

Examen parcial Desarrollo de Software CC-3S2

Alumno: Olivares Ventura Ricardo Leonardo

Código: 20192002A

Las respuesta a cada pregunta también lo he colocado como comentarios en el código

Enlace al repositorio de github:

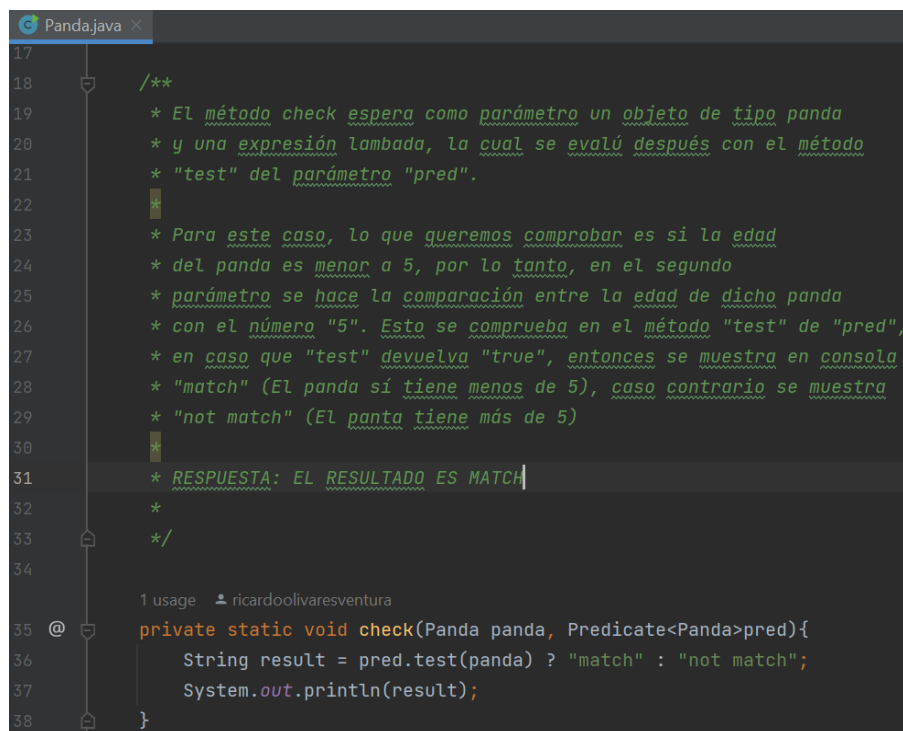
<https://github.com/ricardoolivaresventura/ExamenParcial-3S2>

Pregunta 1 (3 Puntos)

- ¿Cuál es el resultado de la siguiente clase?

```
import java.util.function.*;

public class Panda {
    int age;
    public static void main(String[] args) {
        Panda p1 = new Panda();
        p1.age = 1;
        check(p1, p -> p.age < 5);
    }
    private static void check(Panda panda,
    Predicate<Panda> pred) {
        String result =
        pred.test(panda) ? "match" : "not match";
        System.out.print(result);
    }
}
```



```
Panda.java x
17
18 /**
19  * El método check espera como parámetro un objeto de tipo panda
20  * y una expresión lambda, la cual se evalúa después con el método
21  * "test" del parámetro "pred".
22  *
23  * Para este caso, lo que queremos comprobar es si la edad
24  * del panda es menor a 5, por lo tanto, en el segundo
25  * parámetro se hace la comparación entre la edad de dicho panda
26  * con el número "5". Esto se comprueba en el método "test" de "pred",
27  * en caso que "test" devuelva "true", entonces se muestra en consola
28  * "match" (El panda sí tiene menos de 5), caso contrario se muestra
29  * "not match" (El panda tiene más de 5)
30  *
31  * RESPUESTA: EL RESULTADO ES MATCH
32  *
33  */
34
35 1 usage  ricardoolivaresventura
36 @ private static void check(Panda panda, Predicate<Panda>pred){
37     String result = pred.test(panda) ? "match" : "not match";
38     System.out.println(result);
39 }
```

El método check espera como parámetro un objeto de tipo panda y una expresión lambda, la cual se evalúa después con el método "test" del parámetro "pred". Para este caso, lo que queremos comprobar es si la edad del panda es menor a 5, por lo tanto, en el segundo parámetro se hace la comparación entre la edad de dicho panda con el número "5". Esto se comprueba en el método "test" de "pred", en caso que "test" devuelva "true", entonces se muestra en consola "match" (El panda sí tiene menos de 5), caso contrario se muestra "not match" (El panda tiene más de 5)

RESPUESTA: EL RESULTADO ES MATCH

- ¿Cuál es el resultado del siguiente código?

```
interface Climb {
    boolean isTooHigh(int height, int limit);
}

public class Climber {
    public static void main(String[] args) {
        check((h, m) -> h.append(m).isEmpty(), 5);
    }
    private static void check(Climb climb, int height) {
):    if (climb.isTooHigh(height, 10))
):    System.out.println("too high");
```

```

):    else
):    System.out.println("ok");
):    }
): }
```

```

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

🧑 ricardoolivaresventura
public class Climber {

    /**
     * RESPUESTA:
     * El código tiene un error en la línea donde se ha escrito
     * "h.append(m)". Este error ocurre, porque el método append
     * es de la clase String y el parámetro "h" es un entero al igual que
     * el parámetro m
     */

    🧑 ricardoolivaresventura
    public static void main(String[] args) {
        // |check((h, m) -> h.append(m).isEmpty(), 5);
    }
}

```

RESPUESTA:

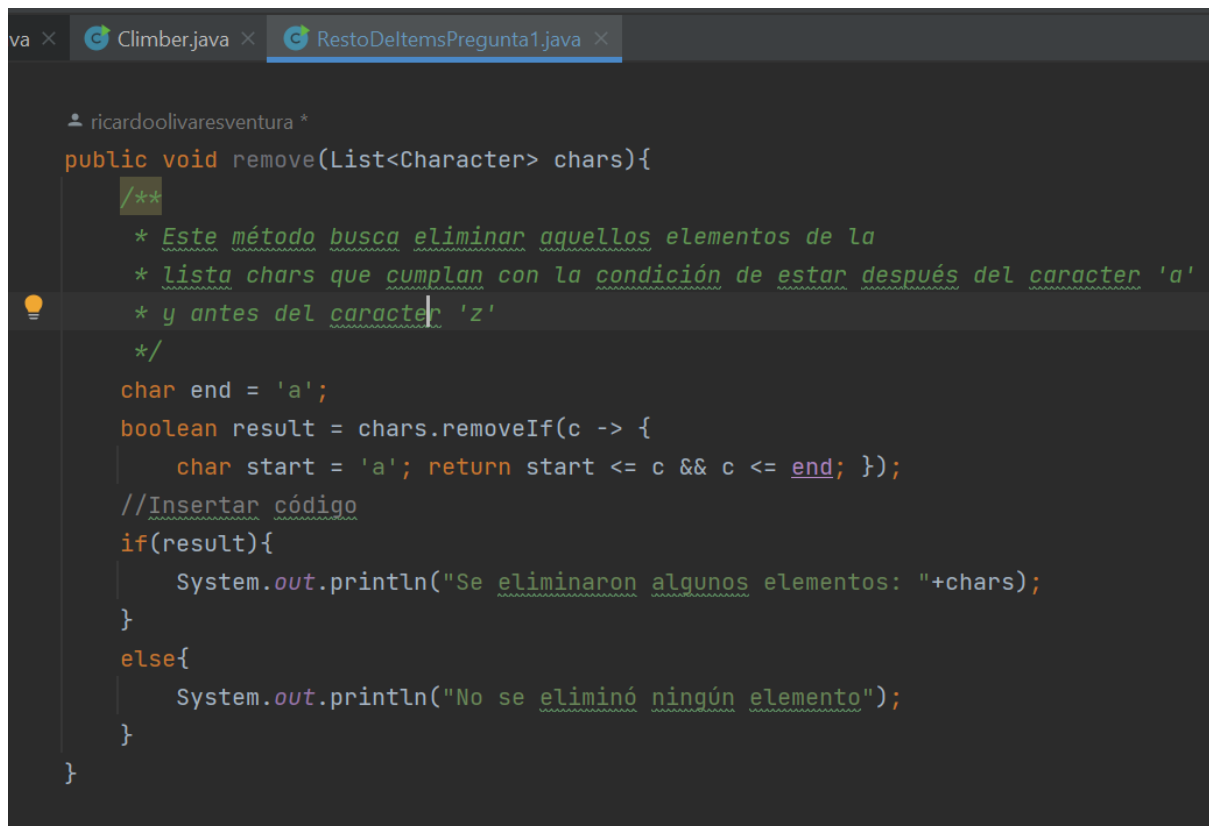
El código tiene un error en la línea donde se ha escrito "h.append(m)". Este error ocurre, porque el método append es de la clase String y el parámetro "h" es un entero al igual que el parámetro m

- 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  
}
```

Este método busca eliminar aquellos elementos de la lista chars que cumplan con la condición de estar después del carácter 'a' y antes del carácter 'z'.

Entonces, en la parte de //Insertar código, vi conveniente agregar un código que diga si se ha eliminado o no algunos elemento de la lista, esto lo podemos hacer con la ayuda del resultado que devuelve el método removeIf, ya que, este método devuelve true en caso se haya eliminado un elemento y false en caso NO se haya eliminado un elemento.



```
va × Climber.java × RestoDeltemsPregunta1.java ×  
ricardoolivaresventura *  
public void remove(List<Character> chars){  
    /**  
     * Este método busca eliminar aquellos elementos de la  
     * lista chars que cumplan con la condición de estar después del carácter 'a'  
     * y antes del carácter 'z'  
     */  
    char end = 'a';  
    boolean result = chars.removeIf(c -> {  
        char start = 'a'; return start <= c && c <= end; });  
    //Insertar código  
    if(result){  
        System.out.println("Se eliminaron algunos elementos: "+chars);  
    }  
    else{  
        System.out.println("No se eliminó ningún elemento");  
    }  
}
```

- ¿Qué puedes decir del siguiente código?

¿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
    }
}
```

La interfaz supplier no toma ningún argumento, pero genera algún valor de tipo T

A: En este caso, la interfaz supplier va a generar un valor de tipo Integer, el cual es "length"

B: Luego, obtenemos dicho valor usando el método .get()

C: En este caso es similar que en el caso "A", con la diferencia en que ahora el valor que se produce es el valor de "j"

D: Luego, se obtiene el valor de "j" con .get() y se muestra por consola

```
RestoDeltemsPregunta1.java ×
1 usage  ricardoolivaresventura *
35 public void methodWithSupplier(){
36     int length = 3;
37     for(int i=0; i<3; i++){
38         if(i%2 == 0){
39             //0,2
40             Supplier<Integer> supplier = () -> length; // A
41             System.out.println(supplier.get()); // B
42         }
43         else{
44             //1
45             int j = i;
46             Supplier<Integer> supplier = () -> j; // C
47             System.out.println(supplier.get()); // D
48         }
49     }
50
51     /**
52      * La interfaz supplier no toma ningún argumento, pero genera algún
53      * valor de tipo T
54      *
55      * A: En este caso, la interfaz supplier va a generar un valor de tipo Integer,
56      * el cual es "length"
57      * B: Luego, obtenemos dicho valor usando el método .get()
```

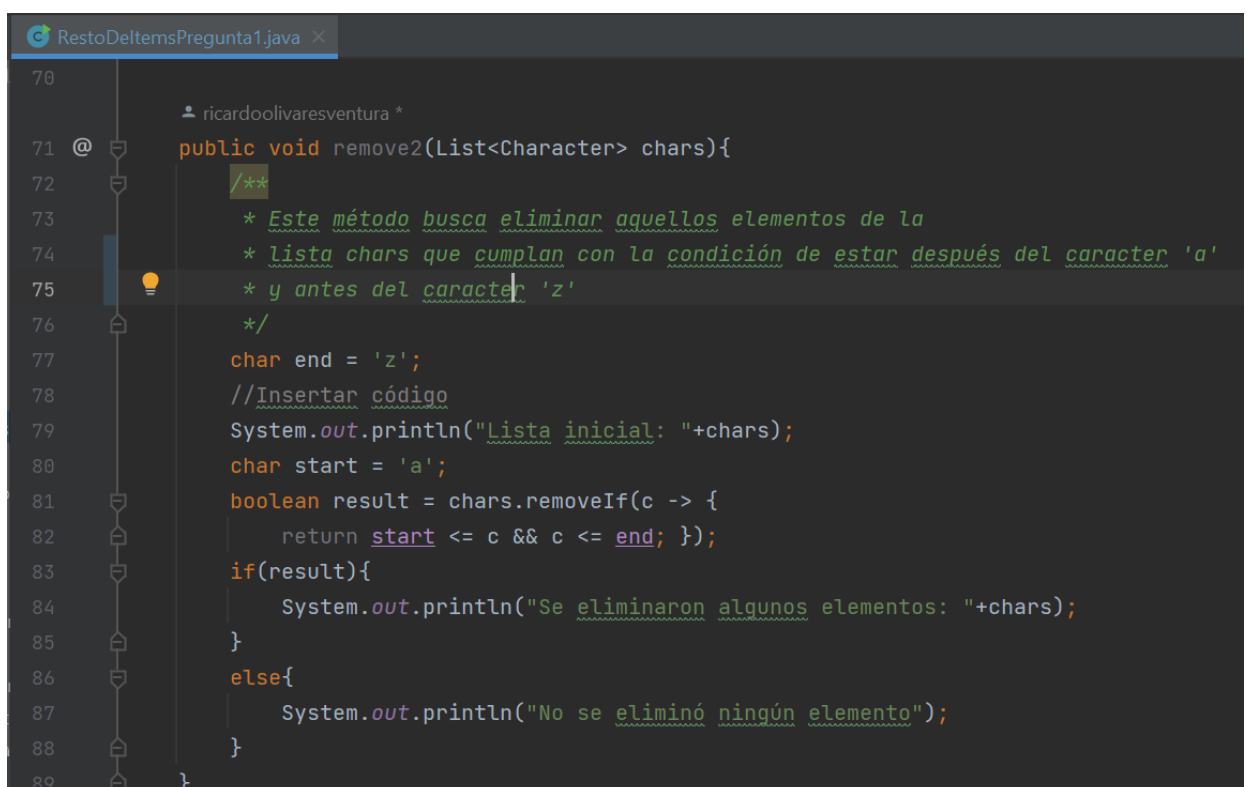
```
RestoDeltemsPregunta1.java ×
43     else{
44         //1
45         int j = i;
46         Supplier<Integer> supplier = () -> j; // C
47         System.out.println(supplier.get()); // D
48     }
49 }
50
51 /**
52  * La interfaz supplier no toma ningún argumento, pero genera algún
53  * valor de tipo T
54  *
55  * A: En este caso, la interfaz supplier va a generar un valor de tipo Integer,
56  * el cual es "length"
57  * B: Luego, obtenemos dicho valor usando el método .get()
58  *
59  * C: En este caso es similar que en el caso "A", con la diferencia
60  * en que ahora el valor que se produce es el valor de "j"
61  *
62  * D: Luego, se obtiene el valor de "j" con .get() y se muestra por consola
63  */
64
65 }
```

- Inserta código sin causar 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; });
}
```

Como ya lo mencionamos líneas arriba, este método busca eliminar aquellos elementos de la lista chars que cumplan con la condición de estar después del carácter 'a' y antes del carácter 'z'.

Entonces, en la parte de //Insertar código, vi conveniente imprimir la lista inicial y además, moví la declaración del **char start = 'a'** afuera del **removeIf** para evitar que gastar memoria en la declaración, ya que, estaríamos declarando la variable en cada iteración del **removeIf**. Además, también si **removeIf** devuelve true, entonces imprimimos el resultado final de la lista para visualizar cómo quedó luego de eliminar los elementos pertinentes, caso contrario solo imprimimos un mensaje que diga que no se eliminó ningún elemento

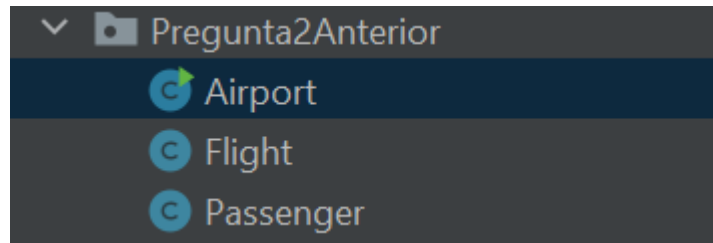


```
RestoDelItemsPregunta1.java x
70
71 @ ricardoolivaresventura *
72 public void remove2(List<Character> chars){
73     /**
74      * Este método busca eliminar aquellos elementos de la
75      * lista chars que cumplan con la condición de estar después del carácter 'a'
76      * y antes del carácter 'z'
77      */
78     char end = 'z';
79     //Insertar código
80     System.out.println("Lista inicial: "+chars);
81     char start = 'a';
82     boolean result = chars.removeIf(c -> {
83         return start <= c && c <= end; });
84     if(result){
85         System.out.println("Se eliminaron algunos elementos: "+chars);
86     }
87     else{
88         System.out.println("No se eliminó ningún elemento");
89     }
90 }
```

Pregunta 2 (12 puntos)

- 2.1) Ejecuta el programa y presenta los resultados y explica qué sucede

El código correspondiente a esta ítem la he colocado dentro de la siguiente carpeta:



```
1 package Pregunta2Anterior;
2
3 public class Airport {
4     public static void main(String[] args) {
5         Flight economyFlight = new Flight( id: "1", flightType: "Economico");
6         Flight businessFlight = new Flight( id: "2", flightType: "Negocios");
7
8         Passenger cesar = new Passenger( name: "Cesar", vip: true);
9         Passenger jessica = new Passenger( name: "Jessica", vip: false);
10
11         businessFlight.addPassenger(cesar);
12         businessFlight.removePassenger(cesar);
13         businessFlight.addPassenger(jessica);
14         economyFlight.addPassenger(jessica);
15
16         System.out.println("Lista de pasajeros de vuelos de negocios:");
17         for (Passenger passenger : businessFlight.getPassengersList()) {
18             System.out.println(passenger.getName());
19         }
20
21         System.out.println("Lista de pasajeros de vuelos economicos:");
22         for (Passenger passenger : economyFlight.getPassengersList()) {
23             System.out.println(passenger.getName());
24         }
25     }
26 }
```



```
25
26  /**
27   * Pregunta 2.1. Ejecuta el programa y presenta los resultados y explica
28   * qué sucede.
29   *
30   * El resultado por consola es:
31   * -----
32   * Lista de pasajeros de vuelos de negocios:
33   * Cesar
34   * Lista de pasajeros de vuelos economicos:
35   * Jessica
36   * -----
37   *
38   * Esto ocurrió porque César es un pasajero VIP, en cambio Jessica no.
39   * Por este motivo, a César sí se le pudo agregar al vuelo de Negocios,
40   * en cambio a Jessica no se le pudo agregar porque no es VIP.
41   *
42   * Además, vemos que en el código se intentó eliminar a César del vuelo,
43   * pero como César es VIP, entonces no se le pudo eliminar.
44   *
45   * También, vemos que a Jessica sí se le pudo agregar al vuelo económico, ya que,
46   * para este tipo de vuelo no hay ninguna restricción
47   *
48   */
49 }
50 }
```

Respuesta:

El resultado por consola es:

Lista de pasajeros de vuelos de negocios: César

Lista de pasajeros de vuelos económicos: Jessica

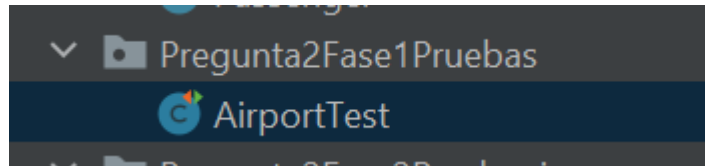
Esto ocurrió porque César es un pasajero VIP, en cambio Jessica no. Por este motivo, a César sí se le pudo agregar al vuelo de Negocios, en cambio a Jessica no se le pudo agregar porque no es VIP

Además, vemos que en el código se intentó eliminar a César del vuelo, pero como él es VIP, entonces no se le puede eliminar de un vuelo

También, vemos que a Jessica sí se le pudo agregar a un vuelo económico, ya que, para este tipo de vuelo no hay ninguna restricción

- **Pregunta 2.2. Si ejecutamos las pruebas con cobertura desde IntelliJ IDEA, ¿cuáles son los resultados que se muestran? ¿Por qué crees que la cobertura del código no es del 100%?**

El código correspondiente a este ítem lo he colocado en la siguiente carpeta



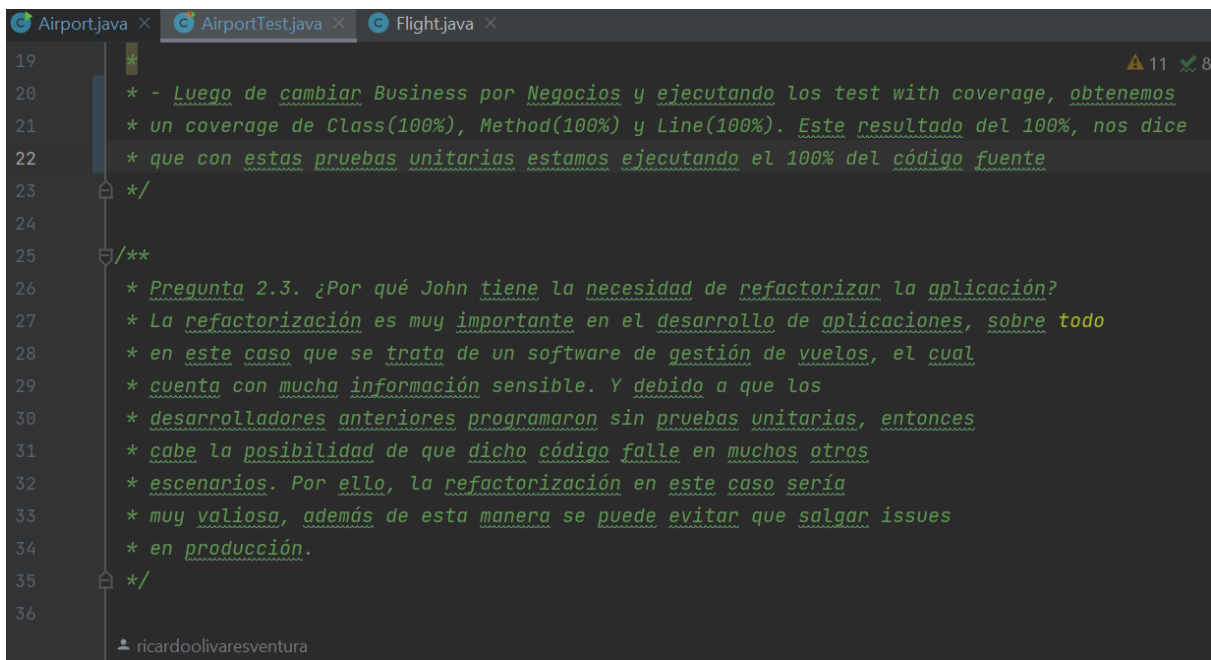
| Element | Class, % | Method, % | Line, % |
|-----------------------|------------|------------|--------------|
| Pregunta2Fase1Pruebas | 100% (3/3) | 100% (8/8) | 100% (29/29) |
| AirportTest | 100% (3/3) | 100% (8/8) | 100% (29/29) |

RESPUESTA:

Al inicio el test fallaba, ya que, se estaba creando un vuelo de tipo “Business”, pero la clase Flight solo soporta en español (Económico y Negocios)

Luego de cambiar Business por Negocios y ejecutando los tests with coverage, obtenemos un coverage de Class(100%), Method(100%) y Line(100%). Este resultado del 100%, nos dice que con estas pruebas unitarias estamos ejecutando el 100% del código fuente

- **Pregunta 2.3 ¿Por qué John tiene la necesidad de refactorizar la aplicación?**



```
19
20 * - Luego de cambiar Business por Negocios y ejecutando los test with coverage, obtenemos
21 * un coverage de Class(100%), Method(100%) y Line(100%). Este resultado del 100%, nos dice
22 * que con estas pruebas unitarias estamos ejecutando el 100% del código fuente
23 */
24
25 /**
26  * Pregunta 2.3. ¿Por qué John tiene la necesidad de refactorizar la aplicación?
27  * La refactorización es muy importante en el desarrollo de aplicaciones, sobre todo
28  * en este caso que se trata de un software de gestión de vuelos, el cual
29  * cuenta con mucha información sensible. Y debido a que los
30  * desarrolladores anteriores programaron sin pruebas unitarias, entonces
31  * cabe la posibilidad de que dicho código falle en muchos otros
32  * escenarios. Por ello, la refactorización en este caso sería
33  * muy valiosa, además de esta manera se puede evitar que salgan issues
34  * en producción.
35 */
36
```

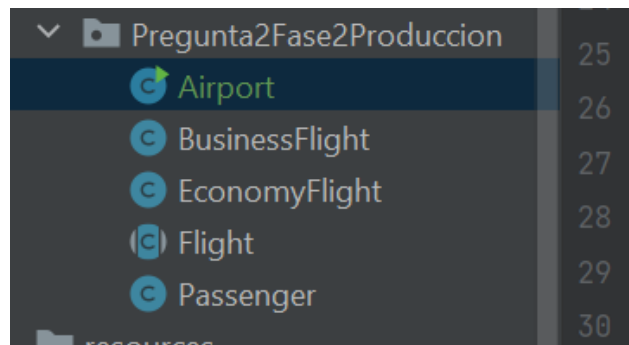
ricardoolivaresventura

Respuesta

La refactorización es muy importante en el desarrollo de aplicaciones, sobre todo en este caso que se trata de un software de gestión de vuelos, el cual cuenta con mucha información sensible. Y debido a que los desarrolladores anteriores programaron sin pruebas unitarias, entonces cabe la posibilidad de que dicho código falle en muchos otros escenarios. Por ello, la refactorización en este caso sería muy valioso, además de esta manera se puede evitar que salgan issues en producción.

- **Pregunta 2.4. Revisa la Fase 2 de la evaluación y realiza la ejecución del programa y analiza los resultados**

El código correspondiente a este ítem se puede encontrar en la siguiente carpeta:



Para este ítem, se tuvo que crear la clase Airport basándonos en la estructura de la clase Airport de la carpeta Anterior de la pregunta 2.

```
public class Airport {  
    public static void main(String[] args) {  
        Flight economyFlight = new EconomyFlight( id: "1");  
        Flight businessFlight = new BusinessFlight( id: "2");  
  
        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.getPassengers()) {  
            System.out.println(passenger.getName());  
        }  
  
        System.out.println("Lista de pasajeros de vuelos economicos:");  
        for (Passenger passenger : economyFlight.getPassengers()) {  
            System.out.println(passenger.getName());  
        }  
    }  
}
```

Respuesta:

La diferencia entre esta clase Airport de la Fase 2 y la clase Airport de la carpeta Anterior de la pregunta 2, es que en esta Fase 2 se está utilizando polimorfismo, siendo la Flight la clase abstracta base.

Además, tenemos clases, que heredan de la clase Flight, para cada uno de los tipos de vuelos. Por ejemplo, tenemos la clase BusinessFlight y EconomyFlight.

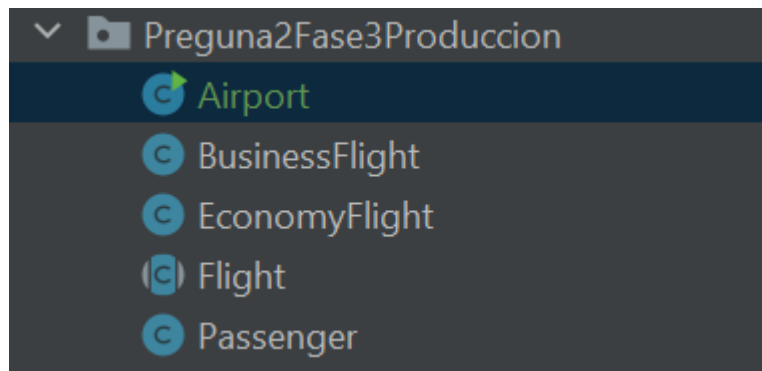
Por ello, al momento de instanciar un vuelo económico, simplemente instanciamos un objeto de la clase EconomyFlight y ya no tenemos que pasarle como parámetro un string que indique el tipo de vuelo. Y para el caso de un vuelo de negocios, tenemos que instanciar un objeto de la clase BusinessFlight y tampoco tenemos que pasarle un string que indique el tipo de vuelo.

Esto hace que el código sea más mantenible, porque podemos tener la lógica de cada tipo de vuelo separada en su propia clase y además, dentro de la clase Flight, podemos tener todas aquellas funciones, atributos, etc que compartan los vuelos, para que de esta manera podamos reutilizar código.

```
Pregunta2Fase2Produccion\Airport.java x Passenger.java x Flight.java x BusinessFlight.java x EconomyFlight.java x
1 package Pregunta2Fase2Produccion;
2
3 /**
4  * Pregunta 2.4 Revisa la Fase 2 de la evaluación y realiza
5  * La diferencia entre esta clase Airport de la Fase 2 y la clase Airport de
6  * la carpeta Anterior de la pregunta 2, es que en esta Fase 2 se está utilizando
7  * polimorfismo, siendo la Flight la clase abstracta base.
8  *
9  * Además, tenemos clases, que heredan de la clase Flight, para cada uno de los
10 * tipos de vuelos. Por ejemplo, tenemos la clase BusinessFlight y EconomyFlight.
11 *
12 * Por ello, al momento de instanciar un vuelo económico, simplemente instanciamos
13 * un objeto de la clase EconomyFlight y ya no tenemos que pasarle como parámetro un
14 * string que indique el tipo de vuelo. Y para el caso de un vuelo de negocios,
15 * tenemos que instanciar un objeto de la clase BusinessFlight y tampoco tenemos
16 * que pasarle un string que indique el tipo de vuelo.
17 *
18 * Esto hace que el código sea más mantenible, porque podemos tener la lógica de
19 * cada tipo de vuelo separada en su propia clase y además, dentro de la clase Flight,
20 * podemos tener todas aquellas funciones, atributos, etc que compartan los vuelos,
21 * para que de esta manera podamos reutilizar código
22 */
```

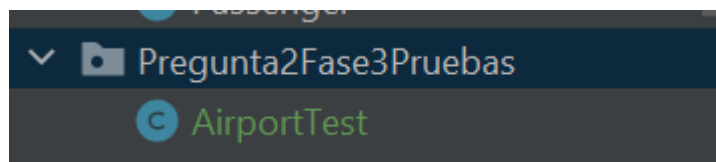
- **Pregunta 2.5. La refactorización y los cambios de la API de propagan a las pruebas. Reescribe el archivo Airport Test de la carpeta Fase 3**

El código correspondiente a este ítem lo puedes encontrar en:



La clase Airport que está en esa carpeta, es la misma que se implementó en el ítem anterior (2.4)

Y el archivo de test se puede encontrar aquí:



En este caso debemos reescribir las pruebas unitarias de tal manera que dichas pruebas pasen satisfactoriamente con el polimorfismo que se implementó durante la refactorización

En la función setUp de la clase EconomyFlightTest, se cambió el new Flight("1", "Economico") por new EconomyFlight("1"). Y en la función setUp de la clase BusinessFlightTest, se cambió el new Flight("2", "Negocios") por new BusinessFlight("2"). De esta manera ahora se soporta el polimorfismo

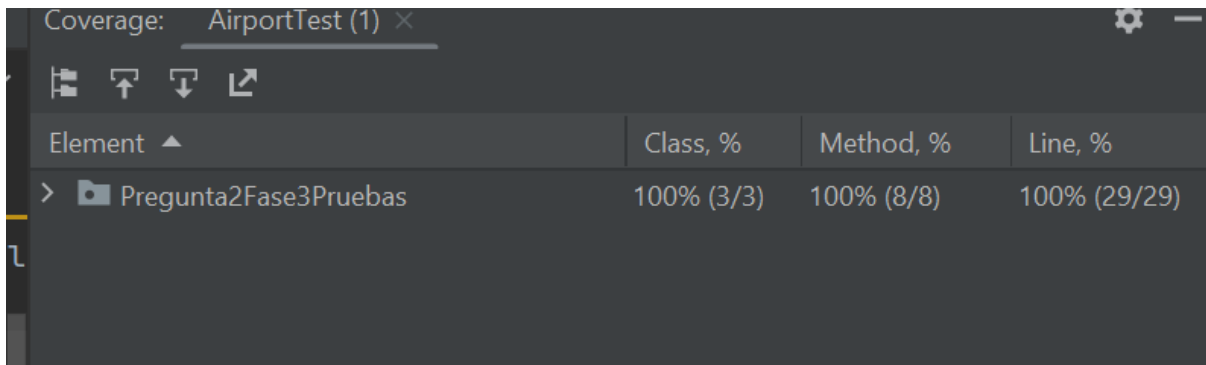
```
public class AirportTest {  
    // Refactorización de la clase AirportTest. Pregunta 2.5  
    @DisplayName("Dado que hay un vuelo economico")  
    @Nested  
    class EconomyFlightTest {  
        13 usages  
        private Flight economyFlight;  
  
        @BeforeEach  
        void setUp(){  
            economyFlight = new EconomyFlight( id: "1");  
        }  
  
        @Test  
        public void testEconomyFlightRegularPassenger() {  
            Passenger jessica = new Passenger( name: "Jessica", vip: false);
```

```
@DisplayName("Dado que hay un vuelo de negocios")  
@Nested  
class BusinessFlightTest {  
    9 usages  
    private Flight businessFlight;  
  
    @BeforeEach  
    void setUp() {  
        businessFlight = new BusinessFlight( id: "2");  
    }  
}
```

Y responder a las siguientes preguntas

- ¿Cuál es la cobertura del código ?
La cobertura es del 100%

Coverage: AirportTest (1) ×



| Element ▲ | Class, % | Method, % | Line, % |
|-------------------------|------------|------------|--------------|
| > Pregunta2Fase3Pruebas | 100% (3/3) | 100% (8/8) | 100% (29/29) |

Lo cual quiere decir que en estas pruebas se está ejecutando todo el código fuente de la Fase 3

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

Sí, claro que sí. Porque ahora tenemos una clase base abstracta, la cual es la clase Flight, de la cual deberán heredar las clases de los distintos tipos de vuelo.

Por ejemplo, al momento de instanciar un vuelo económico, simplemente instanciamos un objeto de la clase EconomyFlight y ya no tenemos que pasarle como parámetro un string que indique el tipo de vuelo. Y para el caso de un vuelo de negocios, tenemos que instanciar un objeto de la clase BusinessFlight y tampoco tenemos que pasarle un string que indique el tipo de vuelo.

Esto hace que el código sea más mantenible, porque podemos tener la lógica de cada tipo de vuelo separada en su propia clase y además, dentro de la clase Flight, podemos tener todas aquellas funciones, atributos, etc que compartan los vuelos, para que de esta manera podamos reutilizar código.

- **Pregunta 2.6. puntos):¿En qué consiste está regla relacionada a la refactorización?. Evita utilizar y copiar respuestas de internet. Explica cómo se relaciona al problema dado en la evaluación.**

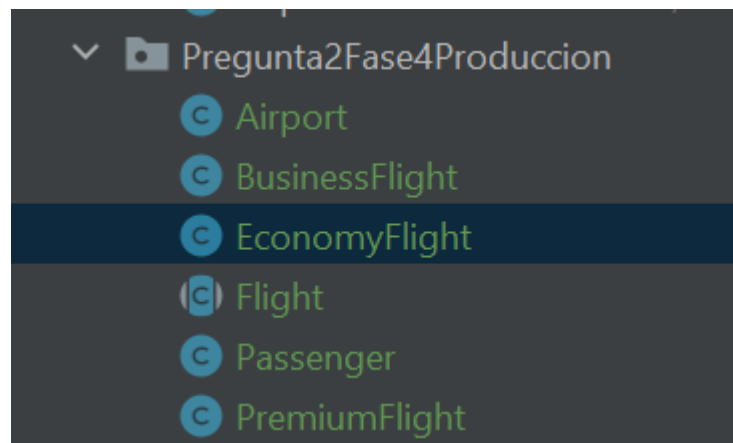
Esta regla sostiene que cuando se tiene una pieza de código que se repite dos veces en todo el código fuente, entonces no es necesario refactorizar, pero cuando dicha pieza de código se repite 3 veces, entonces ahí sí es necesario refactorizar para poder realizar un código más mantenible y limpio.

Esto se debe a que la duplicación de código es considerada una mala práctica en la programación, ya que, cuando quieres realizar un cambio en dicho código, entonces debes buscar en todas las partes del código fuente donde está dicha implementación y realizar el cambio, en cambio si tienes el código en un solo archivo y luego lo importas en otros archivos, entonces cuando desees realizar un cambio lo tendrás que realizar en ese único archivo y el cambio se reflejará en todos los lugares donde se utiliza.

Esto puede ser aplicado en el archivo de AirportTest, ya que, el test para el EconomyFlight y el test para el BusinessFlight tienen piezas de código muy similares y como ahora se debe implementar los tests para el PremiumFlight, entonces cada vez se volverá menos mantenible, y peor aún si en algún futuro hay muchos tipos de viajes

- **.Pregunta 2.7. Escribe el diseño inicial de la clase llamada PremiumFlight y agrega a la Fase 4 en la carpeta producción.**

El código de la clase PremiumFlight se puede encontrar en la siguiente carpeta



En el método addPassenger, debemos comprobar si el usuario es VIP, ya que, solo los usuarios VIP pueden acceder a los vuelos PREMIUM

En el método removePassenger de la clase PremiumFlight siempre devolvemos true, ya que, cualquier pasajero que esté en un vuelo premium puede ser removido

```
public class PremiumFlight extends Flight{

    // Diseño inicial de la clase PremiumFlight. Pregunta 2.7

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

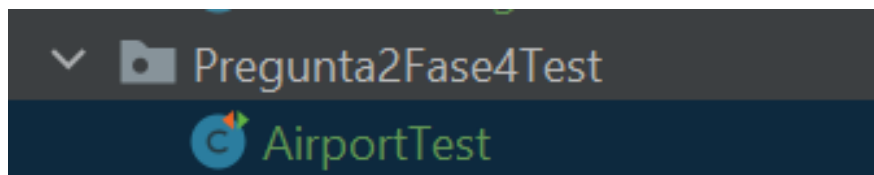
    @Override
    public boolean addPassenger(Passenger passenger) {
        if(passenger.isVip()){
            return passengers.add(passenger);
        }
        return false;
    }

    /**
     * Cualquier pasajero que esté en un vuelo premium puede ser removido
     */
    @Override
    public boolean removePassenger(Passenger passenger) {
        return true;
    }
}
```

- **Pregunta 2.8. Escribe el diseño inicial de la clase llamada PremiumFlight y agrega a la Fase 4 en la carpeta producción.**

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.

El código correspondiente a este test estará en la siguiente carpeta:



```
// Completa la prueba para PremiumFlight de acuerdo a la logica comercial dada. Pregunta 2.8

@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
```

Caso para un pasajero regular y un vuelo PREMIUM

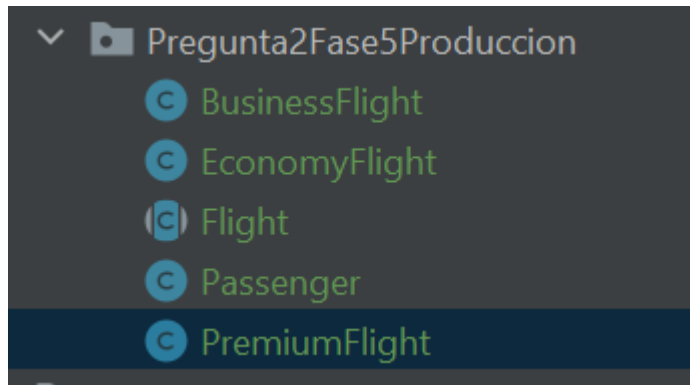
```
@Nested
@DisplayName("Cuando tenemos un pasajero regular")
class RegularPassenger{
    @Test
    @DisplayName("Entonces no podemos agregarlo o eliminarlo de un vuelo PREMIUM")
    public void testPremiumFlightRegularPassenger(){
        assertAll( heading: "Verifica todas las condiciones para un pasajero regular y un vuelo P
            () -> assertEquals( expected: false, premiumFlight.addPassenger(jessica)),
            () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size()),
            () -> assertEquals( expected: false, premiumFlight.removePassenger(jessica)),
            () -> assertEquals( expected: 0, premiumFlight.getPassengersList().size())
        );
    }
}
```

Caso para un pasajero VIP y un vuelo PREMIUM

```
@Nested
@DisplayName("Cuando tenemos un pasajero VIP")
class VipPassenger{
    @Test
    @DisplayName("Sí podemos agregarlo y eliminarlo de un vuelo PREMIUM")
    public void testPremiumFlightVipPassenger(){
        assertAll( heading: "Verifica todas las condiciones para un pasajero VIP y un vuelo PREM
            () -> assertEquals( expected: true, premiumFlight.addPassenger(cesar)),
            () -> assertEquals( expected: 1, premiumFlight.getPassengersList().size()),
            () -> assertEquals( expected: true, premiumFlight.removePassenger(cesar)),
            () -> assertEquals( expected: 1, premiumFlight.getPassengersList().size())
        );
    }
}
```

- **Pregunta 2.9 Agrega la lógica comercial solo para pasajeros VIP en la clase PremiumFlight. Guarda ese archivo en la carpeta Producción de la Fase 5.**

El código de este ítem se puede encontrar en la carpeta Pregunta2Fase5Produccion



PREGUNTA 2.9

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

*/

ricardoolivaresventura *

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

1 usage ricardoolivaresventura

```
    public PremiumFlight(String id) { super(id); }
```

ricardoolivaresventura

```
@Override
```

```
    public boolean addPassenger(Passenger passenger) {
```

```
        if(passenger.isVip()){
```

```
            return passengers.add(passenger);
```

```
        }
```

```
        return false;
```

```
    }
```

ricardoolivaresventura *

```
@Override
```

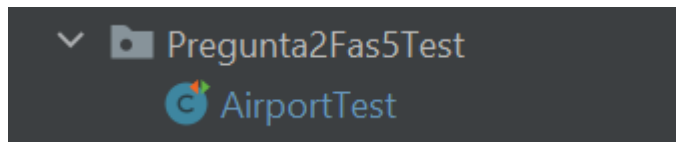
```
    public boolean removePassenger(Passenger passenger) {
```

```
        return true;
```

```
    }
```

```
}
```

- **Pregunta 2.10: El código de esta prueba se encuentra en la siguiente carpeta:**



PREGUNTA 3:

Para esta pregunta debemos considerar que un buen código de prueba se caracteriza por lo siguiente:

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

Además, toda la información, especialmente las entradas y las afirmaciones deben ser lo suficientemente claras.

- **Pregunta 3.1. ¿Cuáles son los problemas de este código de prueba?**

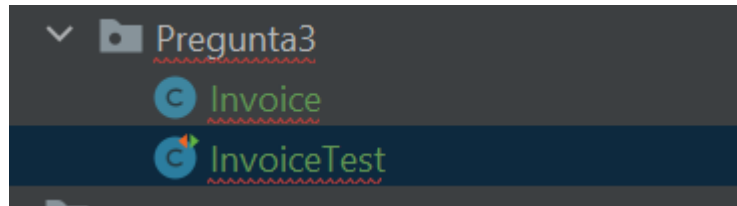
Este código de prueba presenta varios problemas, como los siguientes:

- ❖ El nombre "test1" no es lo suficientemente descriptivo para poder saber qué es lo que está comprobando dicha prueba unitaria
- ❖ Se pudo haber agregado el decorador DisplayName para poder colocar un mensaje que indique qué hace dicha prueba, pero vemos que no se ha colocado.
- ❖ Además, la afirmación que vemos, el cual es 250, no es lo suficientemente claro, porque si un programador ve dicha prueba, no sabría el por qué el valor debe ser 250 a menos que la función calculate tenga un nombre mucho más descriptivo, ya que, de esta

manera se podría conocer lo que se está probando sin necesidad de ir al código fuente

- **Pregunta 3.2. Escriba una versión más legible de este código de prueba. Recuerda llamarlo InvoiceTest.java**

El código de este ítem se encuentra en el siguiente archivo:



```
class InvoiceTestValue {
    2 usages
    private CustomerType customerType;
    3 usages
    private double value;
    2 usages
    private String country;
    2 usages
    private LocalDate createdAt;
    2 usages
    private String customerName;

    @BeforeEach
    void setUp(){
        customerType = CustomerType.COMPANY;
        value = 2500;
        country = "NL";
        createdAt = LocalDate.now();
        customerName = "Ricardo";
    }

    @Test
    @DisplayName("Obtener el valor de dicho recibo")
    void calculateTheInvoiceValue(){
        double ratio = 0.1;
        double expectedValue = value * ratio;
        Invoice invoice = new Invoice(value, country, customerType, createdAt, customerName);
        double receivedValue = invoice.calculate();
        assertEquals(expectedValue, receivedValue);
    }
}
```

- **Pregunta 2.3: Implementa el InvoiceBuilder**

En la clase Invoice se agregó dos atributos más, uno es el createdAt de tipo LocalDate, el cual indica la fecha de creación del recibo y el otro es el customerName, el cual es el nombre de la persona o de la empresa a la se emitirá el recibo

```
1 usage
private final LocalDate createdAt;
1 usage
private final String customerName;
2 usages
public Invoice(double value, String country, CustomerType customerType, LocalDate createdAt,
               String customerName){
    this.value = value;
    this.country = country;
    this.customerType = customerType;
    this.createdAt = createdAt;
    this.customerName = customerName;
}
```

Luego, en la clase InvoiceBuilder se agregó dos métodos más, uno para el createdAt y el otro para el customerName

```
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;
}

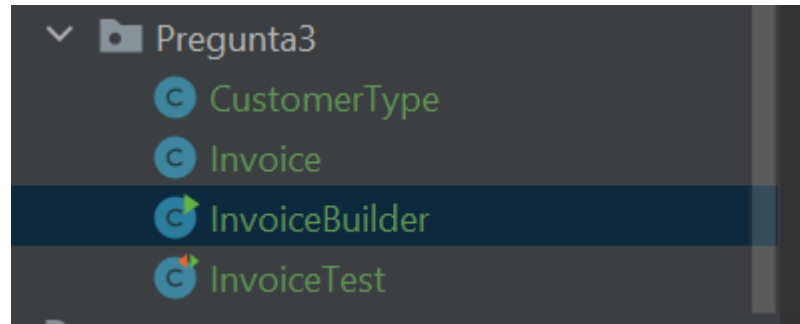
public InvoiceBuilder withCreationData(LocalDate createAt){
    this.createAt = createAt;
    return this;
}

public InvoiceBuilder withCustomerName(String customerName){
    this.customerName = customerName;
    return this;
}

public Invoice build() {
    return new Invoice(value, country, customerType, createAt, customerName);
}
```


- **Pregunta 3.4: Escribe en una línea una factura compleja. Muestra los resultados**

El código de este ítem está en el siguiente archivo:



```
public static void main(String[] args) {  
    Invoice invoice = new InvoiceBuilder()  
        .asCompany()  
        .withCustomerName("FACEBOOK")  
        .withCreationDate(LocalDate.now())  
        .withAValueOf(45348)  
        .withCountry("UK")  
        .build();  
  
    invoice.showInformation();  
}
```

- **Pregunta 3.5: Agrega este listado en el código anterior y muestra los resultados**

En este ítem se agregó el método `fromTheUs`, el cual cambia el `country` del `Invoice`, y el método `anyCompany`, que establece el `customerType` en `COMPANY`, adicionalmente se consideró que este método también cambiaría el `customerName` a "Any Company"

```
}  
  
public Invoice anyCompany(){  
    return new Invoice(value, country, CustomerType.COMPANY, LocalDate.now(), customerName: "Any Company");  
}  
  
2 usages  
public Invoice fromTheUS(){  
    return new Invoice(value, country: "US", customerType, createdAt, customerName);  
}  
  
1 usage  
public Invoice build() {  
    return new Invoice(value, country, customerType, createdAt, customerName);  
}  
  
public static void main(String[] args) {
```

```
public static void main(String[] args) {  
    InvoiceBuilder invoiceBuilder = new InvoiceBuilder();  
    Invoice invoice = new InvoiceBuilder()  
        .asCompany()  
        .withCustomerName("FACEBOOK")  
        .withCreationDate(LocalDate.now())  
        .withAValueOf(45348)  
        .withCountry("UK")  
        .build();  
  
    invoice.showInformation();  
  
    System.out.println("Create company from the US-----");  
    invoice = invoiceBuilder.fromTheUS();  
  
    invoice.showInformation();  
  
    System.out.println("Create any company-----");  
    invoice = invoiceBuilder.anyCompany();  
  
    invoice.showInformation();  
}
```

```
Customer Name: FACEBOOK
Value: 45348.0
Country: UK
Customer type: COMPANY
Created At: 2022-11-13
Create company from the US-----
Customer Name: FACEBOOK
Value: 45348.0
Country: US
Customer type: COMPANY
Created At: 2022-11-13
Create any company-----
Customer Name: Any Company
Value: 45348.0
Country: UK
Customer type: COMPANY
Created At: 2022-11-13
```

```
Process finished with exit code 0
```

```
|
```

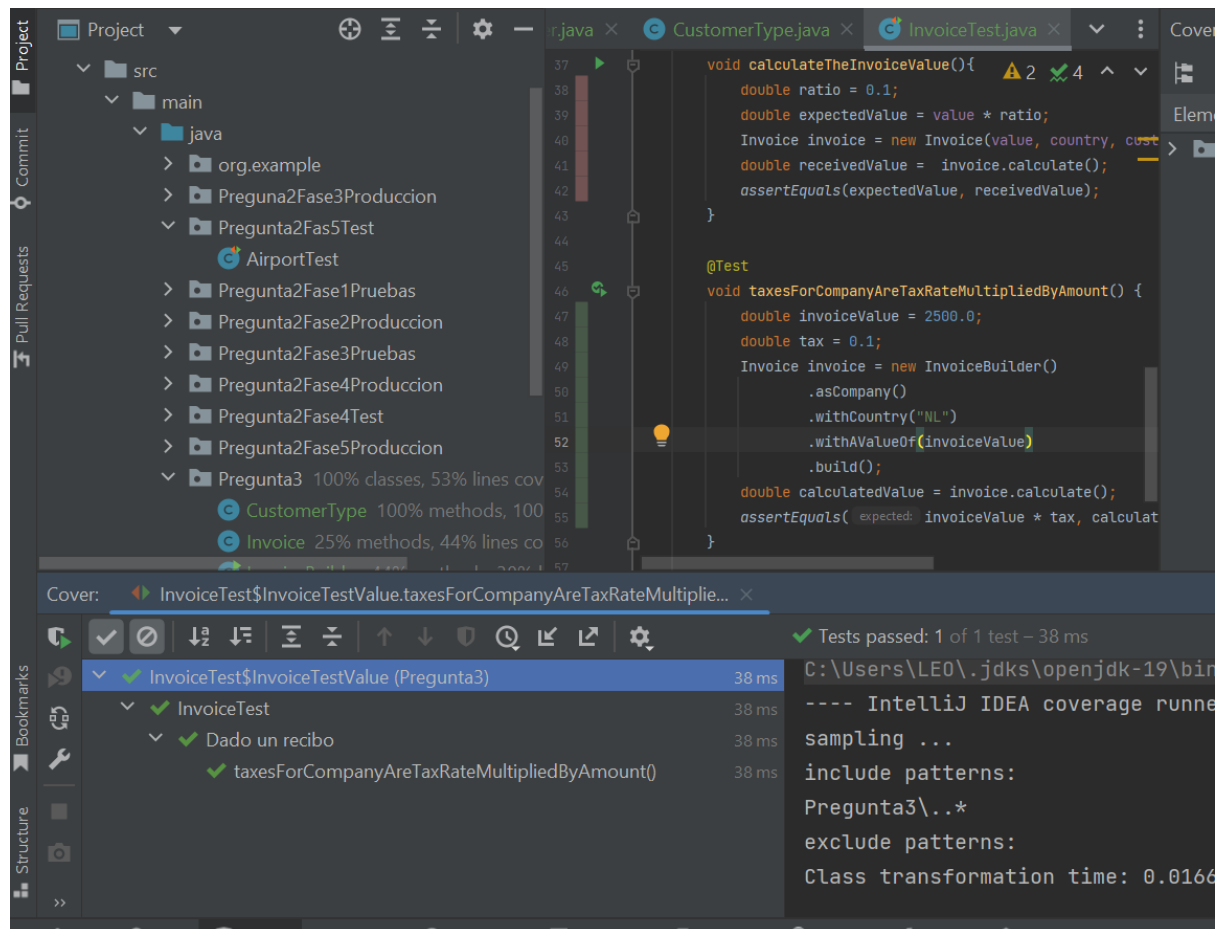
- **Pregunta 3.6.**

Agrega este listado en el código anterior y muestra los resultados

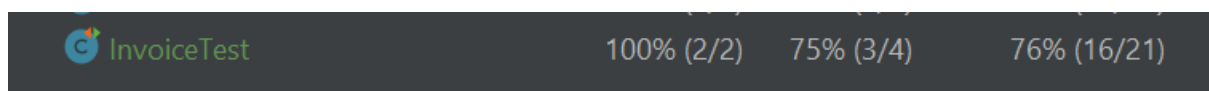
```
@Test
@DisplayName("Obtener el valor de dicho recibo")
void calculateTheInvoiceValue(){
    double ratio = 0.1;
    double expectedValue = value * ratio;
    Invoice invoice = new Invoice(value, country, customerType, createdAt, customerName);
    double receivedValue = invoice.calculate();
    assertEquals(expectedValue, receivedValue);
}

@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();
    assertEquals( expected: invoiceValue * tax, calculatedValue);
}
```

Al ejecutar esta última prueba, en la cual se está utilizando todo lo nuevo que se implementó en el InvoiceBuilder (nuestro generador de datos de prueba), vemos que dicha prueba pasa satisfactoriamente



Además, podemos ver que el coverage es el siguiente



Lo cual tiene sentido, ya que no se está abarcando todo el código fuente de la carpeta o package Pregunta 3