Refactorización de software

(Mikel Calleja)

Link: https://github.com/SikeMike/IS2-Rides24

Unidades de código largas (más de 15 líneas):

Método original:

```
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String
driverName)
                        throws RideAlreadyExistException,
RideMustBeLaterThanTodayException {
                System.out.println(
                                ">> DataAccess: createRide=> from= " + from + " to= " + to + "
driver=" + driverName + " date " + date);
                if (driverName==null) return null;
                try {
                        if (new Date().compareTo(date) > 0) {
                                System.out.println("ppppp");
                                throw new RideMustBeLaterThanTodayException(

ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBeLaterTh
anToday"));
                        }
                        db.getTransaction().begin();
                        Driver driver = db.find(Driver.class, driverName);
                        if (driver.doesRideExists(from, to, date)) {
                                db.getTransaction().commit();
                                throw new RideAlreadyExistException(

ResourceBundle.getBundle("Etiquetas").getString("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
                        return null;
                }}
```

Refactorización:

```
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String
driverName)
      throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
  System.out.println(">> DataAccess: createRide=> from= " + from + " to= " + to + " driver="
+ driverName + " date " + date);
  if (driverName == null) return null;

  validateRideDate(date);

  db.getTransaction().begin();
  Driver driver = db.find(Driver.class, driverName);
  checkIfRideExists(driver, from, to, date);

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

private void validateRideDate(Date date) throws RideMustBeLaterThanTodayException {
if (new Date().compareTo(date) > 0) {
        throw new RideMustBeLaterThanTodayException(
        ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBe
        LaterThanToday")); } }

private void checkIfRideExists(Driver driver, String from, String to, Date date) throws
RideAlreadyExistException {
if (driver.doesRideExists(from, to, date)) {
        db.getTransaction().commit(); throw new RideAlreadyExistException(
        ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
        } }
```

Explicación:

Este método tiene más de 15 líneas. Lo que hice fue dividir en métodos más pequeños para quitar complejidad al método. En mi caso:

- Extraer la validación de fecha a un nuevo método: validateRideDate()
- Extraer la lógica de verificar si el ride ya existe: checkIfRideExists()

Unidades de código complejas (más de 4 puntos de ramificación):

Método original:

```
public boolean isRegistered(String erab, String passwd) {
                TypedQuery<Long> travelerQuery = db.createQuery(
                                "SELECT COUNT(t) FROM Traveler t WHERE
t.username = :username AND t.passwd = :passwd", Long.class);
                travelerQuery.setParameter("username", erab);
                travelerQuery.setParameter("passwd", passwd);
                Long travelerCount = travelerQuery.getSingleResult();
                TypedQuery<Long> driverQuery = db.createQuery(
                                "SELECT COUNT(d) FROM Driver d WHERE
d.username = :username AND d.passwd = :passwd", Long.class);
                driverQuery.setParameter("username", erab);
                driverQuery.setParameter("passwd", passwd);
                Long driverCount = driverQuery.getSingleResult();

                boolean isAdmin = ((erab.compareTo("admin") == 0) &&
(passwd.compareTo(adminPass) == 0));
                return travelerCount > 0 || driverCount > 0 || isAdmin;
        }
```

Refactorización:

```
public boolean isRegistered(String erab, String passwd) {
  return isTravelerRegistered(erab, passwd) || isDriverRegistered(erab, passwd) ||
isAdmin(erab, passwd);
}

private boolean isTravelerRegistered(String erab, String passwd) {
  TypedQuery<Long> travelerQuery = db.createQuery(
    "SELECT COUNT(t) FROM Traveler t WHERE t.username = :username AND t.passwd
= :passwd", Long.class);
  travelerQuery.setParameter("username", erab);
  travelerQuery.setParameter("passwd", passwd);
  return travelerQuery.getSingleResult() > 0;
}

private boolean isDriverRegistered(String erab, String passwd) {
  TypedQuery<Long> driverQuery = db.createQuery(
    "SELECT COUNT(d) FROM Driver d WHERE d.username = :username AND d.passwd
= :passwd", Long.class);
  driverQuery.setParameter("username", erab);
  driverQuery.setParameter("passwd", passwd);
```

```
    return driverQuery.getSingleResult() > 0;
}

private boolean isAdmin(String erab, String passwd) {
    return erab.equals("admin") && passwd.equals(adminPass);
}
```

Explicación:

El método isRegistered() tiene múltiples puntos de ramificación(consulta de viajero,conductor…) lo que hice fue crear métodos auxiliares para delegar comprobaciones.

Demasiados parámetros:

Método original:

```
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverName)
```

Refactorización:

Nueva clase →

```
public class RideDetails {
	private String from;
	private String to;
	private Date date;
	private int nPlaces;
	private float price;
	private String driverName;
	public RideDetails(String from, String to, Date date, int nPlaces, float price, String driverName) {
		super();
		this.from = from;
		this.to = to;
		this.date = date;
		this.nPlaces = nPlaces;
		this.price = price;
		this.driverName = driverName;
	}
	public String getFrom() {
		return from;
	}
```

```
	public String getTo() {
		return to;
	}
	public Date getDate() {
		return date;
	}
	public int getNPlaces() {
		return nPlaces;
	}
	public float getPrice() {
		return price;
	}
	public String getDriverName() {
		return driverName;
	}
}
```

Nuevo método →

```
public Ride createRide(RideDetails details) throws RideAlreadyExistException,
RideMustBeLaterThanTodayException {
		System.out.println(">> DataAccess: createRide=> from= " +
details.getFrom() + " to= " + details.getTo()
				+ " driver=" + details.getDriverName() + " date " +
details.getDate());
		if (details.getDriverName() == null)
			return null;
		validateRideDate(details.getDate());
		db.getTransaction().begin();
		Driver driver = db.find(Driver.class, details.getDriverName());
		checkIfRideExists(driver, details.getFrom(), details.getTo(),
details.getDate());
		Ride ride = driver.addRide(details.getFrom(), details.getTo(),
details.getDate(), details.getNPlaces(),
				details.getPrice());
		db.persist(driver);
		db.getTransaction().commit();
		return ride;
	}
```

Explicación:

En este método hace uso de demasiados parámetros (6) por lo que cree una clase auxiliar que tenga de atributos los valores que usa, encapsulando . También se modifique el método de createRide() aparte que ahora solo use un parámetro cuando tiene que crear un driver o un ride llame a los getters oportunos. Además tuve que cambiar la clase BLFacadeImplementation por su método de createRide y ajustar las pruebas de test.

Código duplicado:

Método original:

```
	public String getMotabyUsername(String erab) {
		TypedQuery<String> driverQuery = db.createQuery("SELECT
d.mota FROM Driver d WHERE d.username = :username",
				String.class);
		driverQuery.setParameter("username", erab);
		List<String> driverResultList = driverQuery.getResultList();
		TypedQuery<String> travelerQuery = db.createQuery("SELECT
t.mota FROM Traveler t WHERE t.username = :username",
				String.class);
		travelerQuery.setParameter("username", erab);
		List<String> travelerResultList = travelerQuery.getResultList();
		/*
		 * TypedQuery<String> adminQuery =
		 * db.createQuery("SELECT a.mota FROM Admin a WHERE
a.username = :username",
		 * String.class); adminQuery.setParameter("username", erab);
List<String>
		 * adminResultList = adminQuery.getResultList();
		 */
		if (!driverResultList.isEmpty()) {
			return driverResultList.get(0);
		} else if (!travelerResultList.isEmpty()) {
			return travelerResultList.get(0);
		} else {
			return "Admin";
		}
	}
```

Refactorización:

```
public String getMotabyUsername(String erab) {
                String driverMota = getMotaForUser("Driver", erab);
                String travelerMota = getMotaForUser("Traveler", erab);

                if (driverMota != null) return driverMota;
                if (travelerMota != null) return travelerMota;
                return "Admin";
            }
            private String getMotaForUser(String userType, String erab) {
                TypedQuery<String> query = db.createQuery("SELECT u.mota
FROM " + userType + " u WHERE u.username = :username", String.class);
                query.setParameter("username", erab);
                List<String> resultList = query.getResultList();
                return resultList.isEmpty() ? null : resultList.get(0);
            }
```

Explicación:

Refactoricé el método getMotabyUsername para mejorar su mantenimiento. Extraje la lógica de las consultas en un nuevo método llamado getMotaForUser, que recibe el tipo de usuario como parámetro y devuelve la mota correspondiente. Esto hizo que el método principal fuera más claro y fácil de entender, y me permitirá agregar nuevos tipos de usuarios en el futuro sin duplicar el código como por ejemplo Admin.