

### Pregunta1a:

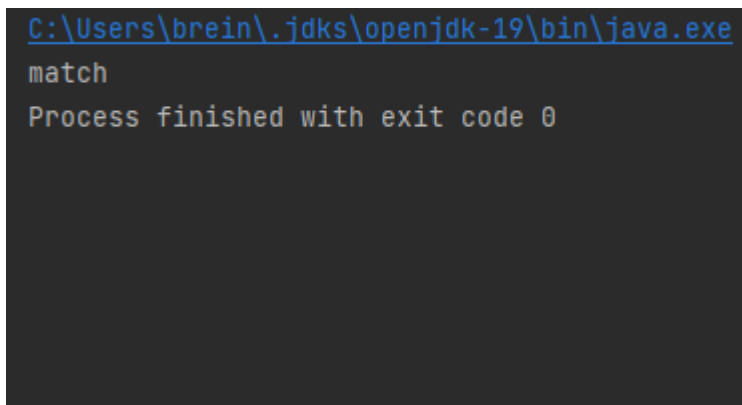
En la línea 10 al método check se le dan los parámetros Panda panda y el Predicate<panda> que se llamara pred

en la línea 12 y 13 el resultado que obtendremos depende de pred, si pred.test(panda) es verdadero entonces result es igual a "match", sino es "not match"

en la línea 7 tenemos que edad es 1

y que en la línea 8 el parámetro "p->p.age<5" nos dice que el Predicate se basará en ese booleano

si la edad es menor a 5 hace "match" sino sera "not match" en este caso la edad es 1 entonces hara "match"



```
C:\Users\brein\.jdk\openjdk-19\bin\java.exe  
match  
Process finished with exit code 0
```

### Pregunta 1b

```

interface Climb {
    1 usage BreinerCatalinoMorales
    boolean isTooHigh(int height, int limit);
}

1 usage BreinerCatalinoMorales *
public class Climber {
    BreinerCatalinoMorales *
    public static void main(String[] args) {
        check((h, m) -> h.append(m).isEmpty(), height: 5);
    }

    1 usage BreinerCatalinoMorales
    private static void check(Climb climb, int height) {
        if (climb.isTooHigh(height, limit: 10))
            System.out.println("too high");
        else
            System.out.println("ok");
    }
}

```

como se observa hay un error al usar .append ya que este solo sirve para strings mientras que m es un entero

pero sin ese error lo que debería hacer el código es devolver "too high" si es que el parámetro que estoy metiendo en este caso height=5 fuera mayor a 10 pero no lo es, caso contrario devolverá "ok".

#### Pregunta 1c

```

import java.util.function.Supplier;
2 usages 1 implementation BreinerCatalinoMorales
interface Secret {
    1 implementation BreinerCatalinoMorales
    void magic(double d);
}

BreinerCatalinoMorales *
class Secret1 implements Secret {
    1 usage
    String str="poof";
    BreinerCatalinoMorales *
    public void magic(double d) {
        Supplier<String> lambda = () -> str;
        System.out.println(lambda.get());
    }
}

```

se utilizo la interfaz supplier que no necesita parametro y devolvera un valor en este caso el string "str" que inicializamos

#### Pregunta1d

```
public class Pregunta1d {  
    BreinerCatalinoMorales  
    public static void main(String[] args){  
        List<Character> chars = new ArrayList<>();  
        chars.add('a');  
        chars.add('b');  
        chars.add('c');  
        chars.add('d');  
        chars.add('z');  
        chars.add('X');  
  
        //Muestra la lista de caracteres  
        System.out.println(chars);  
  
        //Elimina los caracteres  
        remove(chars);  
        System.out.println(chars);  
    }  
}
```

```
//Si la palabra que remove en este caso 'c'  
//es mayor o igual que 'a' Y es menor o igual que 'z'  
// entonces se elimina ese caracter  
1 usage BreinerCatalinoMorales  
public static void remove(List<Character> chars) {  
    char end = 'z';  
    chars.removeIf(c -> {  
        char start = 'a'; return start <= c && c <= end; });  
    }  
}
```

En este código Si la palabra que remove en este caso 'c' es mayor o igual que 'a' Y es menor o igual que 'z' entonces retorna verdadero y procede a eliminar ese carácter

```
[a, b, c, d, z, X]  
[X]
```

como se observa, se eliminaron todas menos el carácter 'X'.

### Pregunta 1e

```
3 ▶ public class Pregunta1e {  
4 ▶     public static void main(String[] args){  
5         int length = 3;  
6         for (int i = 0; i<3; i++) {  
7             if (i%2 == 0) {  
8                 Supplier<Integer> supplier = () -> length; // A  
9                 System.out.println(supplier.get()); // B  
10            } else {  
11                int j = i;  
12                Supplier<Integer> supplier = () -> j; // C  
13                System.out.println(supplier.get()); // D  
14            }  
15        }  
16    }  
17 }
```

En este código se está usando la interfaz Supplier que devolverá un valor  
en la línea 5 se inicializa la longitud con 3  
en la línea 6 se usa el ciclo for que dice  
en la línea 7 8 y 9 Si "i" es múltiplo de 2 el supplier devolverá length  
en la línea 10, si no entonces se inicializa el nuevo valor "j" con "i"  
en la línea 12 se usa supplier para devolver j y en  
la línea 13 se muestra en consola

```
C:\Users\brein\.jdk\openjdk-19\bin\java.exe  
3  
1  
3  
  
Process finished with exit code 0
```

### Pregunta 1f

Como ya se vio en la pregunta 1d, se sigue el mismo procedimiento

```

//Si la palabra que remove en este caso 'c'
//es mayor o igual que 'a' Y es menor o igual que 'z'
// entonces se elimina ese caracter
1 usage  BreinerCatalinoMorales
public static void remove(List<Character> chars) {
    char end = 'z';
    chars.removeIf(c -> {
        char start = 'a'; return start <= c && c <= end; });
}
}

```

En este código Si la palabra que remove en este caso 'c' es mayor o igual que 'a' Y es menor o igual que 'z' entonces retorna verdadero y procede a eliminar ese carácter

## Pregunta 2 ( 12 puntos)

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

```

1  ▶ public class Airport {
2
3  ▶  public static void main(String[] args) {
4      Flight economyFlight = new Flight( id: "1", flightType: "Economico");
5      Flight businessFlight = new Flight( id: "2", flightType: "Negocios");
6
7      Passenger cesar = new Passenger( name: "Cesar", vip: true);
8      Passenger jessica = new Passenger( name: "Jessica", vip: false);
9
10     businessFlight.addPassenger(cesar);
11     businessFlight.removePassenger(cesar);
12     businessFlight.addPassenger(jessica);
13     economyFlight.addPassenger(jessica);
14
15     System.out.println("Lista de pasajeros de vuelos de negocios:");
16     for (Passenger passenger : businessFlight.getPassengersList()) {
17         System.out.println(passenger.getName());
18     }
19
20     System.out.println("Lista de pasajeros de vuelos economicos:");
21     for (Passenger passenger : economyFlight.getPassengersList()) {
22         System.out.println(passenger.getName());
23     }
24 }
25 }
26

```

```

Lista de pasajeros de vuelos de negocios:
Cesar
Lista de pasajeros de vuelos economicos:
Jessica

Process finished with exit code 0

```

Como se observa se muestra en la consola la lista de pasajeros en vuelo de negocios y en vuelo economico

Cesar es de la clase negocio ya que en la línea 7 se dice que cesar si es pasajero vip por lo tanto si puede estar en los vuelos económicos como de negocios mientras que jessica no es vip por lo tanto solo puede estar en el vuelo economico

como se observa en la línea 11 a césar se le quiere eliminar del vuelo de negocios pero no es posible ya que la política dice que no se puede eliminar vips

y en la línea 12 se le quiere agregar a jessica al vuelo de negocios pero no es posible ya que no es vip

## Pregunta 2 (1 punto)

Element ▲	Class, %	Method, %	Line, %
▼ all	83% (10/1...	88% (32/36)	59% (82/138)
Airport	0% (0/1)	0% (0/1)	0% (0/16)
AirportTest	100% (3/3)	100% (8/8)	79% (23/29)
Flight	100% (1/1)	83% (5/6)	68% (13/19)
Passenger	100% (1/1)	100% (3/3)	100% (5/5)

Como se observa en AirportTest en el “Line %” no sale al 100 % esto debido al error de escritura como se observa abajo

```

java.lang.RuntimeException: Tipo desconocido: Business
    at Flight.addPassenger(Flight.java:38)
    at AirportTest$BusinessFlightTest.testBusinessFlightVipPassenger(AirportTest.java:73) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <27 internal lines>

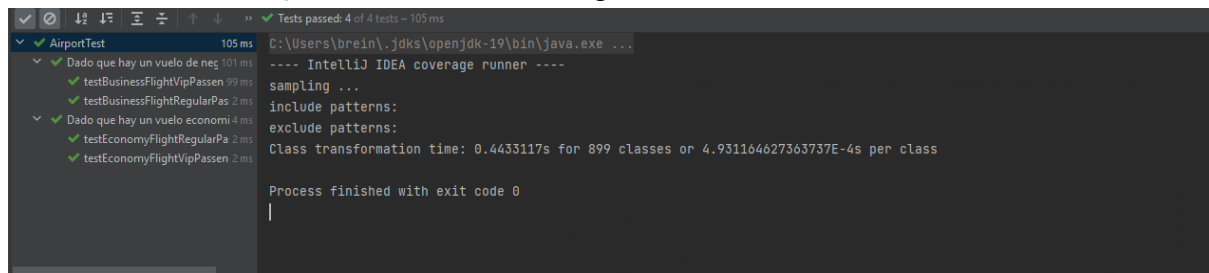
java.lang.RuntimeException: Tipo desconocido: Business
    at Flight.addPassenger(Flight.java:38)
    at AirportTest$BusinessFlightTest.testBusinessFlightRegularPassenger(AirportTest.java:62) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <27 internal lines>

```

Como se observa nos sale 2 errores de prueba por escritura ya que en la clase Flight los tipos de vuelos estan en español mientras que que en la prueba se escribio en ingles, despues de cambiar esto se hizo el coverage y se obtuvo lo siguiente

all	83% (10/12)	88% (32/36)	72% (100/138)
Airport	0% (0/1)	0% (0/1)	0% (0/16)
AirportTest	100% (3/3)	100% (8/8)	100% (29/29)
Flight	100% (1/1)	83% (5/6)	84% (16/19)
Passenger	100% (1/1)	100% (3/3)	100% (5/5)

Como se observa en AirportTest sale 100% coverage



Y en las pruebas todas pasaron

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

La refactorización siempre tiene el sencillo y claro propósito de mejorar el código.como el ejemplo que se nos muestras en vez de usar condicional podemos usar el polimorfismo lo cual nos ayudara de manera más eficiente, este proceso de refactorizar nos ayudará en caso de probar otros escenarios que no se nos ocurra para las pruebas .

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

```

public class AirportPregunta2_4 {

    public static void main(String[] args) {
        Flight economyFlight = new Flight(id: "1", "Economico");
        Flight businessFlight = new Flight(id: "2", "Negocios");

        Passenger cesar = new Passenger( name: "Cesar", vip: true);
        Passenger jessica = new Passenger( name: "Jessica", vip: false);

        businessFlight.addPassenger(cesar);
        businessFlight.removePassenger(cesar);
        businessFlight.addPassenger(jessica);
        economyFlight.addPassenger(jessica);

        System.out.println("Lista de pasajeros de vuelos de negocios:");
        for (Passenger passenger : businessFlight.getPassengersList()) {
            System.out.println(passenger.getName());
        }

        System.out.println("Lista de pasajeros de vuelos de negocios:");
        for (Passenger passenger : economyFlight.getPassengersList()) {
            System.out.println(passenger.getName());
        }
    }
}

```

Se busca usar la misma clase Aripport vista en la fase 1 par ejecutar pero como se observa ahora existen clases para cada tipo de vuelo entonces procedemos a realizar los cambios

```

1 public class AirportPregunta2_4 {
2
3     public static void main(String[] args) {
4         Flight economyFlight = new EconomyFlight(id: "1");
5         Flight businessFlight = new BusinessFlight(id: "2");
6
7         Passenger cesar = new Passenger( name: "Cesar", vip: true);
8         Passenger jessica = new Passenger( name: "Jessica", vip: false);
9
10        businessFlight.addPassenger(cesar);
11        businessFlight.removePassenger(cesar);
12        businessFlight.addPassenger(jessica);
13        economyFlight.addPassenger(jessica);
14
15        System.out.println("Lista de pasajeros de vuelos de negocios:");
16        for (Passenger passenger : businessFlight.getPassengers()) {
17            System.out.println(passenger.getName());
18        }
19
20        System.out.println("Lista de pasajeros de vuelos de negocios:");
21        for (Passenger passenger : economyFlight.getPassengers()) {
22            System.out.println(passenger.getName());
23        }
24    }
25 }
26

```

una vez cambiado observamos los cambios y compilamos



```
C:\Users\brein\.jdk\openjdk-19\bin\java.exe ...  
Lista de pasajeros de vuelos de negocios:  
Cesar  
Lista de pasajeros de vuelos economicos:  
Jessica  
  
Process finished with exit code 0
```

como se observa ya se compiló sin error, además también podemos ver los cambios que se generaron en comparación a la clase Airport de la Fase 1 que utilizamos, para este caso se usó polimorfismo para poder eliminar el TIPO DE VUELO y mejor se crearon las clases BusinessFlight y EconomiFlyght esto nos ayudará a no tener que instanciar el tipo de vuelo al momento de ingresar un pasajero

Todo esto nos ayuda a mejorar el código ya que se usan menos recursos y entenderlo mejor ya que por ejemplo gracias a las clases BussinesFlight y EconomiFlyght sabremos que tipo de vuelo se está tomando, y así cualquier persona que lea el código podrá entender cómo es que funciona

**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:

Se reutilizó AirportTest de la prueba 2 para poder realizar esta pregunta, realizando los respectivos cambios después de aplicar la refactorización

```

import static org.junit.jupiter.api.Assertions.assertEquals;

public class AirportTest {

    @DisplayName("Dado que hay un vuelo economico")
    @Nested
    class EconomyFlightTest {

        13 usages
        private Flight economyFlight;

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

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

            assertEquals( expected: "1", economyFlight.getId());
            assertEquals( expected: true, economyFlight.addPassenger(jessica));
            assertEquals( expected: 1, economyFlight.getPassengersList().size());
            assertEquals( expected: "Jessica", economyFlight.getPassengersList().get(0).getName());

            assertEquals( expected: true, economyFlight.removePassenger(jessica));
            assertEquals( expected: 0, economyFlight.getPassengersList().size());
        }
    }
}

```

```

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

            assertEquals( expected: "1", economyFlight.getId());
            assertEquals( expected: true, economyFlight.addPassenger(cesar));
            assertEquals( expected: 1, economyFlight.getPassengersList().size());
            assertEquals( expected: "Cesar", economyFlight.getPassengersList().get(0).getName());

            assertEquals( expected: false, economyFlight.removePassenger(cesar));
            assertEquals( expected: 1, economyFlight.getPassengersList().size());
        }
    }

    @DisplayName("Dado que hay un vuelo negocios")
    @Nested
    class BusinessFlightTest {

        9 usages
        private Flight businessFlight;

        @BeforeEach
        void setUp() { businessFlight = new BusinessFlight( id: "2"); }
    }
}

```

```

@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());
}
}

```

- ¿Cuál es la cobertura del código ?

Coverage: AirportTest (1) ×			
Element ^	Class, %	Method, %	Line, %
all	100% (14/14)	100% (40/40)	100% (98/98)
AirportTest	100% (3/3)	100% (8/8)	100% (29/29)
BusinessFlight	100% (1/1)	100% (3/3)	100% (5/5)
EconomyFlight	100% (1/1)	100% (3/3)	100% (5/5)
Flight	100% (1/1)	100% (3/3)	100% (5/5)
Passenger	100% (1/1)	100% (3/3)	100% (5/5)

Al realizar la cobertura se observa el 100% en todo, nos quiere decir que estamos usando el 100% del código fuente

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

Si, como se observa en el coverage y comparando con el coverage de la carpeta

“Anterior” el coverage ahora es del 100%, eso quiere decir que al usar el TDD

hemos logrado optimizar lo que va de nuestro código, por ejemplo al usar el

polimorfismo para los tipos de vuelos, esto nos ayudará para casos futuros en que

se desee crear más tipos de vuelos como vuelos Premium por ejemplo,

**Pregunta 6 (0.5 puntos):**

La regla de 3 consiste en que cuando hay 3 piezas de código similares se deben refactorizar para evitar la duplicación ya que la duplicación hace que el código sea difícil de mantener, como se ve al implementar la clase Premium se está usando la misma lógica que se uso para las demás clases como Business y Economi, ahi se podría usar la regla de 3

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

```
public class PremiumFlight extends Flight {  
    public PremiumFlight(String id) { super(id); }  
  
    6 usages  
    @Override  
    public boolean addPassenger(Passenger passenger) {  
        return true;  
    }  
  
    6 usages  
    @Override  
    public boolean removePassenger(Passenger passenger) {  
        return true;  
    }  
}
```

Como se observa se uso la misma lógica para los otros tipos de vuelos como BusinessFlight y EconomiFlight todo esto siguiendo la política del negocio

“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.”

**Pregunta 8 (2 puntos):**

Se construye las pruebas

```

@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 BusinessFlight( 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 {

```

```

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

    @Test
    @DisplayName("Entonces no puede agregarlo o eliminarlo de un vuelo Premium")
    public void testBusinessFlightRegularPassenger() {
        assertAll( heading: "Verifica todas las condiciones para un pasajero regular y un vuelo premium",
            () -> assertEquals( expected: true, premiumFlight.addPassenger(jessica)),
            () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size()),
            () -> assertEquals( expected: false, premiumFlight.removePassenger(jessica)),
            () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size())
        );
    }
}

```

```

@Nested
@DisplayName("Cuando tenemos un pasajero VIP")
class VipPassenger {

    @Test
    @DisplayName("Luego se debe agregar a un vuelo premium")
    public void testBusinessFlightVipPassenger() {
        assertAll( heading: "Verifica todas las condiciones para un pasajero VIP y un vuelo premium",
            () -> assertEquals( expected: false, premiumFlight.addPassenger(cesar)),
            () -> assertEquals( expected: 1, premiumFlight.getPassengersList().size()),
            () -> assertEquals( expected: false, premiumFlight.removePassenger(cesar)),
            () -> assertEquals( expected: 1, premiumFlight.getPassengersList().size())
        );
    }
}
}

```

```

Comparison Failure:
Expected :1
Actual   :0
<Click to see difference>

expected: <false> but was: <true>
Comparison Failure:
Expected :false
Actual   :true
<Click to see difference>

expected: <1> but was: <0>
Comparison Failure:
Expected :1
Actual   :0
<Click to see difference>

org.opentest4j.MultipleFailuresError: Verifica todas las condiciones para un pasajero VIP y un vuelo premium (3 failures)

```

Para un pasajero vip en un vuelo premium se tiene que no se puede remover del vuelo premium pero en la prueba nos retorna verdadero

```

expected: <false> but was: <true>
Comparison Failure:
Expected :false
Actual   :true
<Click to see difference>

expected: <false> but was: <true>
Comparison Failure:
Expected :false
Actual   :true
<Click to see difference>

org.opentest4j.MultipleFailuresError: Verifica todas las condiciones para un pasajero regular y un vuelo premium (2 failures)

```

para un pasajero regular y un vuelo premium, el pasajero no se puede agregar al vuelo premium pero en la prueba nos devuelve verdad, tampoco se puede remover pero en la prueba se observa que si lo hace

Ahora corregiremos el código para pasar las pruebas

```

1 usage
public class PremiumFlight extends Flight {

    1 usage
    public PremiumFlight(String id) { super(id); }

    6 usages
    @Override
    public boolean addPassenger(Passenger passenger) {
        if (passenger.isVip()) {
            return passengers.add(passenger);
        }
        return false;
    }

    6 usages
    @Override
    public boolean removePassenger(Passenger passenger) { return false; }
}

```

✓ AirportTest	90 ms
✓ Dado que hay un vuelo Premium	63 ms
✓ Cuando tenemos un pasajero VIP	57 ms
✓ Luego se debe agregar a un vuelo premium	57 ms
> ✓ Cuando tenemos un pasajero regular	6 ms
> ✓ Dado que hay un vuelo de negocios	13 ms
✓ Dado que hay un vuelo economico	14 ms
> ✓ Cuando tenemos un pasajero VIP	9 ms
✓ Cuando tenemos un pasajero regular	5 ms
✓ Luego puede agregarlo y eliminarlo de un vuelo economico	5 ms

como se observa se pasaron exitosamente usando el TDD

**Pregunta 9 (2 puntos):** En la pregunta 8 ya se había corregido el código siguiendo la lógica de negocios y aplicándolo a la prueba, en esta parte se implementaran a la Fase 5 y se obtiene lo siguiente.

```

1 usage
public class PremiumFlight extends Flight {

    1 usage
    public PremiumFlight(String id) { super(id); }

    6 usages
    @Override
    public boolean addPassenger(Passenger passenger) {
        if (passenger.isVip()) {
            return passengers.add(passenger);
        }
        return false;
    }

    6 usages
    @Override
    public boolean removePassenger(Passenger passenger) { return false; }
}

```

Siguiendo la lógica de negocio se obtuvo lo siguiente

```

class RegularPassenger {

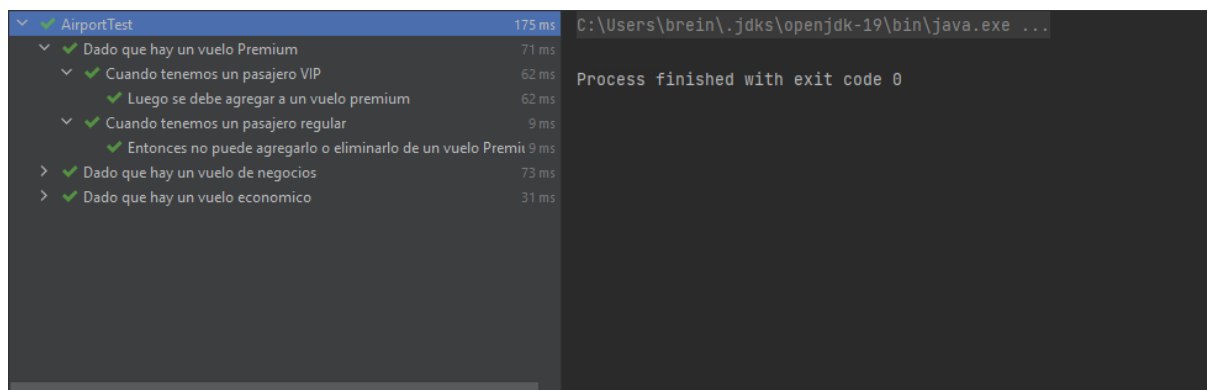
    @Test
    @DisplayName("Entonces no puede agregarlo o eliminarlo de un vuelo Premium")
    public void testBusinessFlightRegularPassenger() {
        assertAll( heading: "Verifica todas las condiciones para un pasajero regular y un vuelo premium",
            () -> assertEquals( expected: false, premiumFlight.addPassenger(jessica)),
            () -> assertEquals( expected: 0, premiumFlight.getPassengersSet().size()),
            () -> assertEquals( expected: false, premiumFlight.removePassenger(jessica)),
            () -> assertEquals( expected: 0, premiumFlight.getPassengersSet().size())
        );
    }
}

@Nested
@DisplayName("Cuando tenemos un pasajero VIP")
class VipPassenger {

    @Test
    @DisplayName("Luego se debe agregar a un vuelo premium")
    public void testBusinessFlightVipPassenger() {
        assertAll( heading: "Verifica todas las condiciones para un pasajero VIP y un vuelo premium",
            () -> assertEquals( expected: true, premiumFlight.addPassenger(cesar)),
            () -> assertEquals( expected: 1, premiumFlight.getPassengersSet().size()),
            () -> assertEquals( expected: false, premiumFlight.removePassenger(cesar)),
            () -> assertEquals( expected: 1, premiumFlight.getPassengersSet().size())
        );
    }
}

```





## 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.

### Pregunta 3 (5 puntos)

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

Primero al implementar los parámetros no se definieron todas las clases, nosotros tuvimos que definir las clases, por ejemplo se creó la clase CustomerType para que se pueda ejecutar el programa, también se tuvieron que cambiar los tipos de datos ya que había error al comparar un valor de tipo **double** con uno de tipo **BigDecimal**

```
org.opentest4j.AssertionFailedError:
Expected :250
Actual   :250.0
<Click to see difference>

<6 internal lines>
at InvoiceTest.test1(InvoiceTest.java:19) <29 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <28 internal lines>
```

como se observa

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

```
import java.math.BigDecimal;

public class InvoiceTest {
    @Test
    void testInvoice() {
        CustomerType tipe=new CustomerType( tipo1: "COMPANY");
        Invoice invoice = new Invoice(new BigDecimal( val: "2500"), country: "NL", tipe);
        BigDecimal v = invoice.calculate();
        assertEquals(new BigDecimal( val: 250),v);
    }
}
```

**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.

```
import java.math.BigDecimal;

6 usages
public class InvoiceBuilder {
    2 usages
    private String country = "NL";
    2 usages
    private CustomerType customerType;
    2 usages
    private double value = 500;

    1 usage
    public InvoiceBuilder withCountry(String country) {
        this.country = country;
        return this;
    }

    1 usage
    public InvoiceBuilder asCompany() {
        CustomerType tipe = new CustomerType( tipo1: "PERSON");
        this.customerType = tipe;
        return this;
    }
}
```

```
1 usage
public InvoiceBuilder withAValueOf(double value) {
    this.value = value;
    return this;
}

1 usage
public Invoice build() {
    return new Invoice(new BigDecimal(value), country, customerType);
}

public static void main(String[] args) {
    InvoiceBuilder invoiceBuilder = new InvoiceBuilder();
    Invoice invoice = new InvoiceBuilder()
        .asCompany()
        .withAValueOf(1203)
        .withCountry("PE")
        .build();
}
}
```

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

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

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

