Laboratorio Refactorización

Autor: Mikel Fajardo

Código inicial:

```
    public boolean gauzatuEragiketa(String username, double amount, boolean deposit) {
        try {
            db.getTransaction().begin();
            User user = getUser(username);
            if (user != null) {
                double currentMoney = user.getMoney();
                if (deposit) {
                    user.setMoney(currentMoney + amount);
                } else {
                    if ((currentMoney - amount) < 0)
                        user.setMoney(0);
                    else
                        user.setMoney(currentMoney - amount);
                }
                db.merge(user);
                db.getTransaction().commit();
                return true;
            }
            db.getTransaction().commit();
            return false;
        } catch (Exception e) {
            e.printStackTrace();
            db.getTransaction().rollback();
            return false;
        }
    }
```

Código final:

```
    public boolean gauzatuEragiketa(String username, double amount, boolean deposit) {
        try {
            db.getTransaction().begin();
            User user = hacerOperacion(username, amount, deposit);
                db.merge(user);
                db.getTransaction().commit();
                return true;
        } catch (Exception e) {
            e.printStackTrace();
            db.getTransaction().rollback();
            return false;
        }
    }
    private User hacerOperacion(String username, double amount, boolean deposit) {
        User user = getUser(username);
            double currentMoney = user.getMoney();
            if (deposit) {
                user.setMoney(currentMoney + amount);
            } else {
                if ((currentMoney - amount) < 0)
                    user.setMoney(0);
                else
                    user.setMoney(currentMoney - amount);
            }
        return user;
    }
```

Podemos apreciar que en el codigo inicial del metodo “gauzatuEragiketa” hay dos “bad smell”, el primero es que hay mas de 15 lineas de codigo ("Write short units of code" (capítulo 2)) y el segundo es que la complejidad ciclomatica del metodo es mayor que 4 ("Write simple units of code" (capítulo 3)). Para resolver ambos “bad smell” lo he hecho es primero eliminar el primer “if” ya que en caso de que no se cumpliese la condicion de este “if” saltaria una excepcion y habria parte del codigo que no se ejecutaria nunca. Despues de hacer esto, ya tendriamos resuelto uno de los “bad smell” ya que la complejidad cilclomatica baharia a 4. Para reducir las lineas del metodo, he refactorizado la parte del codigo que se encargaba de hacer la operación extrayendolo en un metodo “hacerOperacion” que se llama desde el metodo “gauzatuEragiketa”.

Código inicial:

```
public boolean erreklamazioaBidali(String nor, String nori, Date gaur, Booking booking, String textua,
        boolean aurk) {
    try {
        db.getTransaction().begin();

        Complaint erreklamazioa = new Complaint(nor, nori, gaur, booking, textua, aurk);
        db.persist(erreklamazioa);
        db.getTransaction().commit();
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        db.getTransaction().rollback();
        return false;
    }
}
```

Código final:

```
public boolean erreklamazioaBidali(ErreklamazioaBidaliParameter parameterObject) { //Editado por Mikel
    try {
        db.getTransaction().begin();

        Complaint erreklamazioa = new Complaint(parameterObject.getNor(), parameterObject.getNori(), parameterObject.getGaur(),
                parameterObject.getBooking(), parameterObject.getTextua(), parameterObject.isAurk());
        db.persist(erreklamazioa);
        db.getTransaction().commit();
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        db.getTransaction().rollback();
        return false;
    }
}
```

En este método he refactorizado el “bad smell” “Keep unit interfaces small" ya que en este método había mas de cuatro parámetros. Lo que he hecho es refactorizar los parámetros del método creando un nuevo objeto “ErreklamazioaBidaliParameter” donde se guardan todos los parámetros que este método necesita, así este método solo necesitará un parámetro de entrada que será el objeto “ErreklamazioaBidaliParameter”.

Código inicial:

```
this.jCalendar.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent propertychangeevent) {
        if (propertychangeevent.getPropertyName().equals("locale")) {
            jCalendar.setLocale((Locale) propertychangeevent.getNewValue());
        } else if (propertychangeevent.getPropertyName().equals("calendar")) {
            calendarAnt = (Calendar) propertychangeevent.getOldValue();
            calendarAct = (Calendar) propertychangeevent.getNewValue();
```

Código final:

```
this.jCalendar.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent propertychangeevent) {
        Object newValue = propertychangeevent.getNewValue();
        if (propertychangeevent.getPropertyName().equals("locale")) {
            jCalendar.setLocale((Locale) newValue);
        } else if (propertychangeevent.getPropertyName().equals("calendar")) {
            calendarAnt = (Calendar) propertychangeevent.getOldValue();
            calendarAct = (Calendar) newValue;
```

En el código inicial podemos ver que hay duplicidad de código ya que se repite "propertychangeevent.getNewValue()" dos veces. Para evitar esto, lo he refactorizado creando una variable local para esta expresión.

Autor: Euken Sáez

Código inicial:

```
public boolean bookRide(String username, Ride ride, int seats, double desk) { //Euken
    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 final:

```java
public boolean bookRide(String username, Ride ride, int seats, double desk) { //Euken
    try {
        return validBook(username, ride, seats, desk);
    } catch (Exception e) {
        e.printStackTrace();
        db.getTransaction().rollback();
        return false;
    }
}

private boolean validBook(String username, Ride ride, int seats, double desk) {
    db.getTransaction().begin();
    Traveler traveler = getTraveler(username);
    if (traveler == null || ride.getnPlaces() < seats) {
        return false;
    }

    double ridePriceDesk = (ride.getPrice() - desk) * seats;
    double availableBalance = traveler.getMoney();
    if (availableBalance < ridePriceDesk) {
        return false;
    }

    makeBook(ride, seats, desk, traveler, ridePriceDesk, availableBalance);
    return true;
}

private void makeBook(Ride ride, int seats, double desk, Traveler traveler, double ridePriceDesk,
        double availableBalance) {
    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();
}
```

Comentario: Como hemos estudiado este método no es para nada adecuado para mantener un buen código que sea mantenible en el tiempo. Nos centramos en primer lugar en el tamaño de este, donde encontramos un método demasiado extenso (a partir de las 15 líneas de código podemos considerar que es demasiado largo) para mantenerlo. Por ello se ha decidido dividir el método donde el método makeBook realizara los cambios necesarios en la base de datos para realizar la reserva. En segundo lugar, observamos que hay una complejidad ciclo matica de 4, algo no recomendable. Por ello en este caso se ha decidido separar la comprobación de los datos introducidos por parámetro en un método validBook donde se han unido 2 condicionales en 1 y se ha conseguido un mejor código para mantener y realizar cambios.

Código inicial:

```java
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.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) {
        // TODO Auto-generated catch block
        return null;
    }


}
```

Código final:

```
/**
 * This method creates a ride for a driver
 * @param parameterObject TODO
 * @param driverEmail to which ride is added
 *
 * @return the created ride, or null, or an exception
 * @throws RideMustBeLaterThanTodayException if the ride date is before today
 * @throws RideAlreadyExistException         if the same ride already exists for
 *                                           the driver
 */
public Ride createRide(CreateRideParameter parameterObject)
        throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    System.out.println(
            ">> DataAccess: createRide=> from= " + parameterObject.getFrom() + " to= " + parameterObject.getTo() + " driver=" + parameterObject.getDriverName()
            + " date " + parameterObject.getDate());
    if (parameterObject.getDriverName()==null) return null;
    try {
        if (new Date().compareTo(parameterObject.getDate()) > 0) {
            System.out.println("ppppp");
            throw new RideMustBeLaterThanTodayException(
                    ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }

        db.getTransaction().begin();
        Driver driver = db.find(Driver.class, parameterObject.getDriverName());
        if (driver.doesRideExists(parameterObject.getFrom(), parameterObject.getTo(), parameterObject.getDate())) {
            db.getTransaction().commit();
            throw new RideAlreadyExistException(
                    ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
        }
        Ride ride = driver.addRide(parameterObject.getFrom(), parameterObject.getTo(), parameterObject.getDate(),
                parameterObject.getnPlaces(), parameterObject.getPrice());
        // next instruction can be obviated
        db.persist(driver);
        db.getTransaction().commit();

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

}
```

Comentario: Otra practica incorrecta a la hora de realizar código es la utilización de un numero demasiado alto de parámetros (más de 4 se considera incorrecto por norma general), por ello una forma de evitar este problema es la creación de un objeto que obtiene los atributos necesarios. En este caso se crea la clase CreateRideParameter que crea un objeto que permite crear los viajes.