Refactorización: Entrega

Proyecto: Rides

Autores:

- Rubén Gallego
- Mikel Berasategui

Asignatura: Ingeniería de Software II

Enlaces del proyecto:

-Github: https://github.com/RubenGGBC/rides.git

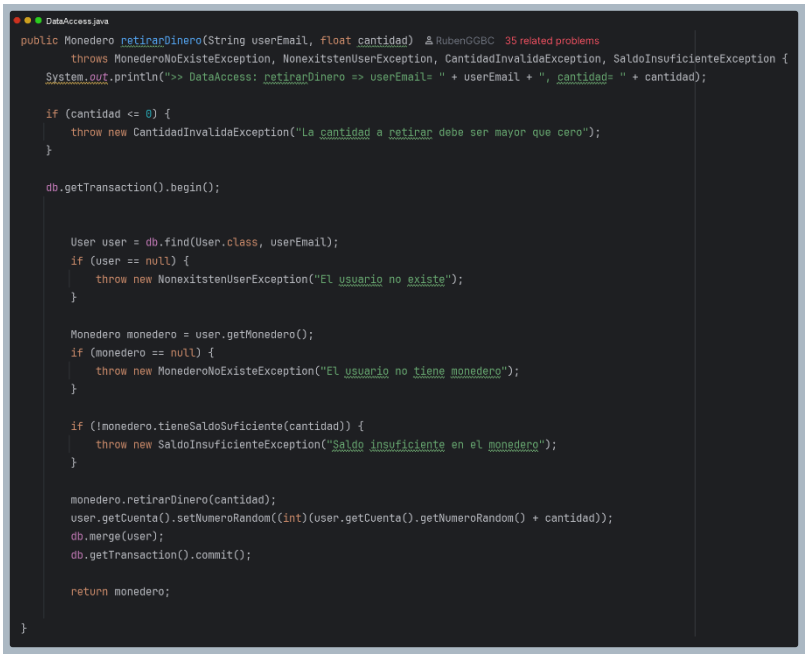
-SonarCloud: https://sonarcloud.io/project/overview?id=rubenggbc_fr

Refactorizaciones de Mikel Berasategui

Refactorización 1: Short units of codes

- Código inicial:

  Realizaremos la refactorización en retirarCuenta (+15 líneas):



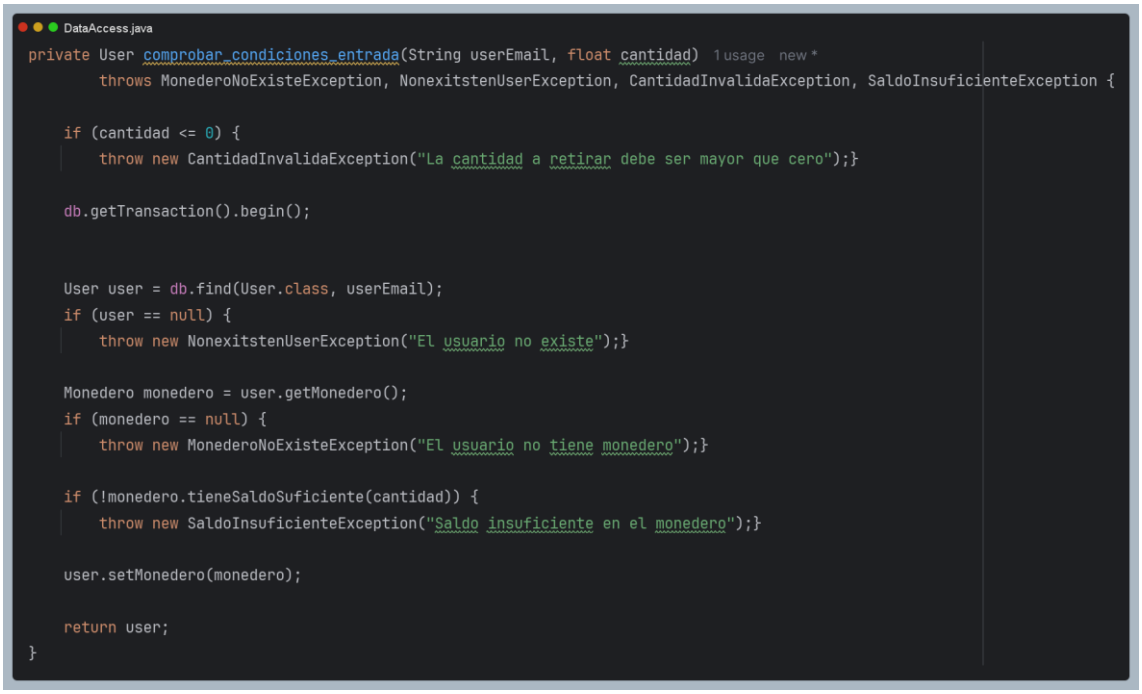
```
DataAccess.java
public Monedero retirarDinero(String userEmail, float cantidad)  RubenGGBC  35 related problems
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException, SaldoInsuficienteException {
    System.out.println(">> DataAccess: retirarDinero => userEmail= " + userEmail + ", cantidad= " + cantidad);

    if (cantidad <= 0) {
        throw new CantidadInvalidaException("La cantidad a retirar debe ser mayor que cero");
    }

    db.getTransaction().begin();


        User user = db.find(User.class, userEmail);
        if (user == null) {
            throw new NonexitstenUserException("El usuario no existe");
        }

        Monedero monedero = user.getMonedero();
        if (monedero == null) {
            throw new MonederoNoExisteException("El usuario no tiene monedero");
        }

        if (!monedero.tieneSaldoSuficiente(cantidad)) {
            throw new SaldoInsuficienteException("Saldo insuficiente en el monedero");
        }

        monedero.retirarDinero(cantidad);
        user.getCuenta().setNumeroRandom((int)(user.getCuenta().getNumeroRandom() + cantidad));
        db.merge(user);
        db.getTransaction().commit();

        return monedero;

}
```

- Código refactorizado

  El Código del método quedaría así:

```
DataAccess.java
public Monedero retirarDinero(String userEmail, float cantidad)  new *
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException, SaldoInsuficienteException {
    System.out.println(">> DataAccess: retirarDinero => userEmail= " + userEmail + ", cantidad= " + cantidad);

    User user = comprobar_condiciones_entrada(userEmail, cantidad);
    Monedero monedero = user.getMonedero();

    monedero.retirarDinero(cantidad);
    user.getCuenta().setNumeroRandom((int)(user.getCuenta().getNumeroRandom() + cantidad));
    db.merge(user);
    db.getTransaction().commit();

    return monedero;

}
```

Sin embargo, también se ha añadido un método auxiliar privado que se llama desde el retirarDinero:



DataAccess.java

```
private User comprobar_condiciones_entrada(String userEmail, float cantidad) 1 usage  new *
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException, SaldoInsuficienteException {

    if (cantidad <= 0) {
        throw new CantidadInvalidaException("La cantidad a retirar debe ser mayor que cero");}

    db.getTransaction().begin();


    User user = db.find(User.class, userEmail);
    if (user == null) {
        throw new NonexitstenUserException("El usuario no existe");}

    Monedero monedero = user.getMonedero();
    if (monedero == null) {
        throw new MonederoNoExisteException("El usuario no tiene monedero");}

    if (!monedero.tieneSaldoSuficiente(cantidad)) {
        throw new SaldoInsuficienteException("Saldo insuficiente en el monedero");}

    user.setMonedero(monedero);

    return user;
}
```

- Descripción de code smell y refactorización:

  Inicialmente, el método retirarDinero cumplía el bad smell descrito en el capítulo 2: “Write short units of code”. En otras palabras, el método tenía más de 15 líneas de código. Por lo tanto, he creado un método auxiliar privado, que se ocupa de revisar los casos de entrada que antes se hacía en el propio retirarDinero, y se le llama desde retirarDinero refactorizado. Ahora, ambos métodos (retirarDinero y el auxiliar) no tienen más de 15 líneas y no cumplen el bad smell.

Refactorización 2: Simple units of codes

- Código inicial:

  Realizaremos la refactorización en el nuevo método auxiliar creado: comprobar_condiciones_entrada (complejidad ciclomática = 5, 4 condicionales + 1)

DataAccess.java

```
private User comprobar_condiciones_entrada(String userEmail, float cantidad) 1 usage  new *
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException, SaldoInsuficienteException {

    if (cantidad <= 0) {
        throw new CantidadInvalidaException("La cantidad a retirar debe ser mayor que cero");}

    db.getTransaction().begin();


    User user = db.find(User.class, userEmail);
    if (user == null) {
        throw new NonexitstenUserException("El usuario no existe");}

    Monedero monedero = user.getMonedero();
    if (monedero == null) {
        throw new MonederoNoExisteException("El usuario no tiene monedero");}

    if (!monedero.tieneSaldoSuficiente(cantidad)) {
        throw new SaldoInsuficienteException("Saldo insuficiente en el monedero");}

    user.setMonedero(monedero);

    return user;
}
```

- Código refactorizado:

  El Código del método comprobar_condiciones_entrada ha quedado así:

DataAccess.java

```
private User comprobar_condiciones_entrada(String userEmail, float cantidad) 1 usage  new *
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException, SaldoInsuficienteException {

    comprobar_cantidad(cantidad);

    db.getTransaction().begin();


    User user = db.find(User.class, userEmail);
    comprobar_user(user);

    Monedero monedero = user.getMonedero();
    comprobar_monedero(monedero, cantidad);

    user.setMonedero(monedero);

    return user;
}
```

Además, se han creado los siguientes métodos auxiliares para realizar la refactorización:


DataAccess.java

```java
private void comprobar_cantidad(float cantidad) throws CantidadInvalidaException {  1 usage  new *
    if (cantidad <= 0) {
        throw new CantidadInvalidaException("La cantidad a retirar debe ser mayor que cero");
    }
}

private void comprobar_user(User user) throws NonexitstenUserException {  1 usage  new *
    if (user == null) {
        throw new NonexitstenUserException("El usuario no existe");
    }
}

private void comprobar_monedero(Monedero monedero, float cantidad) throws MonederoNoExisteException, SaldoInsuficienteException {
    if (monedero == null) {
        throw new MonederoNoExisteException("El usuario no tiene monedero");
    }

    if (!monedero.tieneSaldoSuficiente(cantidad)) {
        throw new SaldoInsuficienteException("Saldo insuficiente en el monedero");
    }
}
```

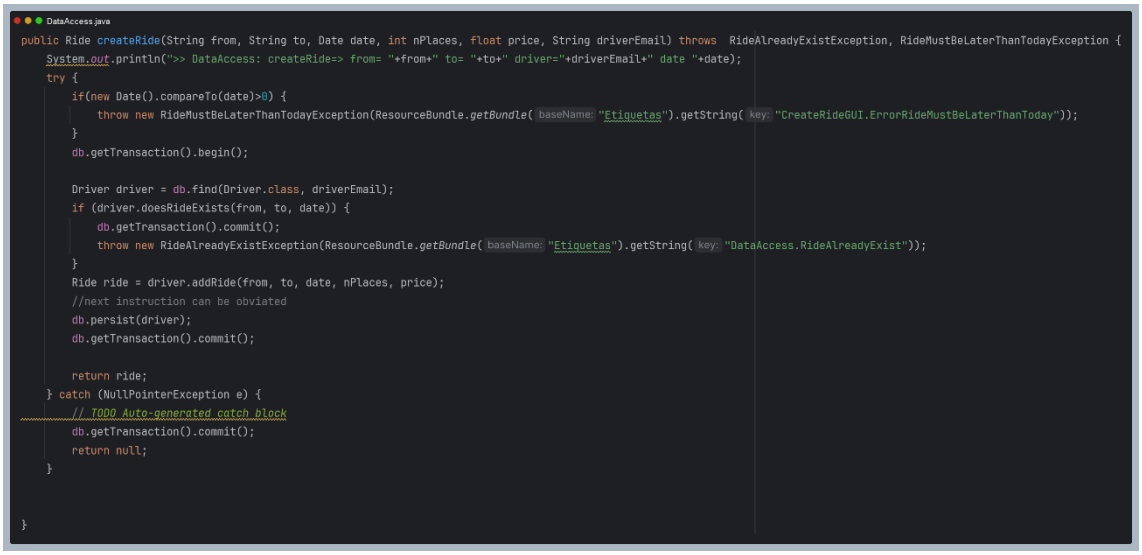
- Descripción de code smell y refactorización:

  Para eliminar el bad smell del capítulo 3: "Write simple units of code", hay que bajar la complejidad ciclomática del método. Por lo tanto, se han creado 3 nuevos métodos auxiliares, y ahora ninguno de ellos supera la complejidad ciclomática de 4.

Refactorización 3: Duplicate code

- Código inicial:

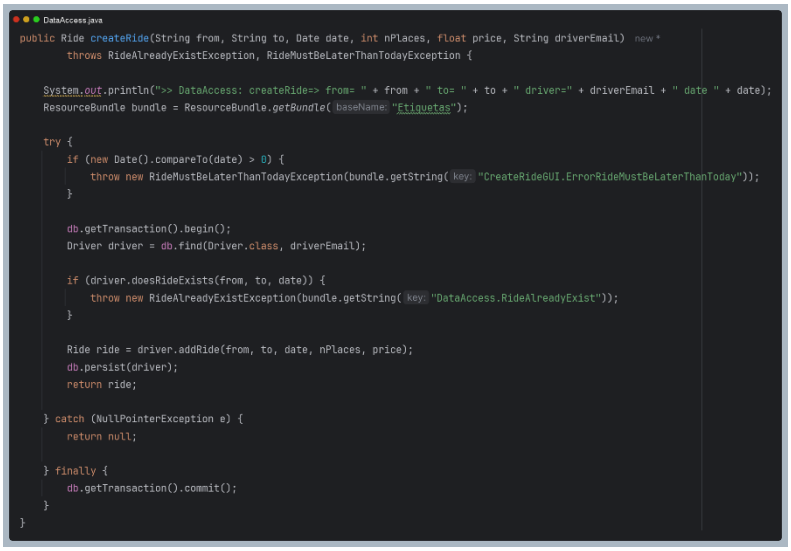
  Realizaremos la refactorización sobre el método createRide, el cual incluye duplicidad de código (se repiten 3 llamadas a db.getTransaction().commit() y 2 llamadas a ResourceBundle.getBundle()):



```
DataAccess.java
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail) throws  RideAlreadyExistException, RideMustBeLaterThanTodayException {
    System.out.println(">> DataAccess: createRide=> from= "+from+" to= "+to+" driver="+driverEmail+" date "+date);
    try {
        if(new Date().compareTo(date)>0) {
            throw new RideMustBeLaterThanTodayException(ResourceBundle.getBundle( baseName: "Etiquetas").getString( key: "CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }
        db.getTransaction().begin();

        Driver driver = db.find(Driver.class, driverEmail);
        if (driver.doesRideExists(from, to, date)) {
            db.getTransaction().commit();
            throw new RideAlreadyExistException(ResourceBundle.getBundle( baseName: "Etiquetas").getString( key: "DataAccess.RideAlreadyExist"));
        }
        Ride ride = driver.addRide(from, to, date, nPlaces, price);
        //next instruction can be obviated
        db.persist(driver);
        db.getTransaction().commit();

        return ride;
    } catch (NullPointerException e) {
        // TODO Auto-generated catch block
        db.getTransaction().commit();
        return null;
    }


}
```

- Código refactorizado:

  Se ha modificado el código de createRide para hacer que las llamadas db.getTransaction().commit() y ResourceBundle.getBundle() se realicen solamente una vez. Para ello, se realiza la llamada de db.getTransaction().commit() en un finally y la de ResourceBundle.getBundle() se guarda en una variable bundle:



```
DataAccess.java
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail)  new *
        throws RideAlreadyExistException, RideMustBeLaterThanTodayException {

    System.out.println(">> DataAccess: createRide=> from= " + from + " to= " + to + " driver=" + driverEmail + " date " + date);
    ResourceBundle bundle = ResourceBundle.getBundle( baseName: "Etiquetas");

    try {
        if (new Date().compareTo(date) > 0) {
            throw new RideMustBeLaterThanTodayException(bundle.getString( key: "CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }

        db.getTransaction().begin();
        Driver driver = db.find(Driver.class, driverEmail);

        if (driver.doesRideExists(from, to, date)) {
            throw new RideAlreadyExistException(bundle.getString( key: "DataAccess.RideAlreadyExist"));
        }

        Ride ride = driver.addRide(from, to, date, nPlaces, price);
        db.persist(driver);
        return ride;

    } catch (NullPointerException e) {
        return null;

    } finally {
        db.getTransaction().commit();
    }
}
```

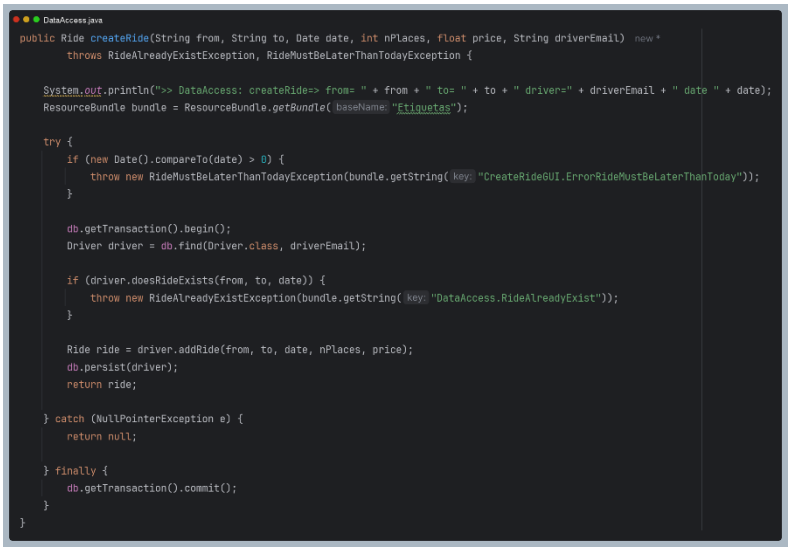
- Descripción de code smell y refactorización:

  Se ha eliminado el bad smell del capítulo 4: “Duplicate code”. En este caso, hemos reducido las llamadas de db.getTransaction().commit() de 3 a 1 mediante un finally. De la misma manera, las llamadas de ResourceBundle.getBundle() se han reducido de 2 a 1 guardándolo en una variable.

Refactorización 4: Keep unit interfaces small

- Código inicial:

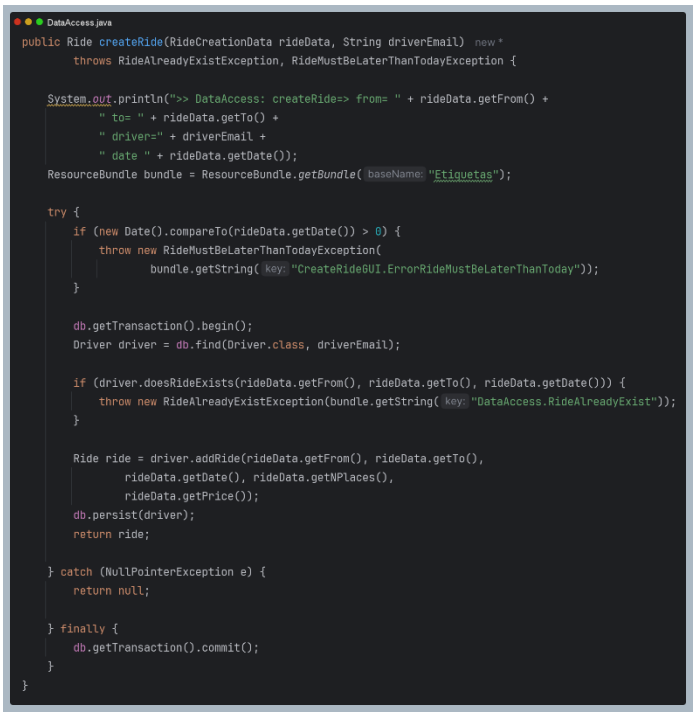
  Vamos a partir del método anterior actualizado, es decir, createRide con el bad smell de las repeticiones solucionado:



```
DataAccess.java
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail)  new *
        throws RideAlreadyExistException, RideMustBeLaterThanTodayException {

    System.out.println(">> DataAccess: createRide=> from= " + from + " to= " + to + " driver=" + driverEmail + " date " + date);
    ResourceBundle bundle = ResourceBundle.getBundle( baseName: "Etiquetas");

    try {
        if (new Date().compareTo(date) > 0) {
            throw new RideMustBeLaterThanTodayException(bundle.getString( key: "CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }

        db.getTransaction().begin();
        Driver driver = db.find(Driver.class, driverEmail);

        if (driver.doesRideExists(from, to, date)) {
            throw new RideAlreadyExistException(bundle.getString( key: "DataAccess.RideAlreadyExist"));
        }

        Ride ride = driver.addRide(from, to, date, nPlaces, price);
        db.persist(driver);
        return ride;

    } catch (NullPointerException e) {
        return null;

    } finally {
        db.getTransaction().commit();
    }
}
```

- Código refactorizado

  Para solucionarlo, se ha actualizado el método al siguiente código:



```
DataAccess.java
public Ride createRide(RideCreationData rideData, String driverEmail)  new *
        throws RideAlreadyExistException, RideMustBeLaterThanTodayException {

    System.out.println(">> DataAccess: createRide=> from= " + rideData.getFrom() +
            " to= " + rideData.getTo() +
            " driver=" + driverEmail +
            " date " + rideData.getDate());
    ResourceBundle bundle = ResourceBundle.getBundle( baseName: "Etiquetas");

    try {
        if (new Date().compareTo(rideData.getDate()) > 0) {
            throw new RideMustBeLaterThanTodayException(
                    bundle.getString( key: "CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }

        db.getTransaction().begin();
        Driver driver = db.find(Driver.class, driverEmail);

        if (driver.doesRideExists(rideData.getFrom(), rideData.getTo(), rideData.getDate())) {
            throw new RideAlreadyExistException(bundle.getString( key: "DataAccess.RideAlreadyExist"));
        }

        Ride ride = driver.addRide(rideData.getFrom(), rideData.getTo(),
                rideData.getDate(), rideData.getNPlaces(),
                rideData.getPrice());
        db.persist(driver);
        return ride;

    } catch (NullPointerException e) {
        return null;

    } finally {
        db.getTransaction().commit();
    }
}
```

Y, aparte, se ha creado una nueva clase: RideCreationData:

RideCreationData.java

```java
package domain;

import java.util.Date;

public class RideCreationData {  3 usages  new *
    private String from;  2 usages
    private String to;  2 usages
    private Date date;  2 usages
    private int nPlaces;  2 usages
    private float price;  2 usages

    public RideCreationData(String from, String to, Date date, int nPlaces, float price) {
        this.from = from;
        this.to = to;
        this.date = date;
        this.nPlaces = nPlaces;
        this.price = price;
    }

    // Getters
    public String getFrom() { return from; }  new *
    public String getTo() { return to; }  new *
    public Date getDate() { return date; }  new *
    public int getNPlaces() { return nPlaces; }  1 usage  new *
    public float getPrice() { return price; }  new *
}
```

- Descripción de code smell y refactorización:

  Inicialmente, el método createRide recibía 6 parámetros de entrada, lo cual cumplía con el bad smell del capítulo 5: "Keep unit interfaces small". Ahora, sin embargo, recibe 2. Para hacer eso posible, se ha creado una nueva clase: RideCreationData, la cual se debe crear (mediante new RideCreationData(...)) antes de llamar a createRide, para pasársela como parámetro. Por eso, también se han tenido que modificar las llamadas a createRide desde otros métodos del programa. Ahora se llamará a createRide así:

BLFacadeImplementation.java

```java
RideCreationData rideData = new RideCreationData(from, to, date, nPlaces, price);
Ride ride = dbManager.createRide(rideData, driverEmail);
```

Refactorizaciones de Rubén Gallego

Refactorización 1: Short units of codes

- Código inicial:

```java
public Monedero ingresarDinero(String userEmail, float cantidad)  RubenGGBC *
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException {
    System.out.println(">> DataAccess: ingresarDinero => userEmail= " + userEmail + ", cantidad= " + cantidad);

    db.getTransaction().begin();

    User user = db.find(User.class, userEmail);
    if (user == null) {
        throw new NonexitstenUserException("El usuario no existe");
    }

    Monedero monedero = user.getMonedero();
    if (monedero == null) {
        // Si el usuario no tiene monedero, creamos uno
        monedero = new Monedero( id: userEmail + "_wallet");
        monedero.setUser(user);
        user.setMonedero(monedero);
    }
    if(user.getCuenta().getNumeroRandom()<cantidad) {
        throw new CantidadInvalidaException("No tienes tanto dinero en la cuenta");
    }

    monedero.ingresarDinero(cantidad);
    user.getCuenta().setNumeroRandom((int)(user.getCuenta().getNumeroRandom() - cantidad));
    db.merge(user);
    db.getTransaction().commit();

    return monedero;
}
```

- Código refactorizado

```java
public Monedero ingresarDinero(String userEmail, float cantidad)  RubenGGBC
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException {
    System.out.println(">> DataAccess: ingresarDinero => userEmail= " + userEmail + ", cantidad= " + cantidad);

    db.getTransaction().begin();

    User user = validarYObtenerUsuario(userEmail);
    Monedero monedero = obtenerOCrearMonedero(user, userEmail);
    validarSaldoEnCuenta(user, cantidad);

    realizarIngresoYActualizarCuenta(monedero, user, cantidad);

    db.merge(user);
    db.getTransaction().commit();
    return monedero;
}
```

- Descripción de code smell y refactorización
    - El problema era debido a que el numero del metodo original superaba con creces las <=15 lineas recomendadas, produciendo un bad smell, para solucionar el problema, he encapsulando porciones antiguas del codigo en diferentes metodos

Refactorización 2: Simple units of codes

- Código inicial

```
DataAccess.java
public Monedero ingresarDinero(String userEmail, float cantidad)  RubenGGBC
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException {
    System.out.println(">> DataAccess: ingresarDinero => userEmail= " + userEmail + ", cantidad= " + cantidad);

    db.getTransaction().begin();

    User user = validarYObtenerUsuario(userEmail);
    Monedero monedero = obtenerOCrearMonedero(user, userEmail);
    validarSaldoEnCuenta(user, cantidad);

    realizarIngresoYActualizarCuenta(monedero, user, cantidad);

    db.merge(user);
    db.getTransaction().commit();
    return monedero;
}
```

- Código refactorizado

```
DataAccess.java
public Monedero ingresarDinero(String userEmail, float cantidad)  RubenGGBC
        throws MonederoNoExisteException, NonexitstenUserException, CantidadInvalidaException {
    System.out.println(">> DataAccess: ingresarDinero => userEmail= " + userEmail + ", cantidad= " + cantidad);

    db.getTransaction().begin();

    User user = validarYObtenerUsuario(userEmail);
    Monedero monedero = obtenerOCrearMonedero(user, userEmail);
    validarSaldoEnCuenta(user, cantidad);

    realizarIngresoYActualizarCuenta(monedero, user, cantidad);

    db.merge(user);
    db.getTransaction().commit();
    return monedero;
}

// MÉTODOS EXTRAÍDOS
private User validarYObtenerUsuario(String userEmail) throws NonexitstenUserException {  RubenGGBC
    User user = db.find(User.class, userEmail);
    if (user == null) {
        throw new NonexitstenUserException("El usuario no existe");
    }
    return user;
}

private Monedero obtenerOCrearMonedero(User user, String userEmail) {  RubenGGBC
    Monedero monedero = user.getMonedero();
    if (monedero == null) {
        monedero = new Monedero( id: userEmail + "_wallet");
        monedero.setUser(user);
        user.setMonedero(monedero);
    }
    return monedero;
}

private void validarSaldoEnCuenta(User user, float cantidad) throws CantidadInvalidaException {  RubenGGBC
    if(user.getCuenta().getNumeroRandom() < cantidad) {
        throw new CantidadInvalidaException("No tienes tanto dinero en la cuenta");
    }
}

private void realizarIngresoYActualizarCuenta(Monedero monedero, User user, float cantidad) {  RubenGGBC
    monedero.ingresarDinero(cantidad);
    user.getCuenta().setNumeroRandom((int)(user.getCuenta().getNumeroRandom() - cantidad));
}
```

- Descripción de code smell y refactorización:
    - Como podemos observar, al reducir el numero de lineas delegando porciones de codigo en diferentes metodos, hemos conseguido reducir la complejidad ciclomatica del metodo, poniendo 1 if por cada metodo auxiliar

Refactorización 3: Duplicate code

- Código inicial:

- Código refactorizado

- Descripción de code smell y refactorización
  - Como podemos ver hemos evitado las porciones de codigo replicado aprovechando los metodos auxiliares que hemos creado para solucionar los 2 primeros bad smells del metodo ingresarDinero(), asociarCuentaBancaria tambien comprueba si el usuario es null y si tiene monedero y en caso de que no sea asi crea uno, antes en los 2 metodos se hacia la misma comprobacion, por lo que al reutilizar el metodo auxiliar que hace esas funciones, hemos podido reducir tanto la complejidad ciclomatica, el numero de lineas y codigo duplicado del metodo.
  - Como mejora adicional podemos ver que se hace una comprobacion de si el driver es null, se podra extraer tambien esas lineas y encapsularlas en otro metodo, esto nos permitira que el metodo cumplira las 4 condiciones, sin embargo no es codigo duplicado ya que es en el unico metodo en el que se hace esa comprobación

Refactorización 4: Keep unit interfaces small

- Código inicial:

```
DataAccess.java
public void createUser(String email, String password, boolean driver, String nombre) throws UserAlredyExistException {
    db.getTransaction().begin();
    User user=new User(email,password,driver,nombre);

    if (db.find(User.class,email)!=null) {
        db.getTransaction().commit();
        throw new UserAlredyExistException("Ya existe un usuario con ese email");
    }else {
        String tipo = "Driver";
        if(!driver) {
            tipo = "Cliente";
        }else {
            Driver conductor=new Driver(email,nombre);
            db.persist(conductor);
        }
        System.out.println(">> DataAccess: createUser=> email= "+email+" tipo= "+tipo+" nombre= "+ nombre);
        db.persist(user);
        db.getTransaction().commit();
    }
}
```

- Código refactorizado

```
DataAccess.java
public void createUser(String email, String password, String nombre) throws UserAlredyExistException {
    db.getTransaction().begin();

    if (db.find(User.class, email) != null) {
        db.getTransaction().commit();
        throw new UserAlredyExistException("Ya existe un usuario con ese email");
    }

    User user = new User(email, password, driver: false, nombre);
    Driver conductor = new Driver(email, nombre);

    db.persist(conductor);
    db.persist(user);

    System.out.println(">> DataAccess: createUser=> email= " + email + " nombre= " + nombre);
    db.getTransaction().commit();
}
```

- Descripción de code smell y refactorización
    - Este code smell como tal no incumplia la recomendacion del numero de parametros, pero estaba en el limite ya que tenia 4 parametros, para solucionar este problema, he aprovechado que el metodo estaba mal diseñado para arreglarlo
    - Conceptualmente el metodo distinguia entre crear usuarios y drivers, sin embargo, el boolean de driver debido a un fallo en

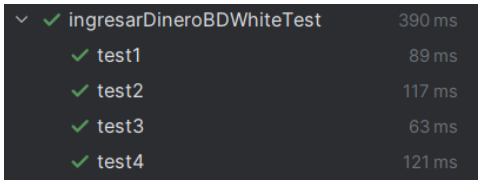
el if siempre era false, por lo que creaba tanto un user como un driver con el mismo correo, al final esto resulto ser mas util y decidi dejarlo, sin embargo ahora crea las 2 entidades y ya, sin importar el parametro de driver, reduciendo asi el numero de parametros a 3

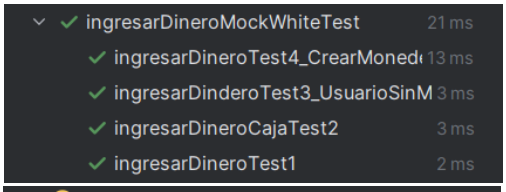
Comprobación del funcionamiento de los tests:

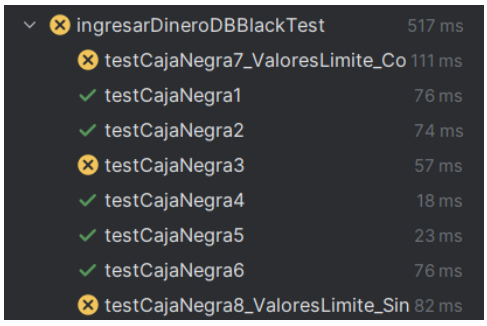
Después de modificar dos métodos sobre los cuales se realizaron tests en la anterior entrega, comprobamos que en ambos casos los resultados son exactamente iguales que en la entrega anterior.

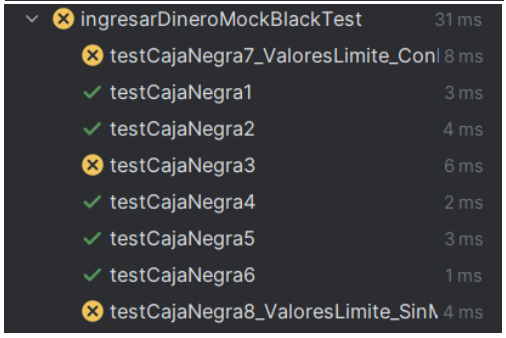
Tests de ingresarDinero

Como no se arreglaron los errores en el código anterior después de la realización de los test, los test del método ingresarDinero sigue fallando en los mismos casos de prueba.









Tests de retirarDinero

Todos los tests muestran un resultado positivo, al igual que en la anterior entrega.