INGENIERÍA DEL SOFTWARE

REFACTORIZACIÓN EN ECLIPSE

AGUSTÍN BELTRÁN DE HEREDIA

MADDI LÓPEZ

GAIZKA ACEDO

12/10/2025

AUTORA: Maddi López

“Duplicate code”

CÓDIGO INICIAL: “Gazteiz” está repetido

```
//Create rides
String Bilbo = "Bilbo";
String Donosti = "Donostia";
driver1.addRide(Donosti, Bilbo, UtilDate.newDate(year,month,15), 7, car1);
driver1.addRide(Donosti, "Gazteiz", UtilDate.newDate(year,month,6), 8, car1);
driver1.addRide(Bilbo, Donosti, UtilDate.newDate(year,month,25), 4, car5);
driver1.addRide(Donosti, "Iruña", UtilDate.newDate(year,month,7), 8, car5);

driver2.addRide(Donosti, Bilbo, UtilDate.newDate(year,month,15), 3, car2);
driver2.addRide(Bilbo, Donosti, UtilDate.newDate(year,month,25), 5, car4);
driver2.addRide("Eibar", "Gasteiz", UtilDate.newDate(year,month,6), 5, car2);

driver3.addRide(Bilbo, Donosti, UtilDate.newDate(year,month,14), 3, car3);
```

CÓDIGO FINAL:

```
//Create rides
String Bilbo = "Bilbo";
String Donosti = "Donostia";
String Gazteiz = "Gazteiz";
driver1.addRide(Donosti, Bilbo, UtilDate.newDate(year,month,15), 7, car1);
driver1.addRide(Donosti, Gazteiz, UtilDate.newDate(year,month,6), 8, car1);
driver1.addRide(Bilbo, Donosti, UtilDate.newDate(year,month,25), 4, car5);
driver1.addRide(Donosti, "Iruña", UtilDate.newDate(year,month,7), 8, car5);

driver2.addRide(Donosti, Bilbo, UtilDate.newDate(year,month,15), 3, car2);
driver2.addRide(Bilbo, Donosti, UtilDate.newDate(year,month,25), 5, car4);
driver2.addRide("Eibar", Gazteiz, UtilDate.newDate(year,month,6), 5, car2);

driver3.addRide(Bilbo, Donosti, UtilDate.newDate(year,month,14), 3, car3);
```

DESCRIPCIÓN:

En el código inicial, el String “Gazteiz” se repetía varias veces a lo largo del código, para evitar esa repetición de código, hemos creado un valor estático, llamado Bilbao, en el que le he metido el valor del string “Gazteiz” para poder sustituir ese atributo por el que estaba repetido.

"Write short units of code"

CÓDIGO INICIAL:

```
public void initializeDB(){

    db.getTransaction().begin();

    try {

       Calendar today = Calendar.getInstance();

       int month=today.get(Calendar.MONTH);
       int year=today.get(Calendar.YEAR);
       if (month==12) { month=1; year+=1;}


        //Create drivers
        Driver driver1=new Driver("driver1@gmail.com","Aitor Fernandez", "123");
        Driver driver2=new Driver("driver2@gmail.com","Ane Gaztañaga", "456");
        Driver driver3=new Driver("driver3@gmail.com","Test driver", "789");

        Car car1 = new Car("1234 ABC", 4, driver1, false);
        Car car2 = new Car("2345 DFG", 4, driver2, false);
        Car car3 = new Car("3456 HIJ", 6, driver3, true);
        Car car4 = new Car("4567 KLM", 9, driver2, true);
        Car car5 = new Car("5678 NÑO", 1, driver1, false);

        driver1.addCar(car1);
        driver1.addCar(car5);
        driver2.addCar(car4);
        driver2.addCar(car2);
        driver3.addCar(car3);


        //Create rides
        driver1.addRide("Donostia", Bilbo, UtilDate.newDate(year,month,15), 7, car1);
        driver1.addRide("Donostia", "Gazteiz", UtilDate.newDate(year,month,6), 8, car1);
        driver1.addRide(Bilbo, "Donostia", UtilDate.newDate(year,month,25), 4, car5);
        driver1.addRide("Donostia", "Iruña", UtilDate.newDate(year,month,7), 8, car5);

        driver2.addRide("Donostia", Bilbo, UtilDate.newDate(year,month,15), 3, car2);
        driver2.addRide(Bilbo, "Donostia", UtilDate.newDate(year,month,25), 5, car4);
        driver2.addRide("Eibar", "Gasteiz", UtilDate.newDate(year,month,6), 5, car2);

        driver3.addRide(Bilbo, "Donostia", UtilDate.newDate(year,month,14), 3, car3);

        Admin admin1 = new Admin("aitzol@gmail.com", "Aitzol", "123");
        Admin admin2 = new Admin("eneko@gmail.com", "Eneko", "123");

        db.persist(driver1);
        db.persist(driver2);
        db.persist(driver3);

        db.persist(admin1);
        db.persist(admin2);

        db.getTransaction().commit();
        System.out.println("Db initialized");
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
```

CÓDIGO FINAL:

```
public void initializeDB() {
    db.getTransaction().begin();
    try {
        Calendar today = Calendar.getInstance();
        int month = today.get(Calendar.MONTH);
        int year = today.get(Calendar.YEAR);
        if (month == 12) { month = 1; year += 1; }

        List<Driver> drivers = createDrivers();
        assignCarsToDrivers(drivers);
        createRides(drivers, month, year);

        List<Admin> admins = createAdmins();

        persistEntities(drivers, admins);

        db.getTransaction().commit();
        System.out.println("Db initialized");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private List<Driver> createDrivers() {
    List<Driver> drivers = new ArrayList<>();
    Driver driver1 = new Driver("driver1@gmail.com","Aitor Fernandez", "123");
    Driver driver2 = new Driver("driver2@gmail.com","Ane Gaztañaga", "456");
    Driver driver3 = new Driver("driver3@gmail.com","Test driver", "789");
    drivers.add(driver1);
    drivers.add(driver2);
    drivers.add(driver3);
    return drivers;


}

private void assignCarsToDrivers(List<Driver> drivers) {
    Driver driver1 = drivers.get(0);
    Driver driver2 = drivers.get(1);
    Driver driver3 = drivers.get(2);

    Car car1 = new Car("1234 ABC", 4, driver1, false);
    Car car2 = new Car("2345 DFG", 4, driver2, false);
    Car car3 = new Car("3456 HIJ", 6, driver3, true);
    Car car4 = new Car("4567 KLM", 9, driver2, true);
    Car car5 = new Car("5678 NÑO", 1, driver1, false);

    driver1.addCar(car1);
    driver1.addCar(car5);
    driver2.addCar(car2);
    driver2.addCar(car4);
    driver3.addCar(car3);
}
```

```java
private void createRides(List<Driver> drivers, int month, int year) {
    Driver driver1 = drivers.get(0);
    Driver driver2 = drivers.get(1);
    Driver driver3 = drivers.get(2);

    driver1.addRide("Donostia", Bilbo, UtilDate.newDate(year, month, 15), 7, driver1.getCars().get(0));
    driver1.addRide("Donostia", "Gazteiz", UtilDate.newDate(year, month, 6), 8, driver1.getCars().get(0));
    driver1.addRide(Bilbo, "Donostia", UtilDate.newDate(year, month, 25), 4, driver1.getCars().get(1));
    driver1.addRide("Donostia", "Iruña", UtilDate.newDate(year, month, 7), 8, driver1.getCars().get(1));

    driver2.addRide("Donostia", Bilbo, UtilDate.newDate(year, month, 15), 3, driver2.getCars().get(0));
    driver2.addRide(Bilbo, "Donostia", UtilDate.newDate(year, month, 25), 5, driver2.getCars().get(1));
    driver2.addRide("Eibar", "Gasteiz", UtilDate.newDate(year, month, 6), 5, driver2.getCars().get(0));

    driver3.addRide(Bilbo, "Donostia", UtilDate.newDate(year, month, 14), 3, driver3.getCars().get(0));
}

private List<Admin> createAdmins() {
    List<Admin> admins = new ArrayList<>();
    Admin admin1 = new Admin("aitzol@gmail.com", "Aitzol", "123");
    Admin admin2 = new Admin("eneko@gmail.com", "Eneko", "123");
    admins.add(admin1);
    admins.add(admin2);
    return admins;

}

private void persistEntities(List<Driver> drivers, List<Admin> admins) {
    for (Driver d : drivers) db.persist(d);
    for (Admin a : admins) db.persist(a);
}
```

DESCRIPCIÓN:

Lo que se ha hecho en la refactorización de initializeDB() ha sido dividir un método largo y complejo en varias unidades más pequeñas y más fáciles de manejar, cada una con menos de 15 líneas y con una responsabilidad clara: crear drivers, asignar coches, crear rides, crear administradores y persistir las entidades. El método principal se encarga únicamente de coordinar estos pasos y finalizar la transacción, conservando la misma funcionalidad, pero logrando un código más claro, fácil de mantener y de testear cada componente por separado, siguiendo las buenas prácticas de “Write short units of code” y “Write simple units of code”.

“Keep unit interfaces small”

CÓDIGO INICIAL:

```
public Ride createRide(String from, String to, Date date, float price, String driverEmail, String carPlate) throws  RideAlreadyExistException, RideMustBeLaterThanTodayException {
```

CÓDIGO FINAL:

```
package dataAccess;

import java.util.Date;

public class RideInfo {
    private String from;
    private String to;
    private Date date;
    private float price;
    private String driverEmail;
    private String carPlate;

    public RideInfo(String from, String to, Date date, float price, String driverEmail, String carPlate) {
        this.from = from;
        this.to = to;
        this.date = date;
        this.price = price;
        this.driverEmail = driverEmail;
        this.carPlate = carPlate;
    }


    public String getFrom() {
        return from;
    }

    public String getTo() {
        return to;
    }

    public Date getDate() {
        return date;

    }
    public float getPrice() {
        return price;
    }

    public String getDriverEmail() {
        return driverEmail;
    }

    public String getCarPlate() {
        return carPlate;
    }
}
public Ride createRide(RideInfo rideInfo) throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    System.out.println(">> DataAccess: createRide=> from= "+rideInfo.getFrom()+" to= "+rideInfo.getTo()+" driver="+rideInfo.getDriverEmail()+" date "+rideInfo.getDate());

    db.getTransaction().begin();
    try {
        Driver driver = db.find(Driver.class, rideInfo.getDriverEmail());
        Car car = db.find(Car.class, rideInfo.getCarPlate());

        validateRide(rideInfo, driver);

        Ride ride = createAndPersistRide(rideInfo, driver, car);

        db.getTransaction().commit();
        return ride;
    } catch (NullPointerException e) {
        db.getTransaction().commit();
        return null;
    }
}
private void validateRide(RideInfo rideInfo, Driver driver) throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    if (new Date().compareTo(rideInfo.getDate()) > 0) {
        throw new RideMustBeLaterThanTodayException(ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBeLaterThanToday"));
    }

    if (driver.doesRideExists(rideInfo.getFrom(), rideInfo.getTo(), rideInfo.getDate())) {
        throw new RideAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
    }
}

private Ride createAndPersistRide(RideInfo rideInfo, Driver driver, Car car) {
    Ride ride = driver.addRide(rideInfo.getFrom(), rideInfo.getTo(), rideInfo.getDate(), rideInfo.getPrice(), car);
    db.persist(driver);
    return ride;
}
```

DESCRIPCIÓN:

He creado una nueva clase llamada RideInfo que incluye en sus atributos todos los datos necesarios para crear un viaje, como el origen, el destino, la fecha, el precio, el correo del conductor y la matrícula del coche, y he modificado el método createRide para que reciba este objeto en lugar de seis parámetros individuales, de manera que dentro del método se utilizan los getters de RideInfo para acceder a cada dato y realizar las comprobaciones y operaciones necesarias, lo que simplifica la interfaz del método, reduce la complejidad y facilita la mantenibilidad.

"Write simple units of code"

CÓDIGO INICIAL:

```
public Ride createRide(RideInfo rideInfo) throws  RideAlreadyExistException, RideMustBeLaterThanTodayException {
    System.out.println(">> DataAccess: createRide=> from= "+rideInfo.getFrom()+" to= "+rideInfo.getTo()+" driver="+rideInfo.getDriverEmail()+" date "+rideInfo.getDate());
    try {
        if(new Date().compareTo(rideInfo.getDate())>0) {
            throw new RideMustBeLaterThanTodayException(ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }
        db.getTransaction().begin();

        Driver driver = db.find(Driver.class, rideInfo.getDriverEmail());
        if (driver.doesRideExists(rideInfo.getFrom(), rideInfo.getTo(), rideInfo.getDate())) {
            db.getTransaction().commit();
            throw new RideAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
        }

        Car car = db.find(Car.class, rideInfo.getCarPlate());
        Ride ride = driver.addRide(rideInfo.getFrom(), rideInfo.getTo(), rideInfo.getDate(), rideInfo.getPrice(), car);
        //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 FINAL:

```
public Ride createRide(RideInfo rideInfo) throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    System.out.println(">> DataAccess: createRide=> from= "+rideInfo.getFrom()+" to= "+rideInfo.getTo()+" driver="+rideInfo.getDriverEmail()+" date "+rideInfo.getDate());

    db.getTransaction().begin();
    try {
        Driver driver = db.find(Driver.class, rideInfo.getDriverEmail());
        Car car = db.find(Car.class, rideInfo.getCarPlate());

        validateRide(rideInfo, driver);

        Ride ride = createAndPersistRide(rideInfo, driver, car);

        db.getTransaction().commit();
        return ride;
    } catch (NullPointerException e) {
        db.getTransaction().commit();
        return null;
    }
}
private void validateRide(RideInfo rideInfo, Driver driver) throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    if (new Date().compareTo(rideInfo.getDate()) > 0) {
        throw new RideMustBeLaterThanTodayException(ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBeLaterThanToday"));
    }

    if (driver.doesRideExists(rideInfo.getFrom(), rideInfo.getTo(), rideInfo.getDate())) {
        throw new RideAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
    }
}

private Ride createAndPersistRide(RideInfo rideInfo, Driver driver, Car car) {
    Ride ride = driver.addRide(rideInfo.getFrom(), rideInfo.getTo(), rideInfo.getDate(), rideInfo.getPrice(), car);
    db.persist(driver);
    return ride;
}
```

DESCRIPCIÓN:

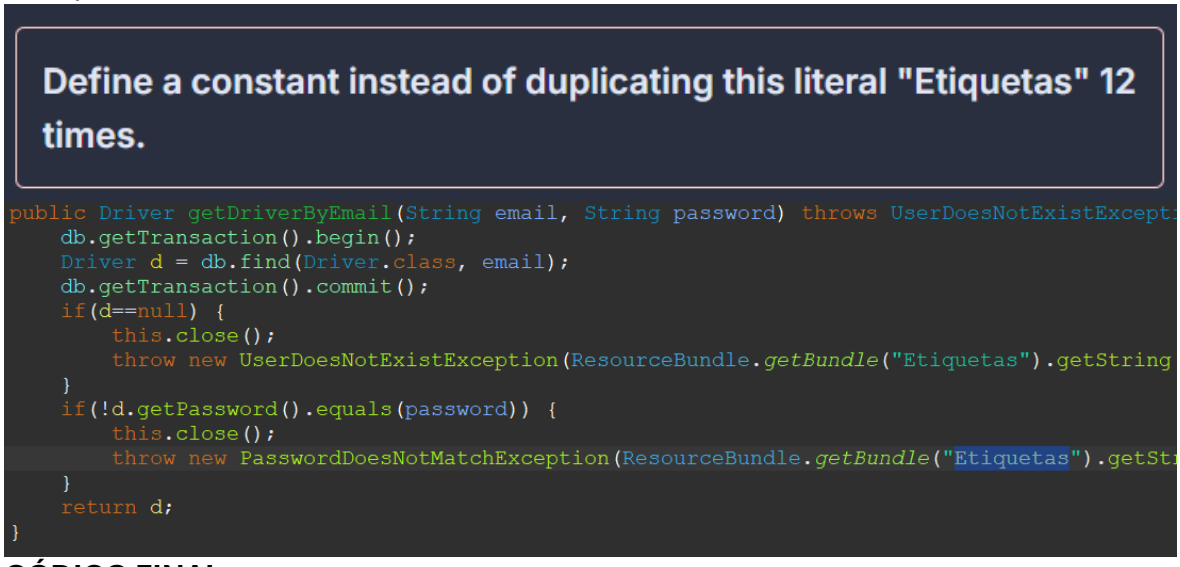
He cambiado el método createRide(RideInfo rideInfo) para que sea más fácil de entender y manejar. Ahora en lugar de tener todo en un solo método, lo he dividido en partes más pequeñas. Por ejemplo, la parte que revisa si la fecha del viaje es correcta y si el ride ya existe se hace en un método aparte llamado validateRide(), y la parte que crea y guarda el ride se hace en otro método llamado createAndPersistRide(). Así cada método hace solo una cosa, es más sencillo de leer, de probar y de modificar. Básicamente lo que he hecho es separar la validación de la creación del ride para que el código sea más claro y menos complicado.

AUTOR: Agustín Beltrán de Heredia

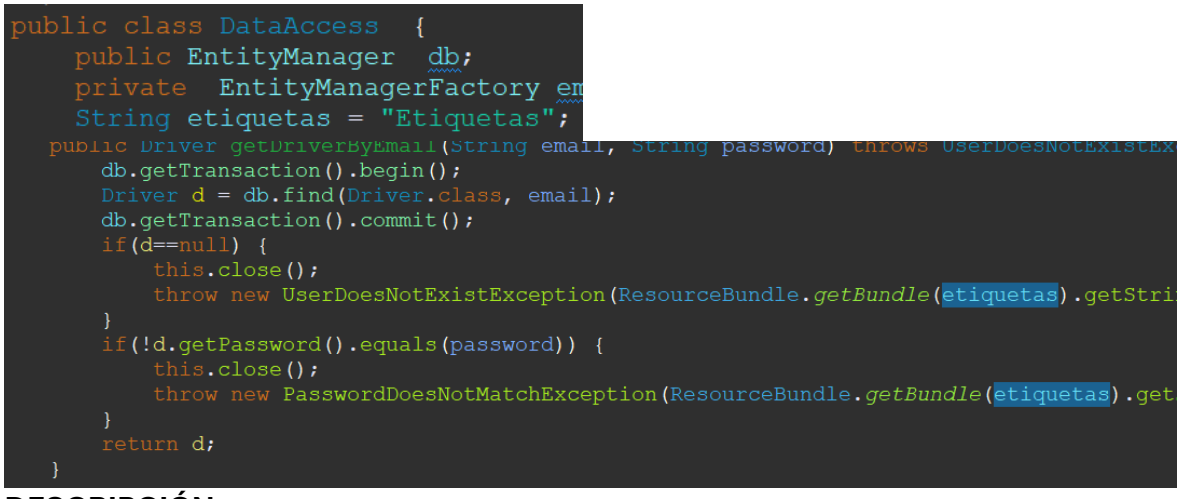
"Duplicate code"

CÓDIGO INICIAL:

Ya que “Etiquetas” se encuentra entre distintos métodos solo pondré la captura de uno, pero cambiaré todos



CÓDIGO FINAL:



DESCRIPCIÓN:

En el código inicial, el string “Etiquetas” se repetía 12 veces a lo largo de todo el código. Para evitar dicha repetición, he creado un valor estático, llamado etiquetas, en el que le he asignado el string repetido para poder sustituirlo evitando así la duplicación.

"Write short units of code"

CÓDIGO INICIAL:

```
public void pay(Reservation res) throws NotEnoughMoneyException{
    db.getTransaction().begin();
    try {
        Traveler t = db.find(Traveler.class, res.getTraveler().getEmail());
        Driver d = db.find(Driver.class, res.getDriver().getEmail());
        float price = res.getHmTravelers()*res.getRide().getPrice();
        if(t.getMoney()-price<0){
            db.getTransaction().commit();
            throw new NotEnoughMoneyException();
        }
        Reservation r = db.find(Reservation.class, res.getReservationCode());
        r.setPayed(true);
        t.setMoney(t.getMoney()-price);
        d.setMoney(d.getMoney()+price);
        Transaction tr = new Transaction(price, d, t);
        d.addTransaction(tr);
        t.addTransaction(tr);
        db.persist(r);
        db.persist(tr);
        db.persist(d);
        db.persist(t);
        db.getTransaction().commit();
    }catch(NullPointerException e) {
        db.getTransaction().commit();
    }
}
```

CÓDIGO FINAL:

```
public void pay(Reservation res) throws NotEnoughMoneyException{
    db.getTransaction().begin();
    try {
        Traveler t = db.find(Traveler.class, res.getTraveler().getEmail());
        Driver d = db.find(Driver.class, res.getDriver().getEmail());
        float price = res.getHmTravelers()*res.getRide().getPrice();

        validateFunds(t,price);

        Reservation r = db.find(Reservation.class, res.getReservationCode());

        processPayment(r,d,t,price);
        db.getTransaction().commit();
    }catch(NullPointerException e) {
        db.getTransaction().commit();
    }
}
//Método auxiliar para pay()
private void validateFunds(Traveler t, float price) throws NotEnoughMoneyException {
    if(t.getMoney()-price<0){
        throw new NotEnoughMoneyException();
    }
}
//Método auxiliar para pay()
private void processPayment(Reservation r, Driver d, Traveler t, float price) {
    r.setPayed(true);
    t.setMoney(t.getMoney()-price);
    d.setMoney(d.getMoney()+price);
    Transaction tr = new Transaction(price, d, t);
    d.addTransaction(tr);
    t.addTransaction(tr);
    db.persist(r);
    db.persist(tr);
    db.persist(d);
    db.persist(t);
}
```

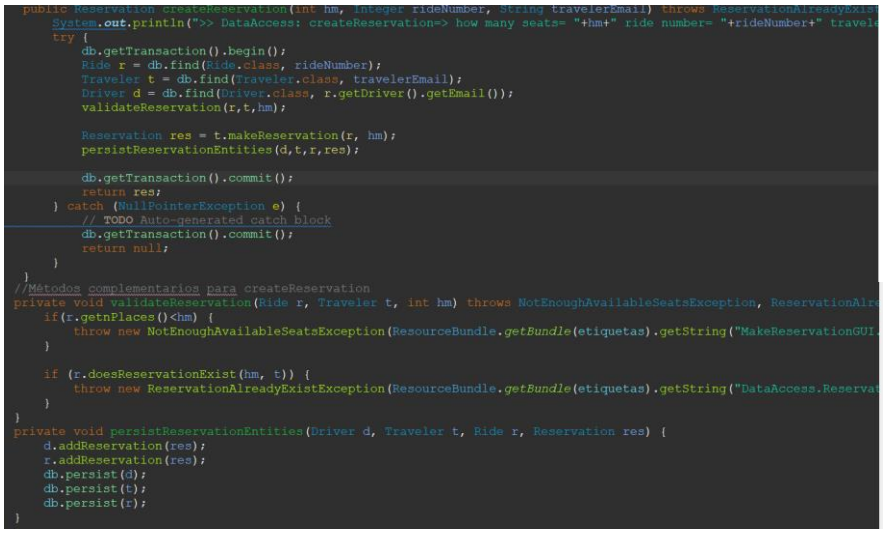
DESCRIPCIÓN: Lo que se ha hecho en la refactorización de pay() ha sido dividir un método largo y complejo en unidades más pequeñas y fáciles de manejar, cada una con menos de 15 líneas y con una responsabilidad clara: verificar que el traveler tenga dinero suficiente para realizar el pago, y actualizar el estado de la reserva, modificando los saldos del conductor y del viajero, después creando la transacción y persistiendo los cambios en la base de datos. Gracias a esta separación aumenta la claridad, organización y mantenibilidad del código, además, no se altera el comportamiento del sistema.

"Write simple units of code"

CÓDIGO INICIAL:

```
public Reservation createReservation(int hm, Integer rideNumber, String travelerEmail) throws ReservationAlreadyExist
    System.out.println(">> DataAccess: createReservation=> how many seats= "+hm+" ride number= "+rideNumber+" travele
    try {
        db.getTransaction().begin();
        Ride r = db.find(Ride.class, rideNumber);
        if(r.getnPlaces()<hm) {
            throw new NotEnoughAvailableSeatsException(ResourceBundle.getBundle(etiquetas).getString("MakeReservatio
        }

        Traveler t = db.find(Traveler.class, travelerEmail);
        Driver d = db.find(Driver.class, r.getDriver().getEmail());

        if (r.doesReservationExist(hm, t)) {
            db.getTransaction().commit();
            throw new ReservationAlreadyExistException(ResourceBundle.getBundle(etiquetas).getString("DataAccess.Rese
        }
        Reservation res = t.makeReservation(r, hm);

        d.addReservation(res);
        r.addReservation(res);
        db.persist(d);
        db.persist(t);
        db.persist(r);
        db.getTransaction().commit();

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

CÓDIGO FINAL:



```
public Reservation createReservation(int hm, Integer rideNumber, String travelerEmail) throws ReservationAlreadyExist
    System.out.println(">> DataAccess: createReservation=> how many seats= "+hm+" ride number= "+rideNumber+" travele
    try {
        db.getTransaction().begin();
        Ride r = db.find(Ride.class, rideNumber);
        Traveler t = db.find(Traveler.class, travelerEmail);
        Driver d = db.find(Driver.class, r.getDriver().getEmail());
        validateReservation(r,t,hm);

        Reservation res = t.makeReservation(r, hm);
        persistReservationEntities(d,t,r,res);

        db.getTransaction().commit();
        return res;
    } catch (NullPointerException e) {
        // TODO Auto-generated catch block
        db.getTransaction().commit();
        return null;
    }
}
//Métodos complementarios para createReservation
private void validateReservation(Ride r, Traveler t, int hm) throws NotEnoughAvailableSeatsException, ReservationAlre
    if(r.getnPlaces()<hm) {
        throw new NotEnoughAvailableSeatsException(ResourceBundle.getBundle(etiquetas).getString("MakeReservationGUI
    }

    if (r.doesReservationExist(hm, t)) {
        throw new ReservationAlreadyExistException(ResourceBundle.getBundle(etiquetas).getString("DataAccess.Reserva
    }
}
private void persistReservationEntities(Driver d, Traveler t, Ride r, Reservation res) {
    d.addReservation(res);
    r.addReservation(res);
    db.persist(d);
    db.persist(t);
    db.persist(r);
}
```

DESCRIPCIÓN: He cambiado el método createReservation para que sea más fácil de entender y manejar. Ahora en lugar de tener todo en un solo método, lo he dividido en partes más pequeñas, ya que se superaba el límite recomendado de 4 branchs por unidad de código según la guideline. Para reducir su complejidad he dividido el método y creado 2 nuevas unidades: validateReservation se encargaba de realizar las validaciones necesarias para poder crear la reserva, y persist ReservationEntities se encarga de gestionar la persistencia de varios objetos en la BD. Por lo tanto, el código ahora es más modular, limpio y fácil de extender o modificar en el futuro.

"Keep unit interfaces small"

CÓDIGO INICIAL:

```
public void addCarToDriver(String driverEmail, String carPlate, int nPlaces, boolean dis) throws CarA
    db.getTransaction().begin();
    Driver d = db.find(Driver.class, driverEmail);
    Car c = db.find(Car.class, carPlate);
    if(c != null) {
        db.getTransaction().commit();
        throw new CarAlreadyExistsException();
    }
    Car car = new Car (carPlate, nPlaces, d, dis);
    d.addCar(car);
    db.persist(car);
    db.persist(d);
    db.getTransaction().commit();
}
```

CÓDIGO FINAL:

```
public void addCarToDriver(String driverEmail, CarInfo carInfo) throws CarAlreadyExistsException
    db.getTransaction().begin();
    Driver d = db.find(Driver.class, driverEmail);
    Car c = db.find(Car.class, carInfo.getCarPlate());
    if(c != null) {
        db.getTransaction().commit();
        throw new CarAlreadyExistsException();
    }
    Car car = new Car (carInfo.getCarPlate(), carInfo.getnPlaces(), d, carInfo.isDis());
    d.addCar(car);
    db.persist(car);
    db.persist(d);
    db.getTransaction().commit();
}
public class CarInfo {
    String carPlate;
    int nPlaces;
    boolean dis;
    public CarInfo(String carPlate, int nPlaces, boolean dis) {
        this.carPlate = carPlate;
        this.nPlaces = nPlaces;
        this.dis = dis;
    }

    public String getCarPlate() {
        return carPlate;
    }
    public int getnPlaces() {
        return nPlaces;
    }
    public boolean isDis() {
        return dis;
    }
```

DESCRIPCIÓN: He creado una nueva clase llamada CarInfo que incluye en sus atributos todos los datos necesarios que necesita un coche en este método: la matricula, el número de asientos y si está disponible o no. Posteriormente, he modificado el método addCarToDriver para que utilice un objeto de dicha clase, utilizando dentro del método los getters para acceder a los datos y realizar las operaciones necesarias. De esta manera la interfaz del método es más pequeña y fácil, ya que solo hay 2 parametros en vez de 4.

AUTOR: GAIZKA ACEDO

"Write short units of code "

CÓDIGO INICIAL:

```
public void removeRideDriver(Integer rideNumber, String email) {
    System.out.println(">> DataAccess: removeRideDriver=> ride number= "+rideNumber+" Driver="+email)
    try {
        db.getTransaction().begin();
        Ride r = db.find(Ride.class, rideNumber);
        List<Reservation>resList=r.getReservations();
        this.returnMoneyTravelers(resList, email);
        Driver d = db.find(Driver.class, email);
        d.removeRide(r.getFrom(), r.getTo(), r.getDate());
        db.remove(r);
        db.persist(d);
        db.getTransaction().commit();
    } catch (NullPointerException e) {
        // TODO Auto-generated catch block
        db.getTransaction().commit();
    }
```

CÓDIGO FINAL:

```
  public void removeRideDriver(Integer rideNumber, String email) {
      System.out.println(">> DataAccess: removeRideDriver=> ride number= "+rideNumber+" Driver="+email)
      removeRide2(rideNumber, email);
  }

  private void removeRide2(Integer rideNumber, String email) {
      try {
          db.getTransaction().begin();
          Ride r = db.find(Ride.class, rideNumber);
          List<Reservation> resList = r.getReservations();
          this.returnMoneyTravelers(resList, email);
          Driver d = db.find(Driver.class, email);
          d.removeRide(r.getFrom(), r.getTo(), r.getDate());
          db.remove(r);
          db.persist(d);
          db.getTransaction().commit();
      } catch (NullPointerException e) {
          db.getTransaction().commit();
      }
```

DESCRIPCIÓN: El método tiene 20 líneas de código y lo parto en dos creando otro método que calcule y luego lo llama al final

“Write simple units of code ”

CÓDIGO INICIAL:

```
private boolean doesAlertExist(Traveler tra, String jatorria, String helmuga) {
    Traveler t = db.find(Traveler.class, tra.getEmail());
    Alert momentukoa = new Alert(jatorria, helmuga, t);
    List<Alert>alertList = t.getAlerts();
    System.out.println(alertList.size());
    boolean found = false;
    int i = 0;
    while(!found && i<alertList.size()) {
        if(alertList.get(i).equals(momentukoa)) {
            found = true;
        }
        i++;
    }
    return found;
}
```

CÓDIGO FINAL:

```
private boolean doesAlertExist(Traveler tra, String jatorria, String helmuga) {
    Traveler traveler = db.find(Traveler.class, tra.getEmail());
    Alert targetAlert = new Alert(jatorria, helmuga, traveler);
    List<Alert> alerts = traveler.getAlerts();
    System.out.println(alerts.size());
    return containsAlert(alerts, targetAlert);
}

private boolean containsAlert(List<Alert> alerts, Alert targetAlert) {
    for (Alert alert : alerts) {
        if (alert.equals(targetAlert)) {
            return true;
        }
    }
    return false;
}
```

DESCRIPCIÓN: Separo el método en dos para sacar los bucles fuera y luego llamarlos el otro método. Hago esto para bajar la complejidad del método.

“Duplicated code”

CÓDIGO INICIAL:

```
public List<Transaction> getTravelerTransactions(String email){
    db.getTransaction().begin();
    Traveler t = db.find(Traveler.class, email);
    db.getTransaction().commit();
    return t.getTransactions();
}

public List<Transaction> getDriverTransactions(String email){
    db.getTransaction().begin();
    Driver d = db.find(Driver.class, email);
    db.getTransaction().commit();
    return d.getTransactions();
}
```

CÓDIGO FINAL:

```
public List<Transaction> getTravelerTransactions(String email){
    return getTransactions(Traveler.class, email);
}

public List<Transaction> getDriverTransactions(String email){
    return getTransactions(Driver.class, email);
}

private <T> List<Transaction> getTransactions(Class<T> userClass, String email) {
    db.getTransaction().begin();
    T user = db.find(userClass, email);
    db.getTransaction().commit();

    if (user instanceof Traveler) {
        return ((Traveler) user).getTransactions();
    } else if (user instanceof Driver) {
        return ((Driver) user).getTransactions();
    }
    return new ArrayList<>();
}
```

DESCRIPCIÓN: En vez de tener dos métodos idénticos, pero para dos tipos de objetos diferentes, he metido la condición y dependiendo del tipo de objeto que sean van por un lado u otro mediante el instanceof

"Keep unit interfaces small"

CÓDIGO INICIAL:

```
public void addRatingToTraveler(String e, int z, Integer resCode) {
    db.getTransaction().begin();
    Reservation r = db.find(Reservation.class, resCode);
    Traveler t = db.find(Traveler.class, e);
    r.setRatedD(true);
    t.addRating(z);
    db.persist(t);
    db.persist(r);
    db.getTransaction().commit();
}
```

CÓDIGO FINAL:

```
public void addRatingToTraveler(String email, int rating, Integer reservationCode) {
    db.getTransaction().begin();
    Reservation reservation = db.find(Reservation.class, reservationCode);
    Traveler traveler = db.find(Traveler.class, email);
    updateRating(reservation, traveler, rating);
    db.persist(traveler);
    db.persist(reservation);
    db.getTransaction().commit();
}

private void updateRating(Reservation reservation, Traveler traveler, int rating) {
    reservation.setRatedD(true);
    traveler.addRating(rating);
}
```

DESCRIPCIÓN: He separado dos líneas para que no supere el máximo de 4 atributos y además he puesto los nombres de los atributos al completo en vez de las acortadas como antes.