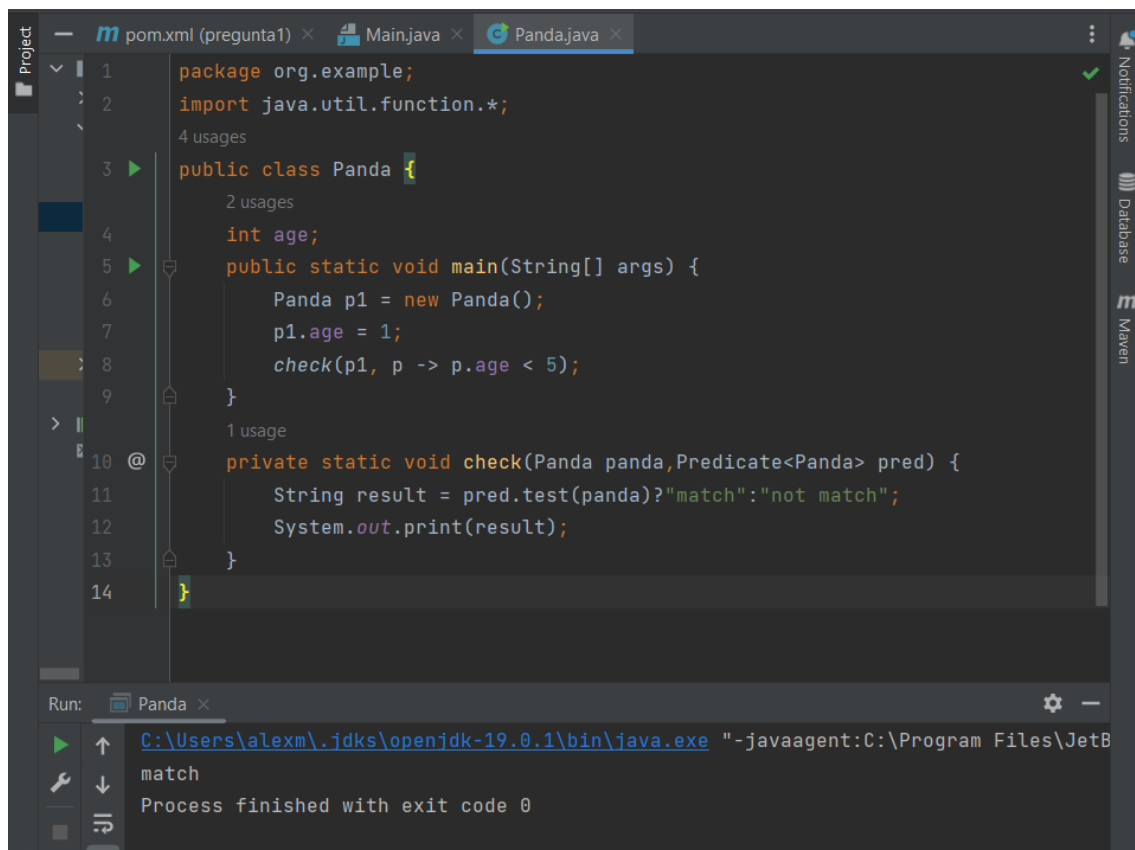


## PARCIAL DESARROLLO DE SOFTWARE

### Pregunta 1

a) ¿Cuál es el resultado de la siguiente clase? (clase Panda)

Retorna match ya que check comprueba si la edad es menor a 5, como la edad(age) es 1, entonces imprime match



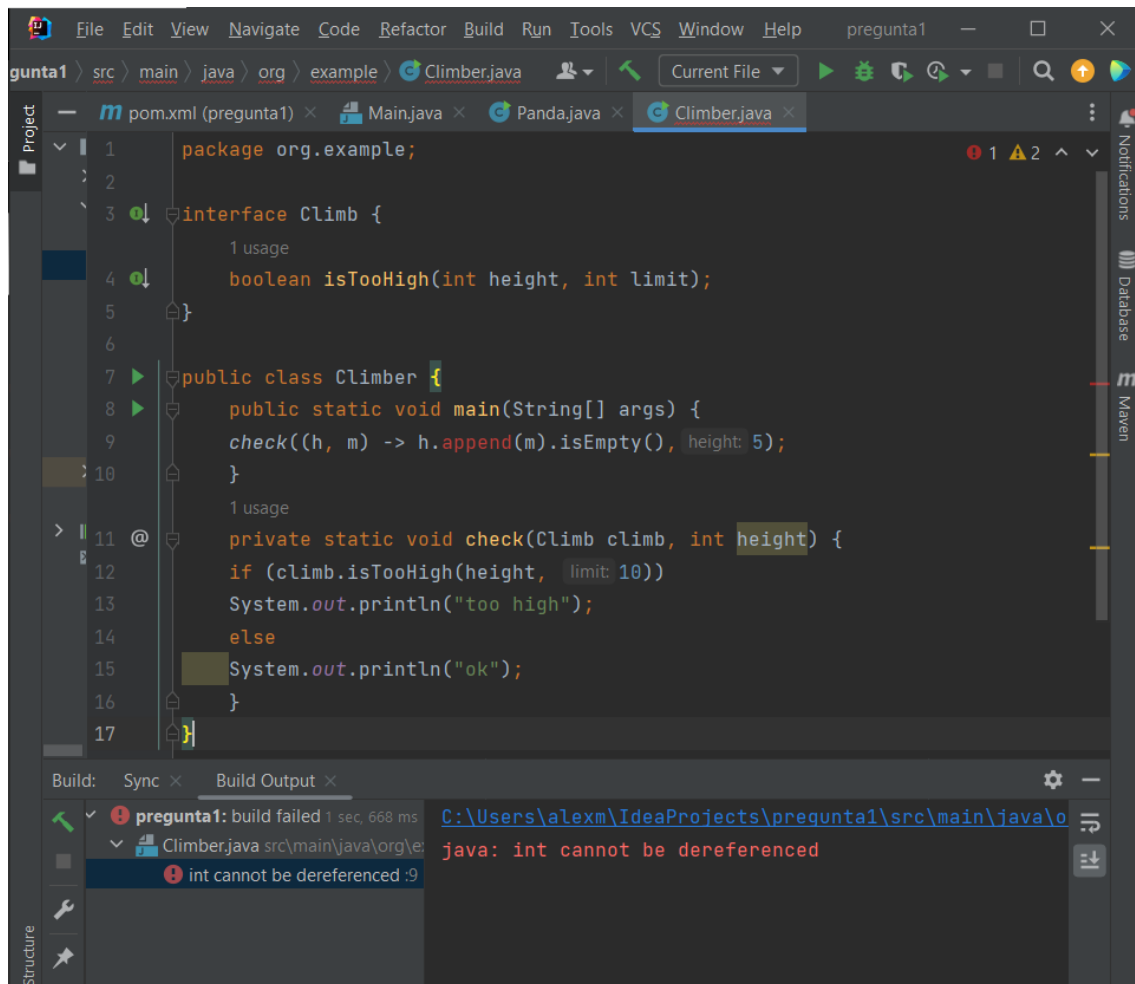
```
1 package org.example;
2 import java.util.function.*;
3 public class Panda {
4     int age;
5     public static void main(String[] args) {
6         Panda p1 = new Panda();
7         p1.age = 1;
8         check(p1, p -> p.age < 5);
9     }
10    private static void check(Panda panda, Predicate<Panda>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

Run: Panda

```
C:\Users\alexm\.jdk\openjdk-19.0.1\bin\java.exe "-javaagent:C:\Program Files\JetB
match
Process finished with exit code 0
```

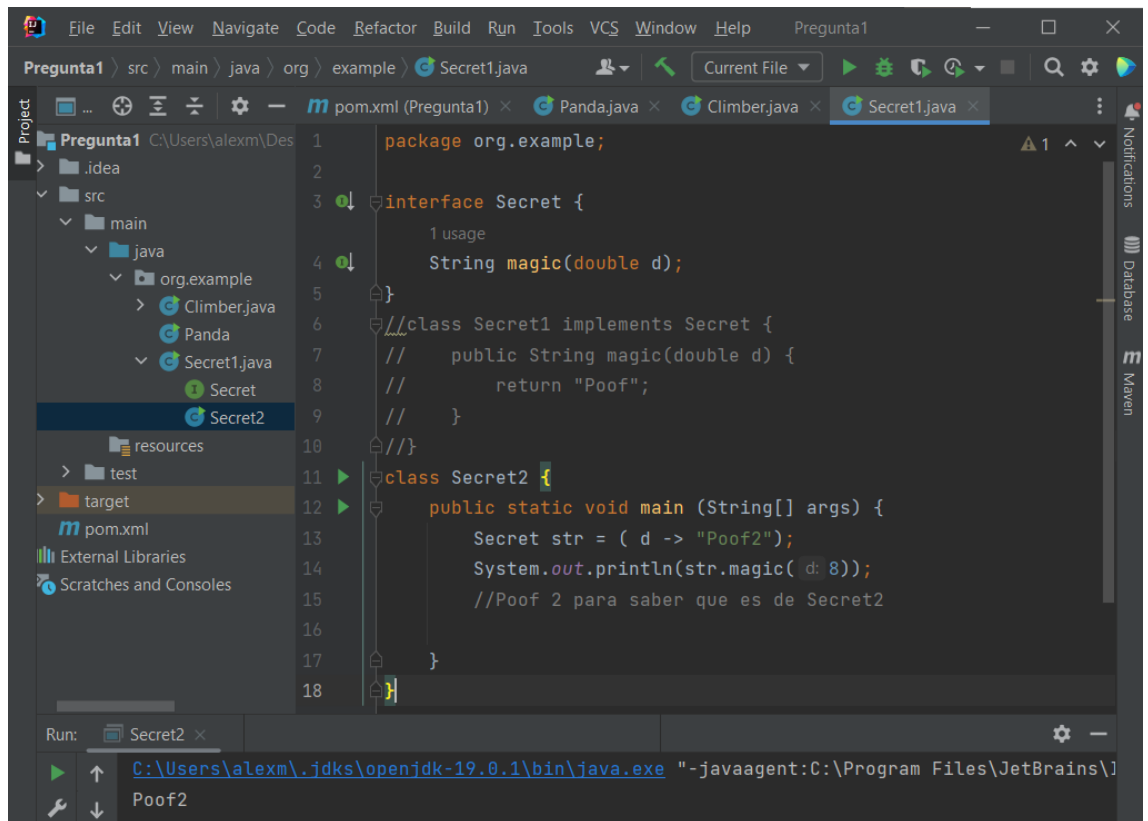
b) ¿Cuál es el resultado del siguiente código?

Error, ya que en la línea 9 no se especifica que tipo de dato es h, así que se toma como entero por defecto el cual es desreferenciado



```
/Climber.java:9: error: int cannot be dereferenced
check((h, m) -> h.append(m).isEmpty(),5);
               ^
1 error
```

c)



d)

## PREGUNTA 2

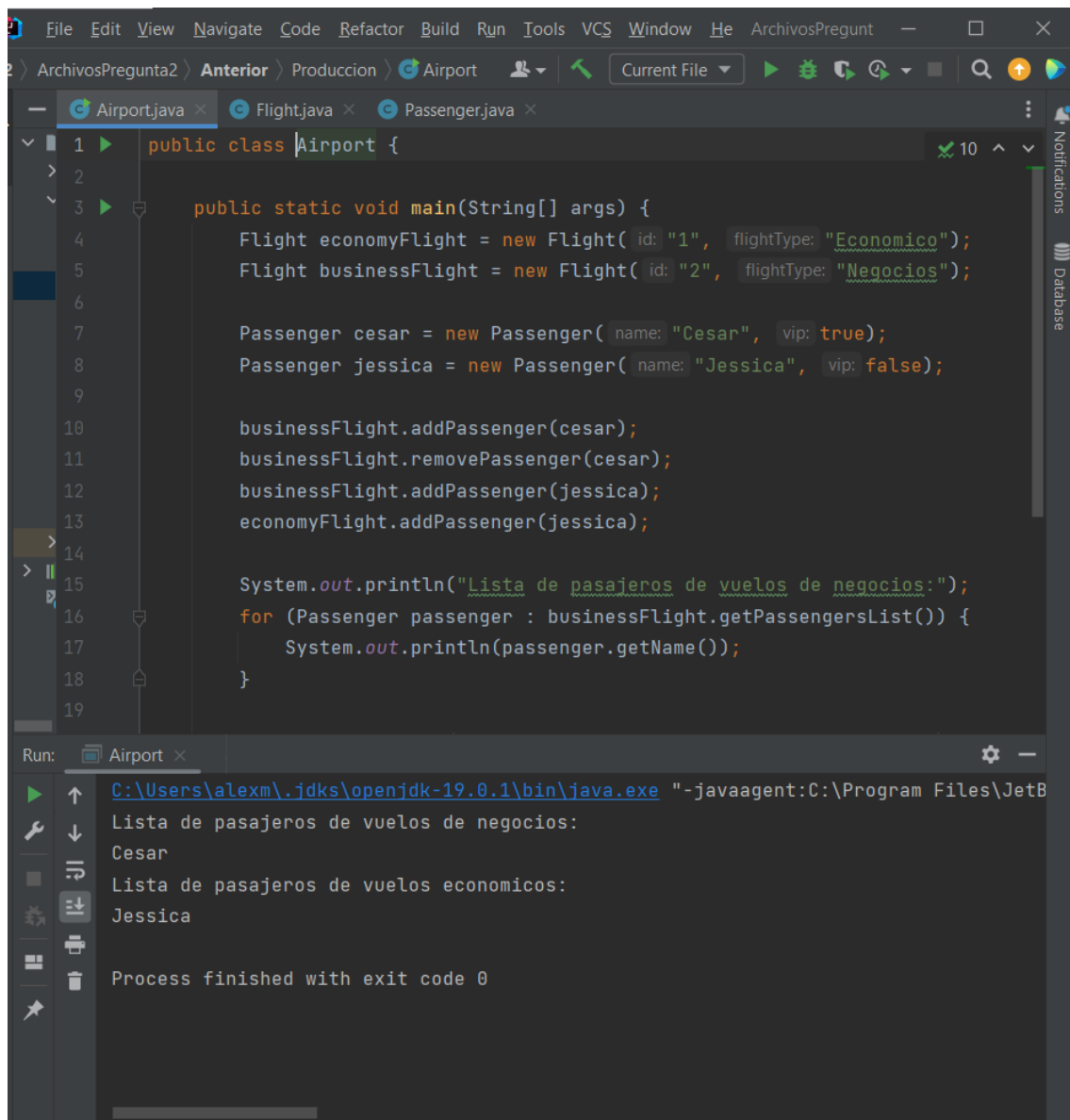
1. Se ejecuta el código e imprime:

Lista de pasajeros de vuelos de negocios: Cesar

Lista de pasajeros de vuelos económicos: Jessica

En la línea 10 podemos ver que añade un pasajero “Cesar” a negocios, el cual sí puede ser añadido ya que en la línea 7 se ve que añade a Cesar y que para el booleano vip es true, lo que

nos indica que Cesar es VIP, en la línea 11 remueve a Cesar de un viaje de negocios, sin embargo, los socios VIP no pueden ser removidos, esto se ve en Flight en removePassenger, por lo que al ejecutar el programa Cesar se mantiene en la lista de pasajeros de vuelos de negocio. En la línea 12 añade a Jessica a un vuelo de negocios, sin embargo, en la línea 8 el booleano vip es false, por lo que no se puede añadir a Jessica a un vuelo de negocios y es por ello que no sale en la lista de pasajeros de vuelos de negocio, en la línea 13 añade a Jessica a un vuelo económico, lo que si está permitido y se ve en Flight en addPassenger, por lo que Jessica aparece en la lista de pasajeros de vuelos económicos.



The screenshot shows the IntelliJ IDEA IDE with the `Airport.java` file open. The code defines an `Airport` class with a `main` method. It creates two `Flight` objects: `economyFlight` (type "Economico") and `businessFlight` (type "Negocios"). It also creates two `Passenger` objects: `cesar` (vip: true) and `jessica` (vip: false). The code adds `cesar` to `businessFlight`, removes him, adds `jessica` to `businessFlight`, and adds `jessica` to `economyFlight`. Finally, it prints the list of passengers for business flights and then for economic flights.

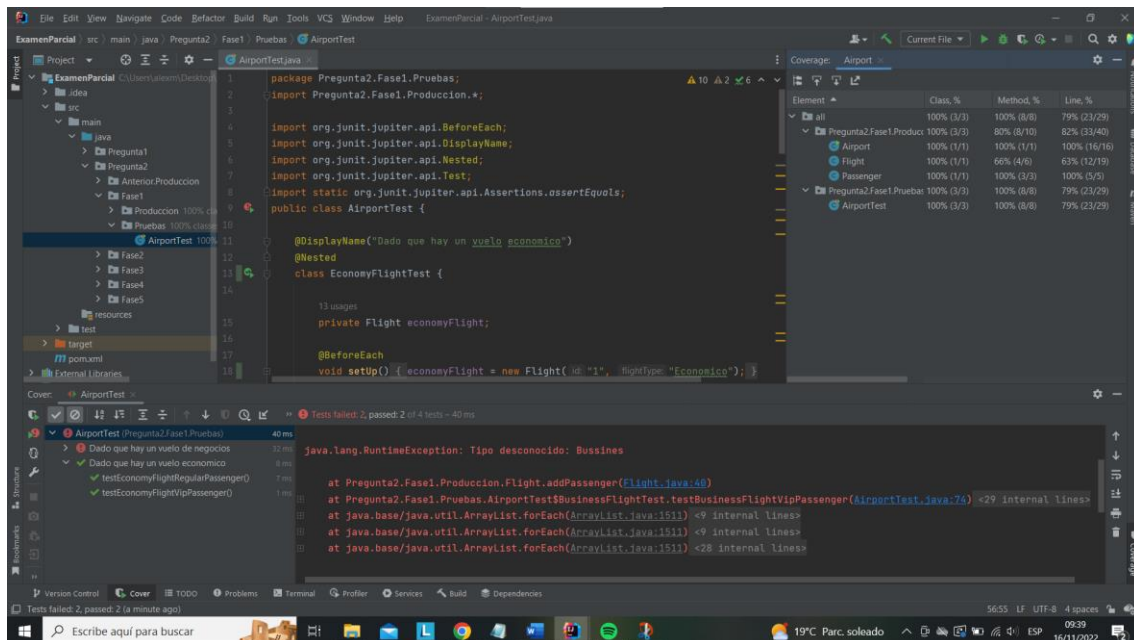
```
1 public class Airport {
2
3     public static void main(String[] args) {
4         Flight economyFlight = new Flight( id: "1", flightType: "Economico");
5         Flight businessFlight = new Flight( id: "2", flightType: "Negocios");
6
7         Passenger cesar = new Passenger( name: "Cesar", vip: 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 }
```

The Run window shows the output of the program:

```
C:\Users\alexm\.jdk\openjdk-19.0.1\bin\java.exe "-javaagent:C:\Program Files\JetB
Lista de pasajeros de vuelos de negocios:
Cesar
Lista de pasajeros de vuelos economicos:
Jessica
Process finished with exit code 0
```

2. 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%?

Al usar las pruebas de cobertura no tienen un resultado del 100%. Después de usar coverage como se vio en los enlaces del examen, en la consola se muestra un error que es Tipo desconocido: Bussines, ya que se tiene "Negocios" y "Bussines", lo correcto sería usar "Negocios" ya que en Flight se declara de esa forma.



### 3. ¿Por qué John tiene la necesidad de refactorizar la aplicación?

Hay varios factores, primero el error anterior de Business necesita ser cambiado, luego como se vio en clase de refactorización, se tienen 3 vuelos diferentes en una sola clase, si se quiere cambiar en un futuro o añadir mas tipos de vuelo sería muy engorroso, la clase sería muy voluminosa y de esta forma en un futuro podría generar errores ya que esta tendría que cambiarse y se necesitaría flightType en el cual tendríamos que añadir casos que no han sido ejecutados.

### 4. Revisa la Fase 2 de la evaluación y realiza la ejecución del programa y analiza los resultados.

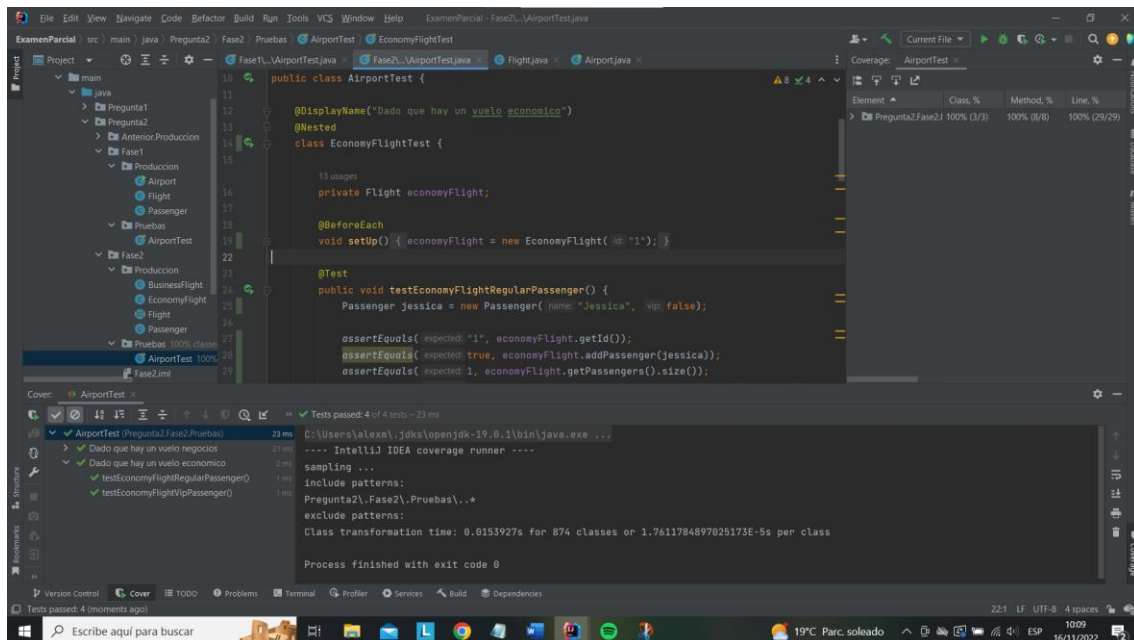
Como se puede ver en la imagen, en la fase dos se reemplaza la clase Flight original de la Fase1 por dos nuevas clases que son BusinessFlight y EconomyFlight, por ello en AirportTest en @BeforeEach

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

ya no se especifica el tipo de vuelo como en la Fase1

```
@BeforeEach
void setUp() {
    economyFlight = new Flight("1", "Economico");
}
```

Luego de ejecutarlo se ve que todas las pruebas pasaron exitosamente, ello quiere decir que la refactorización de la clase Flight no afectó y se ve una cobertura del 100%



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

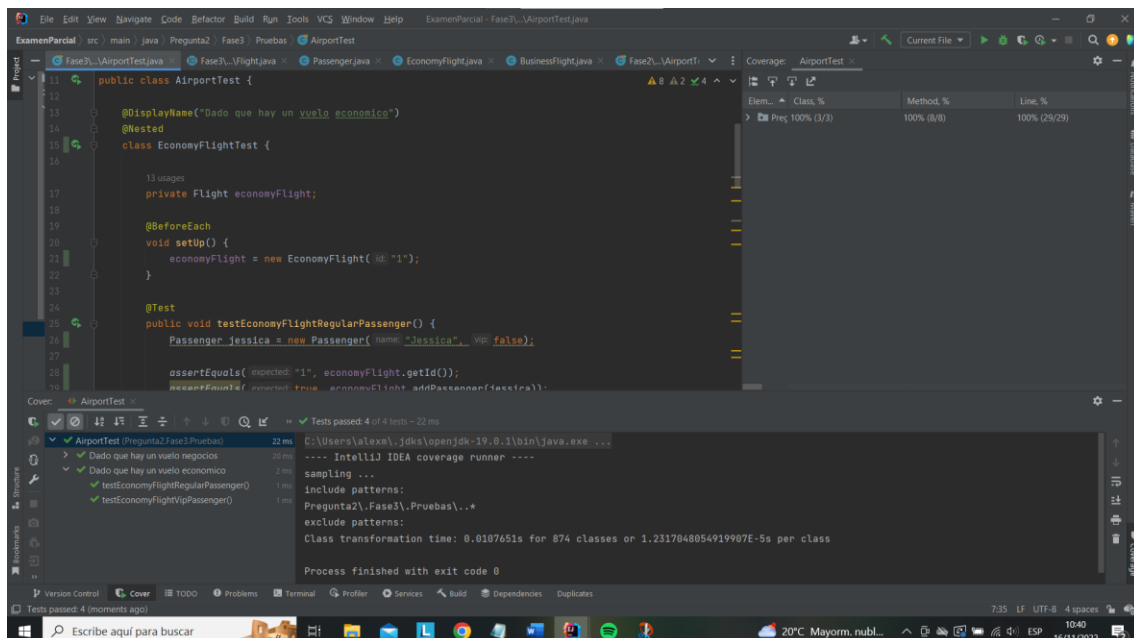
Y responde las siguientes preguntas:

- ¿Cuál es la cobertura del código?

Al realizar la prueba de cobertura esta tiene una cobertura del 100% como se ve en la imagen al lado derecho

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

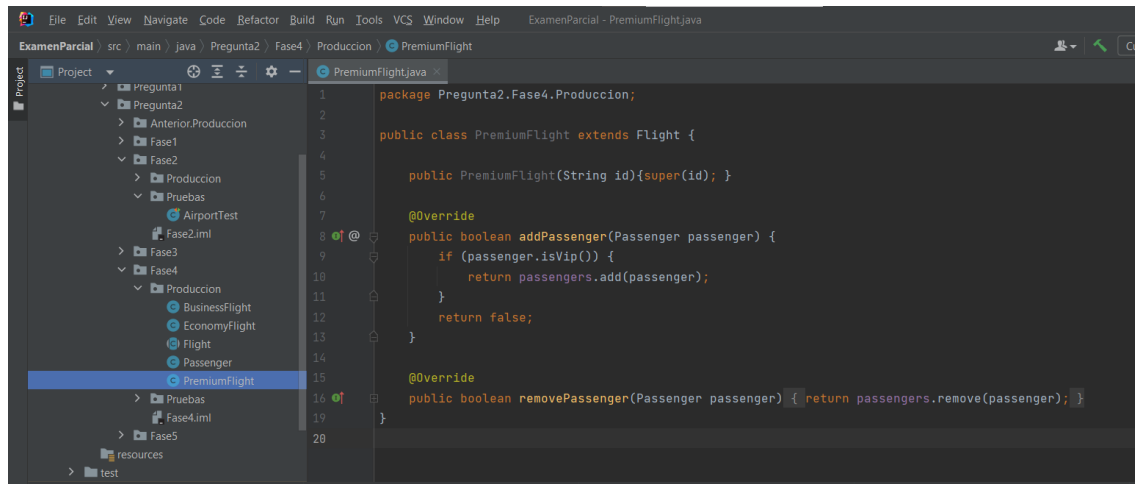
Si, hubieron algunos cambios como getPassengers a getPassengersList que se definió así en Flight en la Fase3, también la refactorización permitió que se tengas dos subclases por lo que ya no tenemos que especificar cada tipo de vuelo para cada caso, de esta manera el programa cumple con la escalabilidad.



6. ¿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.

La regla de tres consiste en que si hay 3 códigos que son similares entonces se debe refactorizar, estas similitudes también se cuentan como strikes, el problema de que los códigos se repitan es que si cambiamos la regla de uno entonces también tenemos que cambiarlo en todos los que contengan el duplicado, con la regla de tres podemos evitar la duplicación de código.

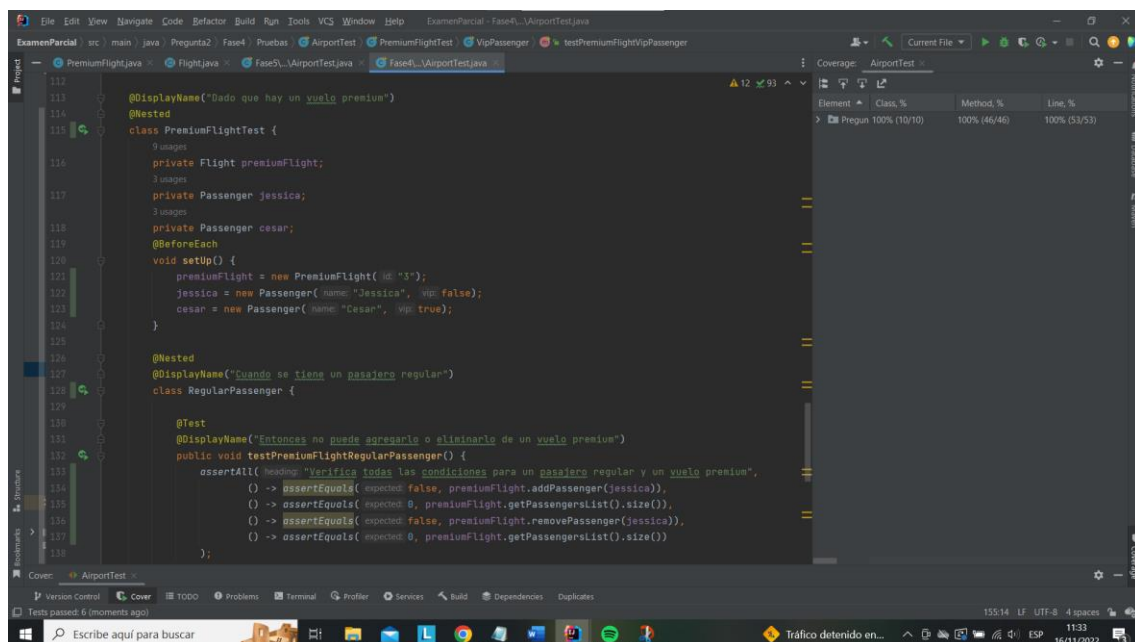
7. Escribe el diseño inicial de la clase llamada PremiumFlight y agrega a la Fase 4 en la carpeta producción.



```
1 package Pregunta2.Fase4.Produccion;
2
3 public class PremiumFlight extends Flight {
4
5     public PremiumFlight(String id){super(id); }
6
7     @Override
8     public boolean addPassenger(Passenger passenger) {
9         if (passenger.isVip()) {
10             return passengers.add(passenger);
11         }
12         return false;
13     }
14
15     @Override
16     public boolean removePassenger(Passenger passenger) { return passengers.remove(passenger); }
17 }
18
19
20
```

8. 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.

Siguiendo la distribución de las pruebas anteriores hacemos lo mismo para un vuelo Premium, una para cada tipo de pasajero (regular y VIP)



```
111 @DisplayName("Dado que hay un vuelo premium")
112 @Nested
113 class PremiumFlightTest {
114     private Flight premiumFlight;
115     private Passenger jessica;
116     private Passenger cesar;
117     @BeforeEach
118     void setUp() {
119         premiumFlight = new PremiumFlight("1");
120         jessica = new Passenger("Jessica", vip: false);
121         cesar = new Passenger("Cesar", vip: true);
122     }
123
124     @Nested
125     @DisplayName("Cuando se tiene un pasajero regular")
126     class RegularPassenger {
127
128         @Test
129         @DisplayName("Entonces no puede agregarlo o eliminarlo de un vuelo premium")
130         public void testPremiumFlightRegularPassenger() {
131             assertEquals("Verifica todas las condiciones para un pasajero regular y un vuelo premium",
132                 () -> assertEquals(expected: false, premiumFlight.addPassenger(jessica)),
133                 () -> assertEquals(expected: 0, premiumFlight.getPassengersList().size()),
134                 () -> assertEquals(expected: false, premiumFlight.removePassenger(jessica)),
135                 () -> assertEquals(expected: 0, premiumFlight.getPassengersList().size())
136             );
137         }
138     }
139 }
```

9. Agrega la lógica comercial solo para pasajeros VIP en la clase PremiumFlight. Guarda ese archivo en la carpeta Producción de la Fase 5.

```
package Pregunta2.Fase5.Produccion;

public class PremiumFlight extends Flight {

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

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

    @Override
    public boolean removePassenger(Passenger passenger) { return passengers.remove(passenger); }
}
```

10. 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.

Al igual que en pruebas anteriores usamos Set, de esta forma se puede verificar que el pasajero fue añadido una sola vez y no se vuelva a añadir.

```
@DisplayName("Dado que hay un vuelo premium")
@Nested
class PremiumFlightTest {
    1 usage
    private Flight premiumFlight;
    3 usage
    private Passenger jessica;
    5 usage
    private Passenger cesar;
    @BeforeEach
    void setUp() {
        premiumFlight = new PremiumFlight("3");
        jessica = new Passenger("Jessica", vip: false);
        cesar = new Passenger("Cesar", vip: true);
    }

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

        @Test
        @DisplayName("Entonces no puede agregarlo o eliminarlo de un vuelo premium")
        public void testPremiumFlightRegularPassenger() {
            assertEquals("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())
            );
        }
    }
}
```