

Proyecto 2.2 - Refactorización

Autores: Mikel León, Mikel Bernal y Anne Baquedano

- Github: <https://github.com/anneBaque/ridesFinal>

- SonarCloud: https://sonarcloud.io/project/overview?id=is2-baquedano_rides

1. “Write short units of code”	2
1.1 Problema uno.....	2
1.2 Problema dos.....	3
1.3 Problema tres.....	10
2. “Write simple units of code”	12
2.1 Problema uno.....	12
2.2 Problema dos.....	12
3. “Duplicate code”	14
3.1 Problema uno.....	14
3.2 Problema dos.....	15
3.3 Problema tres.....	17
4. “Keep unit interfaces small”	19
4.1 Problema uno.....	19
4.2 Problema dos.....	20
4.3 Problema tres.....	22

1. “Write short units of code”

1.1 Problema uno

Código Inicial:

```
public boolean acceptReservation(Reservation reserva) {  
    >>     TypedQuery<Ride> query = db.createQuery("SELECT r FROM Ride r WHERE r.rideNumber=?1", Ride.class); ...  
    >>     query.setParameter(1, reserva.getIdRide());  
    >>     Reservation actu = db.find(Reservation.class, reserva.getIdRes());  
    >>     Ride viaje = query.getSingleResult();  
    >>     db.getTransaction().begin();  
  
    >>     boolean reservado = true;  
    >>     actu.setProcesado(true);  
  
    >>     if (viaje == null || viaje.getPlazasOcupadas() == viaje.getnPlaces()) {  
    >>         System.out.println(reserva.getIdRide() + " no está en la base de datos");  
    >>         reservado = false;  
    >>     } else {  
    >>         viaje.setPlazasOcupadas(1+viaje.getPlazasOcupadas());  
    >>         actu.setEstado(true);  
    >>         System.out.println(viaje + " ha sido actualizado");  
    >>     }  
    >>     db.getTransaction().commit();  
    >>     return reservado;  
}
```

Código refactorizado:

```
public boolean acceptReservation(Reservation reserva) {  
    >>     TypedQuery<Ride> query = db.createQuery("SELECT r FROM Ride r WHERE r.rideNumber=?1", Ride.class); ...  
    >>     query.setParameter(1, reserva.getIdRide());  
    >>     Reservation actu = db.find(Reservation.class, reserva.getIdRes());  
    >>     Ride viaje = query.getSingleResult();  
    >>     db.getTransaction().begin();  
  
    >>     boolean reservado = true;  
    >>     actu.setProcesado(true);  
  
    >>     if (viaje == null || viaje.getPlazasOcupadas() == viaje.getnPlaces()) {  
    >>         reservado = false;  
    >>     } else {  
    >>         viaje.setPlazasOcupadas(1+viaje.getPlazasOcupadas());  
    >>         actu.setEstado(true);  
    >>     }  
    >>     db.getTransaction().commit();  
    >>     return reservado;  
}
```

El “Bad Smell” detectado es que el código es más largo que 15 líneas, exactamente 16. Eso hace que el código sea más difícil de leer y entender. Si nos fijamos en el código hay dos escrituras en consola que nos avisan si está en la base de datos el viaje o si la información ha sido actualizada. Estos han sido puestas a la hora de hacer el método para hacer pruebas, pero a niveles prácticos no sirven para nada. Y con el nuevo sistema de pruebas aprendido sobran totalmente. Por ello quitándolos conseguimos un método de 14 líneas que funciona correctamente.

Autora: Anne Baquedano

1.2 Problema dos

Código inicial:

en Rides24/src/main/java/gui/ApplicationLauncher.java método main

```
public static void main(String[] args) {
```

```
    ConfigXML c=ConfigXML.getInstance();
```

```
    System.out.println(c.getLocale());
```

```
    Locale.setDefault(new Locale(c.getLocale()));
```

```
    System.out.println("Locale: "+Locale.getDefault());
```

```
    Driver driver=new Driver("driver3@gmail.com","Test Driver");
```

```
    MainGUI a=new MainGUI(driver);
```

```
    a.setVisible(true);
```

```
    if(c.isDatabaseInitialized()) {
```

```
        Path carpeta = Paths.get("imagenes");
```

```
        //Eliminamos las imagenes de la base de datos antigua
```

```
        if (Files.exists(carpeta) && Files.isDirectory(carpeta)) {
```

```
            try (DirectoryStream<Path> archivos = Files.newDirectoryStream(carpeta)) { //iterar  
                sobre el directorio
```

```
                for (Path archivo : archivos) {
```

```
                    try{
```

```
                        Files.delete(archivo);
```

```
        } catch (IOException e) {
            System.err.println("Error al borrar el archivo: " + archivo + " -> " + e.getMessage());
        }
    }

} catch (IOException e) {
    System.err.println("Error al leer " + carpeta + ": " + e.getMessage());
}

try {
    Files.delete(carpeta); // Eliminar la carpeta después de vaciarla
} catch (IOException e) {
    System.err.println("Error al borrar la carpeta: " + carpeta + " -> " +
e.getMessage());
}

}

}

try {

BLFacade appFacadeInterface;
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

if (c.isBusinessLogicLocal()) {

    DataAccess da= new DataAccess();
    appFacadeInterface=new BLFacadeImplementation(da);

}

}
```

```
else { //If remote

String serviceName= "http://"+c.getBusinessLogicNode() +":"+
c.getBusinessLogicPort()+"/ws/"+c.getBusinessLogicName()+"?wsdl";

URL url = new URL(serviceName);

//1st argument refers to wsdl document above
//2nd argument is service name, refer to wsdl document above

QName qname = new QName("http://businessLogic/",
"BLFacadeImplementationService");

Service service = Service.create(url, qname);

appFacadeInterface = service.getPort(BLFacade.class);

}

MainGUI.setBussinessLogic(appFacadeInterface);

}catch (Exception e){

a.jLabelSelectOption.setText("Error: "+e.toString());

a.jLabelSelectOption.setForeground(Color.RED);
```

```
System.out.println("Error in ApplicationLauncher: "+e.toString());  
}  
//a.pack();  
  
}
```

Código refactorizado:

```
public class ApplicationLauncher {  
  
    public static void main(String[] args) {  
  
        ConfigXML c=ConfigXML.getInstance();  
  
        setUpLocale(c);  
  
        if(c.isDatabaseInitialized()) {  
            cleanOldDbImages(c);  
        }  
  
        Driver driver=new Driver("driver3@gmail.com","Test Driver");  
        MainGUI a=new MainGUI(driver);  
        a.setVisible(true);  
  
        initializeBusinessLogic(c, a);  
    }  
}
```

```
}
```

```
public static void setUpLocale(ConfigXML config) {  
    System.out.println(config.getLocale());  
    Locale.setDefault(new Locale(config.getLocale()));  
    System.out.println("Locale: "+Locale.getDefault());  
}
```

```
public static void cleanOldDbImages(ConfigXML config) {  
    Path carpeta = Paths.get("imagenes");  
    //Eliminamos las imagenes de la base de datos antigua  
    if (Files.exists(carpeta) && Files.isDirectory(carpeta)) {  
        try (DirectoryStream<Path> archivos = Files.newDirectoryStream(carpeta)) {  
            //iterar sobre el directorio  
            for (Path archivo : archivos) {  
                try{  
                    Files.delete(archivo);  
                } catch (IOException e) {  
                    System.err.println("Error al borrar el archivo: " + archivo + " -> " + e.getMessage());  
                }  
            }  
        } catch (IOException e) {  
            System.err.println("Error al leer " + carpeta + ": " + e.getMessage());  
        }  
  
        try {  
            Files.delete(carpeta); // Eliminar la carpeta después de vaciarla  
        } catch (IOException e) {
```

```

        System.err.println("Error al borrar la carpeta: " + carpeta + " -> " +
e.getMessage());

    }

}

}

public static void initializeBusinessLogic(ConfigXML c, MainGUI a) {
try {
BLFacade appFacadeInterface;
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

if (c.isBusinessLogicLocal()) {
DataAccess da= new DataAccess();
appFacadeInterface=new BLFacadeImplementation(da);
}

else { //If remote

String serviceName= "http://"+c.getBusinessLogicNode() +":"+
c.getBusinessLogicPort()+"/ws/"+c.getBusinessLogicName()+"?wsdl";
URL url = new URL(serviceName);

//1st argument refers to wsdl document above
//2nd argument is service name, refer to wsdl document above

 QName qname = new QName("http://businessLogic/",
"BLFacadeImplementationService");
Service service = Service.create(url, qname);
appFacadeInterface = service.getPort(BLFacade.class);
}
}

```

```
MainGUI.setBussinessLogic(appFacadeInterface);

}catch (Exception e){

a.jLabelSelectOption.setText("Error: "+e.toString());

a.jLabelSelectOption.setForeground(Color.RED);

System.out.println("Error in ApplicationLauncher: "+e.toString());

}

}

}
```

Originalmente el método main tenía demasiadas responsabilidades, lo que hacía que su código tuviese muchas líneas y de distintas tareas. Era difícil de leer y mantener. La refactorización consiste en separar las responsabilidades y sacarlas fuera del método, creando un método exclusivo para cada responsabilidad. Configuración de idioma, limpieza de imagen antigua de BD, inicialización de la lógica de negocio...

Ahora main solo orquesta el flujo.

Autor: Mikel León Ramos

1.3 Problema tres

Código inicial:

```
public static void initializeBusinessLogic(ConfigXML c, MainGUI a) {
    try {
        BLFacade appFacadeInterface;
        UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

        if (c.isBusinessLogicLocal()) {
            DataAccess da= new DataAccess();
            appFacadeInterface=new BLFacadeImplementation(da);
        }

        else { //If remote
            String serviceName= "http://"+c.getBusinessLogicNode() +":"+ c.getBusinessLogicPort()+"ws/"+c.getBusinessLogicName()+"?wsdl";
            URL url = new URL(serviceName);

            //1st argument refers to wsdl document above
            //2nd argument is service name, refer to wsdl document above
            QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");
            Service service = Service.create(url, qname);
            appFacadeInterface = service.getPort(BLFacade.class);
        }

        MainGUI.setBussinessLogic(appFacadeInterface);

    }catch (Exception e) {
        a.jLabelSelectOption.setText("Error: "+e.toString());
        a.jLabelSelectOption.setForeground(Color.RED);

        System.out.println("Error in ApplicationLauncher: "+e.toString());
    }
}
```

Código refactorizado:

```

public static void initializeBusinessLogic(ConfigXML c, MainGUI a) {
    try {
        setUILookAndFeel();
        BLFacade appFacadeInterface = c.isBusinessLogicLocal()
            ? createLocalBusinessLogic()
            : createRemoteBusinessLogic(c);
        MainGUI.setBusinessLogic(appFacadeInterface);
    } catch (Exception e) {
        handleInitializationError(a, e);
    }
}

private static void setUILookAndFeel() throws Exception {
    UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
}

private static BLFacade createLocalBusinessLogic() {
    DataAccess da = new DataAccess();
    return new BLFacadeImplementation(da);
}

private static BLFacade createRemoteBusinessLogic(ConfigXML c) throws Exception {
    String serviceName = "http://" + c.getBusinessLogicNode() + ":" +
        c.getBusinessLogicPort() + "/ws/" +
        c.getBusinessLogicName() + "?wsdl";
    URL url = new URL(serviceName);
    QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");
    Service service = Service.create(url, qname);
    return service.getPort(BLFacade.class);
}

private static void handleInitializationError(MainGUI a, Exception e) {
    a.jLabelSelectOption.setText("Error: " + e.toString());
    a.jLabelSelectOption.setForeground(Color.RED);
    System.err.println("Error in ApplicationLauncher: " + e.toString());
}

```

El método `initializeBusinessLogic(ConfigXML c, MainGUI a)` vulnera la convención que marca 15 líneas de código como máximo para un método. Ello causa un *bad smell*, que resulta fácilmente solucionable seccionando el método en trozos más pequeños, donde `initializeBusinessLogic(ConfigXML c, MainGUI a)` sigue siendo el método principal, y el resto son auxiliares.

Autor: Mikel Bernal Arnaldes

2. “Write simple units of code”

2.1 Problema uno

```
public DataAccess() {  
    if (c.isDatabaseInitialized()) {  
        String fileName=c.getDbFilename();  
        try {  
            Files.delete(Paths.get(fileName));  
            System.out.println("File deleted");  
        }catch(NoSuchFileException e) {  
            System.out.println("No existe: "+ e.getMessage());  
        }catch(DirectoryNotEmptyException e) {  
            System.out.println("Directorio no vacío: "+ e.getMessage());  
        }catch (IOException e) {  
            System.out.println("Error al borrar: "+ e.getMessage());  
        }  
    }  
    open();  
    if (c.isDatabaseInitialized())initializeDB();  
  
    System.out.println("DataAccess created => isDatabaseLocal: "+c.isDatabaseLocal()+" isDatabaseInitialized: "+c.isDatabaseInitialized());  
  
    close();  
}  
  
...  
.. public DataAccess(){  
    if (c.isDatabaseInitialized()) {  
        borrarBD();  
    }  
    open();  
    if (c.isDatabaseInitialized())initializeDB();  
    System.out.println("DataAccess created => isDatabaseLocal: "+c.isDatabaseLocal()+" isDatabaseInitialized: "+c.isDatabaseInitialized());  
    close();  
}  
...  
.. public void borrarBD(){  
    String fileName=c.getDbFilename();  
    try {  
        Files.delete(Paths.get(fileName));  
        System.out.println("File deleted");  
    }catch(NoSuchFileException e) {  
        System.out.println("No existe: "+ e.getMessage());  
    }catch(DirectoryNotEmptyException e) {  
        System.out.println("Directorio no vacío: "+ e.getMessage());  
    }catch (IOException e) {  
        System.out.println("Error al borrar: "+ e.getMessage());  
    }  
}
```

La constructora de la clase de DataAccess tiene cinco de complejidad ciclomática, lo que hace que sea más difícil de modificar y testear. Por ello, la solución es crear un método que borre la base de datos. Así la constructora puede llamar a ese método, lo que hace que su complejidad pase a dos, mientras que el del método nuevo es de cuatro.

Autora: Anne Baquedano

2.2 Problema dos

Código original:

```

public static void cleanOldDbImages(ConfigXML config) {
    Path carpeta = Paths.get("imagenes");
    //Eliminamos las imagenes de la base de datos antigua
    if (Files.exists(carpeta) && Files.isDirectory(carpeta)) {
        try (DirectoryStream<Path> archivos = Files.newDirectoryStream(carpeta)) { //iterar sobre el directorio
            for (Path archivo : archivos) {
                try{
                    Files.delete(archivo);
                } catch (IOException e) {
                    System.err.println("Error al borrar el archivo: " + archivo + " -> " + e.getMessage());
                }
            }
        } catch (IOException e) {
            System.err.println("Error al leer " + carpeta + ": " + e.getMessage());
        }

        try {
            Files.delete(carpeta); // Eliminar la carpeta después de vaciarla
        } catch (IOException e) {
            System.err.println("Error al borrar la carpeta: " + carpeta + " -> " + e.getMessage());
        }
    }
}

```

Código refactorizado:

```

public static void cleanOldDbImages(ConfigXML config) {
    Path carpeta = Paths.get("imagenes");

    if (!Files.exists(carpeta) || !Files.isDirectory(carpeta)) {
        return;
    }

    deleteFilesInDirectory(carpeta);
    deleteDirectory(carpeta);
}

private static void deleteFilesInDirectory(Path carpeta) {
    try (DirectoryStream<Path> archivos = Files.newDirectoryStream(carpeta)) {
        for (Path archivo : archivos) {
            deleteFile(archivo);
        }
    } catch (IOException e) {
        System.err.println("Error al leer " + carpeta + ": " + e.getMessage());
    }
}

private static void deleteFile(Path archivo) {
    try {
        Files.delete(archivo);
    } catch (IOException e) {
        System.err.println("Error al borrar el archivo: " + archivo + " -> " + e.getMessage());
    }
}

private static void deleteDirectory(Path carpeta) {
    try {
        Files.delete(carpeta);
    } catch (IOException e) {
        System.err.println("Error al borrar la carpeta: " + carpeta + " -> " + e.getMessage());
    }
}

```

Este *bad smell* surge porque el método *cleanOldDbImages* tiene una complejidad ciclomática de 5+1, cuando el máximo fijado por convención es 4. Para solucionar este problema, abstraemos los *try-catch* en métodos auxiliares.

Autor: Mikel Bernal Arnaldes

3. “Duplicate code”

3.1 Problema uno

```
public boolean storeCar(Car coche) {  
    boolean añadido=false;  
    db.beginTransaction().begin();  
    TypedQuery<Car> query =db.createQuery("SELECT c FROM Car c WHERE c.emailConductor=?1", Car.class);  
    query.setParameter(1, coche.getEmailConductor());  
    List<Car> existe=query.getResultList();  
    if (existe.isEmpty()) {  
        db.persist(coche);  
        añadido=true;  
    }  
    db.beginTransaction().commit();  
    return añadido;  
}  
  
public Car findCar(String email){  
    TypedQuery<Car> query =db.createQuery("SELECT c FROM Car c WHERE c.emailConductor=?1", Car.class);  
    query.setParameter(1, email);  
    Car coche = query.getSingleResult();  
    return coche;  
}  
  
public boolean storeCar(Car coche) {  
    boolean añadido=false;  
    db.beginTransaction().begin();  
    Car existe=findCar(coche.getEmailConductor());  
    if (existe != null) {  
        db.persist(coche);  
        añadido=true;  
    }  
    db.beginTransaction().commit();  
    return añadido;  
}
```

Aquí podemos ver dos métodos que repiten código lo que empeora la mantenibilidad. La solución encontrada es que el método para guardar coches llame al de buscarlos. Así conseguimos no repetir el código y que el primer método sea más corto, por lo que es más leíble.

Autora: Anne Baquedano

3.2 Problema dos

Código inicial

```
en src/main/java/dataAccess/ObjectdbManagerServer.java método constructor  
ObjectdbManagerServer:  
  
//line 76  
  
try {  
  
    Runtime.getRuntime().exec("java -cp "+objectDbpath+" com.objectdb.Server -port "+  
c.getDatabasePort()+" stop");  
  
} catch (Exception ioe) {  
  
    System.out.println (ioe);  
  
}
```

//line 113

```
try {  
  
    Runtime.getRuntime().exec("java -cp "+objectDbpath+" com.objectdb.Server -port "+  
c.getDatabasePort()+" start");  
  
} catch (Exception ioe) {  
  
    System.out.println (ioe);  
  
}  
  
,
```

Código refactorizado:

```
executeCommand("start");  
executeCommand("stop");  
  
public void executeCommand(String action) {  
  
    try {  
  
        Runtime.getRuntime().exec("java -cp "+objectDbpath+" com.objectdb.Server -port "+  
c.getDatabasePort()+" " + action);  
  
    } catch (Exception ioe) {  
  
        System.out.println (ioe);  
  
    }  
  
}
```

```
 } catch (Exception ioe) {  
     System.out.println (ioe);  
 }  
 }
```

El bad smell detectado es el código duplicado. En el método ObjectdbManagerServer() en dos ocasiones se repiten varias líneas de código para ejecutar el mismo comando que únicamente cambia en la acción (start/stop). En la refactorización se ha creado un método adicional executeCommand al que se le pasa por parámetro la acción en forma de string. En el método original se ha sustituido las líneas de código repetido por la llamada al nuevo método.

Autor: Mikel León Ramos

3.3 Problema tres

Código original:

```
public boolean eraseReservation(String email, Ride ride) {
    boolean res=false;
    db.getTransaction().begin();
    TypedQuery<Reservation> query = db.createQuery("SELECT r FROM Reservation r WHERE r.pasEmail=?1 AND r.idRide=?2 ",Reservation.class);
    query.setParameter(1, email);
    query.setParameter(2, ride.getRideNumber());
    Reservation auxiliar = query.getSingleResult();
    if(auxiliar!=null) {
        db.remove(auxiliar);
    }
    Ride sarakatumba = db.find(Ride.class, ride.getRideNumber());
    sarakatumba.setPlazasOcupadas(sarakatumba.getPlazasOcupadas()-1);
    res=true;
}
db.getTransaction().commit();
return res;
}

public boolean existsReservation(Reservation res) {
    boolean existe = true;
    TypedQuery<Reservation> query = db.createQuery("SELECT r FROM Reservation r WHERE r.pasEmail=?1 AND r.idRide=?2 ",Reservation.class);
    query.setParameter(1, res.getPasEmail());
    query.setParameter(2, res.getIdRide());
    List<Reservation> lista = query.getResultList();
    if(lista.isEmpty()) {
        existe = false;
    }
    return existe;
}
```

Refactorización:

Definimos una variable privada queryReservaEmailRide:

```
public class DataAccess {
    private EntityManager db;
    private EntityManagerFactory emf;
    private String queryreservaEmail = "SELECT r FROM Reservation r WHERE r.pasEmail=?1";
    private String queryReservaEmailRide = "SELECT r FROM Reservation r WHERE r.pasEmail=?1 AND r.idRide=?2";
```

Y sustituimos en los queries correspondientes:

```
public boolean eraseReservation(String email, Ride ride) {
    boolean res=false;
    db.getTransaction().begin();
    TypedQuery<Reservation> query = db.createQuery(queryReservaEmailRide, Reservation.class);
    query.setParameter(1, email);
    query.setParameter(2, ride.getRideNumber());
    Reservation auxiliar = query.getSingleResult();
    if(auxiliar!=null) {
        db.remove(auxiliar);

        Ride sarakatumba = db.find(Ride.class, ride.getRideNumber());
        sarakatumba.setPlazasOcupadas(sarakatumba.getPlazasOcupadas()-1);
        res=true;
    }
    db.getTransaction().commit();
    return res;
}
```

```
public boolean existsReservation(Reservation res) {  
    boolean existe = true;  
    TypedQuery<Reservation> query = db.createQuery(queryReservaEmailRide, Reservation.class);  
    query.setParameter(1, res.getPasEmail());  
    query.setParameter(2, res.getIdRide());  
    List<Reservation> lista = query.getResultList();  
    if(lista.isEmpty()) {  
        existe = false;  
    }  
    return existe;  
}
```

En este ejemplo, observamos un caso de duplicidad de código en queries. Realizamos el mismo query en dos métodos diferentes, por lo que podemos abstraerlo y definir una variable privada de tipo String para evitar repeticiones.

Autor: Mikel Bernal Arnaldes

4. “Keep unit interfaces small”

4.1 Problema uno

```
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail) 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.beginTransaction().begin();

        Driver driver = db.find(Driver.class, driverEmail);
        if (driver.doesRideExists(from, to, date)) {
            db.beginTransaction().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.beginTransaction().commit();

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

public Ride createRide(Ride r, String driverEmail) throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    System.out.println(">> DataAccess: createRide=> from= "+r.getFrom()+" to= "+r.getTo()+" driver= "+driverEmail+" date "+r.getDate());
    try {
        if(new Date().compareTo(r.getDate())>0) {
            throw new RideMustBeLaterThanTodayException(ResourceBundle.getBundle("Etiquetas").getString("CreateRideGUI.ErrorRideMustBeLaterThanToday"));
        }
        db.beginTransaction().begin();
        Driver driver = db.find(Driver.class, driverEmail);
        if (driver.doesRideExists(r.getFrom(), r.getTo(), r.getDate())) {
            db.beginTransaction().commit();
            throw new RideAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
        }
        Ride ride = driver.addRide(r.getFrom(), r.getTo(), r.getDate(), r.getnPlaces(), r.getPrice());
        //next instruction can be obviated
        db.persist(driver);
        db.beginTransaction().commit();

        return ride;
    } catch (NullPointerException e) {
        // TODO Auto-generated catch block
        db.beginTransaction().commit();
        return null;
    }
}
```

En este caso el “Bad Smell” que hay es que el número de parámetros es mayor que cuatro, seis concretamente. Por eso hemos decidido pasar un objeto tipo Ride, así hemos conseguido que el número de parámetros baje a dos. Esto no solo ha supuesto cambios en la clase Data Access, sino que también ha sido necesario cambiar código en el BLFacadeImplementation para que pase los parámetros correspondientes. También ha habido cambios en las pruebas que se usa esa función de la base de datos. Después de los cambios correspondientes se ha comprado su funcionamiento y es el correcto.

Autora: Anne Baquedano

4.2 Problema dos

Código inicial:

en la clase Rides24/src/main/java/businessLogic/BLFacadeImplementation.java el método BLFacadeImplementation

```
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail ) throws RideMustBeLaterThanTodayException, RideAlreadyExistException{
```

```
    dbManager.open();
    Ride ride = new Ride(from, to, date, nPlaces, price);
    ride=dbManager.createRide(ride, driverEmail);
    dbManager.close();
    return ride;
}
```

como consecuencia del cambio tambien tendremos que modificar la interfaz BLFacade:

```
public Ride createRide( String from, String to, Date date, int nPlaces, float price, String driverEmail) throws RideMustBeLaterThanTodayException, RideAlreadyExistException;
```

y el método jButtonCreateActionPerformed de la clase CreateRideGUI:

```
Ride r=facade.createRide(fieldOrigin.getText(), fieldDestination.getText(),
UtilDate.trim(jCalendar.getDate()), inputSeats, price, driver.getEmail());
```

Código refactorizado:

en el método creatRide de la clase BLFacadeImplementation (donde estaba el bad smell):

```
public Ride createRide(Ride r, String driverEmail ) throws
RideMustBeLaterThanTodayException, RideAlreadyExistException{



dbManager.open();
Ride ride=dbManager.createRide(r, driverEmail);
dbManager.close();
return ride;
}
```

en la interfaz BLFacade:

```
public Ride createRide( Ride r, String driverEmail) throws
RideMustBeLaterThanTodayException, RideAlreadyExistException;
```

en la clase CreateRideGUI:

```
Ride rParam = new Ride(fieldOrigin.getText(), fieldDestination.getText(),
UtilDate.trim(jCalendar.getDate()), inputSeats, price);
Ride r=facade.createRide(rParam, driver.getEmail());
```

Descripción:

El bad smell se basa en el exceso de parámetros de un método, en este caso createRide tenía 6 parámetros, de los cuales 5 se pueden agrupar ya que forman parte de los atributos de una clase ya existente en el proyecto. La refactorización ha consistido en agrupar esos 5 parámetros sobrantes en un objeto Ride, el sexto parámetro no ha sido modificado ya que se usa de forma independiente.

Además, se ha tenido que modificar las llamadas a este método desde las distintas clases del sistema.

Autor: Mikel León Ramos

4.3 Problema tres

Tenemos el siguiente método `addRide`, que contiene 5 parámetros de entrada:

```
public Ride addRide(String from, String to, Date date, int nPlaces, float price) {  
    Ride ride=new Ride(from,to,date,nPlaces,price, this);  
    rides.add(ride);  
    return ride;  
}
```

Refactorizamos, concatenando los Strings `from` y `to` en un único `from-to (route)`:

```
public Ride addRide(String route, Date date, int nPlaces, float price) {  
    // Dividir route en from y to  
    String[] parts = route.split("-");  
    String from = parts[0];  
    String to = parts[1];  
  
    Ride ride=new Ride(from,to,date,nPlaces,price, this);  
    rides.add(ride);  
    return ride;  
}
```

Y por tanto, modificamos las llamadas al método `addRide` para que cumplan con los cambios. Estos son algunos ejemplos:

```
//Create rides  
driver1.addRide(ridesDonostia, "Bilbo", UtilDate.newDate(year,month,15), 4, 7);  
driver1.addRide(ridesDonostia, "Gazteiz", UtilDate.newDate(year,month,6), 4, 8);  
driver1.addRide("Bilbo", ridesDonostia, UtilDate.newDate(year,month,25), 4, 4);  
driver1.addRide(ridesDonostia, "Iruña", UtilDate.newDate(year,month,7), 4, 8);  
  
//Create rides  
driver1.addRide(ridesDonostia + "-Bilbo", UtilDate.newDate(year,month,15), 4, 7);  
driver1.addRide(ridesDonostia + "-Gazteiz", UtilDate.newDate(year,month,6), 4, 8);  
driver1.addRide("Bilbo-" + ridesDonostia, UtilDate.newDate(year,month,25), 4, 4);  
driver1.addRide(ridesDonostia + "-Iruña", UtilDate.newDate(year,month,7), 4, 8);
```

```

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

        Driver driver = db.find(Driver.class, driverEmail);
        if (driver.doesRideExists(r.getFrom(), r.getTo(), r.getDate())) {
            db.getTransaction().commit();
            throw new RideAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
        }
        Ride ride = driver.addRide(r.getFrom(), r.getTo(), r.getDate(),(int) r.getnPlaces(), r.getPrice());
        //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;
    }
}

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

        Driver driver = db.find(Driver.class, driverEmail);
        if (driver.doesRideExists(r.getFrom(), r.getTo(), r.getDate())) {
            db.getTransaction().commit();
            throw new RideAlreadyExistException(ResourceBundle.getBundle("Etiquetas").getString("DataAccess.RideAlreadyExist"));
        }
        Ride ride = driver.addRide(r.getFrom() + "-" + r.getTo(), r.getDate(),(int) r.getnPlaces(), r.getPrice());
        //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;
    }
}

```

Este *bad smell* se produce porque el método *addRides* tiene cinco parámetros de entrada, superando los 4 que se fijan como máximo por convención. La solución pasa por concatenar los dos Strings iniciales en uno único, y dividirlos dentro del método.