

**Universidad Nacional de Lanús**

**Departamento de Desarrollo Productivo y Tecnológico**

**Trabajo Practico N° 1**

**Carrera:** Licenciatura en Sistemas.

**Materia:** Prueba de Software.

**Año:** 2024.

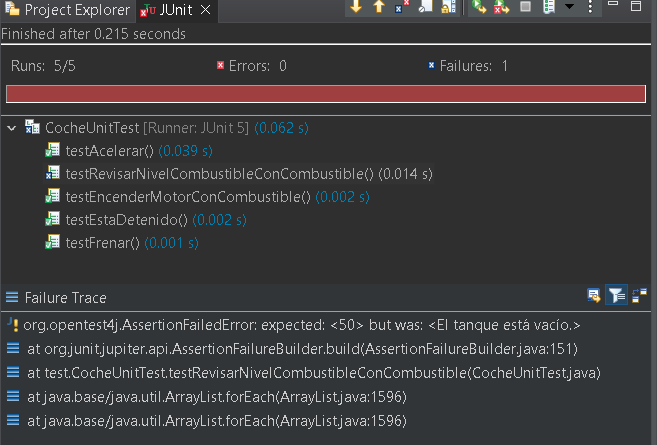
**Docente:** M. Ing. Eduardo Diez.  
 Lic. Pablo San Román.

**Integrantes:** Alex Cadima. DNI:42434535Ignacio Mansilla. DNI: 41557864

**Repositorio:** <https://github.com/Alexcadima/Prueba-Software.git>

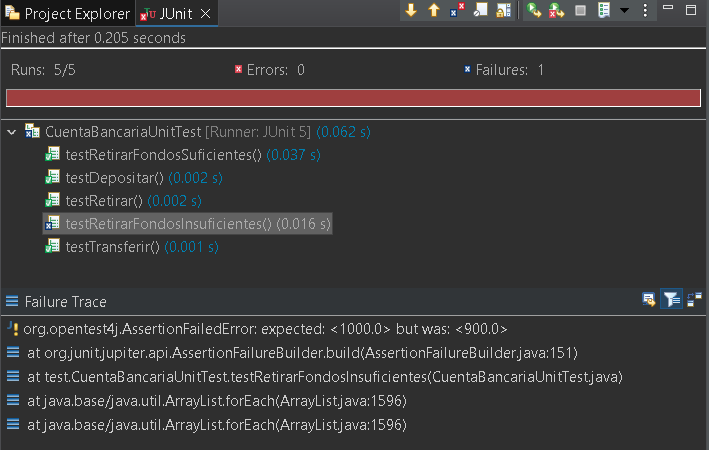
**Ejercicio N°1**

Clase Coche:



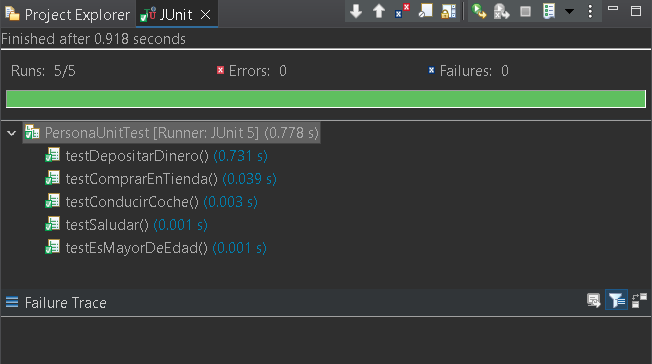
El método arroja un error, dicho error se debe por la espera de un nivel de combustible igual a 50 pero dicho coche arranca con un nivel de combustible en 0.

Clase CuentaBancaria:



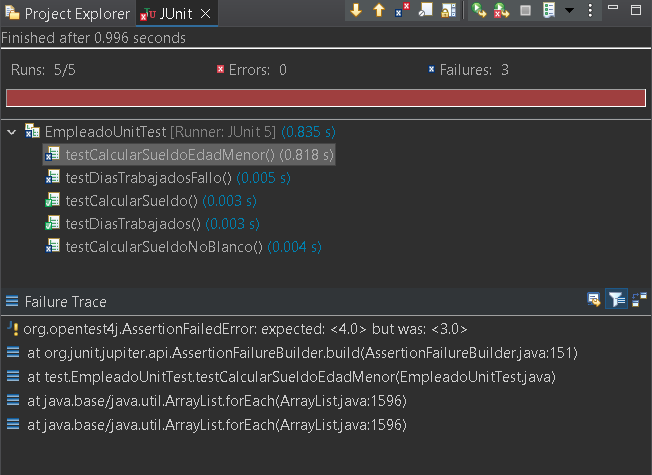
El error se debe al esperar un resultado que sabemos que no obtendremos.

Clase Persona:



Sin problemas en esta clase

Clase Empleado:

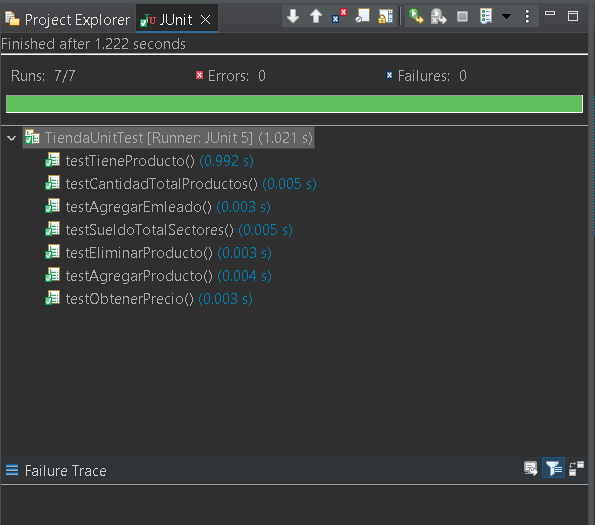


Los tres métodos dan error al esperar resultados erróneos, como por ejemplo esperar 10 días trabajados cuando solo fue 1.

Otro método es por no cumplir con los requisitos de la edad para en este caso tener 19 cuando piden 30 o mayor.

El ultimo método es esperar un sueldo que se multiplique por 2 pero al no estar en blanco no nos dará el resultado esperado.

Clase Tienda:

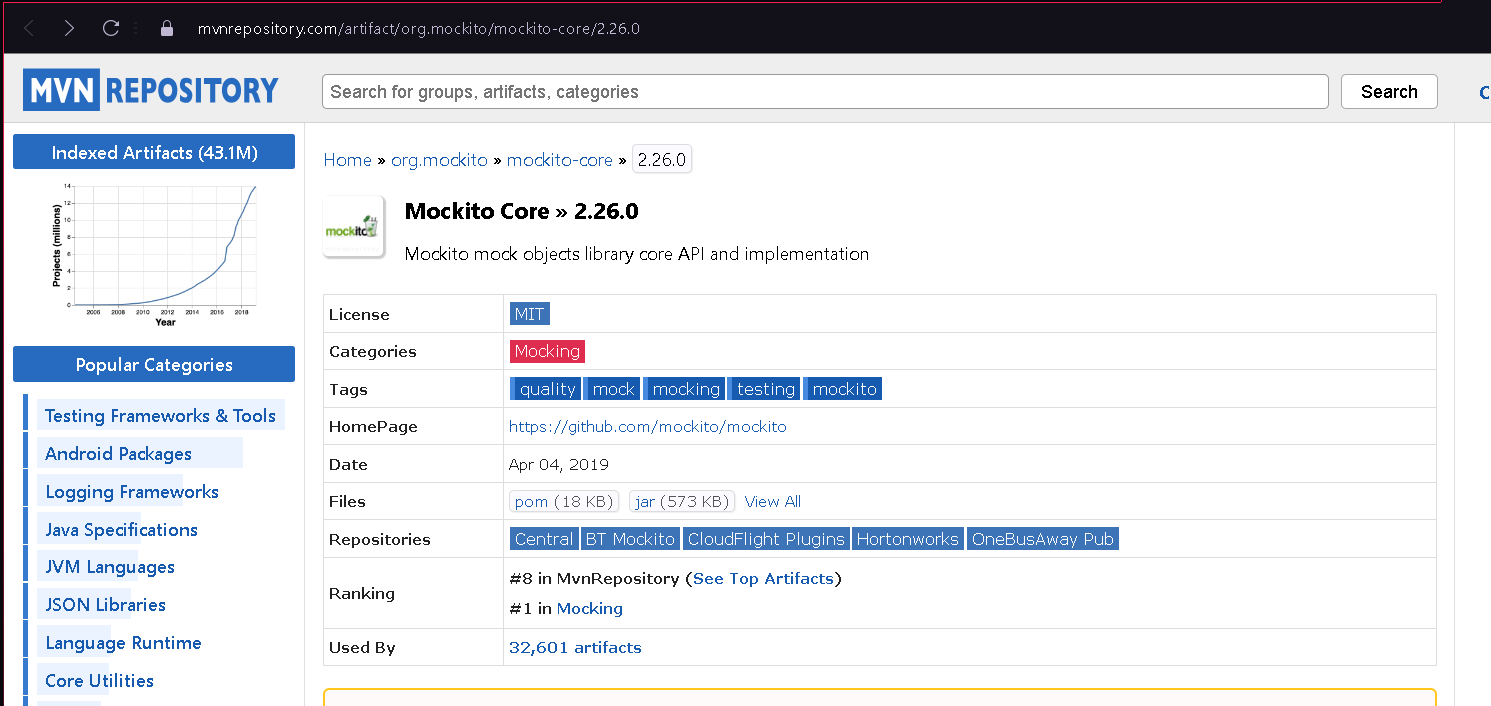


Sin ningún problema.

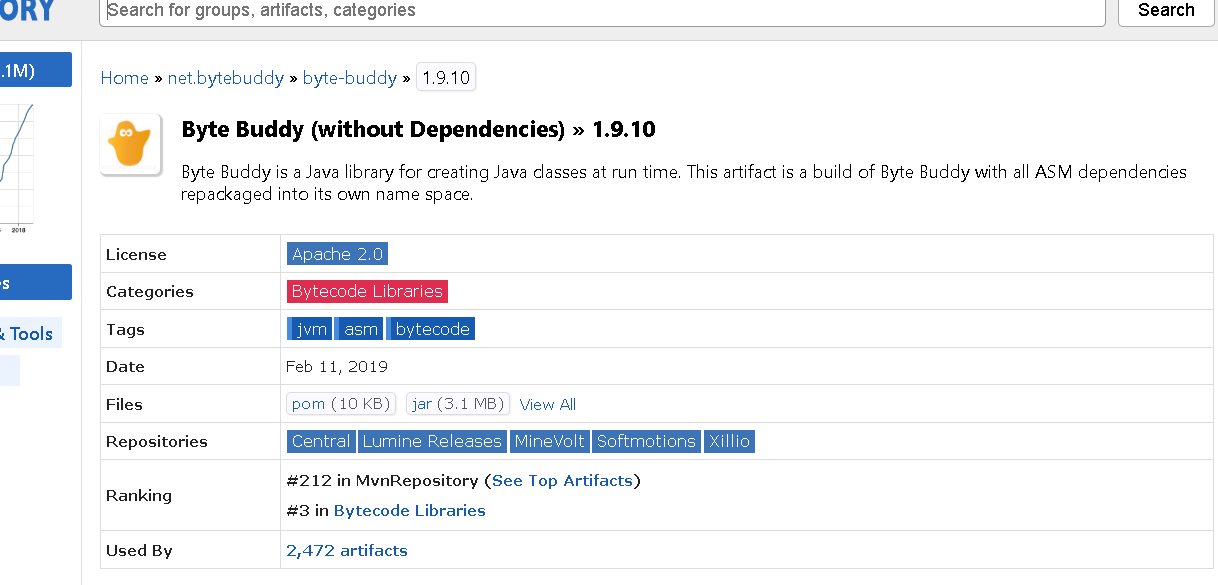
**Ejercicio N°2**

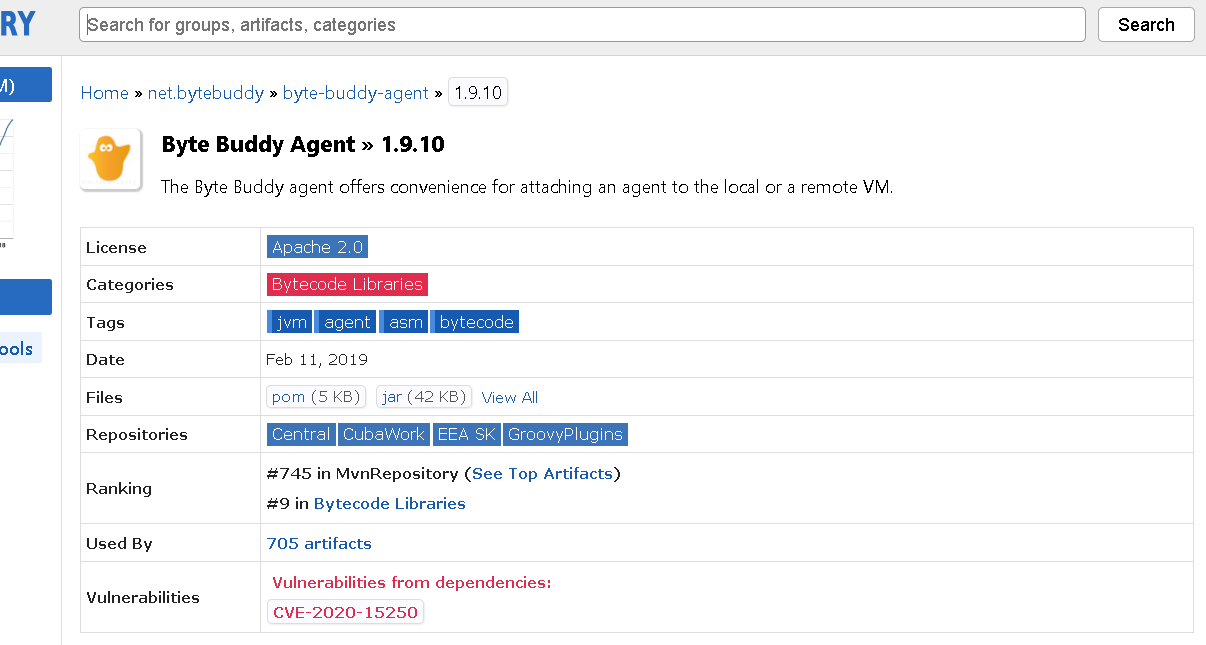
**Instalación de Mockito**

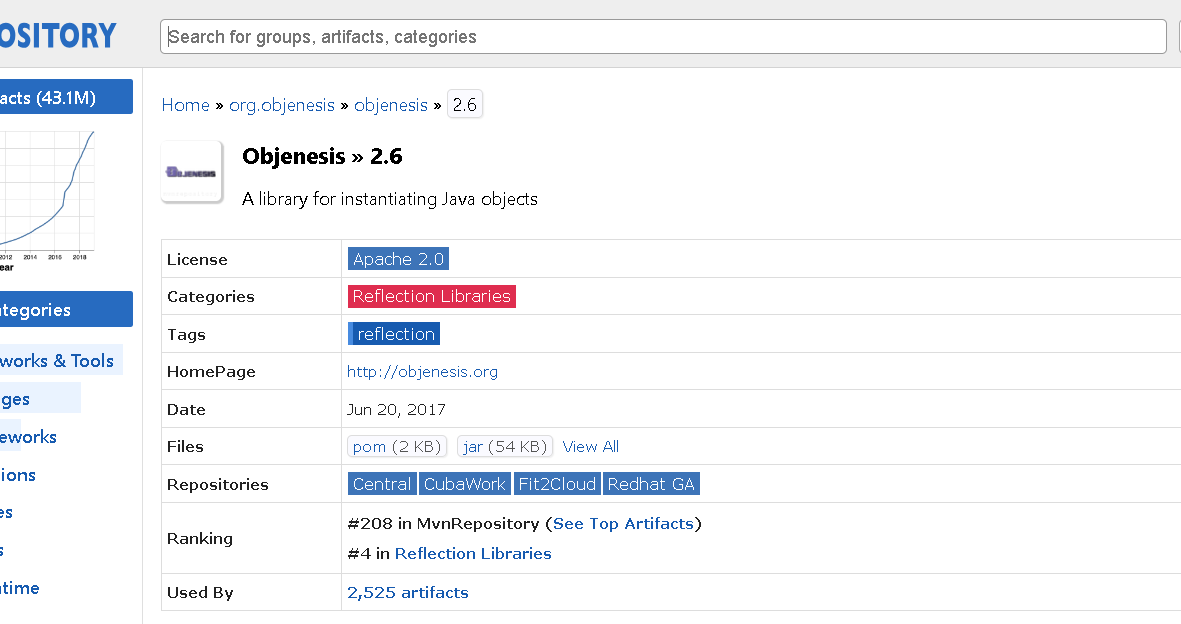
Primero descargamos la librería mockito y sus dependencias. Ingresamos a la página Maven Repository (<https://mvnrepository.com/artifact/org.mockito/mockito-core>) y descargamos la versión de mockito core que nos guste:

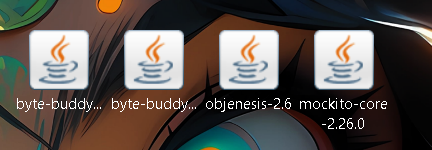


Le damos a la opción Jar la cual nos servirá para eclipse y en la misma página buscamos y descargamos sus dependencias “Byte Buddy”, “Byte Buddy Java Agent” y “Objenesis”.

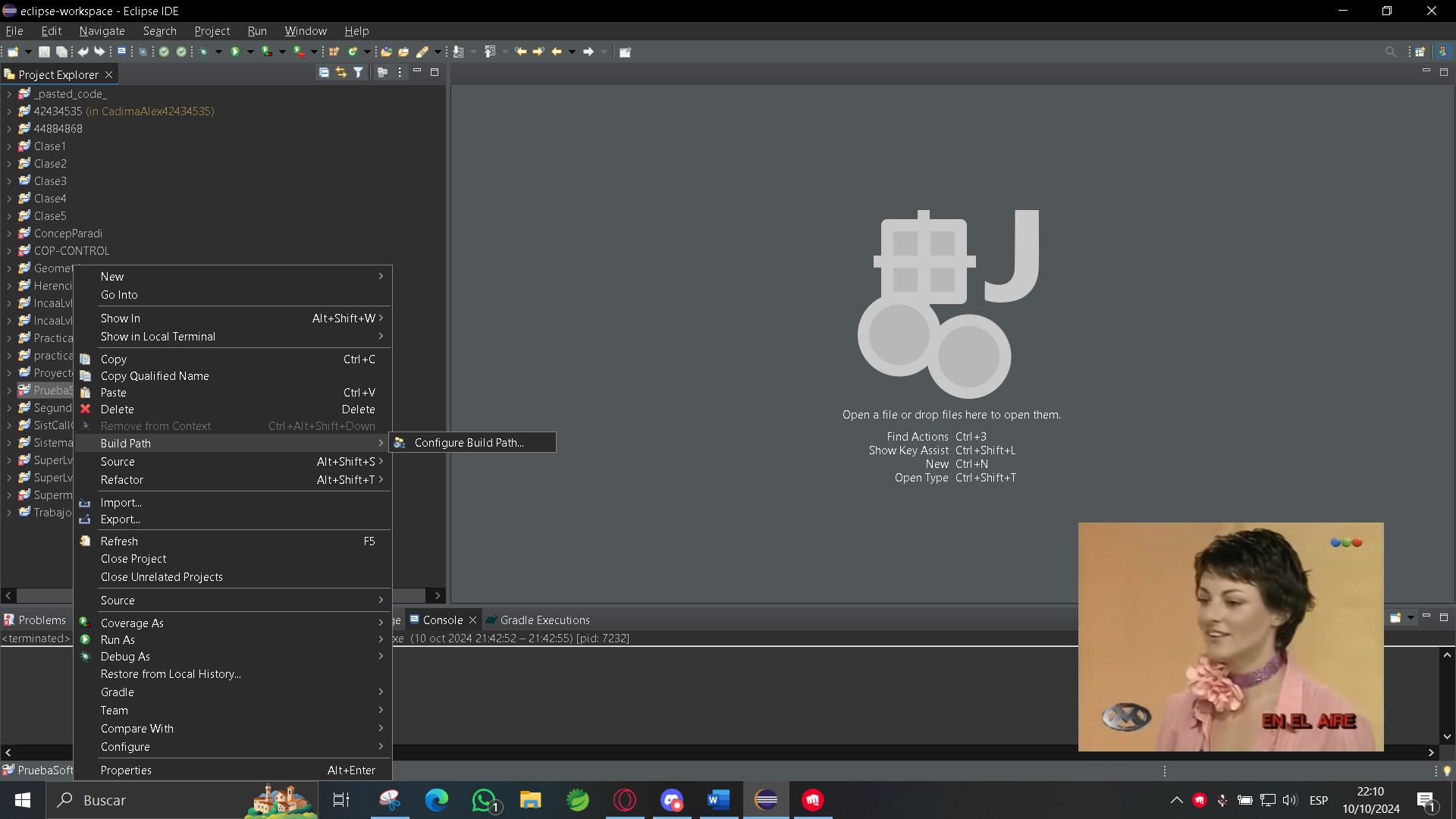




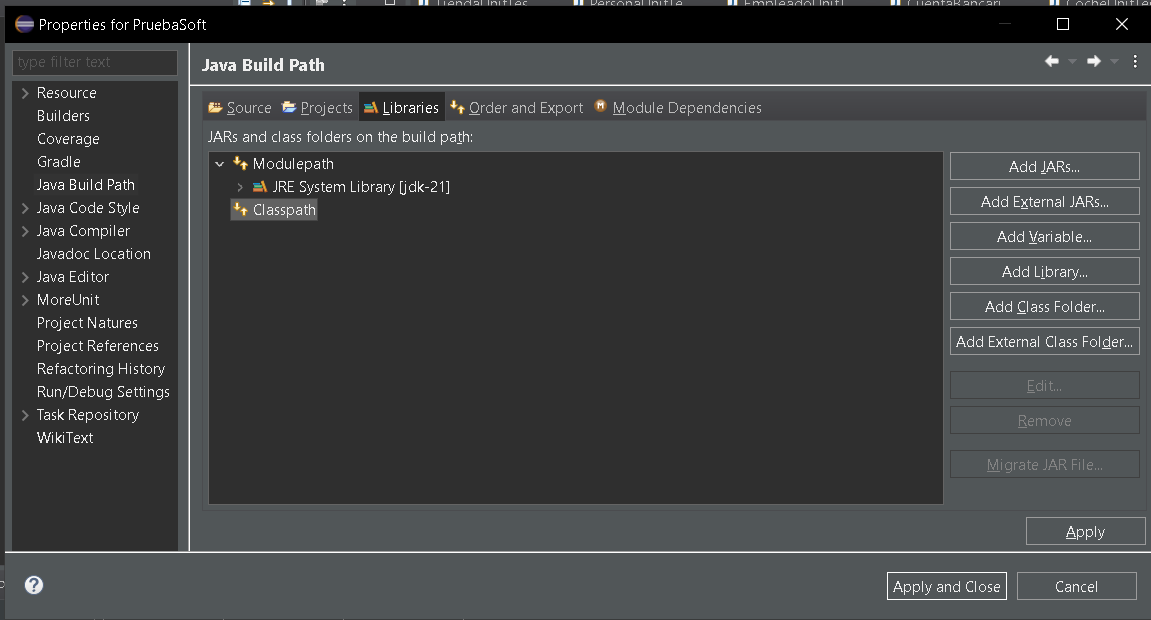
Una vez hecho todo esto nos quedara los siguientes archivos:

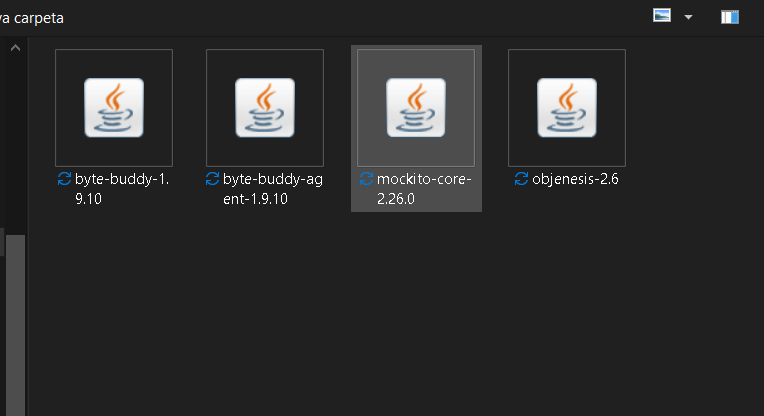


Ahora abrimos eclipse, hacemos click derecho a nuestro proyecto, vamos a la opción “Build Path” y entramos a su configuración

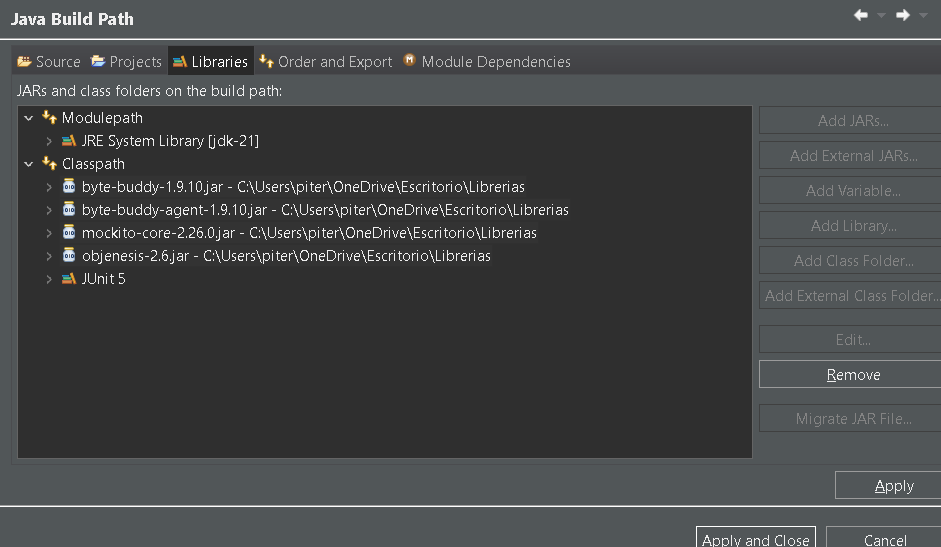


Se nos abrirá la siguiente ventana y le daremos a la opción que dice “add external JARs”



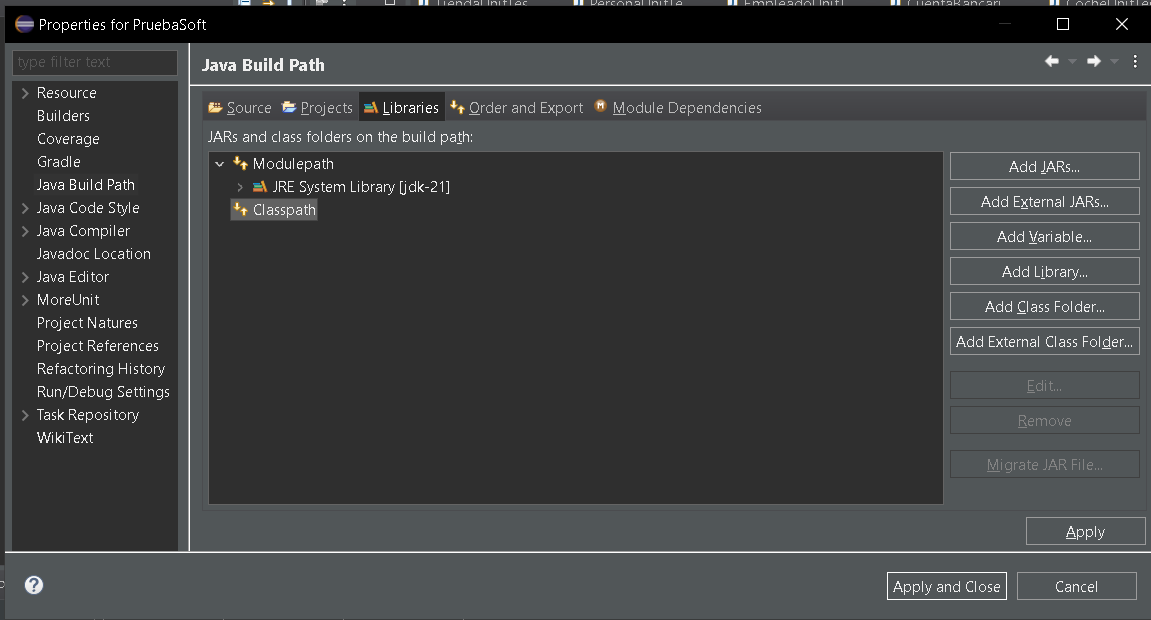


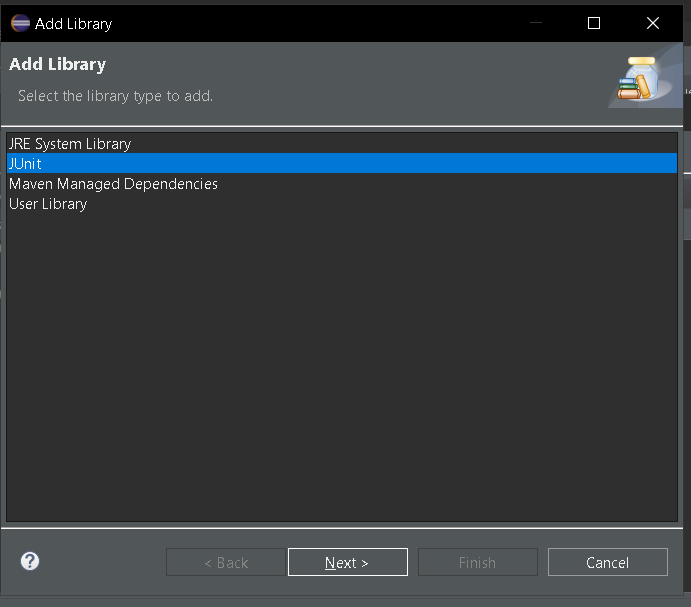
Agregamos las 4 librerías y le damos a Apply y Apply and Close. Con esto terminamos de configurar mockito para poder ahora usarlo sin ningún problema.

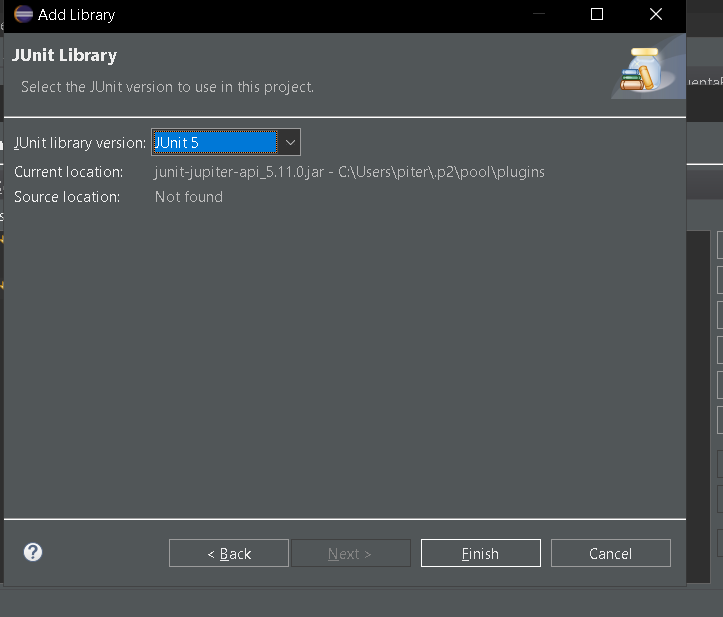


**Importante**

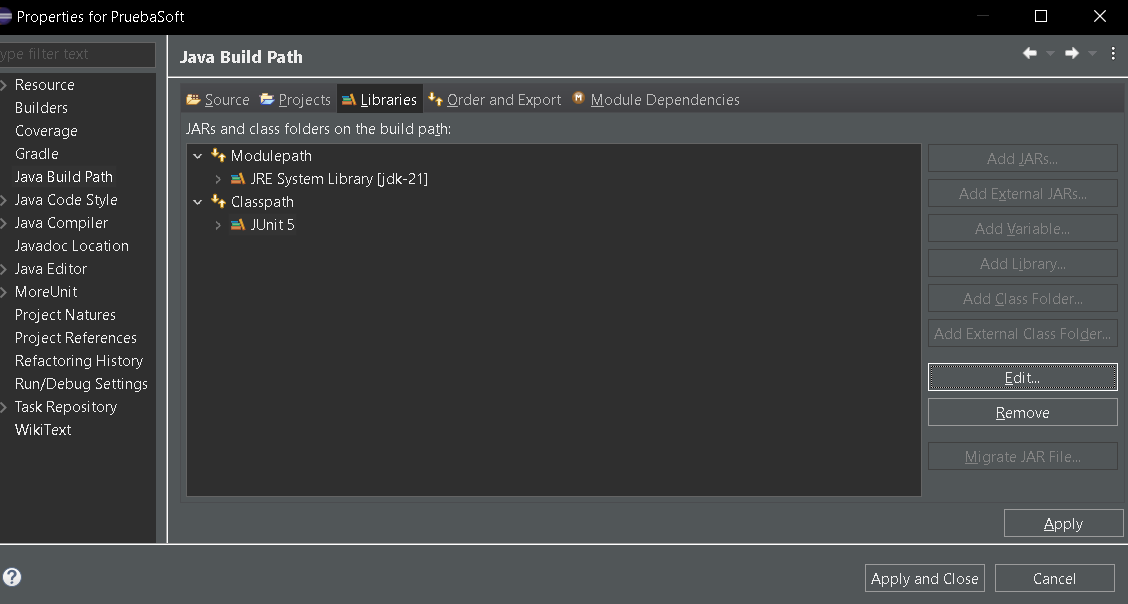
Si no tiene la librería de Junit 5 agregada lo hacemos desde la misma ventana pero entramos desde la opción Add Library



Entramos y seleccionamos la opción Junit con la versión, en este caso 5

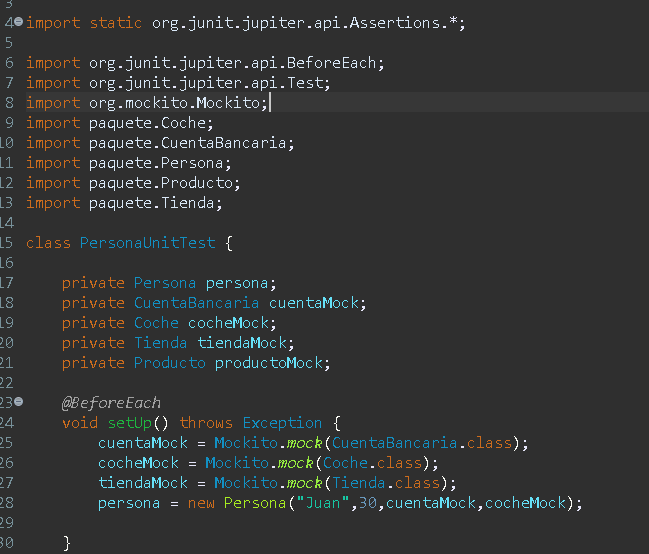


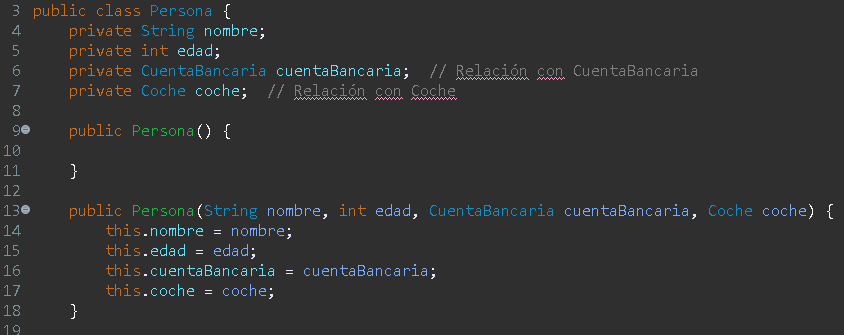
Le damos a finish y ya tenemos la librería Junit.



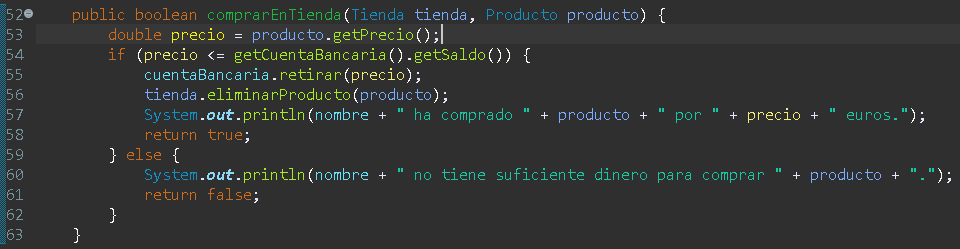
**Uso de Mockito**

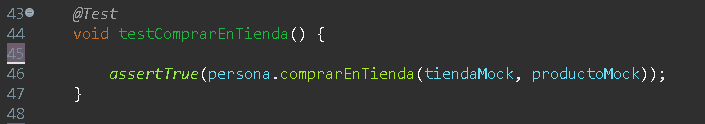
Tenemos que aislar completamente cada test, para ello vamos a usar el concepto **de stub (o talon)** . Un stub hace referencia a una clase que simula ser otra pero que solo tiene implementada una pequeña parte de su funcionalidad. Lo suficiente para gestionar el método al que invocamos.

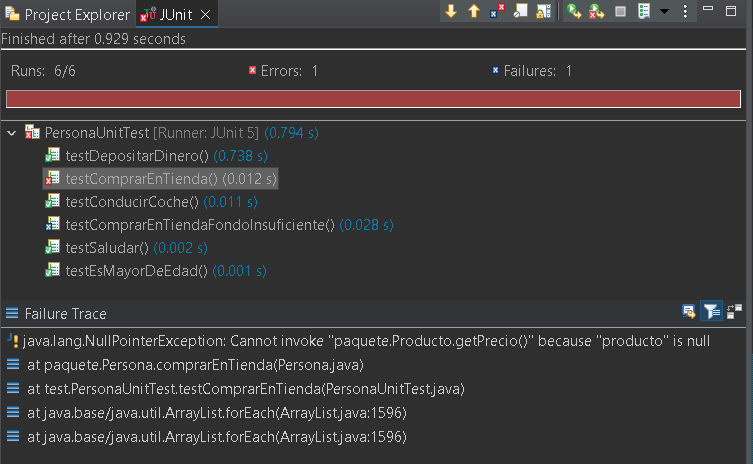


declaramos las variables y usamos el método mock para generar un**Mockito Stub** para coche, tienda y cuenta que seran objetos vacios que simulan ser objetos reales. Esto ya que la clase persona trabaja con dichos objetos.

Vamos a intentar usar el método comprarEnTienda, pero este necesita de 2 clase, producto y tienda, a continuación, mostramos el método que verifica que el monto que esta en la cuenta bancaria sea mayor al precio del producto, de lo contrario no nos dejara comprarlo.



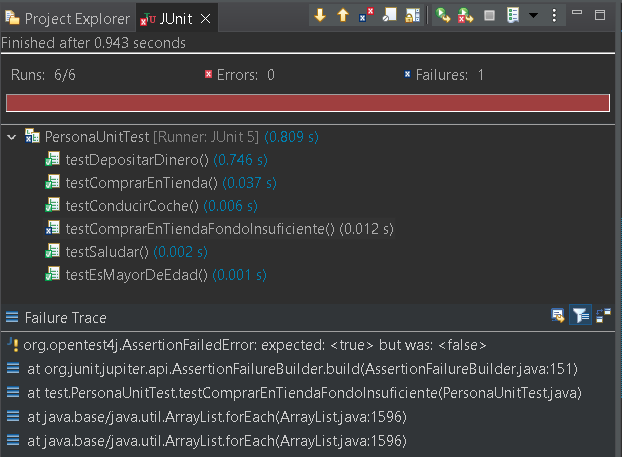




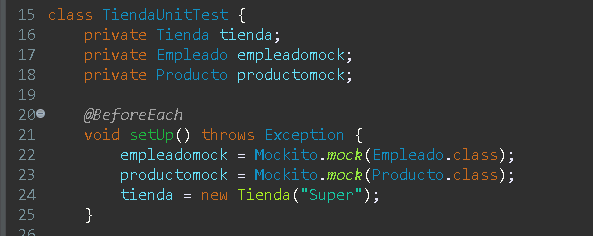
Si ejecutamos el testo nos dará error diciendo que no pudo hacer el getPrecio del producto porque el producto es nulo, esto ya que el producto es simulado no es el objeto real.

Con Mockito vamos a configurar a través de **thenReturn** que cuando usemos algunos gets nos devuelva un valor fijo en este c aso el precio del producto, el saldo de la cuenta bancaria y el nombre del producto. Esto ya que como vimos estos datos son necesario para el método que vamos a testear que es **“comprarEnTienda”** y con el ***assertTrue*** verificamos que el método nos dará true 

Ejecutamos esto y nos ya nos dará un resultado sin errores, de igual manera creamos otro test en donde el saldo de la cuenta bancaria sea menor al precio del producto.



Hacemos lo mismo con el test unitario de Tienda, declaramos las variables que usaremos y simulamos las clases con Mockito.

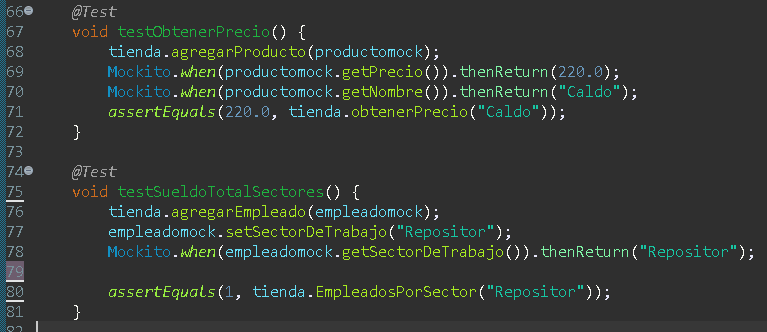


Los siguientes tests utilizan las funciones ya antes vistas como son Mockito.mock(clase en particular) para simularla y Mockito.when para definir el resultado de un método. Estos métodos son los que usaremos en todo los test cuando haga falta. Además de usar los assertEquals para saber si el resultado esperado es el resultado obtenido del método y el assertTrue para verificar que el resultado es de tipo booleano true.

En los dos últimos test usamos mockito para los gets y le pasamos el nombre del producto que queramos que devuelva.

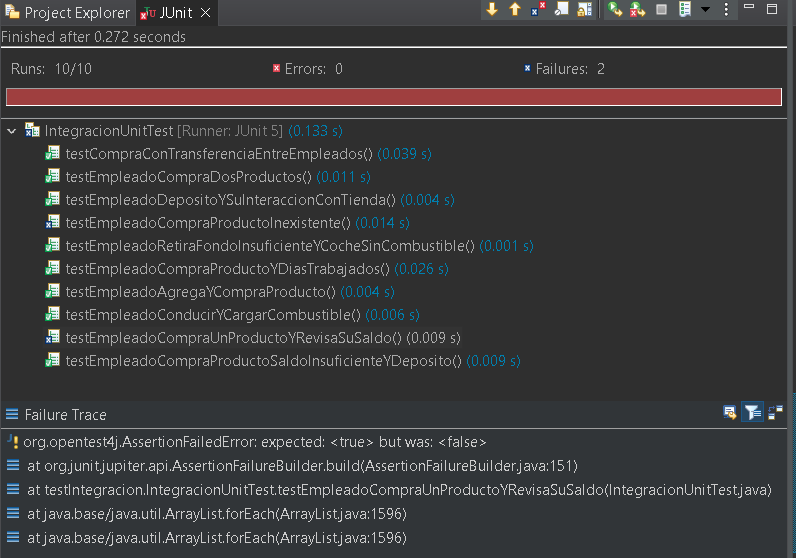


Y en estos dos test para obtener el precio, nombre y nos devuelva el sector al que pertenece cada empleado.

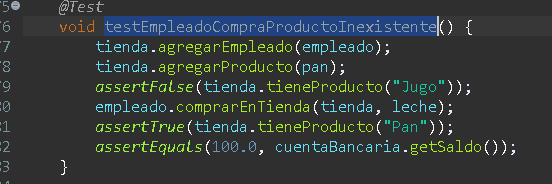


**Ejercicio 3**

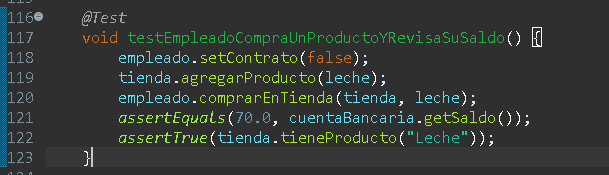
Ahora veremos los resultados de la prueba de integracion, recordemos que en la prueba de integracion no usaremos Mockito ya que nos interesa saber el funcionamiento de todas las clases.



Explicaremos los errores que nos tira la ejecución, en este caso solo dos métodos nos dan errores.



Vemos que agregamos dos productos a la tienda y tratamos de comprar un producto que no existe, dándonos ya un error de esa línea. Luego compramos un producto que, si existe, pero el valor de este es de 30.0, que se descuenta del saldo de la cuenta bancaria, por ende, esperamos un saldo de 100.0 cuando es menos debido a la compra del producto.



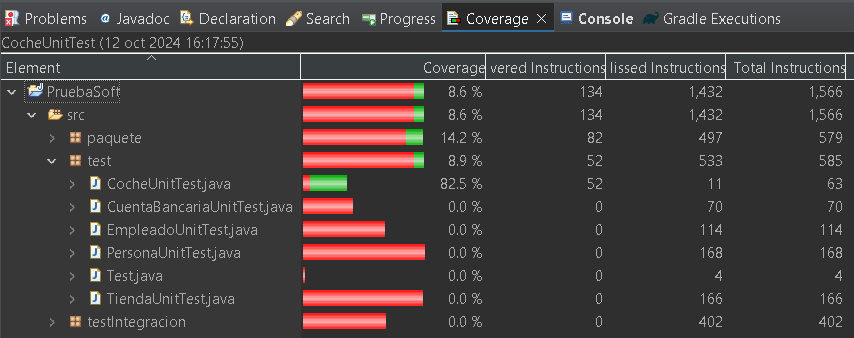
En este segundo test es parecido al primer, pero en este el saldo esperado es el correcto, pero nos dará error en la línea ***assertTrue*** nos dará un false ya que una vez comprado el producto “leche” dejo de existir en la tienda.

**Ejercicio 4**

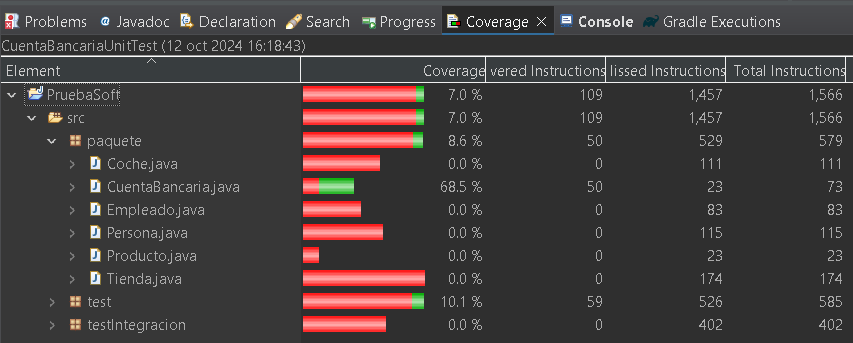
Ahora veremos la cobertura de código que logramos mediante todos los test.

**Tests Unitarios.**

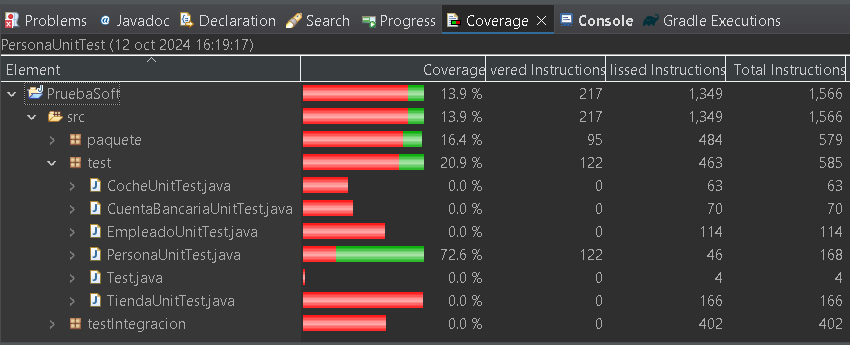
Coche.



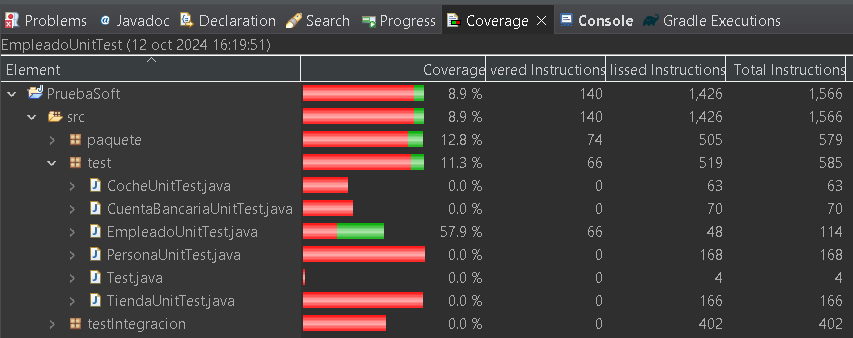
Cuenta Bancaria.



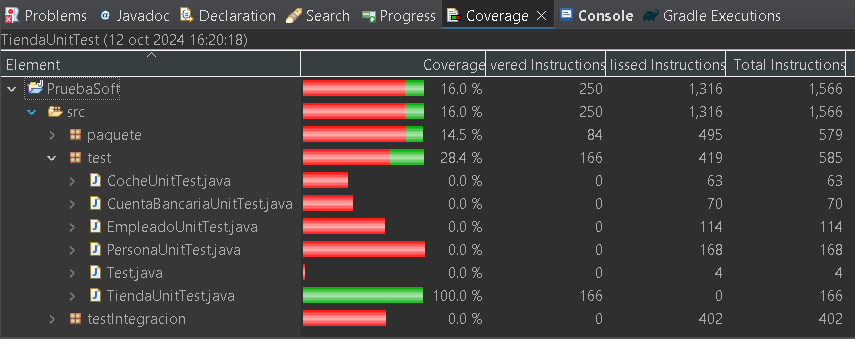
Persona.



Empleado.



Tienda.



Por último, tenemos el test de integracion nos dará una cobertura del 83.6% la cual en nuestra opinión es mas que aceptable para este modelo de proyecto.

