ERREFAKTORIZAZIOA (RIDES24)

Egileak: **Ander Mena, Ekhi Ugarte, Haritz Eizagirre**
Data: **2025/11/08**
Irakaslea: Jon Iturrioz
Irakasgaia: Software Ingeniaritza II

AURKIBIDEA

1. TALDEKIDEA: Haritz Eizaguirre

1. Write short units of code

- Hasierako kodea:

```
public Reservation createReservation(int nTravelers, Integer rideNumber, String travelerEmail)
throws ReservationAlreadyExistException, NotEnoughAvailableSeatsException{
                System.out.println(">> DataAccess: createReservation=> how many seats=
"+nTravelers+" ride number= "+rideNumber+" traveler="+travelerEmail);
                try {
                        db.getTransaction().begin();
                        Ride r = db.find(Ride.class, rideNumber);
                        if(r.getnPlaces()<nTravelers) {
                                throw new
NotEnoughAvailableSeatsException(ResourceBundle.getBundle("Etiquetas").getString("MakeRese
rvationGUI.jButtonError2"));
                        }

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

                        if (r.doesReservationExist(nTravelers, t)) {
                                db.getTransaction().commit();
                                throw new
ReservationAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.
ReservationAlreadyExist"));
                        }
                        Reservation res = t.makeReservation(r, nTravelers);
                        System.out.println("res: "+ res);
                        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;
                }
        }
```

- Errefaktorizatutako kodea:

```
        public Reservation createReservation(int nTravelers, Integer rideNumber, String
travelerEmail) throws ReservationAlreadyExistException, NotEnoughAvailableSeatsException{
            System.out.println(">> DataAccess: createReservation=> how many seats=
"+nTravelers+" ride number= "+rideNumber+" traveler="+travelerEmail);
            try {
                db.getTransaction().begin();
                Ride r = db.find(Ride.class, rideNumber);
                checkAvailableSeats(r, nTravelers);
                Traveler t = db.find(Traveler.class, travelerEmail);
                Driver d = db.find(Driver.class, r.getDriver().getEmail());
                ensureReservationDoesNotExist(r, nTravelers, t);
                Reservation res = persistNewReservation(r, t, d, nTravelers);
                db.getTransaction().commit();
                return res;
            } catch (NullPointerException e) {
                db.getTransaction().commit();
                return null;
            }
        }

        private void checkAvailableSeats(Ride r, int nTravelers) throws
NotEnoughAvailableSeatsException {
            if (r.getnPlaces() < nTravelers) {
                throw new
NotEnoughAvailableSeatsException(ResourceBundle.getBundle("Etiquetas").getString("MakeRese
rvationGUI.jButtonError2"));
            }
        }

        private void ensureReservationDoesNotExist(Ride r, int nTravelers, Traveler t) throws
ReservationAlreadyExistException {
            if (r.doesReservationExist(nTravelers, t)) {
                throw new
ReservationAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.
ReservationAlreadyExist"));
            }
        }

        private Reservation persistNewReservation(Ride r, Traveler t, Driver d, int nTravelers) {
            Reservation res = t.makeReservation(r, nTravelers);
            d.addReservation(res);
            r.addReservation(res);
            db.persist(d);
            db.persist(t);
            db.persist(r);
            return res;
        }
```

createReservation metodoa oso luzea denez hainbat metodo laguntzaile berri sortu ditugu aurreko metodoa txikiagoa eta ulertzeko errazagoa izateko. Gainera, horrela mantenigarriagoa izango da.

2. Duplicate code

- Hasierako kodea:

```
//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);
```

- Errefaktorizatutako kodea:

```
private static final String BILBO = "Bilbo";
private static final String DONOSTIA = "Donostia";
private static final String GASTEIZ = "Gasteiz";
private static final String IRUÑA = "Iruña";
private static final String EIBAR = "Eibar";

//Create rides
driver1.addRide(DONOSTIA, BILBO, UtilDate.newDate(year,month,15), 7, car1);
driver1.addRide(DONOSTIA, GASTEIZ, 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);
```

Lehenengo kodean hirien izenak string literalen bidez jartzen dira eta gainera horietako batzuk askotan errepikatzen dira, horretarako konstanteak sortu ditugu string literalen ordez jartzeko.
Konstanteak erabiltzeak (private static final), strings literalen ordez, akats tipografikoak eta balioen bikoizketa saihesten ditu. Gainera mantentzea errazten du, izen bat aldatzen bada, leku batean bakarrik aldatzen baita.

3. Keep unit interfaces small

- Hasierako kodea:

```
public Ride createRide(String from, String to, Date date, float price, String driverEmail, String carPlate) 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("Etiquetas").getString("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("Etiquetas").getString("DataAccess.RideAlreadyExist"));
                        }

                        Car car = db.find(Car.class, carPlate);
                        Ride ride = driver.addRide(from, to, date, price, 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;
                }
        }
```

- Errefaktorizatutako kodea:

```
package dataAccess;
import java.util.Date;
public class RideCreationRequest {
   private final String from;
   private final String to;
   private final Date date;
   private final float price;
   private final String driverEmail;
   private final String carPlate;

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

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

        Car car = db.find(Car.class, request.getCarPlate());
        Ride ride = driver.addRide(request.getFrom(), request.getTo(), request.getDate(),
request.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;
  }
}
```

createRide metodoak parametro gehiegi jasotzen ditu. Horretarako, RideCreationRequest klasea sortu dugu ondoren createRide metodoan parametro guztiak batera sartu ahal izateko eta horrela mantenigarritasuna hobetzeko.

2. TALDEKIDEA: Ander Mena

1. Write short units of code

- Hasierako kodea:

```
        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();
                }
        }
```

- Errefaktorizatutako kodea:

```
        public void pay(Reservation res) throws NotEnoughMoneyException{
                db.getTransaction().begin();
```

```
                try {
                        Traveler traveler = db.find(Traveler.class,
res.getTraveler().getEmail());
                        Driver driver = db.find(Driver.class, res.getDriver().getEmail());
                        float price = calculateReservationPrice(res);

                        validateTravelerHasSufficientFunds(traveler, price);

                        Reservation reservation = db.find(Reservation.class,
res.getReservationCode());
                        processPayment(reservation, traveler, driver, price);

                        db.getTransaction().commit();
                }catch(NullPointerException e) {
                        db.getTransaction().commit();
                }
        }

        private float calculateReservationPrice(Reservation res) {
                return res.getnTravelers() * res.getRide().getPrice();
        }

        private void validateTravelerHasSufficientFunds(Traveler traveler, float price)
throws NotEnoughMoneyException {
                if(traveler.getMoney() - price < 0) {
                        db.getTransaction().commit();
                        throw new NotEnoughMoneyException();
                }
        }

        private void processPayment(Reservation reservation, Traveler traveler, Driver
driver, float price) {
                reservation.setPayed(true);
                transferMoneyBetweenUsers(traveler, driver, price);
                Transaction transaction = new Transaction(price, driver, traveler);
                recordTransaction(reservation, transaction, traveler, driver);
        }
```

DataAccess klaseko “pay” metodoa laburtu da, bere barne logika kanpo-metodo laguntzaileetan sartuz. Metodo laguntzaile hauei izen esanguratsuak eman zaizkie egiten dutena ulerterraza izan dadin.

2. Duplicate code

- Hasierako kodea:

```
if(!r.isAccepted()) {
jLabelError.setText(ResourceBundle.getBundle("Etiquetas").getStri
ng("CheckOwnReservationsGUI.jLabelError3"));
}else {
...
```

CheckOwnReservationsGUI.java klasean 4 bider agertzen da errepikatuta kode zati desberdinetan "CheckOwnReservationsGUI.jLabelError3" string-a. String hau “hardcodeatu” beharrean, egokiagoa da konstante bat definitzea.

- Errefaktorizatutako kodea:

```
private static final String JLABEL_ERROR3 =
"CheckOwnReservationsGUI.jLabelError3";

...

}else {
jLabelError.setText(ResourceBundle.getBundle("Etiquetas").getString(JLABEL_ERROR3)
);
}
```

Orain etiketa hori eskuratu nahi den bakoitzean, string-a idatzi ordez, konstanteari egingo zaio erreferentzia, kode duplikatua saihestuz.

3. Keep unit interfaces small

- Hasierako kodea:

```
        public void addCarToDriver(String driverEmail, String carPlate, int nPlaces,
boolean dis) throws CarAlreadyExistsException{
                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();
        }
```

Ikus dezakegun bezala, metodoak 4 parametro ditu. Parametroak ordezkatuko ditugu CarRequest klase berri bateko objektu batekin.

- Errefaktorizatutako kodea:

Klase berria

```
package domain;
public class CarRequest {
   private final String carPlate;
   private final int nPlaces;
   private final Driver driver;
   private final boolean hasDiscount;

   public CarRequest(String carPlate, int nPlaces, Driver driver, boolean hasDiscount) {
      this.carPlate = carPlate;
      this.nPlaces = nPlaces;
      this.driver = driver;
      this.hasDiscount = hasDiscount;
   }

   public String getCarPlate() { return carPlate; }
   public int getNPlaces() { return nPlaces; }
   public Driver getDriver() { return driver; }
   public boolean hasDiscount() { return hasDiscount; }
}
```

```
        public void addCarToDriver(CarRequest request) throws
CarAlreadyExistsException{
                db.getTransaction().begin();
                Driver d = db.find(Driver.class, request.getDriver().getEmail());
                Car c = db.find(Car.class, request.getCarPlate());
                if(!(c == null)) {
                        db.getTransaction().commit();
                        throw new CarAlreadyExistsException();
                }
                Car car = new Car(request.getCarPlate(), request.getNPlaces(), d,
request.hasDiscount());
                d.addCar(car);
                db.persist(car);
                db.persist(d);
                db.getTransaction().commit();
        }
```

Orain 4 parametro pasa beharrean CarRequest objektua pasako dugu, eta parametroak objektu horren bitartez eskuratuko ditugu.

3. TALDEKIDEA: Ekhi Ugarte

1. Write shorts units of code

- Hasierako kodea:

```
	public void acceptComplaint(Complaint co) {
		db.getTransaction().begin();
		Complaint complaint = db.find(Complaint.class, co.getId());
		Driver d = db.find(Driver.class, complaint.getDriverEmail());
		Traveler t = db.find(Traveler.class, complaint.getTravelerEmail());
		Reservation res = db.find(Reservation.class,
co.getRes().getReservationCode());
		float price = res.getHmTravelers()*res.getRide().getPrice();
		d.setMoney(d.getMoney()-price);
		t.setMoney(t.getMoney()+price);
		d.removeComplaint(complaint);
		t.removeComplaint(complaint);
		Transaction tra = new Transaction(price, t, d);
		d.addTransaction(tra);
		t.addTransaction(tra);
		db.persist(tra);
		db.persist(t);
		db.persist(d);
		db.remove(complaint);
		db.getTransaction().commit();
	}
```

- Errefaktorizatutako kodea:

```
	public void acceptComplaint(Complaint co) {
		db.getTransaction().begin();
		Complaint complaint = db.find(Complaint.class, co.getId());
		Driver driver = db.find(Driver.class, complaint.getDriverEmail());
		Traveler traveler = db.find(Traveler.class, complaint.getTravelerEmail());
		Reservation reservation = db.find(Reservation.class,
co.getRes().getReservationCode());

		float refundAmount = calculateReservationPrice(reservation);
		processComplaintRefund(driver, traveler, complaint, refundAmount);

		db.getTransaction().commit();
	}


	private void processComplaintRefund(Driver driver, Traveler traveler, Complaint
complaint, float refundAmount) {
		refundMoneyToTraveler(driver, traveler, refundAmount);
		removeComplaintFromUsers(driver, traveler, complaint);
```

```
                Transaction refundTransaction = new Transaction(refundAmount, traveler,
driver);
                recordComplaintRefund(refundTransaction, driver, traveler, complaint);
        }

        private void refundMoneyToTraveler(Driver driver, Traveler traveler, float amount) {
                driver.setMoney(driver.getMoney() - amount);
                traveler.setMoney(traveler.getMoney() + amount);
        }

        private void removeComplaintFromUsers(Driver driver, Traveler traveler, Complaint
complaint) {
                driver.removeComplaint(complaint);
                traveler.removeComplaint(complaint);
        }
```

DataAccess klaseko “acceptComplaint” metodoa errefaktorizatu da bere barne-logika metodo laguntzailetara ateraz. Lehenago sortu dugun calculateReservationPrice erabilita eta refundMoneyToTraveler zein removeComplaintFromUsers berrien bitartez, jatorrizko metodoa sinplifikatu da, berriro ere izen esanguratsuak ezarrita.

2. Duplicate code

ActionListener kodea errepikatu egiten zen, errepikapena saiesteko metodo lagungarri bat sortu dut, createLocaleActionListener izenekoa:

```
private ActionListener createLocaleActionListener(String localeCode) {
                return new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                Locale.setDefault(new Locale(localeCode));
                                System.out.println("Locale: " + Locale.getDefault());
                                paintAgain();
                        }
                };
        }


        rdbtnNewRadioButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                Locale.setDefault(new Locale("en"));
                                System.out.println("Locale: "+Locale.getDefault());
                                paintAgain();                                }
                });

rdbtnNewRadioButton.addActionListener(createLocaleActionListener("en"));
```

3. Keep unit interfaces small

Hau addRating metodoko bertsio zaharra, 5 parametro pasa behar ziren.

```
private void addRating(String email, int rating, Integer reservationCode, Class<?>
userClass, boolean isDriver) {
    db.getTransaction().begin();
    Reservation reservation = db.find(Reservation.class, reservationCode);
    Object user = db.find(userClass, email);

    if (isDriver) {
        ((Driver) user).addRating(rating);
        reservation.setRatedT(true);
    } else {
        ((Traveler) user).addRating(rating);
        reservation.setRatedD(true);
    }

    db.persist(user);
    db.persist(reservation);
    db.getTransaction().commit();
}

public void addRatingToDriver(String email, int rating, Integer reservationCode) {
    addRating(email, rating, reservationCode, Driver.class, true);
}

public void addRatingToTraveler(String email, int rating, Integer reservationCode) {
    addRating(email, rating, reservationCode, Traveler.class, false);
}
```

AddRating metodo pribatuak zuen 5 parametroak elkartu ditut RatingRequest izeneko parametro-objektu batean eta orain addRating(RatingRequest req) erabiltzen da. Arrazoiak: parametro-kopurua murriztu (errazago ulertu eta mantentzeko) eta unitate-interfazeak txikiak izan daitezen.

Hau da objektu berria:

```
package dataAccess;

public class RatingRequest {
    private final String email;
    private final int rating;
    private final Integer reservationCode;
    private final Class<?> userClass;
    private final boolean isDriver;

    public RatingRequest(String email, int rating, Integer reservationCode, Class<?>
userClass, boolean isDriver) {
```

```
        this.email = email;
        this.rating = rating;
        this.reservationCode = reservationCode;
        this.userClass = userClass;
        this.isDriver = isDriver;
    }

    public String getEmail() {
        return email;
    }

    public int getRating() {
        return rating;
    }

    public Integer getReservationCode() {
        return reservationCode;
    }

    public Class<?> getUserClass() {
        return userClass;
    }

    public boolean isDriver() {
        return isDriver;
    }
}
```

eta hau kode errefaktorizatua:

```
private void addRating(dataAccess.RatingRequest req) {
    db.getTransaction().begin();
    Reservation reservation = db.find(Reservation.class, req.getReservationCode());
    Object user = db.find(req.getUserClass(), req.getEmail());

    if (req.isDriver()) {
        ((Driver) user).addRating(req.getRating());
        reservation.setRatedT(true);
    } else {
        ((Traveler) user).addRating(req.getRating());
        reservation.setRatedD(true);
    }

    db.persist(user);
    db.persist(reservation);
    db.getTransaction().commit();
}

public void addRatingToDriver(String email, int rating, Integer reservationCode) {
    addRating(new dataAccess.RatingRequest(email, rating, reservationCode, Driver.class,
true));
}
```

```
public void addRatingToTraveler(String email, int rating, Integer reservationCode) {
    addRating(new dataAccess.RatingRequest(email, rating, reservationCode,
Traveler.class, false));
}
```

Write simple units of code:

Gure proiektuan ez dago 4 baino gehiagoko konplexutasun ziklomatikoa duen metodorik, beraz bad smell hau ez da azaltzen proiektuan guk errefaktorizatzeko.