Refactorización

por Ramón Casado y Leire Sesma

BookRide

Código Inicial

```
public boolean bookRide(String username, Ride ride,
int seats, double desk) {
    try {
         db.getTransaction().begin();
         Traveler traveler = getTraveler(username);
         if (traveler == null) {
              return false;
         }
         if (ride.getnPlaces() < seats) {
              return false;
         }
         double ridePriceDesk =
            (ride.getPrice() - desk) * seats;
         double availableBalance = traveler.getMoney();
         if (availableBalance < ridePriceDesk) {
              return false;
         }
         Booking booking = new Booking(
            ride, traveler, seats);
         booking.setTraveler(traveler);
         booking.setDeskontua(desk);
         db.persist(booking);
         ride.setnPlaces(ride.getnPlaces() - seats);
         traveler.addBookedRide(booking);
         traveler.setMoney(
            availableBalance - ridePriceDesk);
```

```
            traveler.setIzoztatutakoDirua(
               traveler.getIzoztatutakoDirua()
                +ridePriceDesk);
            db.merge(ride);
            db.merge(traveler);
            db.getTransaction().commit();
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            db.getTransaction().rollback();
            return false;
        }
}
```

Código Refactorizado

```
 public boolean bookRide(String username, Ride ride, int seats,
double desk) {
        try {
            Traveler traveler = getTraveler(username);
            if (traveler == null)
                return false;
            if (ride.getnPlaces() < seats)
                return false;
            return createBooking(ride, seats, desk, traveler);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
 }

 public boolean createBooking(Ride ride, int seats, double desk,
Traveler traveler) {
        double ridePriceDesk = (ride.getPrice() - desk) * seats;
        double availableBalance = traveler.getMoney();
        if (availableBalance < ridePriceDesk)
            return false;
```

```
        db.getTransaction().begin();
        Booking booking = registerBooking(ride, seats, desk,traveler);
        payBooking(booking, ridePriceDesk, availableBalance);
        db.getTransaction().commit();
        return true;
    }

    public void payBooking(Booking booking, double ridePriceDesk,
    double availableBalance) {
        Ride r = booking.getRide();
        int s = booking.getSeats();
        Traveler t = booking.getTraveler();
        r.setnPlaces(r.getnPlaces() - s);
        t.addBookedRide(booking);
        t.setMoney(availableBalance - ridePriceDesk);
        t.setIzoztatutakoDirua(t.getIzoztatutakoDirua()
         +ridePriceDesk);
        db.merge(r);
        db.merge(t);
    }

    public Booking registerBooking(Ride ride, int seats,
    double desk, Traveler traveler) {
        Booking booking = new Booking(ride, traveler, seats);
        booking.setTraveler(traveler);
        booking.setDeskontua(desk);
        db.persist(booking);
        return booking;
    }
```

Errores detectados y método de refactorización

El método bookRide contiene dos errores que he solucionado. El método contenía demasiadas líneas de código. El método tenía demasiadas responsabilidades. Además de crear el objeto Booking en la base de datos, también se encargaba de comprobar si el traveler tiene suficiente dinero para pagar el viaje, de restar los asientos disponibles para el ride y de descontarle el dinero al traveler. Estas responsabilidades han sido redistribuidas a métodos propios de modo que solo contengan una responsabilidad y de que los bloques de código sean menores de 14 líneas.

Refactorizado por Ramón Casado

CreateRide

Código Inicial

```
public Ride createRide(String from, String to, Date date,
int nPlaces, float price, String driverName)
     throws RideAlreadyExistException,
RideMustBeLaterThanTodayException {
     if (driverName == null)
          return null;
     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, 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) {
          return null;
     }
}
```

Código Refactorizado

```
public Ride createRide(List<String> info, Date date,
int nPlaces, float price)
```

```
                throws RideAlreadyExistException,
                RideMustBeLaterThanTodayException {
    if (info.get(2) == null)
          return null;
    try {
          if (new Date().compareTo(date) > 0) {
                String err=ResourceBundle.getBundle("Etiquetas")
                    .getString("CreateRideGUI.
               ErrorRideMustBeLaterThanToday");
                throw new RideMustBeLaterThanTodayException(err);
          }
          db.getTransaction().begin();
          Driver driver = db.find(Driver.class, info.get(2));
          if (driver.doesRideExists(
              info.get(0), info.get(1), date)) {
                db.getTransaction().commit();
                String err=ResourceBundle.getBundle("Etiquetas")
                   .getString("DataAccess.RideAlreadyExist");
                throw new RideAlreadyExistException(err);
          }
          Ride ride = driver.addRide(info.get(0), info.get(1),
             date, nPlaces, price);
          // next instruction can be obviated
          db.persist(driver);
          db.getTransaction().commit();
          return ride;
    } catch (NullPointerException e) {
          return null;
    }
}
```

Errores detectados y método de refactorización

El método create ride contiene un error de demasiados parámetros en su interfaz. El método contenía en la interfaz información de lugar de salida, lugar de llegada, fecha, número de asientos y conductor. Se ha reducido la interfaz de 6 elementos a 4 agrupando la información de lugar de salida, lugar de llegada y conductor en una lista de Strings ya que los tres compartían tipo. Ha hecho falta adaptar la implementación de la lógica de negocio y de un par de líneas de la interfaz GUI. También ha hecho falta adaptar los tests para que utilicen el método correctamente y se ha comprobado que sus resultados no han cambiado.

Refactorizado por Ramón Casado

GetDriver

Código Inicial

```
public Driver getDriver(String erab) {
    TypedQuery<Driver> query =
   db.createQuery("SELECT d FROM Driver d
      WHERE d.username = :username",
      Driver.class);
    query.setParameter("username", erab);
    return query.getSingleResult();
}
```

```
public List<Booking> getBookingFromDriver(
   String username) {
    try {
         db.getTransaction().begin();
         TypedQuery<Driver> query =
            db.createQuery("SELECT d FROM Driver d
            WHERE d.username = :username",
            Driver.class);
         query.setParameter("username", username);
         Driver driver = query.getSingleResult();
         List<Ride> rides=driver.getCreatedRides();
         List<Booking> bookings = new ArrayList<>();
         for (Ride ride : rides) {
              if (ride.isActive()) {
                   bookings.addAll(ride.getBookings());
              }
         }
         db.getTransaction().commit();
         return bookings;
    } catch (Exception e) {
         e.printStackTrace();
```

```
            db.getTransaction().rollback();
            return null;
        }
    }
```

```
public List<Ride> getRidesByDriver(String username) {
    try {
            db.getTransaction().begin();
            TypedQuery<Driver> query = db.createQuery(
               "SELECT d FROM Driver d
               WHERE d.username = :username",
               Driver.class);
            query.setParameter("username", username);
            Driver driver = query.getSingleResult();
            List<Ride> rides = driver.getCreatedRides();
            List<Ride> activeRides = new ArrayList<>();
            for (Ride ride : rides) {
                 if (ride.isActive()) {
                      activeRides.add(ride);
                 }
            }
            db.getTransaction().commit();
            return activeRides;
    } catch (Exception e) {
            e.printStackTrace();
            db.getTransaction().rollback();
            return null;
    }
}
```

Código Refactorizado

```
public List<Ride> getRidesByDriver(String username) {
    try {
```

```
            db.getTransaction().begin();
            Driver driver = getDriver(username);
            List<Ride> rides = driver.getCreatedRides();
            List<Ride> activeRides = new ArrayList<>();
            for (Ride ride : rides) {
                 if (ride.isActive()) {
                      activeRides.add(ride);
                 }
            }
            db.getTransaction().commit();
            return activeRides;
        } catch (Exception e) {
            e.printStackTrace();
            db.getTransaction().rollback();
            return null;
        }
}
```

Errores detectados y método de refactorización

El error en este caso se trata de código duplicado, en lugar de ejecutar la función getDriver(username) se estaba replicando el código de la misma función. No he utilizado un método de refactorización más allá de sustituir el fragmento por la ejecución del método. Aunque parece un error simple sin mucha gravedad, es un error muy peligroso ya que si decidimos cambiar el tipo Driver podría resultar en tener que cambiar todas las ocasiones en las que se busca un Driver. En el caso de la versión refactorizada, solo es necesario refactorizar getDriver.

Refactorizado por Ramón Casado

CancelRide

Código inicial:

```
public void cancelRide(Ride ride) {
    try {
        db.getTransaction().begin();

        for (Booking booking : ride.getBookings()) {
            if (booking.getStatus().equals("Accepted") || booking.getStatus().equals("NotDefined")) {
                double price = booking.prezioaKalkulatu();
                Traveler traveler = booking.getTraveler();
                double frozenMoney = traveler.getIzoztatutakoDirua();
                traveler.setIzoztatutakoDirua(frozenMoney - price);

                double money = traveler.getMoney();
                traveler.setMoney(money + price);
                db.merge(traveler);
                db.getTransaction().commit();
                addMovement(traveler, "BookDeny", price);
                db.getTransaction().begin();
            }
            booking.setStatus("Rejected");
            db.merge(booking);
        }
        ride.setActive(false);
        db.merge(ride);

        db.getTransaction().commit();
    } catch (Exception e) {
        if (db.getTransaction().isActive()) {
            db.getTransaction().rollback();
        }
        e.printStackTrace();
    }
}
```

Código refactorizado:

```
public void cancelRide(Ride ride) {
    try {
        db.getTransaction().begin();
        cancelRideBookings(ride);
        deactivateRide(ride);
        db.getTransaction().commit();
    } catch (Exception e) {
        handleTransactionFailure(e);
    }
}

private void cancelRideBookings(Ride ride) {
    for (Booking booking : ride.getBookings()) {
        if (shouldProcessBooking(booking)) {
            processBookingCancellation(booking);
        }
        booking.setStatus("Rejected");
        db.merge(booking);
    }
}

private boolean shouldProcessBooking(Booking booking) {
    return booking.getStatus().equals("Accepted") || booking.getStatus().equals("NotDefined");
}

private void processBookingCancellation(Booking booking) {
    double price = booking.prezioaKalkulatu();
    Traveler traveler = booking.getTraveler();
    updateTravelerFunds(traveler, price);
    addMovement(traveler, "BookDeny", price);
}
```

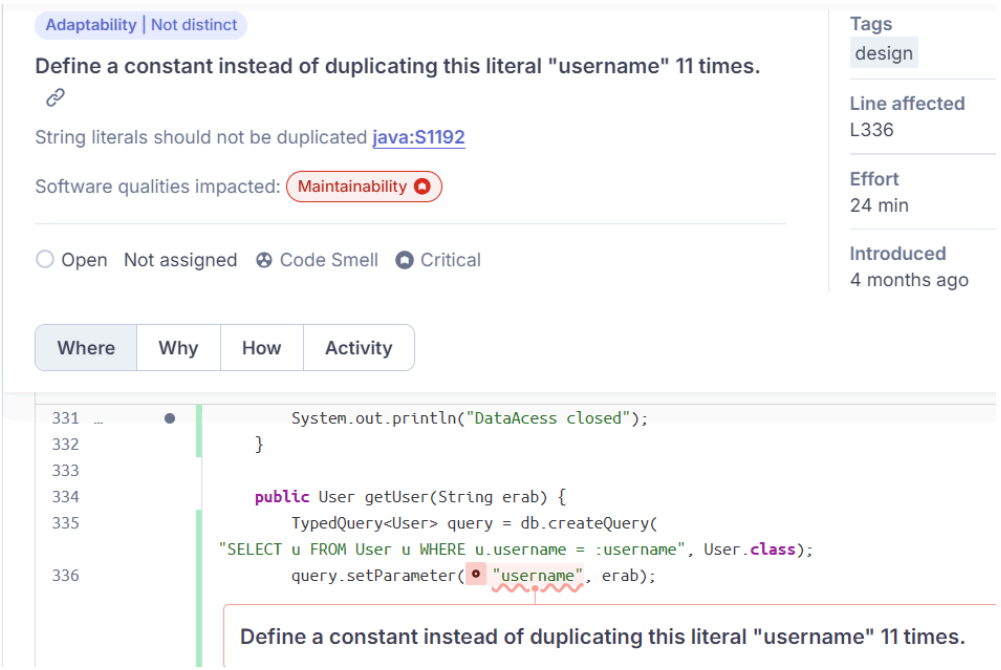
```
private void updateTravelerFunds(Traveler traveler, double price) {
    double frozenMoney = traveler.getIzoztatutakoDirua();
    traveler.setIzoztatutakoDirua(frozenMoney - price);

    double money = traveler.getMoney();
    traveler.setMoney(money + price);
    db.merge(traveler);
    db.getTransaction().commit();
    db.getTransaction().begin();
}

private void deactivateRide(Ride ride) {
    ride.setActive(false);
    db.merge(ride);
}

private void handleTransactionFailure(Exception e) {
    if (db.getTransaction().isActive()) {
        db.getTransaction().rollback();
    }
    e.printStackTrace();
}
```

Con esta refactorización he hecho que cada método contenga menos de 15 líneas de código y he dividido las responsabilidades en diferentes métodos para que cada uno se encargue de una cosa diferente.

Refactorizado por Leire Sesma

GetUser(y demás)

Código Inicial:



Código refactorizado:

```
public void setParam(TypedQuery<?> query, String erab) {
    query.setParameter("username", erab);
}
```

He creado este método para que cada método que lo necesite, llame a este método para que haga la operación. Así en lugar de en cada método poner un query.setParameter, se le llama a este método y lo hace una sola vez. He sustituido el código por esta llamada en cada método que lo necesitaba. Con esto hemos solucionado la duplicación.

Refactorizado por Leire Sesma

UpdateAlertaAurkituak

Código Inicial:

```
public boolean updateAlertaAurkituak(String username) {
    try {
        db.getTransaction().begin();

        boolean alertFound = false;
        TypedQuery<Alert> alertQuery = db.createQuery("SELECT a FROM Alert a WHERE a.traveler.username = :username",
                Alert.class);
        setParam(alertQuery, username);
        List<Alert> alerts = alertQuery.getResultList();

        TypedQuery<Ride> rideQuery = db
                .createQuery("SELECT r FROM Ride r WHERE r.date > CURRENT_DATE AND r.active = true", Ride.class);
        List<Ride> rides = rideQuery.getResultList();

        for (Alert alert : alerts) {
            boolean found = false;
            for (Ride ride : rides) {
                if (UtilDate.datesAreEqualIgnoringTime(ride.getDate(), alert.getDate())
                        && ride.getFrom().equals(alert.getFrom()) && ride.getTo().equals(alert.getTo())
                        && ride.getnPlaces() > 0) {
                    alert.setFound(true);
                    found = true;
                    if (alert.isActive())
                        alertFound = true;
                    break;
                }
            }


             if (!found) {
                 alert.setFound(false);
             }
             db.merge(alert);
         }

         db.getTransaction().commit();
         return alertFound;
    } catch (Exception e) {
        e.printStackTrace();
        db.getTransaction().rollback();
        return false;
    }
}
```

Código refactorizado:

```
public boolean updateAlertaAurkituak(String username) {
    try {
        db.getTransaction().begin();

        List<Alert> alerts = getAlertsByU(username);
        List<Ride> rides = getActiveFutureRides();

        boolean alertFound = processAlerts(alerts, rides);

        db.getTransaction().commit();
        return alertFound;
    } catch (Exception e) {
        e.printStackTrace();
        db.getTransaction().rollback();
        return false;
    }
}

private List<Alert> getAlertsByU(String username) {
    TypedQuery<Alert> alertQuery = db.createQuery("SELECT a FROM Alert a WHERE a.traveler.username = :username", Alert.class
    setParam(alertQuery, username);
    return alertQuery.getResultList();
}

private List<Ride> getActiveFutureRides() {
    TypedQuery<Ride> rideQuery = db.createQuery("SELECT r FROM Ride r WHERE r.date > CURRENT_DATE AND r.active = true", Ride
    return rideQuery.getResultList();
}


private boolean processAlerts(List<Alert> alerts, List<Ride> rides) {
    boolean alertFound = false;

    for (Alert alert : alerts) {
        boolean found = updateAlertWithRides(alert, rides);
        db.merge(alert);

        if (found && alert.isActive()) {
            alertFound = true;
        }
    }
    return alertFound;
}

private boolean updateAlertWithRides(Alert alert, List<Ride> rides) {
    for (Ride ride : rides) {
        if (isMatchingRide(alert, ride)) {
            alert.setFound(true);
            return true;
        }
    }
    alert.setFound(false);
    return false;
}

private boolean isMatchingRide(Alert alert, Ride ride) {
    return UtilDate.datesAreEqualIgnoringTime(ride.getDate(), alert.getDate())
        && ride.getFrom().equals(alert.getFrom())
        && ride.getTo().equals(alert.getTo())
        && ride.getnPlaces() > 0;
}
```

El método principal ahora se centra en la lógica de inicio de la transacción y de retorno. Las consultas y la lógica para procesar las alertas se delegan a otros métodos. Ahora cada método cuenta también con menos de 15 líneas.

Refactorizado por Leire Sesma