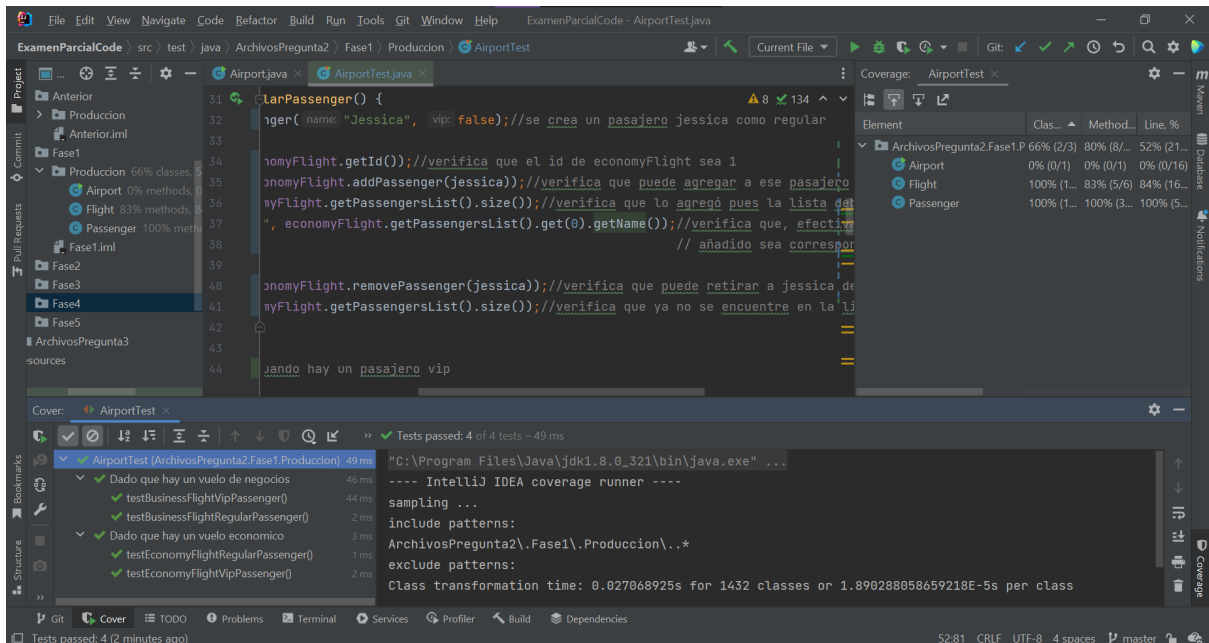
Al inspeccionar la lógica comercial de las figuras anteriores, John comprende que tiene que verificar los escenarios de agregar/eliminar pasajeros proporcionando pruebas para dos tipos de vuelo y dos tipos de pasajeros. Entonces, multiplicando dos tipos de vuelo por dos tipos de pasajeros, esto significa cuatro pruebas en total. Para cada una de las pruebas, tiene que verificar las posibles operaciones de añadir y quitar.

John sigue la lógica comercial para un vuelo económico y utiliza la capacidad de prueba anidada de JUnit 5, ya que las pruebas comparten similitudes entre ellas y se pueden agrupar: pruebas para vuelos económicos y pruebas para vuelos comerciales.

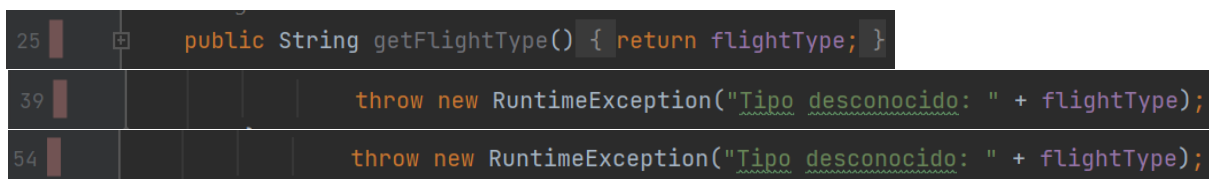
John sigue la lógica comercial para un vuelo comercial y la traduce en el código de la carpeta **Pruebas** de la **Fase 1**.

Pregunta 2 (1 punto) Si ejecutamos las pruebas con cobertura desde IntelliJ IDEA, ¿cuales son los resultados que se muestran?, ¿Por qué crees que la cobertura del código no es del 100%?. Puedes revisar <https://www.jetbrains.com/help/idea/code-coverage.html> y https://en.wikipedia.org/wiki/Code_coverage para responder la pregunta.

Consideremos que las pruebas puestas en cada fase han sido cambiadas para que se encuentren en src\test\java, así podemos tener lo siguiente al correr el test con cobertura:



Por lo visto, todas las pruebas han sido aceptadas. Si nos fijamos en el cuadro de cobertura, vemos que la clase Passenger si ha sido cubierta al 100%, y que la clase Flight tiene una cobertura incompleta. esto por lo siguiente:



No se ha probado los casos para cuando se ingresa un tipo de vuelo inválido - o sea, diferente de Economico o Negocios - y tampoco se hace uso del método que retorna al tipo de vuelo.

También vemos que la clase Airport no ha sido cubierta - esto por el porcentaje que indica 0% - y exactamente, no la hemos probado, pues tiene el método main que sirve como para mostrar un funcionamiento prueba de la lógica que tienen Passenger y Flight unidos .

Según la fuente recomendada, recordemos que para que exista una cobertura al 100% se tienen que cubrir todos los criterios principales de cobertura y este no ha sido el caso.

Observación: Si en la Fase 2 ha sido eliminado Airport, es porque los test pueden probar lo mismo que hacía Airport prácticamente.

Pregunta 3 (0.5 punto)¿ Por qué John tiene la necesidad de refactorizar la aplicación?.

Justo como se mencionó en la pregunta anterior, en la clase Flight no ha sido ejecutado el método getFlightType y tampoco el caso que resulta de default que lanza una excepción.

Por la estructura que poseen los métodos add y remove Passenger de Flight vemos que se depende de la variable fligthType al momento de tomar las decisiones, de modo que, cada vez que queramos añadir un nuevo tipo de vuelo, la clase va a tener que presentar cambios continuos que se darán en cada decisión condicional.

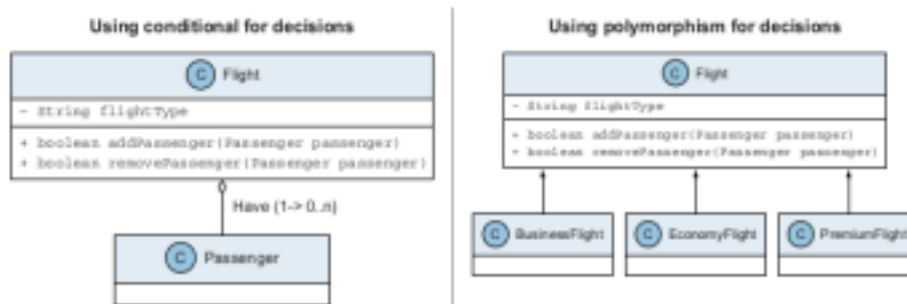
Jhon tiene que eliminar a los elementos que no se utilizan y evitar este tipo de dependencia que no permitirá que el código sea extensible y fácil de cambiar, por ello surge la necesidad de refactorizar.

Justamente lo que vemos para la Fase 2 es una implementación que se basa en el polimorfismo y en el principio de SOLID open/close.

Refactorización de la aplicación de gestión de vuelos

Para la refactorización es necesario 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.

El principio en acción aquí se llama principio abierto/cerrado. En la práctica, significa que el diseño que se muestra a la izquierda requerirá cambios en la clase existente cada vez que agreguemos un nuevo tipo de vuelo. Estos cambios pueden reflejarse en cada decisión condicional que se tome en función del tipo de vuelo. Además, nos vemos obligados a confiar en el campo `flightType` e introducir casos predeterminados no ejecutados.



Con el diseño de la derecha, que se refactoriza reemplazando el condicional con polimorfismo, no necesitamos una evaluación `flightType` o un valor predeterminado en las instrucciones de cambio del listado `Flight` anterior.

John, por supuesto, se preguntará: "¿Cómo puedo estar seguro de que estoy haciendo lo correcto y no afectar la funcionalidad que ya funciona?" La respuesta es que pasar las pruebas brinda la seguridad de que la funcionalidad existente no se modifica.

Pregunta 4 (0.5 puntos): Revisa la **Fase 2** de la evaluación y realiza la ejecución del programa y analiza los resultados.

Al ejecutar el test de la Fase 2 con cobertura, tenemos lo siguiente:

The screenshot shows the IntelliJ IDEA IDE with the `ExamenParcialCode` project open. The `src/test/java/ArchivosPregunta2/Fase2/Produccion/AirportTest.java` file is selected. The `BusinessFlightTest` class is visible in the editor, showing a `setUp()` method that creates a `BusinessFlight` instance.

The **Coverage** window on the right shows the following data:

Element	Class, %	Method, %	Line, %
ArchivosPregunta2.Fase2.Produccion	100% (4/4)	100% (12/12)	100% (20/20)
Flight	100% (1/1)	100% (3/3)	100% (5/5)
Passenger	100% (1/1)	100% (3/3)	100% (5/5)
EconomyFlight	100% (1/1)	100% (3/3)	100% (5/5)
BusinessFlight	100% (1/1)	100% (3/3)	100% (5/5)

The **Test Results** window at the bottom shows the following test results:

- ✓ `AirportTest (ArchivosPregunta2.Fase2.Produccion)` 24 ms
- ✓ `Dado que hay un vuelo negocios` 23 ms
- ✓ `Dado que hay un vuelo economico` 1 ms
- ✓ `testEconomyFlightRegularPassenger()` 1 ms
- ✓ `testEconomyFlightVipPassenger()` 1 ms

The status bar at the bottom indicates "Tests passed: 4 (moments ago)".

Por lo visto, al igual que el caso anterior, todas las pruebas han sido aceptadas. Pero, si nos fijamos en el cuadro de cobertura, vemos que todas las clases han sido cubiertas al 100%, lo cual sucede gracias a la refactorización realizada que nos permite tener una mejor calidad de código. Ahora tenemos una mejor mantenibilidad y podremos agregar más tipos de vuelo sin modificar nuestras clases ya creadas.

=====

La refactorización se logrará manteniendo la clase Flight base de la Fase 3 y, para cada tipo condicional, agregando una clase separada para extender Flight. John cambiará addPassenger y removePassenger a métodos abstractos y delegará su implementación a subclasses. El campo flightType ya no es significativo y se eliminará.

Los archivos de este análisis se encuentran en la **Fase 3**.

John presenta una clase EconomyFlight que amplía Flight e implementa los métodos abstractos heredados addPassenger y removePassenger.

```
public class EconomyFlight extends Flight {  
...  
}
```

También presenta una clase BusinessFlight que amplía Flight e implementa los métodos abstractos heredados addPassenger y removePassenger.

```
public class BusinessFlight extends Flight {  
...  
}
```

Pregunta 5 (3 puntos) La refacción y los cambios de la API se propagan a las pruebas. Reescribe el archivo Airport Test de la carpeta **Fase 3**.

```
public class AirportTest {  
...  
}
```

Y responde las siguientes preguntas:

- ¿Cuál es la cobertura del código ?
- ¿ La refactorización de la aplicación TDD ayudó tanto a mejorar la calidad del código?.

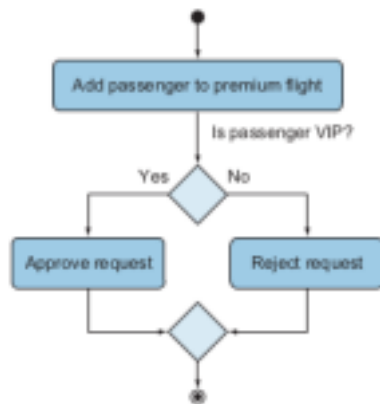
Lo correspondiente a la Fase 3 ha sido implementado en la Fase 2 y tal como se mencionó en la anterior respuesta, se presenta una cobertura del 100%, lo cual sucede gracias a la refactorización realizada que nos permite tener una mejor calidad de código. Ahora tenemos una mejor mantenibilidad y podremos agregar más tipos de vuelo sin modificar nuestras clases ya creadas. Efectivamente, la refactorización ha ayudado.

=====

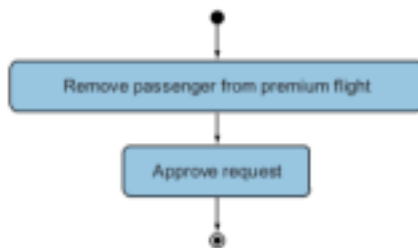
Nuevas características usando TDD

Después de mover el software a TDD y refactorizarlo, John es responsable de la implementación de nuevas funciones requeridas por el cliente que amplían las políticas de la aplicación.

Las primeras características nuevas que John implementará 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.



Agreguemos un nuevo tipo y llamémoslo PremiumFlight, simplemente extendiendo la clase base y definiendo su comportamiento. **De acuerdo con el principio abierto/cerrado, la jerarquía estará abierta para extensiones** (podemos agregar fácilmente nuevas clases) pero cerrada para modificaciones (las clases existentes, comenzando con la clase base Flight, no se modificarán).

John se da cuenta de que esta nueva característica tiene similitudes con las anteriores. Le gustaría aprovechar más el estilo de trabajo TDD y hacer más refactorización, esta vez, para las pruebas. Esto está en el espíritu de la Regla de Tres, como lo establece Don Roberts):

[https://en.wikipedia.org/wiki/Rule_of_three_\(computer_programming\)](https://en.wikipedia.org/wiki/Rule_of_three_(computer_programming))

Pregunta 6 (0.5 puntos):¿En qué consiste esta regla relacionada a la refactorización?. Evita utilizar y copiar respuestas de internet. Explica cómo se relaciona al problema dado en la evaluación.

John 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.

John considera que, tras recibir el requerimiento para la implementación de este tercer tipo de vuelo, es el momento de agrupar un poco más las pruebas existentes utilizando la anotación JUnit 5 @Nested y luego implementar el requisito de vuelo premium de forma similar.

La regla de tres consiste básicamente en "Three strikes and you refactor", tal como menciona en la fuente. Esto puede ser entendido como: la primera vez al implementar algún funcionamiento se puede decir que se encuentra "adecuado", la segunda vez que hagamos algo similar a ese funcionamiento tendremos una mala

práctica por generar una duplicación, pero aún podemos mantenerlo para evitar fallos al arriesgarnos a seleccionar una abstracción incorrecta de forma prematura, pero al momento de tener que implementar alguna funcionalidad similar a las otras dos anteriores es necesario refactorizar, por ello el ¡tres strikes y refactorizamos!

Ahora, en este caso, el tercer strike se presenta gracias a que se introducen nuevos requisitos- la generación del vuelo Premium Flight- entonces lo que hemos implementado tanto para Business y Economy Flight tiene que ser refactorizado, para evitar la duplicación, a través de un diseño apropiado ya que hemos encontrado nuestro tercer “strike”.

=====

En la carpeta **Fase 4** se muestra la clase AirportTest refactorizada antes de pasar al trabajo para el vuelo premium.

Así es como John ha refactorizado las pruebas existentes, para facilitar continuar trabajando en estilo TDD e introducir la nueva lógica comercial de vuelo premium requerida. Si ejecutamos las pruebas ahora, podemos seguir fácilmente la forma en que funcionan y cómo verifican la lógica comercial.

John pasa ahora a la implementación de la clase PremiumFlight y su lógica.

El creará PremiumFlight como una subclase de Flight y override los métodos addPassenger y removePassenger, que 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.

Pregunta 7 (1 punto): Escribe el diseño inicial de la clase llamada PremiumFlight y agrega a la **Fase 4** en la carpeta producción.

Se puede presenciar el diseño inicial propuesto por Jhon en el código de producción de la Fase 4. Este cumple con lo pedido ya que solo devuelven false, recordar que solo es un diseño inicial para centrarnos en las pruebas.

```
public class PremiumFlight extends Flight { //ahora PremiumFlight extiende a Flight
    //crea al constructor que llama al constructor de la superclase
    new *
    public PremiumFlight(String id) { super(id); }

    // Diseño inicial de la clase PremiumFlight. Pregunta 7
    DagmarLV
    @Override
    public boolean addPassenger(Passenger passenger) {
        return false;
    }
    new *
    @Override
    public boolean removePassenger(Passenger passenger) {
        return false;
    }
}
```

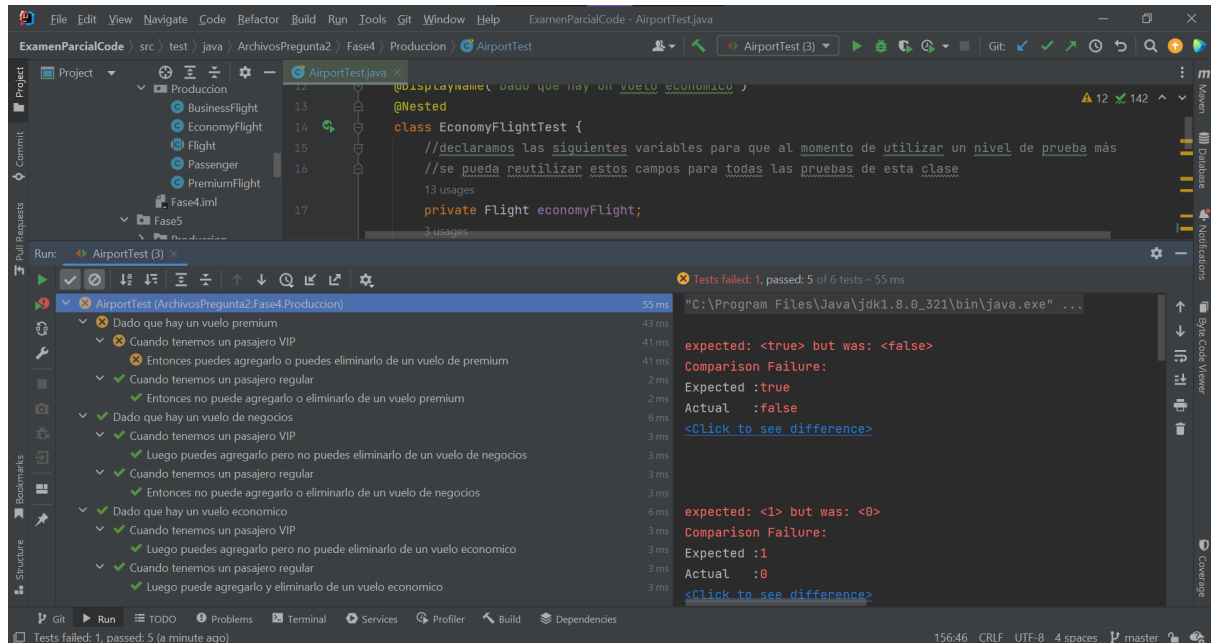
=====

Pregunta 8 (2 puntos): Ayuda a John e implementa las pruebas de acuerdo con la lógica comercial de

vuelos premium de las figuras anteriores. Adjunta tu código en la parte que se indica en el código de la **Fase 4**. Después de escribir las pruebas, John las ejecuta.

Se puede presenciar la prueba de acuerdo con la lógica anterior propuesta por Jhon en el código de pruebas de la Fase 4.

Justamente al ejecutar las pruebas, presentamos error ya que el código de producción todavía no se encuentra completo con la lógica necesaria, pues nos estamos centrando en las pruebas.



Recuerda, él está trabajando al estilo TDD, por lo que las pruebas son lo primero.

El hecho de que una de las pruebas esté fallando no es un problema (debes conseguir eso). 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.

John entiende que solo tiene que concentrarse en el pasajero VIP. Citando nuevamente a Kent Beck, "TDD te ayuda a prestar atención a los problemas correctos en el momento correcto para que puedas hacer que sus diseños sean más limpios, puedes refinar tus diseños a medida que aprendes. TDD te permite ganar confianza en el código con el tiempo".

Entonces, John regresa a la clase PremiumFlight y agrega la lógica comercial solo para pasajeros VIP.

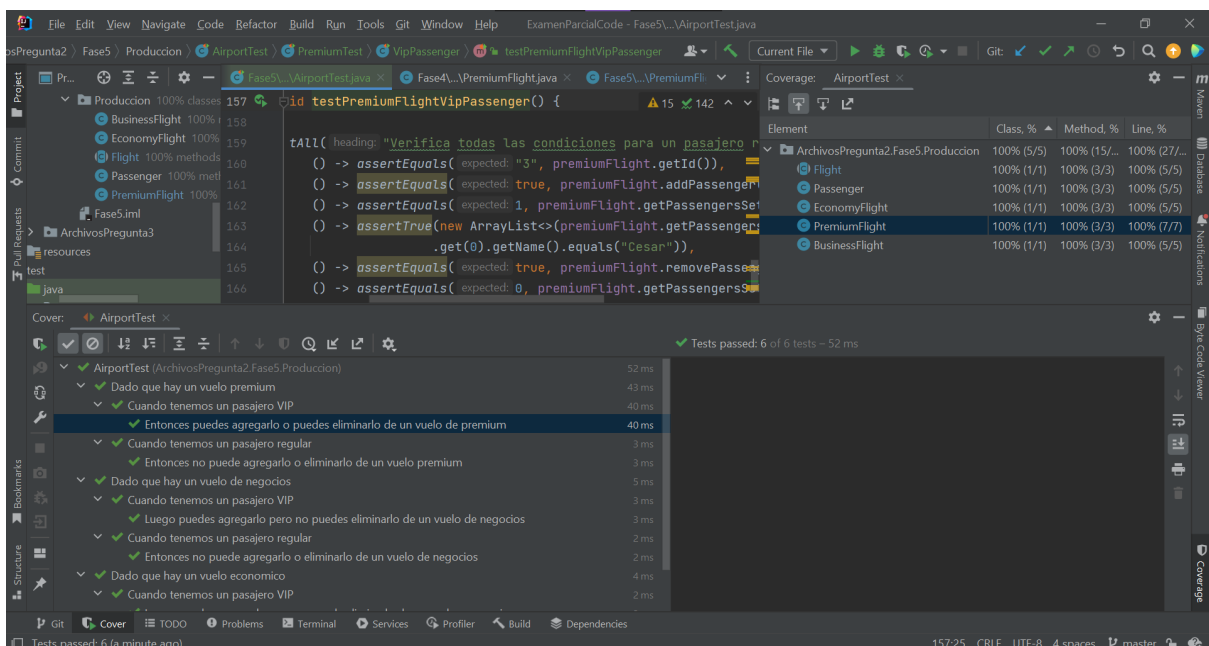
Pregunta 9 (2 puntos): Agrega la lógica comercial solo para pasajeros VIP en la clase PremiumFlight. Guarda ese archivo en la carpeta Producción de la **Fase 5**. Ejecuta el código anterior copiando la respuesta de la pregunta 8 en la carpeta de Pruebas de la Fase.

Para implementar el código de producción se sigue la lógica propuesta, por lo que para añadir o retirar un pasajero se evalúa un condicional que verifica si es que es vip y si lo es, procede con la acción requerida. Al momento de ejecutar las pruebas podemos presenciar que todas las pruebas han pasado correctamente y además con el 100% de cobertura.

```

1 package ArchivosPregunta2.Fase5.Produccion;
2
3 DagmarLV *
4 public class PremiumFlight extends Flight {
5     1 usage DagmarLV
6     public PremiumFlight(String id) { super(id); }
7
8     // Diseño de la lógica comercial para los pasajeros VIP.Pregunta 9
9     DagmarLV *
10    @Override
11    public boolean addPassenger(Passenger passenger) {
12        if(passenger.isVip()){solo si es vip se podrá agregar
13            return passengers.add(passenger);
14        }
15        return false;//en caso contrario no
16    }
17    DagmarLV *
18    @Override
19    public boolean removePassenger(Passenger passenger) {
20        if(passenger.isVip()){solo si es vip se podrá eliminar
21            return passengers.remove(passenger);
22        }
23        return false;//en caso contrario no
24    }
25 }

```



=====

5. Agregar un pasajero solo una vez

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. John necesita asegurarse 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 John la implementará al estilo TDD.

John comenzará la implementación de esta nueva función agregando la prueba para verificarla. Intentará repetidamente agregar el mismo pasajero a un vuelo, como se muestra en la siguiente lista. Detallaremos solo el caso de un pasajero regular agregado repetidamente a un vuelo económico, todos los demás casos son similares.

Sugerencia: Revisa el archivo de prueba de la Fase 5 AirportTest dada en la evaluación.

Si hacemos las pruebas, fallan. Todavía no existe una lógica comercial para evitar agregar un pasajero más de una vez

Para garantizar la unicidad de los pasajeros en un vuelo, John 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. La clase de Flight cambia como se muestra a continuación.

```
public abstract class Flight {  
    [...]  
    Set<Passenger> passengers = new HashSet<>();  
    [...]  
    public Set<Passenger> getPassengersSet() {  
        return Collections.unmodifiableSet(passengers);  
    }  
    [...]  
}
```

Revisa la clase Flight de la carpeta producción de la Fase 5.

Pregunta 10 (1 punto) Ayuda a John a crear una nueva prueba para verificar que un pasajero solo se puede agregar una vez a un vuelo de manera que John ha implementado esta nueva característica en estilo TDD.

Consideremos que la clase Flight ha sido refactorizada, ahora, para verificar que un solo pasajero solo pueda ser agregado una vez podemos hacer uso del @RepeatedTest() para que pruebe el añadir al pasajero regular - jessica - dentro de un bucle y posterior a ello compruebe que sólo se encuentra un elemento en el vuelo y que este sea correspondiente a Jessica.

The screenshot shows the IntelliJ IDEA IDE with the AirportTest.java file open. The code in the file is as follows:

```
55 repetition(); i++) {  
56  
57  
58  
59  
60 agregar solamente una vez un pasajero regular a un vuelo  
61 flight.getId(),//verificamos que es el vuelo economico  
62 flight.getPassengersSet().size(),//vemos que el tamaño sea  
63 rgersSet().contains(jessica),//que el set contenga a j  
64 /Flight.getPassengersSet() //que el nombre corresponda a  
65  
66  
67
```

The test results at the bottom show that all tests passed:

- Tests passed: 9 of 9 tests - 56 ms
- Tests passed: 9 (moments ago)

The coverage report on the right shows the following data:

Element	Class...	Method, %	Line, %
ArchivosPregunta2.Fase5.Prod	100% (5/5)	100% (15...	100% (27...
Flight	100% (1/1)	100% (3/3)	100% (5/5)
Passenger	100% (1/1)	100% (3/3)	100% (5/5)
EconomyFlight	100% (1/1)	100% (3/3)	100% (5/5)
PremiumFlight	100% (1/1)	100% (3/3)	100% (7/7)
BusinessFlight	100% (1/1)	100% (3/3)	100% (5/5)

=====

Pregunta 3 (5 puntos)

Un buen código de prueba se caracteriza por lo siguiente de acuerdo a la literatura relacionada:

- Las pruebas deben ser rápidas
- Las pruebas deben ser cohesivas, independientes y aisladas
- Las pruebas deben tener una razón de existir
- Las pruebas deben ser repetibles
- Las pruebas deben tener aseveraciones sólidas
- Las pruebas deben romperse si el comportamiento cambia
- Las pruebas deben tener una sola y clara razón para fallar
- Las pruebas deben ser fáciles de escribir
- Las pruebas deben ser fáciles de leer

Hay dos prácticas cuando realizas pruebas adicionales que debes seguir para que las pruebas sean legibles: asegurarse de que toda la información (especialmente las entradas y las afirmaciones) sea lo suficientemente clara y el uso generadores de datos de prueba cada vez que construyes estructuras de datos complejas.

Ilustremos estas dos ideas con un ejemplo. La siguiente lista muestra una clase Invoice.

```
public class Invoice {  
  
    private final double value;  
    private final String country;  
    private final CustomerType customerType;  
    public Invoice(double value, String country, CustomerType customerType) {  
        this.value = value;  
        this.country = country;  
        this.customerType = customerType;  
    }  
    public double calculate() {  
        double ratio = 0.1;  
        return value * ratio;  
    }  
}
```

El código de prueba no es muy claro para el método de **calculate()** y podría parecerse a la siguiente lista.

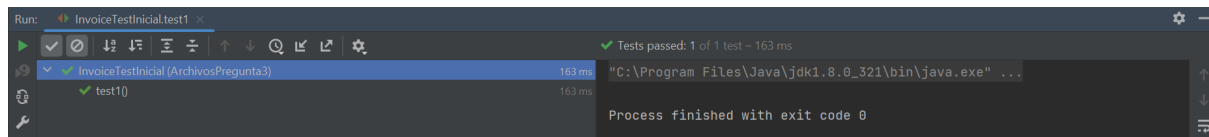
```
@Test  
void test1() {  
    Invoice invoice = new Invoice(new BigDecimal("2500"), "NL",  
        CustomerType.COMPANY);  
    double v = invoice.calculate();  
    assertThat(v).isEqualTo(250);  
}
```

Pregunta 1 (0.5 puntos) : ¿Cuáles son los problemas de este código de prueba?.

Para empezar al momento de querer ejecutar surge un error:

```
C:\Users\Nicole\Documents\GitHub\Actividades-CC3S2-\ExamenParcial-3S2\ExamenPar  
java: incompatible types: java.math.BigDecimal cannot be converted to double
```

Esto puede ser arreglado mediante `new BigDecimal("2500").doubleValue()` y al momento de correr la prueba esta pasa correctamente



Pero ahora centrémonos en la estructura de la prueba.

Empezando por el nombre, este no explica lo que está probando la prueba y el Double tampoco tiene un nombre adecuado porque no explica el significado que conlleva.

Además, si nos fijamos en la inicialización del objeto, no resulta claro el comportamiento de todos los parámetros que contiene, lo que puede generar confusión al momento de querer comprender el funcionamiento de la prueba que va de acuerdo con el comportamiento de sus variables.

=====

Pregunta 2 (1 punto) : Escribe una versión más legible de este código prueba. Recuerda llamarlo InvoiceTest.java

Siguiendo la lógica propuesta en la parte inferior, podemos armar un código de prueba de la forma:

```
@Test
@DisplayName("Dada una Compañía para obtener su facturación")
void invoiceForCompanies() {
    Invoice invoice = new InvoiceBuilder()
        .asCompany()
        .withCountry("NL")
        .withAValueOf(2500)
        .build();
    double calculatedValue = invoice.calculate();

    assertThat(calculatedValue).isEqualTo(250); // 2500 * 0.1 = 250
}
```

Ahora el código muestra una mejor claridad para su entendimiento, porque ahora se evidencia que está sirviendo para hallar la facturación de una compañía y sus elementos se encuentran bastante claros en cuanto a su significado, ahora entendemos que se construye la factura para una compañía con el país "NL" y con un valor de 2500 y el double calculatedValue explica su comportamiento e incluso la aserción contiene un comentario que llega a explicar de dónde viene el 250.

=====

Para sistemas empresariales, donde tenemos métodos similares a los de los negocios, como calcular los impuestos o calcular el precio final, cada método de prueba (o partición) cubre una regla de negocios diferente. Esos pueden expresarse en el nombre de ese método de prueba.

El uso de InvoiceBuilder.java expresa claramente de qué se trata esta factura: es una factura para una empresa (como lo establece claramente el método asCompany()), "NL" es el país de esa factura, y la factura tiene un valor de 2500. El resultado del comportamiento va a una variable cuyo nombre lo dice todo (calculatedValue). La aserción contiene un comentario que explica de dónde viene el 250.

```
public class InvoiceBuilder {
    private String country = "NL";
    private CustomerType customerType = CustomerType.PERSON;
    private double value = 500;
    public InvoiceBuilder withCountry(String country) {
        this.country = country;
        return this;
    }
}
```

```

public InvoiceBuilder asCompany() {
    this.customerType = CustomerType.COMPANY;
    return this;
}
public InvoiceBuilder withAValueOf(double value) {
    this.value = value;
    return this;
}
public Invoice build() {
    return new Invoice(value, country, customerType);
}
}

```

InvoiceBuilder es un ejemplo de implementación de un generador de datos de prueba. El constructor nos ayuda a crear escenarios de prueba proporcionando una API clara y expresiva. El uso de interfaces fluidas (como `asCompany().withAValueOf(...)`) también es una opción de implementación común. En cuanto a su implementación, InvoiceBuilder es una clase de Java.

Pregunta 3 (1 punto) : Implementa InvoiceBuilder.java. Siéntete libre de personalizar sus constructores. Un truco común es hacer que el constructor construya una versión común de la clase sin requerir la llamada a todos los métodos de configuración.

Siguiendo lo presentado en la sección anterior, tenemos a la clase InvoiceBuilder que contiene a los atributos `country`, `costumertype` y `value`, y los tiene inicialmente predeterminados con el país "NL", `CustomerType.PERSON` y un `value` de 500 y tenemos a sus constructores para cambiar los valores de aquellos atributos como sea necesario.

```

3      public class InvoiceBuilder {
4          private String country = "NL";
5          private CustomerType customerType = CustomerType.PERSON;
6          private double value = 500;
7          public InvoiceBuilder withCountry(String country) {
8              this.country = country;
9              return this;
10         }
11         public InvoiceBuilder asCompany() {
12             this.customerType = CustomerType.COMPANY;
13             return this;
14         }
15         public InvoiceBuilder withAValueOf(double value) {
16             this.value = value;
17             return this;
18         }
19         public Invoice build() {
20             return new Invoice(value, country, customerType);

```

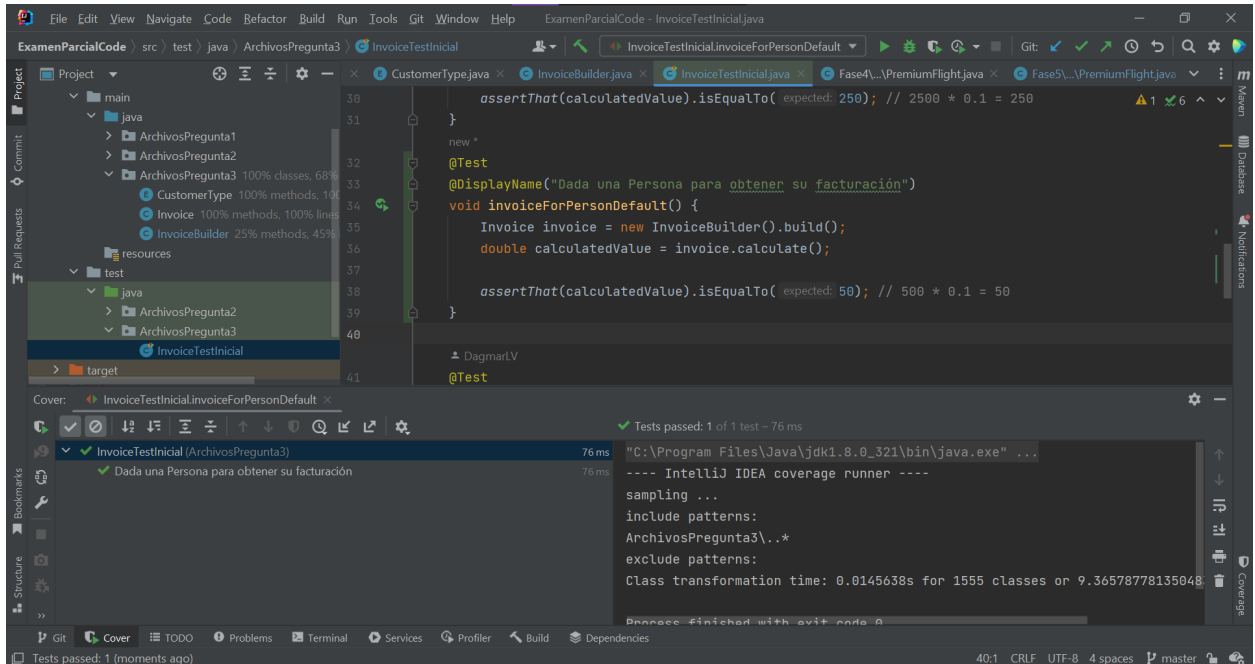
Pregunta 4 (0.5 puntos) : Escribe en una línea una factura compleja. Muestra los resultados

En una línea se podría inicializar lo siguiente: `Invoice invoice = new InvoiceBuilder().build();`

Y se genera una factura con los datos preestablecidos en la clase de InvoiceBuilder.

Para probarlo podemos crear un Test `invoiceForPersonDefault` de la forma que sigue:

```
private String country = "NL";
2 usages
private CustomerType customerType = CustomerType.PERSON;
2 usages
private double value = 500;
```



Otros desarrolladores pueden escribir métodos abreviados que construyan otros accesorios comunes para la clase. En el listado siguiente, el método `anyCompany()` devuelve una factura que pertenece a una empresa (y el valor predeterminado para los demás campos). El método `fromTheUS()` genera una factura para alguien en los EE. UU.

```
public Invoice anyCompany() {
    return new Invoice(value, country, CustomerType.COMPANY);
}

public Invoice fromTheUS() {
    return new Invoice(value, "US", customerType);
}
```

Pregunta 5 (1 punto) : Agrega este listado en el código anterior y muestra los resultados

Para probarlo podemos crear dos nuevos test que utilicen los métodos agregados y serían de la forma:

```

40      @Test
41      void invoiceForAnyCompany() {
42          Invoice invoice = new InvoiceBuilder().anyCompany();
43          double calculatedValue = invoice.calculate();
44
45          assertThat(calculatedValue).isEqualTo( expected: 50); // 500 * 0.1 = 50
46      }
47
48      new *
49      @Test
50      void invoiceForFromTheUS() {
51          Invoice invoice = new InvoiceBuilder().fromTheUS();
52          double calculatedValue = invoice.calculate();
53
54          assertThat(calculatedValue).isEqualTo( expected: 50); // 500 * 0.1 = 50

```

En el listado siguiente usamos las variables `invoiceValue` y `tax` en la aserción para mejorar la explicación de tus resultados.

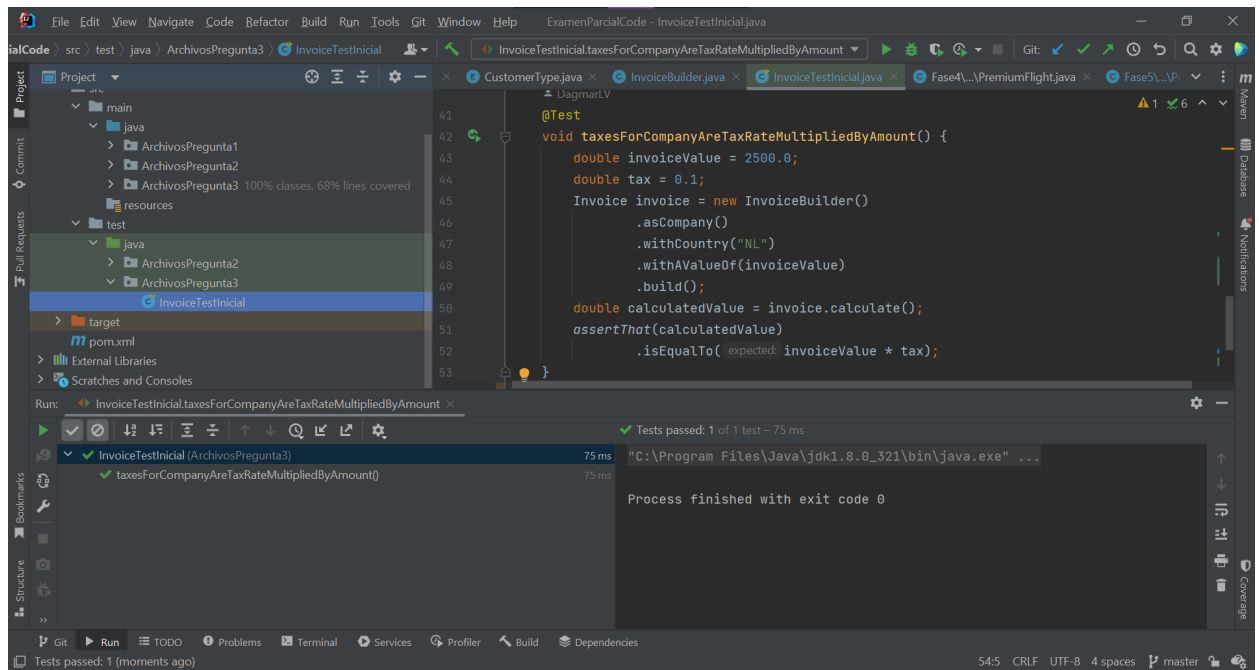
```

@Test
void taxesForCompanyAreTaxRateMultipliedByAmount() {
    double invoiceValue = 2500.0;
    double tax = 0.1;
    Invoice invoice = new InvoiceBuilder()
        .asCompany()
        .withCountry("NL")
        .withAValueOf(invoiceValue)
        .build();
    double calculatedValue = invoice.calculate();
    assertThat(calculatedValue)
        .isEqualTo(invoiceValue * tax);
}

```

Pregunta 6 (1 punto): Agrega este listado en el código anterior y muestra los resultados .

En esta nueva versión para probar la facturación de compañías, se declara la variable `invoiceValue` y la variable `tax` para que sean utilizados por el `assert` de forma explicativa y así evitar utilizar valores introducidos que no poseen una explicación concreta.



Observación: en esta pregunta debes responder todas las preguntas para que se puntúe. No se admiten respuestas incompletas.