

DOCUMENTACIÓN REFACTORIZACIÓN RIDES24COMPLETE

**Ingeniería de Software II
2024-25**

Jon Ander Iturrioz

RECURSOS


MALOS OLORES CORREGIDOS:

- [CONCEPTO 1: "Write short units of code" \(capítulo 2\)](#)
- [CONCEPTO 2: "Write simple units of code" \(capítulo 3\)](#)
- [CONCEPTO 3: "Duplicate code" \(capítulo 4\)](#)
- [CONCEPTO 4: "Keep unit interfaces small" \(capítulo 5\)](#)

NOTA PARA EL PROFESOR:

Todas las imágenes tienen un link adjunto para poder abrirlas en otra pestaña. En caso de que falte alguna, he dejado la carpeta en el apartado de RECURSOS de la siguiente página.

RECURSOS

- Repositorio de GitHub:
 - <https://github.com/JonAnderIturrioz/Rides24Complete>
- Proyecto de SonarCloud:
 - <https://sonarcloud.io/project/overview?id=rides24>
- Carpeta con imágenes (en caso de que no se vean bien):
 -  Refactorización imágenes

MALOS OLORES CORREGIDOS:

CONCEPTO 1: "Write short units of code" (capítulo 2)

- Explicación:

Para que la mantenibilidad de código sea alta, se recomienda que cada unidad tenga un máximo de 15 líneas. En este ejemplo, la función *gauzatuEragiketa* tiene una longitud de unas 21 líneas.

Para remediar el problema, se ha creado un nuevo módulo que lleva a cabo una funcionalidad específica de la función (en nuestro caso, el cálculo del dinero del usuario).

- Antes:

```
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;
    }
}
```

- Despues:

```
public boolean gauzatuEragiketa(String username, double amount, boolean deposit) {
    try {
        db.getTransaction().begin();
        User user = getUser(username);
        if (user != null) {

            gauzatuEragiketaDiruaKalkulatu(user, amount, deposit);

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

public void gauzatuEragiketaDiruaKalkulatu(User user, double amount ,boolean deposit) {
    double currentMoney = user.getMoney();
    if (deposit) {
        user.setMoney(currentMoney + amount);
    } else {
        if ((currentMoney - amount) < 0)
            user.setMoney(0);
        else
            user.setMoney(currentMoney - amount);
    }
}
```

CONCEPTO 2: "Write simple units of code" (capítulo 3)

- Explicación:

La idea de este concepto es reducir el número de caminos posibles que tiene una unidad, es decir, su complejidad ciclomática. Cada unidad no debería tener una complejidad mayor a 4.

La función *updateAlertaAurkituak* de la clase *DataAccess* tiene una complejidad ciclomática de 7, por lo que tenemos que dividirlo. Vamos a sacar el bloque entero del bucle exterior a otra función, y de ahí el bucle interior a otra.

- Antes:

```
DataAccess.java x
969
970 public boolean updateAlertaAurkituak(String username) {
971     try {
972         db.getTransaction().begin();
973
974         boolean alertFound = false;
975         TypedQuery<Alert> alertQuery = db.createQuery("SELECT a FROM Alert a WHERE a.traveler.username = :username",
976             Alert.class);
977         alertQuery.setParameter("username", username);
978         List<Alert> alerts = alertQuery.getResultList();
979
980         TypedQuery<Ride> rideQuery = db
981             .createQuery("SELECT r FROM Ride r WHERE r.date > CURRENT_DATE AND r.active = true", Ride.class);
982         List<Ride> rides = rideQuery.getResultList();
983
984         for (Alert alert : alerts) {
985             boolean found = false;
986             for (Ride ride : rides) {
987                 if (UtilDate.datesAreEqualIgnoringTime(ride.getDate(), alert.getDate())
988                     && ride.getFrom().equals(alert.getFrom()) && ride.getTo().equals(alert.getTo())
989                     && ride.getnPlaces() > 0) {
990                     alert.setFound(true);
991                     found = true;
992                     if (alert.isActive())
993                         alertFound = true;
994                     break;
995                 }
996             }
997             if (!found) {
998                 alert.setFound(false);
999             }
1000             db.merge(alert);
1001         }
1002
1003         db.getTransaction().commit();
1004         return alertFound;
1005     } catch (Exception e) {
1006         e.printStackTrace();
1007         db.getTransaction().rollback();
1008         return false;
1009     }
1010 }
```

- Despues:

```
DataAccess.java ×
966
969 public boolean updateAlertaAurkituak(String username) {
970     try {
971         db.getTransaction().begin();
972
973         boolean alertFound = false;
974         TypedQuery<Alert> alertQuery = db.createQuery("SELECT a FROM Alert a WHERE a.traveler.username = :username",
975             Alert.class);
976         alertQuery.setParameter("username", username);
977         List<Alert> alerts = alertQuery.getResultList();
978
979         TypedQuery<Ride> rideQuery = db
980             .createQuery("SELECT r FROM Ride r WHERE r.date > CURRENT_DATE AND r.active = true", Ride.class);
981         List<Ride> rides = rideQuery.getResultList();
982
983         alertFound = alertaZerrendaZeharkatu(alerts, rides);
984
985         db.getTransaction().commit();
986         return alertFound;
987     } catch (Exception e) {
988         e.printStackTrace();
989         db.getTransaction().rollback();
990         return false;
991     }
992 }
993
994 public boolean alertaZerrendaZeharkatu(List<Alert> alerts, List<Ride> rides) {
995     boolean alertFound = false;
996
997     for (Alert alert : alerts) {
998
999         boolean found = alertaEguneratu(alert, rides);
1000
1001         if (alert.isActive()) {
1002             alertFound = true;
1003         }
1004         if (!found) {
1005             alert.setFound(false);
1006         }
1007         db.merge(alert);
1008     }
1009
1010     return alertFound;
1011 }
1012
1013
1014 public boolean alertaEguneratu(Alert alert, List<Ride> rides) {
1015
1016     for (Ride ride : rides) {
1017         if (UtilDate.datesAreEqualIgnoringTime(ride.getDate(), alert.getDate())
1018             && ride.getFrom().equals(alert.getFrom()) && ride.getTo().equals(alert.getTo())
1019             && ride.getnPlaces() > 0) {
1020             alert.setFound(true);
1021
1022             return true;
1023         }
1024     }
1025     return false;
1026 }
```

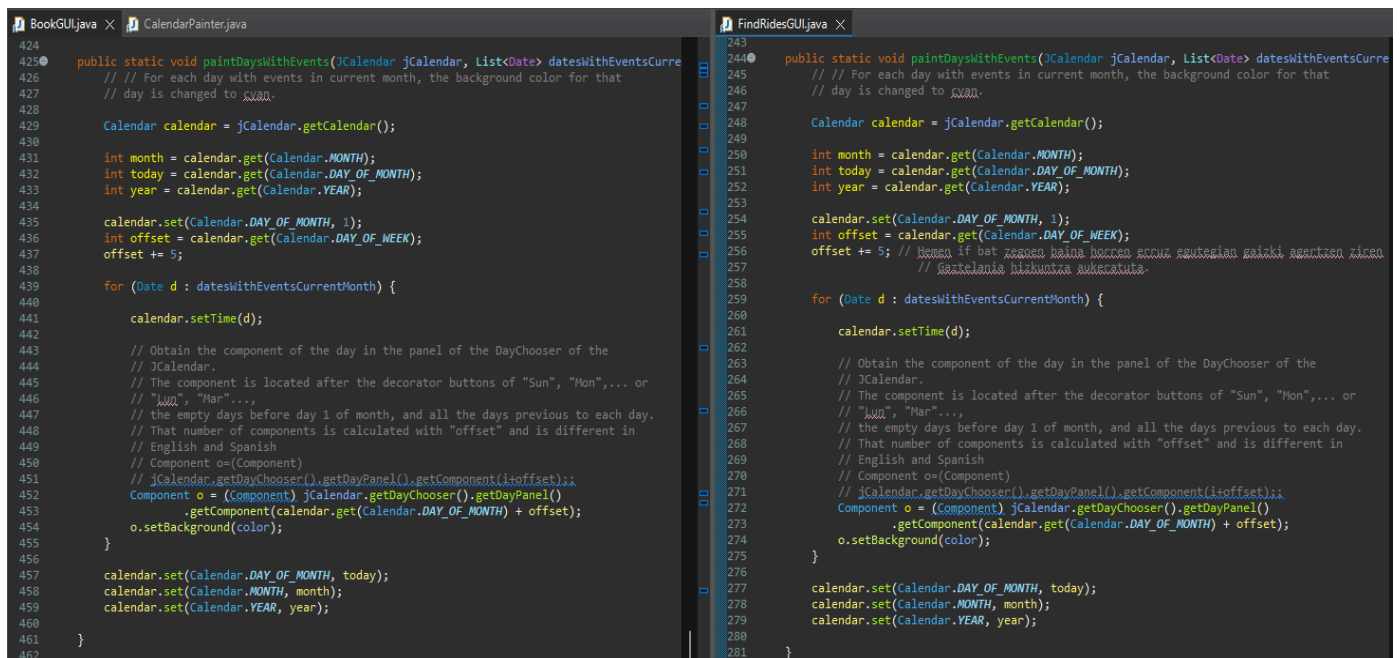
CONCEPTO 3: “Duplicate code” (capítulo 4)

- Explicación:

Algunas funcionalidades pueden ser necesarias en varias unidades y clases diferentes, lo que a veces resulta en código duplicado. En este caso, el método *paintDaysWithEvents* está repetido, letra por letra, en *FindRidesGUI* y *BookGUI*.

Para eliminar código redundante, podemos extraer el método a una nueva clase al que vamos a llamar *CalendarPainter*. Esta nueva clase tendrá una única responsabilidad: colorear los días indicados en el objeto *jCalendar* recibido mediante el método *paintDaysWithEvents* ya mencionado.

- Antes:



The screenshot shows two Java source files side-by-side in an IDE. The left file is `BookGUI.java` and the right file is `FindRidesGUI.java`. Both files contain an identical method named `paintDaysWithEvents`. The method signature is `public static void paintDaysWithEvents(JCalendar jCalendar, List<Date> datesWithEventsCurrentMonth, Color color)`. The code inside the method is identical in both files, including comments in Spanish and English, and the logic for setting the calendar and painting the days. The lines are numbered from 424 to 462 in `BookGUI.java` and 243 to 281 in `FindRidesGUI.java`.

```
jCalendar1.getDate();  
paintDaysWithEvents(jCalendar1, datesWithRidesCurrentMonth, Color.CYAN);
```


- Despues:

```

CalendarPainter.java X
1 package gui;
2
3 import java.awt.Color;
4 import java.awt.Component;
5 import java.util.Calendar;
6 import java.util.Date;
7 import java.util.List;
8
9 import com.toedter.calendar.JCalendar;
10
11 public class CalendarPainter {
12
13     public void paintDaysWithEvents(JCalendar jCalendar, List<Date> datesWithEventsCurrentMonth, Color color) {
14
15         Calendar calendar = jCalendar.getCalendar();
16
17         int month = calendar.get(Calendar.MONTH);
18         int today = calendar.get(Calendar.DAY_OF_MONTH);
19         int year = calendar.get(Calendar.YEAR);
20
21         calendar.set(Calendar.DAY_OF_MONTH, 1);
22         int offset = calendar.get(Calendar.DAY_OF_WEEK);
23         offset += 5;
24
25         for (Date d : datesWithEventsCurrentMonth) {
26
27             calendar.setTime(d);
28
29             Component o = (Component) jCalendar.getDayChooser().getDayPanel()
30                 .getComponent(calendar.get(Calendar.DAY_OF_MONTH) + offset);
31             o.setBackground(color);
32         }
33
34         calendar.set(Calendar.DAY_OF_MONTH, today);
35         calendar.set(Calendar.MONTH, month);
36         calendar.set(Calendar.YEAR, year);
37
38     }
39 }
40

```

```

BookGUI.java X
83 private static CalendarPainter painter = new CalendarPainter();
84
85 public BookGUI(String username) {
86     BookGUI.setBusinessLogic(TravelerGUI.getBusinessLogic());
87
FindRidesGUI.java X
57
58 private static CalendarPainter painter = new CalendarPainter();
59
60 public FindRidesGUI() {
61     this.getContentPane().setLayout(null);
62

```

```

jCalendar1.getDate());
painter.paintDaysWithEvents(jCalendar1, datesWithRidesCurrentMonth, Color.CYAN);

```

CONCEPTO 4: "Keep unit interfaces small" (capítulo 5)

- Explicación:

Si una unidad recibe muchos parámetros, es difícil ver lo que va a hacer cada uno a simple vista. Debido a ello, cada método no debería recibir más de 4 parámetros.

En nuestro ejemplo, *createRide* nos pide 6 parámetros, todos necesarios para crear un objeto Ride. En vez de pasar todos estos valores a la lógica de negocios que, a su vez, va a dárselos a la capa de acceso de datos, donde se crea el objeto Ride antes de introducirlo a nuestro BD, podemos hacer que simplemente reciba un objeto Ride ya creado.

Si hace falta, la clase Ride posee los getters necesarios para extraer todos los valores que requería antes de la refactorización.

- Antes:

```
223 * @throws RideMustBeLaterThanTodayException
224 * @throws RideAlreadyExistsException
225 *
226 */
227 public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverName)
228     throws RideAlreadyExistsException, RideMustBeLaterThanTodayException {
229     System.out.println(
230         "DataAccess: createRide() from " + from + " to " + to + " drivers" + driverName + " date " + date);
231     if (driverName == null) return null;
232     try {
233         if (new Date().compareTo(date) > 0) {
234             System.out.println("ppppp");
235             throw new RideMustBeLaterThanTodayException(
236                 ResourceBundle.getBundle("tiquetas").getString("CreateRideUI.ErrorRideMustBeLaterThanToday"));
237         }
238         db.getTransaction().begin();
239         Driver driver = db.find(Driver.class, driverName);
240         if (driver == null || driver.isDead() || (from, to, date) != null) {
241             db.getTransaction().commit();
242             throw new RideAlreadyExistsException(
243                 ResourceBundle.getBundle("tiquetas").getString("DataAccess.RideAlreadyExist"));
244         }
245         Ride ride = driver.addRide(from, to, date, nPlaces, price);
246         // next instruction can be omitted
247         db.persist(driver);
248         db.getTransaction().commit();
249         return ride;
250     } catch (NullPointerException e) {
251         // 10000 auto-generated catch block
252         return null;
253     }
254 }
255 /**
256  * This method retrieves the rides from two locations on a given date
257  *
258  * @param from the origin location of a ride
259  * @param to the destination location of a ride
260  * @param date the date of the ride
261  * @return collection of rides
262  */
263 public List<Ride> getRides(String from, String to, Date date) {
264     System.out.println("DataAccess: getActiveRides() from " + from + " to " + to + " date " + date);
265     List<Ride> res = new ArrayList<>();
266     TypedQuery<Ride> query = db.createQuery(
267         "SELECT r FROM Ride r WHERE r.from = ?1 AND r.to = ?2 AND r.date = ?3 AND r.active = true", Ride.class);
268     query.setParameter(1, from);
269     query.setParameter(2, to);
270     query.setParameter(3, date);
271     List<Ride> rides = query.getResultList();
272     for (Ride ride : rides) {
273         System.out.println(ride);
274     }
275     TypedQuery<Ride> query = db.createQuery(
276         "SELECT r FROM Ride r WHERE r.from = ?1 AND r.to = ?2 AND r.date = ?3 AND r.active = true", Ride.class);
277     query.setParameter(1, from);
278     query.setParameter(2, to);
279     query.setParameter(3, date);
280     List<Ride> rides = query.getResultList();
281     for (Ride ride : rides) {
282         System.out.println(ride);
283     }
284 }
```

```
83 * @inheritDoc
84 *
85 */
86 @Override
87 public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverName)
88     throws RideMustBeLaterThanTodayException, RideAlreadyExistsException {
89     dbManager.open();
90     Ride ride = dbManager.createRide(from, to, date, nPlaces, price, driverName);
91     dbManager.close();
92     return ride;
93 }
94 }
```

```
58 *
59 * Return the created ride, or null, or a
60 * @throws RideMustBeLaterThanTodayException if the ride date is before today
61 * @throws RideAlreadyExistsException if the same ride already exists for
62 * the driver
63 */
64 @Override
65 public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverName)
66     throws RideMustBeLaterThanTodayException, RideAlreadyExistsException;
67
68 /**
69  * This method retrieves the rides from two locations on a given date
70  */
71 }
```

```
211
212
213 private void jButtonCreateActionPerformed(ActionEvent e) {
214     jLabelMsg.setText("");
215     String error = field_errors();
216     if (error != null) {
217         jLabelMsg.setText(error);
218     } else {
219         try {
220             BIFacade facade = new BIFacade();
221             InputTexts = (ArrayList<String>) comboBoxDest.getSelectedItems();
222             float price = Float.parseFloat(jTextFieldPrice.getText());
223
224             @SuppressWarnings("unused")
225             Ride r = facade.createRide(fieldOrigin.getText(), fieldDestination.getText(),
226                 InputTexts.get(0).getText(), InputTexts.get(1).getText(), price, driver.getName());
227             jLabelMsg.setText(ResourceBundle.getBundle("tiquetas").getString("CreateRideUI.AideCreated"));
228         } catch (RideMustBeLaterThanTodayException e1) {
229             jLabelMsg.setText(e1.getMessage());
230         } catch (RideAlreadyExistsException e1) {
231             jLabelMsg.setText(e1.getMessage());
232         }
233     }
234 }
```

- Despues:

```

DataAccess.java X
223 * @throws RideMustBeLaterThanTodayException
224 * @throws RideAlreadyExistException
225 *
226 */
227 public Ride createRide(Ride ride)
228     throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
229     String from = ride.getFrom();
230     String to = ride.getTo();
231     Date date = ride.getDate();
232     int nPlaces = ride.getNPlaces();
233     float price = (float) ride.getPrice();
234     String driverName = ride.getDriver().getUsername();
235
236     System.out.println(
237         ">>> DataAccess: createRide> from= " + from + " to= " + to + " driver= " + driverName + " date= " + date);
238     if (driverName==null) return null;
239     try {
240         if (new Date().compareTo(date) > 0) {
241             System.out.println("ppppp");
242             throw new RideMustBeLaterThanTodayException(
243                 ResourceBundle.getBundle("Etiquetas").getString("CreateRideUI.errorRideMustBeLaterThanToday"));
244         }
245         db.getTransaction().begin();
246         driver = db.find(DataSource.class, driverName);
247         if (driver.doesRideExist(from, to, date)) {
248             db.getTransaction().commit();
249             throw new RideAlreadyExistException(
250                 ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
251         }
252         driver.addRide(from, to, date, nPlaces, price);
253         // next instruction can be obviated
254         db.persist(driver);
255         db.getTransaction().commit();
256         return ride;
257     } catch (NullPointerException e) {
258         // 1000 Auto-generated catch block
259         return null;
260     }
261 }
262
263 /**
264  * This method retrieves the rides from two locations on a given date
265  *
266  * @param from the origin location of a ride
267  * @param to the destination location of a ride
268  * @param date the date of the ride
269  * @return collection of rides
270  */
271 public List<Ride> getRides(String from, String to, Date date) {
272     System.out.println(">>> DataAccess: getActiveRides> from= " + from + " to= " + to + " date= " + date);
273     List<Ride> res = new ArrayList<>();
274     TypedQuery<Ride> query = db.createQuery(
275
BifacadeImplementation.java X
83 * @return the created ride, or null, or a
84 */
85 @Override
86 public Ride createRide(Ride ride)
87     throws RideMustBeLaterThanTodayException, RideAlreadyExistException {
88     dbManager.open();
89     dbManager.createRide(ride);
90     dbManager.close();
91     return ride;
92 }
93
94 Bifacade.java X
95 * @return the created ride, or null, or a
96 * @throws RideMustBeLaterThanTodayException if the ride date is before today
97 * @throws RideAlreadyExistException if the same ride already exists for
98 * the driver
99 */
100 @Override
101 public Ride createRide(Ride ride)
102     throws RideMustBeLaterThanTodayException, RideAlreadyExistException;
103
104 /**
105  * This method retrieves the rides from two locations on a given date
106  */
107
108 CreateRideGUI.java X
210 private void jButtonCreateActionPerformed(ActionEvent e) {
211     jLabelMsg.setText("");
212     String error = fieldErrors();
213     if (error != null)
214         jLabelMsg.setText(error);
215     else
216         try {
217             Bifacade facade = new Bifacade();
218             int inputSeats = (int) comboBoxSeats.getSelectedItem();
219             float price = Float.parseFloat(jTextFieldPrice.getText());
220             @SuppressWarnings("unused")
221             Ride r = facade.createRide(
222                 new Ride(fieldOrigin.getText(), fieldDestination.getText(),
223                     UtilDate.tria(Calendar.getDate()), inputSeats, price, driver));
224             jLabelMsg.setText(ResourceBundle.getBundle("Etiquetas").getString("CreateRideUI.RideCreated"));
225         } catch (RideMustBeLaterThanTodayException e1) {
226             jLabelMsg.setText(e1.getMessage());
227         } catch (RideAlreadyExistException e1) {
228             jLabelMsg.setText(e1.getMessage());
229         }
230 }

```