

# 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: }}
```

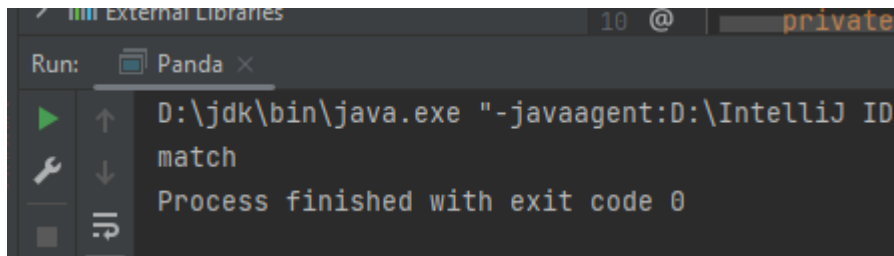
```

package org.example;
import java.util.function.*;
4 usages
public class Panda {
    2 usages
    int age; //edad
    public static void main(String[] args) {
        Panda p1 = new Panda(); //instancia p1
        p1.age = 1; //atributo age=1
        check(p1, p -> p.age < 5); //se crea un expresion lambda la cual comprueba si age es menor a 5
    }
    1 usage
    private static void check(Panda panda,
                              Predicate<Panda> pred) {

        String result =
            pred.test(panda)?"match":"not match";//se hace uso del interfaz predicate el cual contiene un metodo test()
                                                    //retorna true si la entrada coincide con pred sno retorna false
        System.out.print(result); //imprime el resultado
    }
}

```

En este caso el resultado es MATCH ya que age es menor que 5



¿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: }

```

```

1 package org.example;
2
3 interface Climb {
4     boolean isTooHigh(int height, int limit);
5 }
6
7 public class Climber{
8     public static void main(String[] args){
9         check((h,m)->h.append(m).isEmpty(), height: 5); //existe error, ya que check tiene como entrada(climb,int)
10                                                         //ya que no se especifica el tipo de h
11     }
12     private static void check(Climb climb, int height){
13         if(climb.isTooHigh(height, limit: 10)){
14             System.out.println("too high");
15         }
16         else{
17             System.out.println("ok");
18         }
19     }
20 }

```

Ocurre un error de compilación en la línea 9 ya que no se especifica el tipo de la variable h

¿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";
    }
}

```

puede ser reemplazada por lo sgte:

```
Secret a=(e)->{return "poof";};
```

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
}

```

```

package org.example;
import java.util.ArrayList;
import java.util.List;
2 usages
public class remove {
    1 usage
    public void remove(List<Character> chars){
        char end='z';
        chars.removeIf(c->{
            char start = 'a';

            return start <= c && c<=end;
        });
        System.out.print("fin:");
        if(chars.isEmpty()) System.out.println("vacio");
    }
    public static void main(String[] args){
        List<Character> chars= new ArrayList<>();
        chars.add('k');
        chars.add('r');
        chars.add('i');
        chars.add('t');
        chars.add('z');
        chars.add('a');
        chars.add('n');
        System.out.println("inicio: "+chars);
        remove r=new remove();
        r.remove(chars);
    }
}

```

Se agregó un condicional que asegura que todos los elementos fueron removidos, también se agregó un main para poder aplicar un ejemplo y corroborar que lo agregado no causa ningún error de compilación.

```

Run: remove ×
D:\jdk\bin\java.exe "-javaagent:D:\IntelliJ IDEA Comm
inicio: [k, r, i, t, z, a, n]
fin:vacio
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
    }
}
}

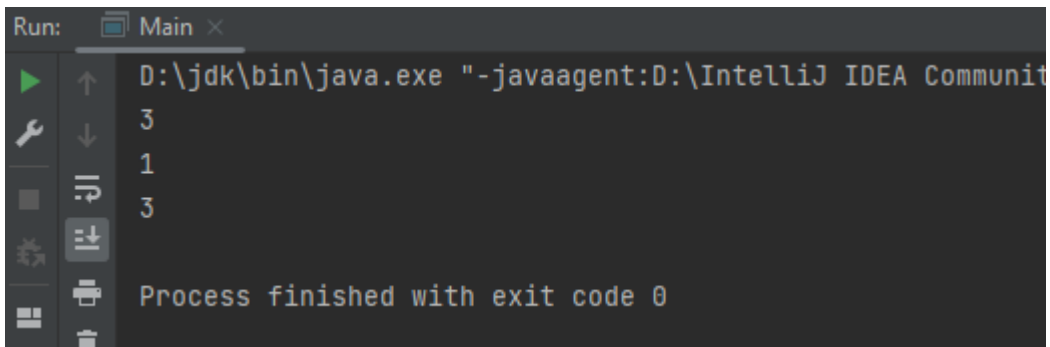
```

```

package org.example;
import java.util.function.Supplier;
public class Main{
    public void main(String[] args){
        int length=3;
        for(int i=0;i<3;i++){
            if(i%2==0){
                Supplier<Integer> supplier=()->length; //escribe sobre el metodo get que retorne length
                System.out.println(supplier.get()); //imprime length
            }else{
                int j=i;
                Supplier<Integer>supplier=()->j; //escribe sobre el metodo get que retorne j
                System.out.println((supplier.get())); //imprime j
            }
        }
    }
}

```

No hay ningun error de compilación. Primero se declara una variable de tipo entero lenght el cual sera retornado por la expresión lambda Supplier <Integer> supplier=()->length. Lo cual no causa ningun tipo de error de compilacion ya que lenght es una variable final. Al igual en la parte de else se declara una variable de tipo entero j que redeclara a la variable i por ende pasa a ser una variable final y así que no produzca ningun error de compilacion en la expresion lambda Supplier <Integer> supplier=()->j.



```

Run: Main x
D:\jdk\bin\java.exe "-javaagent:D:\IntelliJ IDEA Communit
3
1
3
Process finished with exit code 0

```

### Inserta código sin causar un error de compilación

```

public void remove(List<Character> chars) {
    char end = 'z';
    // Insertar código
}

```

```

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

```

```

package org.example;
import java.util.ArrayList;
import java.util.List;
2 usages
public class remove {
    1 usage
    public void remove(List<Character> chars){
        char end='z';
        chars.removeIf(c->{
            char start = 'a';

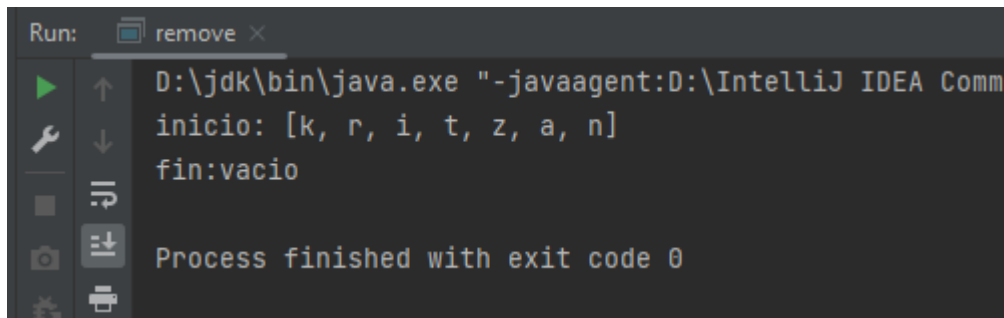
            return start <= c && c<=end;

        });
        System.out.print("fin:");
        if(chars.isEmpty()) System.out.println("vacio");
    }

    public static void main(String[] args){
        List<Character> chars= new ArrayList<>();
        chars.add('k');
        chars.add('r');
        chars.add('i');
        chars.add('t');
        chars.add('z');
        chars.add('a');
        chars.add('n');
        System.out.println("inicio: "+chars);
        remove r=new remove();
        r.remove(chars);
    }
}

```

Se agregó un condicional que asegura que todos los elementos fueron removidos, también se agregó un main para poder aplicar un ejemplo y corroborar que lo agregado no causa ningún error de compilación.



```
Run: remove x
D:\jdk\bin\java.exe "-javaagent:D:\IntelliJ IDEA Comm
inicio: [k, r, i, t, z, a, n]
fin:vacio
Process finished with exit code 0
```

**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

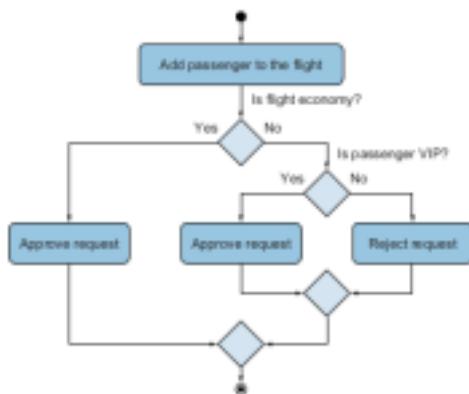
Como hemos comentado a lo largo de las actividades, Tested Data Systems ( empresa de ejemplo) está desarrollando una aplicación de gestión de vuelos para uno de sus clientes. Actualmente, la aplicación puede crear y configurar vuelos, agregar pasajeros y eliminarlos de los vuelos. En esta

evaluación, veremos escenarios que siguen el trabajo diario de los desarrolladores.

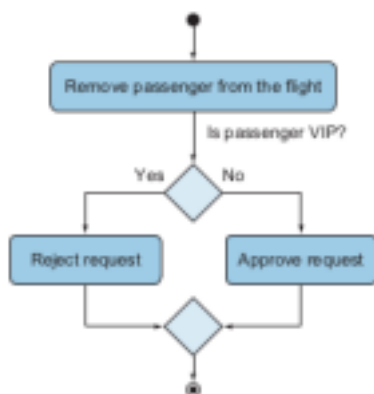
Comenzaremos con la aplicación que no es TDD, que se supone que debe hacer varias cosas, como seguir las políticas de la empresa para pasajeros regulares y VIP. Necesitamos comprender la aplicación y asegurarnos de que realmente está implementando las operaciones esperadas. Entonces, tenemos que cubrir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

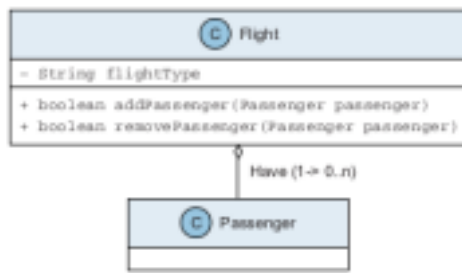


También existe una política para la eliminación de pasajeros de los vuelos: un pasajero regular puede ser eliminado de un vuelo, pero un pasajero VIP no puede ser eliminado. Como podemos ver en estos dos diagramas de actividades, la lógica empresarial inicial se centra en la toma de decisiones.



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.

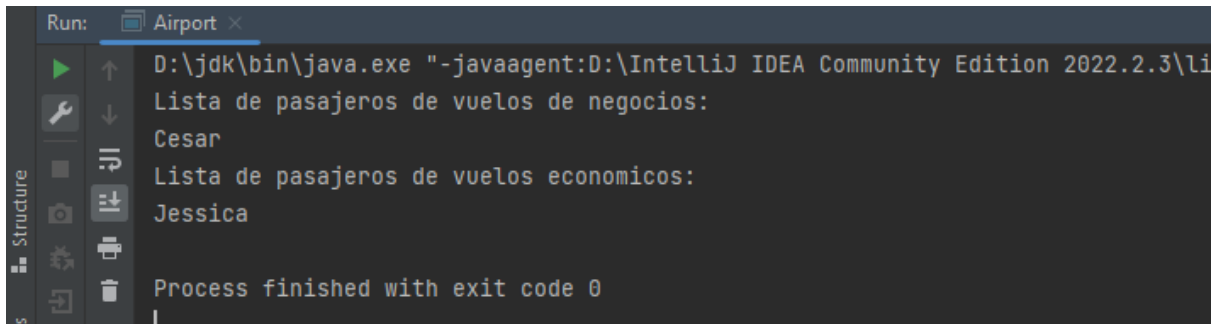
**Pregunta 1 (0.5 puntos):** Ejecuta el programa y presenta los resultados y explica qué sucede.

```

1  class Airport {
2
3  public static void main(String[] args) {
4      Flight economyFlight = new Flight( id: "1", flightType: "Economico"); //se instancia economyFlight como flight
5      Flight businessFlight = new Flight( id: "2", flightType: "Negocios"); //se instancia businessFlight como flight
6
7      Passenger cesar = new Passenger( name: "Cesar", vip: true); //se instancia cesar como passenger
8      Passenger jessica = new Passenger( name: "Jessica", vip: false); //se instancia jessica como passenger
9
10     businessFlight.addPassenger(cesar); //se agrega el pasajero cesar a vuelo businessflight
11     businessFlight.removePassenger(cesar); //se remueve el pasajero cesar del vuelo businessflight
12     businessFlight.addPassenger(jessica); //se agrega la pasajera jessica al vuelo businessflight
13     economyFlight.addPassenger(jessica); //se agrega la pasajera jessica al vuelo economyflight
14
15     System.out.println("Lista de pasajeros de vuelos de negocios:");
16     for (Passenger passenger : businessFlight.getPassengersList()) { //recorre la lista de pasajeros del vuelo businessflight
17         System.out.println(passenger.getName());
18     }
19
20     System.out.println("Lista de pasajeros de vuelos economicos:");
21     for (Passenger passenger : economyFlight.getPassengersList()) { //recorre la lista de pasajeros de vuelo economyflight
22         System.out.println(passenger.getName());
23     }
24 }
  
```

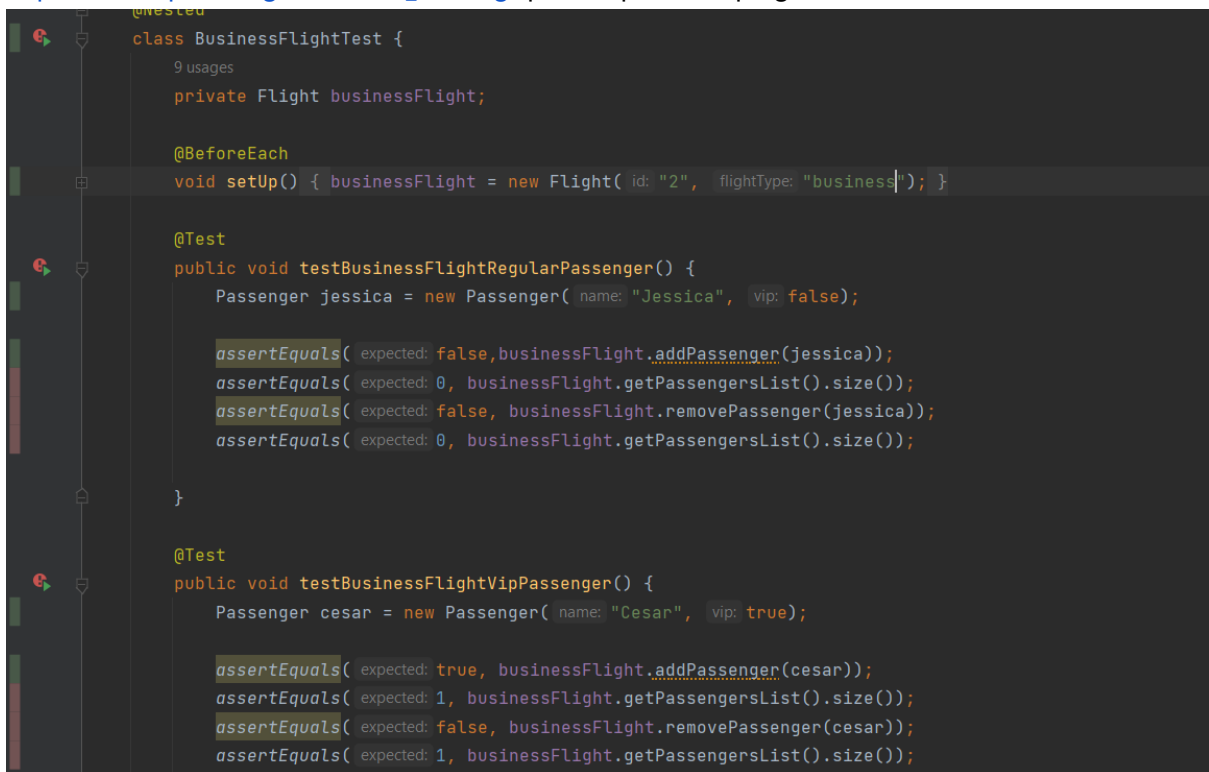
En las líneas 4 y 5 se instancian `economyFlight` y `businessFlight` ambas de la clase `Flight`. En las líneas 7 y 8 se instancian `cesar` (vip) y `jessica` (no vip) ambas de la clase `Passenger`. En la línea 10 se agrega el pasajero Cesar al vuelo `businessFlight`, lo cual si se puede hacer ya que Cesar es vip y puede ser agregado a cualquier tipo de vuelo. En la línea 11 se remueve Cesar sin embargo no se ejecuta ya que al ser un pasajero vip este no puede ser removido de ningún tipo de vuelo. En la línea 12 se agrega la pasajera Jessica al vuelo `businessFlight` sin embargo no se ejecuta ya que es una pasajera no vip y solo los pasajeros vips tienen acceso a los vuelos de negocios. En la línea 13 se agrega la pasajera jessica al vuelo `economyFlight`. En las líneas 16 y 17 se declara un `for` que recorre la lista de pasajeros del vuelo `businessFlight` e imprime el nombre. Lo mismo sucede en las líneas 21 y 22 se declara un `for` que recorre la lista de pasajeros del vuelo `economyFlight` e imprime sus nombres.

Finalmente al ejecutar este programa se obtiene:



```
Run: Airport x
D:\jdk\bin\java.exe "-javaagent:D:\IntelliJ IDEA Community Edition 2022.2.3\li
Lista de pasajeros de vuelos de negocios:
Cesar
Lista de pasajeros de vuelos economicos:
Jessica
Process finished with exit code 0
```

**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](https://en.wikipedia.org/wiki/Code_coverage) para responder la pregunta.



```
class BusinessFlightTest {
    9 usages
    private Flight businessFlight;

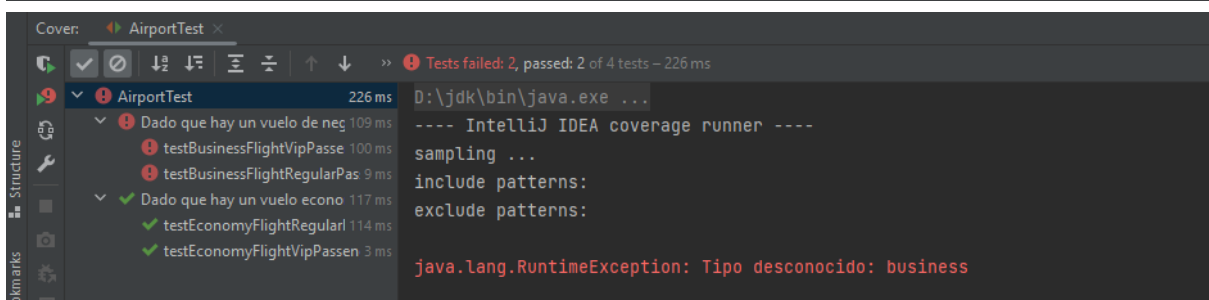
    @BeforeEach
    void setUp() { businessFlight = new Flight( id: "2", flightType: "business"); }

    @Test
    public void testBusinessFlightRegularPassenger() {
        Passenger jessica = new Passenger( name: "Jessica", vip: false);

        assertEquals( expected: false, businessFlight.addPassenger(jessica));
        assertEquals( expected: 0, businessFlight.getPassengersList().size());
        assertEquals( expected: false, businessFlight.removePassenger(jessica));
        assertEquals( expected: 0, businessFlight.getPassengersList().size());
    }

    @Test
    public void testBusinessFlightVipPassenger() {
        Passenger cesar = new Passenger( name: "Cesar", vip: true);

        assertEquals( expected: true, businessFlight.addPassenger(cesar));
        assertEquals( expected: 1, businessFlight.getPassengersList().size());
        assertEquals( expected: false, businessFlight.removePassenger(cesar));
        assertEquals( expected: 1, businessFlight.getPassengersList().size());
    }
}
```



```
Cover: AirportTest x
Tests failed: 2, passed: 2 of 4 tests - 226 ms

AirportTest 226 ms
  Dado que hay un vuelo de neg 109 ms
    testBusinessFlightVipPasse 100 ms
    testBusinessFlightRegularPas 9 ms
  Dado que hay un vuelo econo 117 ms
    testEconomyFlightRegular 114 ms
    testEconomyFlightVipPassen 3 ms

D:\jdk\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
exclude patterns:
java.lang.RuntimeException: Tipo desconocido: business
```

La cobertura no es de 100% ya que al instanciar el businessFlight se nombra a su atributo flightType

business. Cuando se ejecuta el código el método addPassenger va a la excepción de vuelo de tipo desconocido. Una corrección sería cambiar el tipo de vuelo a Negocios o Económico ya que son los 2 tipos de vuelo que existen hasta ahora.

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

Para que la cobertura del código piedra sea del 100%. En este caso se debería

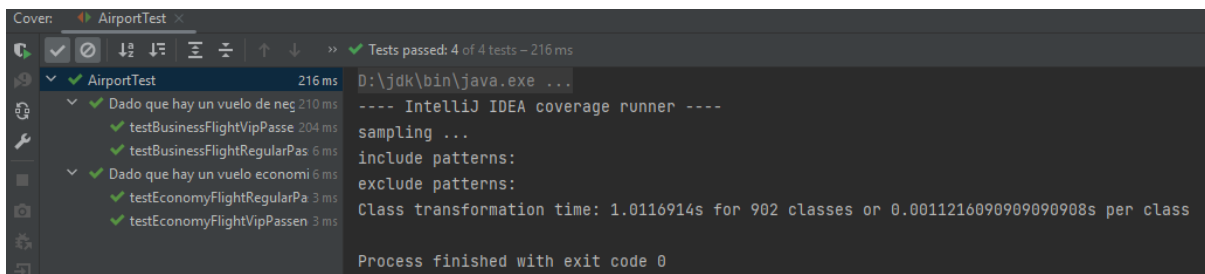
cambiar el tipo de vuelo con uno de los casos que se establecieron, es decir,

Económico o Negocios. Si se realiza dicho cambio entonces la cobertura del código

es del 100%

```
class BusinessFlightTest {
    9 usages
    private Flight businessFlight;

    @BeforeEach
    void setUp() { businessFlight = new Flight( id: "2", flightType: "Negocios"); }
```



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

En la fase 2 a comparación de la fase 1 la clase flight pasa a ser una clase abstracta y contiene atributos y metodos de un vuelo en general por ejemplo el id. También se crean las clases BusinessFlight y EconomyFlight que extienden de la clase flight ambas tiene el método agregar pasajeros y remover pasajero. estos serán modificado con las condiciones que se establecieron dependiendo del tipo de vuelo . en AirportTest se puede nota que en la linea 18 primero se instancia economyflight como flighth y luego instancia como EconomyFlight. Realizado esto se especifica que tipo de vuelo es y que métodos se deben usar.

```

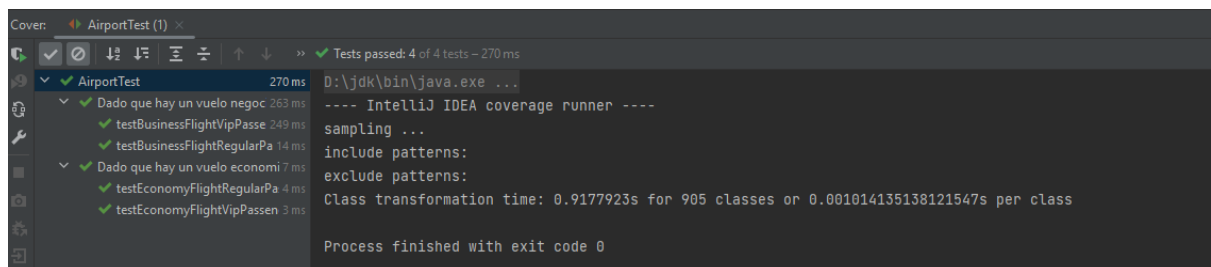
class EconomyFlightTest {

    13 usages
    private Flight economyFlight;

    @BeforeEach
    void setUp() { economyFlight = new EconomyFlight( id: "1"); }
}

```

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 subclases. El campo flightType ya no es significativo y se eliminará.



Cover: AirportTest (1) x

Tests passed: 4 of 4 tests – 270 ms

--- IntelliJ IDEA coverage runner ---

sampling ...

include patterns:

exclude patterns:

Class transformation time: 0.9177923s for 905 classes or 0.001014135138121547s per class

Process finished with exit code 0

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

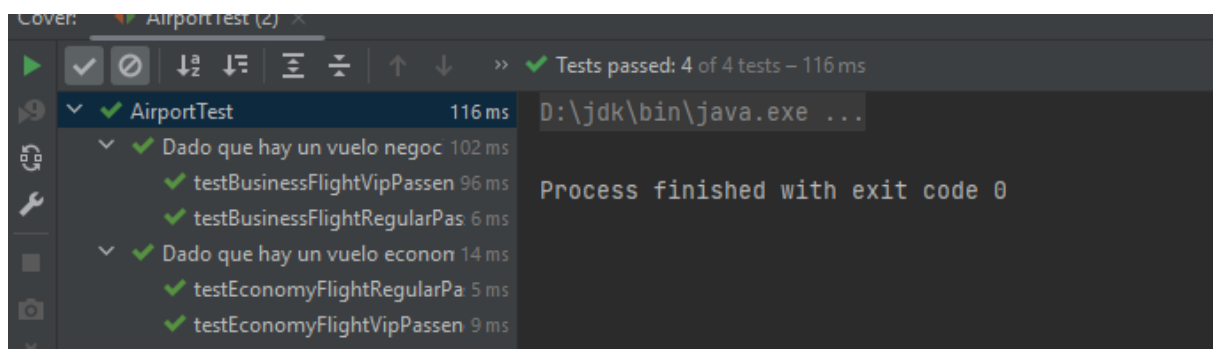
```

public class AirportTest {
...
}

```

Y responde las siguientes preguntas:

- ¿Cuál es la cobertura del código ?



Cover: AirportTest (2) x

Tests passed: 4 of 4 tests – 116 ms

--- IntelliJ IDEA coverage runner ---

sampling ...

include patterns:

exclude patterns:

Class transformation time: 0.9177923s for 905 classes or 0.001014135138121547s per class

Process finished with exit code 0

La cobertura del código es del 100%

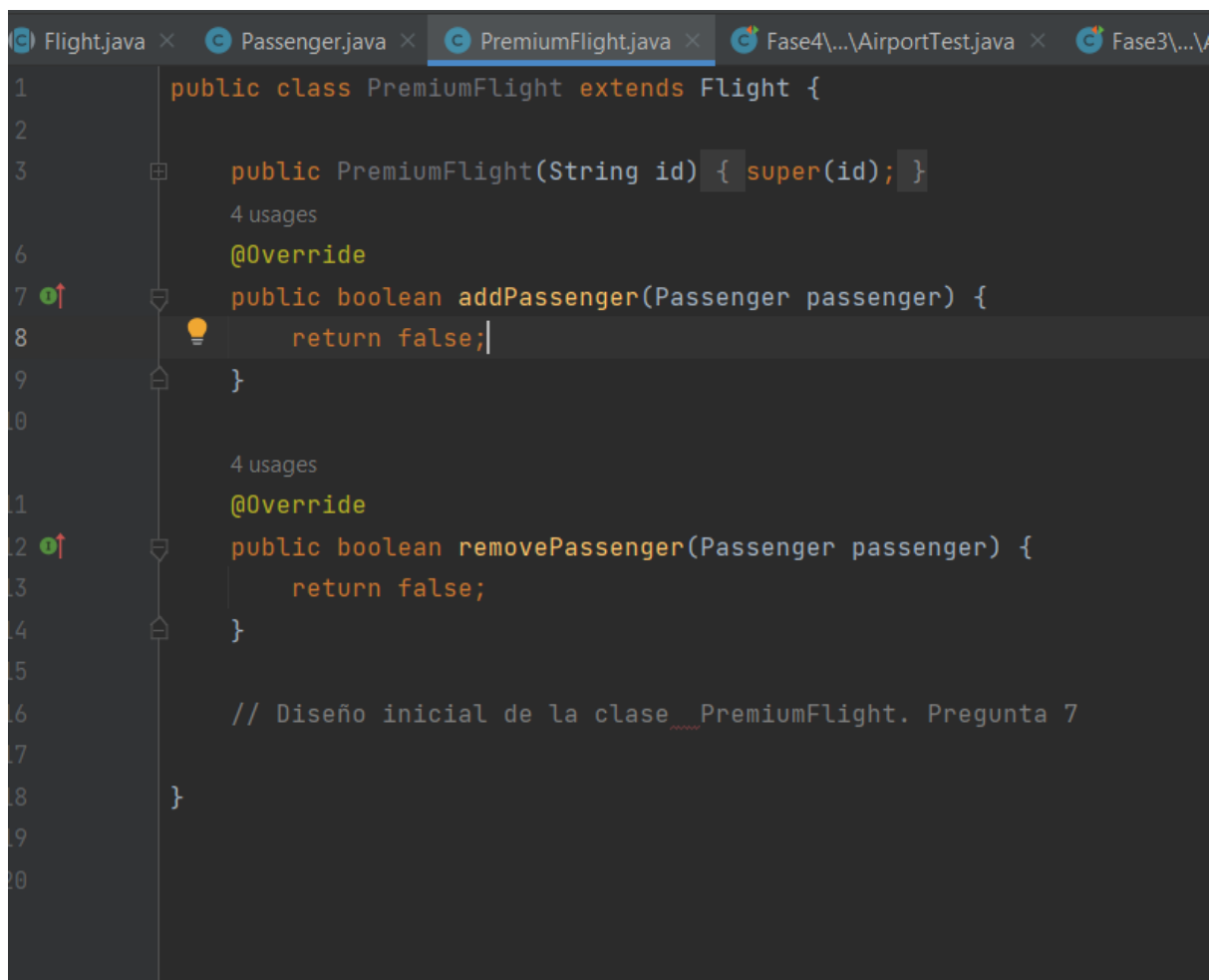
- ¿La refactorización de la aplicación TDD ayudó tanto a mejorar la calidad del código?.

Si, aumento en cierta medida el porcentaje de cobertura.

**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.

Consiste en que se refactoriza el código la tercera vez que se implementa y así evitar la duplicación.

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



```
1 public class PremiumFlight extends Flight {
2
3     public PremiumFlight(String id) { super(id); }
4     4 usages
5
6     @Override
7     public boolean addPassenger(Passenger passenger) {
8         return false;
9     }
10
11     4 usages
12
13     @Override
14     public boolean removePassenger(Passenger passenger) {
15         return false;
16     }
17
18     // Diseño inicial de la clase PremiumFlight. Pregunta 7
19
20 }
```

**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.

```
Passenger.java x PremiumFlight.java x AirportTest.java x Comparison Failure x

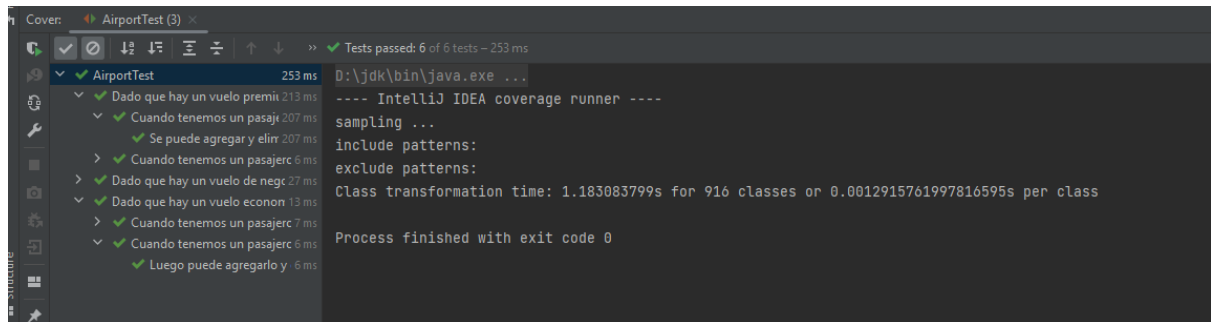
@DisplayName("Dado que hay un vuelo premium")
@Nested
class PremiumFlightTest {
    9 usages
    private Flight premiumFlight;
    3 usages
    private Passenger jessica;
    3 usages
    private Passenger cesar;

    @BeforeEach
    void setUp() {
        premiumFlight = new PremiumFlight( id: "3");
        jessica = new Passenger( name: "Jessica", vip: false);
        cesar = new Passenger( name: "Cesar", vip: true);
    }

    @Nested
    @DisplayName("Cuando tenemos un pasajero regular")
    class RegularPassenger {

        @Test
        public void testPremiumFlightRegularPassenger() {
            assertAll(
                () -> assertEquals( expected: false, premiumFlight.addPassenger(jessica)),
                () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size()),
                () -> assertEquals( expected: false, premiumFlight.removePassenger(jessica)),
                () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size())
            );
        }
    }
}
```

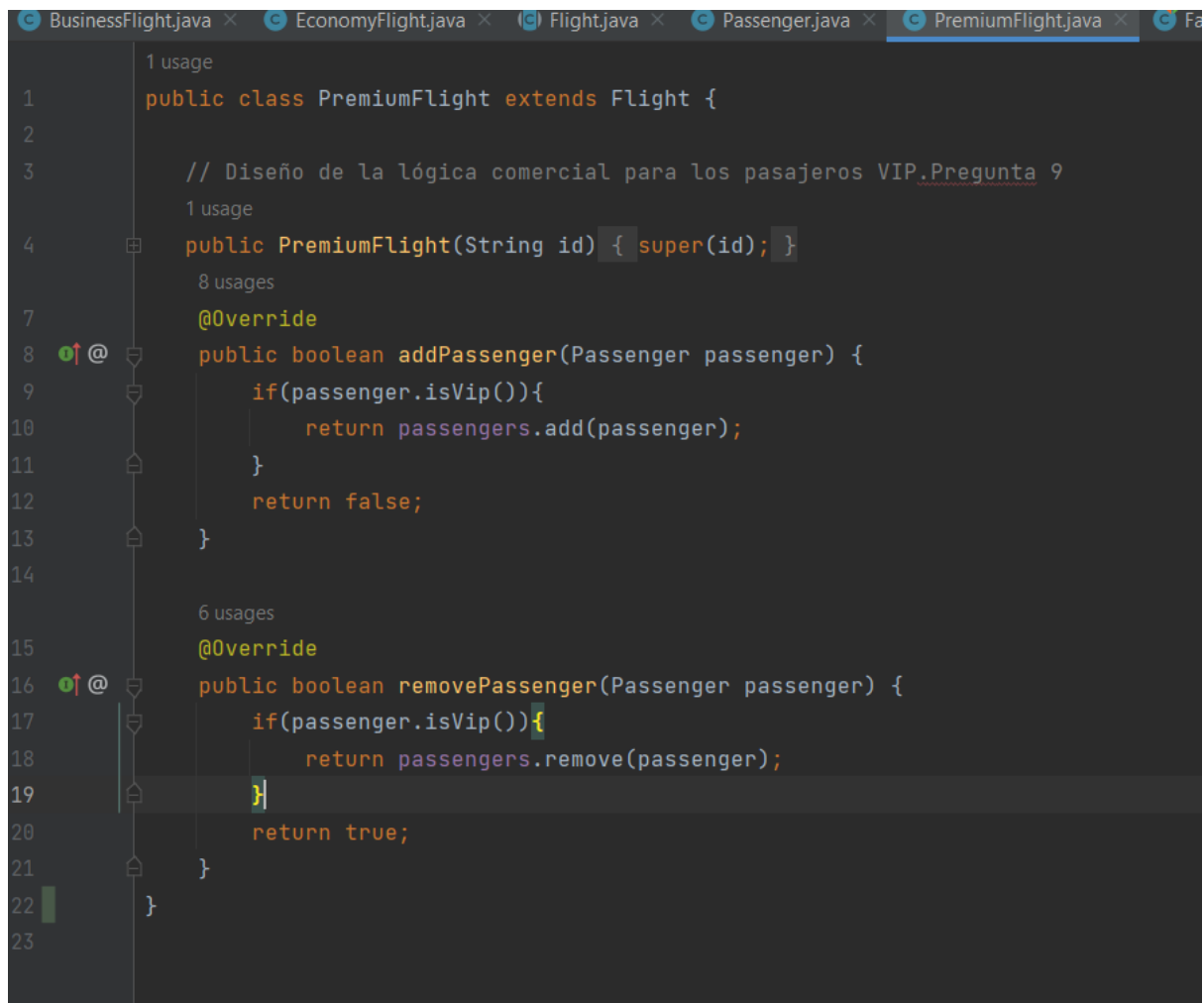
```
Flight.java x Passenger.java x PremiumFlight.java x AirportTest.java x Comparison Failure x
134      () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size())
135      );
136      }
137  }
138
139  @Nested
140  @DisplayName("Cuando tenemos un pasajero VIP")
141  class VipPassenger {
142
143      @Test
144      public void testPremiumFlightVipPassenger() {
145          assertAll(
146              () -> assertEquals( expected: false, premiumFlight.addPassenger(cesar)),
147              () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size()),
148              () -> assertEquals( expected: false, premiumFlight.removePassenger(cesar)),
149              () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size())
150          );
151      }
152  }
153
154  }
155  }
156
157
158  // Completa la prueba para PremiumFlight de acuerdo a la logica comercial dada. Pregunta 8
159
160
```



**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**.

```
public class PremiumFlight extends Flight {
```

```
...
}
```



para el tipo de vuelo premium su id será "3". También cumple las condiciones establecidas: si el pasajero

es vip entonces podrá ser agregado, caso contrario no se podrá y se moverá un pasajero si se requiere.

Y su prueba correspondiente es:

```
151
152
153     @Nested
154     @DisplayName("Cuando tenemos un pasajero regular")
155     class RegularPassenger {
156
157         @Test
158         @DisplayName("No puedes agregarlo pero si eliminarlo de un vuelo premium")
159         public void testPremiumFlightRegularPassenger() {
160             assertEquals(
161                 expected: false, premiumFlight.addPassenger(jessica)),
162                 expected: 0, premiumFlight.getPassengersSet().size()),
163                 expected: true, premiumFlight.removePassenger(jessica)),
164                 expected: 0, premiumFlight.getPassengersSet().size())
165             );
166         }
167     }
168
169     @Nested
170     @DisplayName("Cuando tenemos un pasajero VIP")
171     class VipPassenger {
172
173         @Test
174         @DisplayName("Se puede agregar y eliminar de un vuelo premium")
175         public void testPremiumFlightVipPassenger() {
176             assertEquals(
177                 expected: true, premiumFlight.addPassenger(cesar)),
178                 expected: 1, premiumFlight.getPassengersSet().size()),
179                 expected: true, premiumFlight.removePassenger(cesar)),
180                 expected: 0, premiumFlight.getPassengersSet().size())
181             );
182         }
183     }
184 }
```

```
Tests passed: 23 of 23 tests - 2 sec 193 ms
AirportTest 2 sec 193 ms
  Dado que hay un vuelo premium 2 sec 73 ms
    Entonces no puedo agregar un pasajero regular 1 sec 986 ms
      repetition 1 of 8 1 sec 914 ms
      repetition 2 of 8 10 ms
      repetition 3 of 8 9 ms
      repetition 4 of 8 25 ms
      repetition 5 of 8 8 ms
      repetition 6 of 8 7 ms
      repetition 7 of 8 6 ms
      repetition 8 of 8 7 ms
    Cuando tenemos un pasajero regular 78 ms
      Se puede agregar y eliminar de un vuelo premium 78 ms
    Cuando tenemos un pasajero VIP 9 ms
      No puedes agregarlo pero si eliminarlo de un vuelo premium 9 ms
  Dado que hay un vuelo de negocios 101 ms
  Dado que hay un vuelo economico 19 ms
    Cuando tenemos un pasajero regular 19 ms
      Luego puedes agregarlo 7 ms
    Entonces no puedes agregar un pasajero regular 12 ms
      repetition 1 of 5 3 ms
      repetition 2 of 5 3 ms
      repetition 3 of 5 2 ms
      repetition 4 of 5 2 ms
      repetition 5 of 5 2 ms
```

**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.

Se agregaron pruebas para evitar que se repita un pasajero en un vuelo premium

```
@RepeatedTest(8)
public void testPremiumFlightVipPassengerAddedOnlyOnce(RepetitionInfo repetitionInfo) {
    for (int i = 0; i < repetitionInfo.getCurrentRepetition(); i++) {
        premiumFlight.addPassenger(cesar);
    }

    assertEquals(heading: "Verifica que un pasajero VIP se pueda agregar a un vuelo premium solo una vez",
        () -> assertEquals(expected: 1, premiumFlight.getPassengersSet().size()),
        () -> assertTrue(premiumFlight.getPassengersSet().contains(cesar)),
        () -> assertTrue(new ArrayList<>(premiumFlight.getPassengersSet()).get(0).getName().equals("Cesar")
    );
}
}
```

```
Tests passed: 23 of 23 tests - 2 sec 193 ms
AirportTest 2 sec 193 ms
  Dado que hay un vuelo pn 2 sec 73 ms
    Entonces no puedo agregarlo 1 sec 986 ms
      repetition 1 of 8 1 sec 914 ms
      repetition 2 of 8 10 ms
      repetition 3 of 8 9 ms
      repetition 4 of 8 25 ms
      repetition 5 of 8 8 ms
      repetition 6 of 8 7 ms
      repetition 7 of 8 6 ms
      repetition 8 of 8 7 ms
    Cuando tenemos un pasajero 78 ms
      Se puede agregar y eliminar 78 ms
    Cuando tenemos un pasajero 9 ms
      No puedes agregarlo porque 9 ms
  Dado que hay un vuelo de negocios 101 ms
  Dado que hay un vuelo economico 19 ms
    Cuando tenemos un pasajero 19 ms
      Luego puedes agregarlo 7 ms
      Entonces no puedes agregarlo 12 ms
        repetition 1 of 5 3 ms
        repetition 2 of 5 3 ms
        repetition 3 of 5 2 ms
        repetition 4 of 5 2 ms
        repetition 5 of 5 2 ms

---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
exclude patterns:
Class transformation time: 1.4747866s for 934 classes or 0.001579000642398287s per class
Process finished with exit code 0
```

### 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) :** ¿Cuales son los problemas de este código de prueba?.

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

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.

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

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

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 .

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