

Grado en Ingeniería Informática
Facultad de Informática
UPV/EHU

MANTENIMIENTO SOFTWARE REFACTORIZACIÓN DEL SOFTWARE

eman ta zabal zazu



Universidad del País Vasco Euskal Herriko Unibertsitatea

Mikel Pallin
Iker López

Índice

1. Introducción	3
2. Write short units of code	4
2.1. Bad Smell 1 (Mikel Pallin)	4
2.2. Bad Smell 2 (Iker López)	7
3. Write simple units of code	11
3.1. Bad Smell 3 (Mikel Pallin)	11
3.2. Bad Smell 4 (Iker López)	13
4. Duplicate code	16
4.1. Bad Smell 5 (Mikel Pallin)	16
4.2. Bad Smell 6 (Iker López)	17
5. Keep unit interfaces small	18
5.1. Bad Smell 7 (Mikel Pallin)	18
5.2. Bad Smell 6 (Iker López)	20
6. Test después de Refactorización	21

1. Introducción

Para este proyecto, deberemos detectar y refactorizar posibles '*Bad Smells*' que encontremos en nuestro proyecto Rides24, más concretamente en la clase DataAccess. Para solucionar estos '*Bad Smells*' se utilizará la refactorización de eclipse, tal y como se ha enseñado en el laboratorio 2.2. Cada participante realizará 4 refactorizaciones, una por cada tipo de '*Bad Smell*' descrito en el libro '*Building Maintainable Software*', siendo estos los siguientes:

- **Write short units of code:** Un método no puede contar con más de 15 líneas de código.
- **Write simple units of code:** Un método no debe contener más de 4 puntos de ramificación (if, for, while, etc.).
- **Duplicate code:** La clase DataAccess no debe tener las mismas líneas de código utilizadas más de una vez.
- **Keep unit interfaces small:** Un método no debe tener más de 4 parámetros.

Para cada refactorización hecha, se mostrará: El código inicial del método, el resultado del código una vez refactorizado, una descripción del '*Bad Smell*' detectado, una descripción de la refactorización utilizada para arreglarlo y el nombre del integrante que haga solucionado el '*Bad Smell*'.

2. Write short units of code

2.1. Bad Smell 1 (Mikel Pallin)

El método escogido para la refactorización ha sido *getStopsAndDestinations()*, el cual ha sido trabajado previamente, por lo que se conoce su estructura y los posibles '*Bad Smells*' que pueda llegar a tener. Ahora nos centraremos en que el método no debe de tener más de 15 líneas de código y como podremos ver a continuación, este cuenta con muchas más.

Código inicial:

```
public List<String> getStopsAndDestinations(String from){
    List<String> res = new ArrayList<String>();
    int numStop = 0;

    TypedQuery<Ride> query1 = db.createQuery("SELECT r FROM Ride r WHERE r.stops.name=?1", Ride.class);
    TypedQuery<Ride> query2 = db.createQuery("SELECT r FROM Ride r WHERE r.from=?2", Ride.class);
    query1.setParameter(1, from);
    query2.setParameter(2, from);

    List<Ride> rides1 = query1.getResultList();
    List<Ride> rides2 = query2.getResultList();

    for(Ride r: rides1) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }

        for(Stop s: r.getStops()) {
            if(s.getName().equals(from)) {
                numStop = s.getNumStop();
            }
        }

        for(Stop s: r.getStops()) {
            if(s.getNumStop() > numStop) {
                if(!res.contains(s.getName())) {
                    res.add(s.getName());
                }
            }
        }
    }

    for (Ride r: rides2) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }
        for(Stop s: r.getStops()) {
            if(!res.contains(s.getName())) {
                res.add(s.getName());
            }
        }
    }

    return res;
}
```

Código Refactorizado:

```
public List<String> getStopsAndDestinations(String from){
    List<String> res = new ArrayList<String>();
    int numStop = 0;

    TypedQuery<Ride> query1 = db.createQuery("SELECT r FROM Ride r WHERE r.stops.name=?1", Ride.class);
    TypedQuery<Ride> query2 = db.createQuery("SELECT r FROM Ride r WHERE r.from=?2", Ride.class);
    query1.setParameter(1, from);
    query2.setParameter(2, from);

    List<Ride> rides1 = query1.getResultList();
    List<Ride> rides2 = query2.getResultList();

    getStopsAfterFromAndDestinations(from, res, numStop, rides1);
    getAllStopsAndDestination(res, rides2);

    return res;
}

private void getStopsAfterFromAndDestinations(String from, List<String> res, int numStop, List<Ride> rides1) {
    for(Ride r: rides1) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }

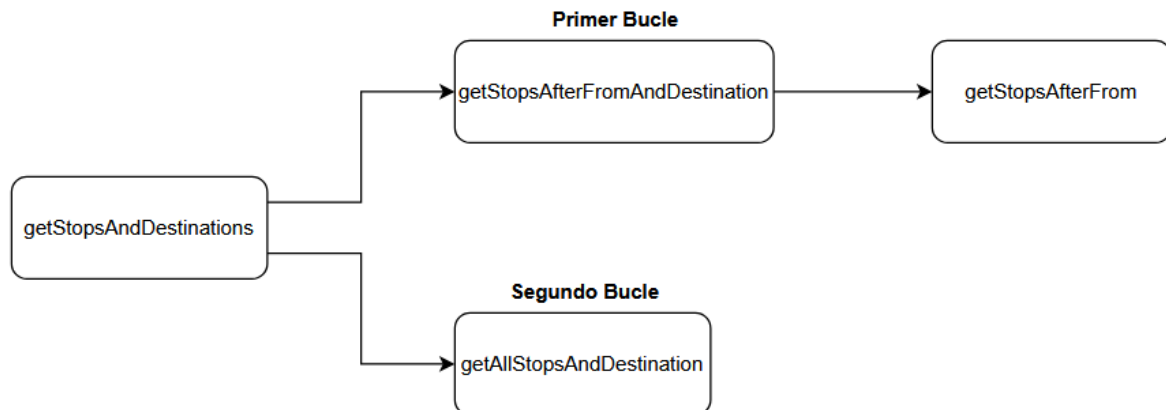
        for(Stop s: r.getStops()) {
            if(s.getName().equals(from)) {
                numStop = s.getNumStop();
            }
        }
        getStopsAfterFrom(res, numStop, r);
    }
}

private void getStopsAfterFrom(List<String> res, int numStop, Ride r) {
    for(Stop s: r.getStops()) {
        if(s.getNumStop() > numStop) {
            if(!res.contains(s.getName())) {
                res.add(s.getName());
            }
        }
    }
}

private void getAllStopsAndDestination(List<String> res, List<Ride> rides2) {
    for (Ride r: rides2) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }
        for(Stop s: r.getStops()) {
            if(!res.contains(s.getName())) {
                res.add(s.getName());
            }
        }
    }
}
```

Explicación Refactorización:

Para solucionar el '*Bad Smell*' previamente descrito, se ha optado por separar el método en cuatro métodos más pequeños. Los dos bucles principales se han refactorizado en dos métodos separados. El primer bucle también contaba con otros dos bucles en su interior, haciéndolo más largo de lo debido. Por lo que se ha optado por separar el segundo bucle en otro nuevo método, quedando la siguiente estructura de llamadas:



Toda la refactorización del método se ha creado utilizando la siguiente función de eclipse: **Refactor > Extract Method**. Así, se ha conseguido que cada método cuente, como el libro '*Building Maintainable Software*' indica, con menos de 15 líneas de código.

2.2. Bad Smell 2 (Iker López)

En el apartado uno ya se trabajó con el método `getThisMonthDatesWithRides()`, por lo que su estructura y posibles 'Bad Smells' son conocidos. Este método presenta varias responsabilidades y más de 15 líneas de código, lo que complica su comprensión y mantenimiento. Por ello, se ha decidido refactorizarlo dividiendo su lógica en métodos auxiliares más pequeños, simplificando los bucles y las consultas, con el objetivo de mejorar la legibilidad, facilitar el testeado y cumplir con la recomendación de mantener unidades de código cortas.

Código inicial:

```
public List<Date> getThisMonthDatesWithRides(String from, String to, Date date) {
    System.out.println(">>> DataAccess: getEventsMonth");
    List<Date> res = new ArrayList<>();

    Date firstDayMonthDate = Util.Date.firstDayMonth(date);
    Date lastDayMonthDate = Util.Date.lastDayMonth(date);

    TypedQuery<Date> query = db.createQuery(
        "SELECT DISTINCT r.date FROM Ride r WHERE r.from=?1 AND r.to=?2 AND r.date BETWEEN ?3 AND ?4",
        Date.class);

    query.setParameter(1, from);
    query.setParameter(2, to);
    query.setParameter(3, firstDayMonthDate);
    query.setParameter(4, lastDayMonthDate);
    List<Date> dates = query.getResultList();
    System.out.println(dates);

    if (dates.isEmpty()) {
        TypedQuery<Date> query2 = db.createQuery(
            "SELECT r.date FROM Ride r WHERE (r.from=?1 OR r.to=?2) AND (r.stops.name=?3 OR r.stops.name=?4) AND r.date BETWEEN ?5 AND ?6",
            Date.class);

        query2.setParameter(1, from);
        query2.setParameter(2, to);
        query2.setParameter(3, from);
        query2.setParameter(4, to);
        query2.setParameter(5, firstDayMonthDate);
        query2.setParameter(6, lastDayMonthDate);
        List<Date> dates2 = query2.getResultList();
        System.out.println(dates2);

        if (dates2.isEmpty()) {
            TypedQuery<Date> query3 = db.createQuery(
                "SELECT DISTINCT r.date FROM Ride r JOIN r.stops s WHERE s.name IN (?2, ?3) AND s.date BETWEEN ?4 AND ?5",
                Date.class);

            query3.setParameter(2, from);
            query3.setParameter(3, to);
            query3.setParameter(4, firstDayMonthDate);
            query3.setParameter(5, lastDayMonthDate);
            List<Date> dates3 = query3.getResultList();
            System.out.println(dates3);

            for (Date d : dates3) {
                res.add(d);
            }
        } else {
            for (Date d : dates2) {
                res.add(d);
            }
        }
    }

    else {
        for (Date d : dates) {
            res.add(d);
        }
    }

    return res;
}
```


Código Refactorizado:

```
public List<Date> getThisMonthDatesWithRides(String from, String to, Date date) {
    System.out.println(">>> DataAccess: getEventsMonth");
    List<Date> res = new ArrayList<>();

    Date firstDayMonthDate= UtilDate.firstDayMonth(date);
    Date lastDayMonthDate= UtilDate.lastDayMonth(date);

    List<Date> dates = getDirectRideDates(from, to, firstDayMonthDate, lastDayMonthDate);

    if(dates.isEmpty()) {
        List<Date> dates2 = getRideDatesWithStops(from, to, firstDayMonthDate, lastDayMonthDate);

        if(dates2.isEmpty()) {
            List<Date> dates3 = getRideDatesFromStopsTable(from, to, firstDayMonthDate, lastDayMonthDate);

            addDates(res, dates3);
        }
        else {
            addDates(res, dates2);
        }
    }
    else {
        addDates(res, dates);
    }
}

return res;
}

private void addDates(List<Date> res, List<Date> dates3) {
    for (Date d: dates3) {
        res.add(d);
    }
}

private List<Date> getRideDatesFromStopsTable(String from, String to, Date firstDayMonthDate,
    Date lastDayMonthDate) {
    TypedQuery<Date> query3 = db.createQuery("SELECT DISTINCT r.date FROM Ride r JOIN r.stops s WHERE s.name IN (?2, ?3) AND s.date BETWEEN ?4 AND ?5", Date.class);

    query3.setParameter(2, from);
    query3.setParameter(3, to);
    query3.setParameter(4, firstDayMonthDate);
    query3.setParameter(5, lastDayMonthDate);
    List<Date> dates3 = query3.getResultList();
    System.out.println(dates3);
    return dates3;
}

private List<Date> getRideDatesWithStops(String from, String to, Date firstDayMonthDate, Date lastDayMonthDate) {
    TypedQuery<Date> query2 = db.createQuery("SELECT r.date FROM Ride r WHERE (r.from=?1 OR r.to=?2) AND (r.stops.name=?3 OR r.stops.name=?6) AND r.date BETWEEN ?4 AND ?5", Date.class);

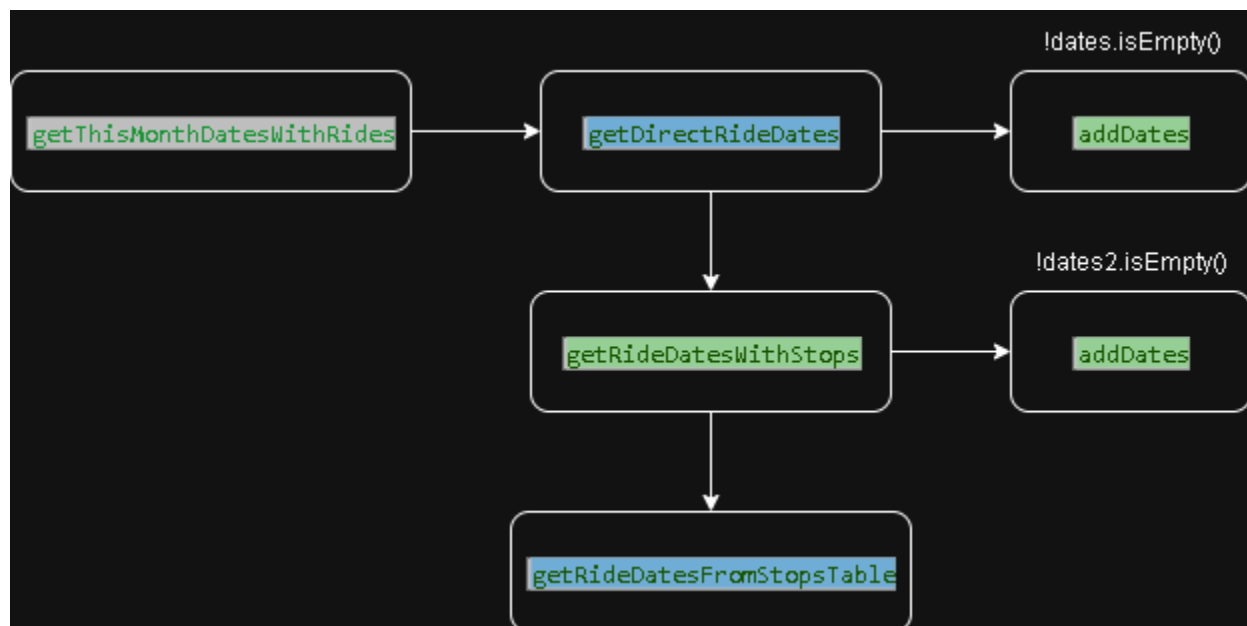
    query2.setParameter(1, from);
    query2.setParameter(2, to);
    query2.setParameter(3, from);
    query2.setParameter(6, to);
    query2.setParameter(4, firstDayMonthDate);
    query2.setParameter(5, lastDayMonthDate);
    List<Date> dates2 = query2.getResultList();
    System.out.println(dates2);
    return dates2;
}

private List<Date> getDirectRideDates(String from, String to, Date firstDayMonthDate, Date lastDayMonthDate) {
    TypedQuery<Date> query = db.createQuery("SELECT DISTINCT r.date FROM Ride r WHERE r.from=?1 AND r.to=?2 AND r.date BETWEEN ?3 AND ?4", Date.class);

    query.setParameter(1, from);
    query.setParameter(2, to);
    query.setParameter(3, firstDayMonthDate);
    query.setParameter(4, lastDayMonthDate);
    List<Date> dates = query.getResultList();
    System.out.println(dates);
    return dates;
}
```

Explicación Refactorización:

La refactorización del método `getThisMonthDatesWithRides()` se centró en decidir su lógica en partes más pequeñas: obtener las fechas de las distintas consultas y añadir los resultados a la lista final. Cada una de estas partes se convirtió en un método independiente (*`getDirectRideDates`, `getRideDatesWithStops`, `getRideDatesFromStopsTable` y `addDates`*), de manera que cada método cumple una única función y es más sencillo de leer. Además, se corrigió un detalle en las comprobaciones con `dates.isEmpty()`, usando `!dates.isEmpty()` para asegurarnos de que las listas con datos se procesen correctamente. Con estos cambios, el método es más fácil de seguir, de mantener y de probar.



La refactorización se realizó usando **Refactor > Extract Method** en Eclipse, logrando que cada método tenga menos de 15 líneas y sea más claro y mantenible.

3. Write simple units of code

3.1. Bad Smell 3 (Mikel Pallin)

Para esta parte, vamos a centrarnos en el método `getStopsAfterFromAndDestination()`, previamente creado. Este método cuenta con 4 ramificaciones, siendo estas 2 *for* y 2 *if*. El libro *'Building Maintainable Software'* indica que el máximo de ramificaciones por método debería ser 4, y aunque el método lo cumpla, vamos a bajar su ramificación para que el método se vuelva más mantenible a futuro.

Código inicial:

```
private void getStopsAfterFromAndDestinations(String from, List<String> res, int numStop, List<Ride> rides1) {
    for(Ride r: rides1) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }

        for(Stop s: r.getStops()) {
            if(s.getName().equals(from)) {
                numStop = s.getNumStop();
            }
        }
        getStopsAfterFrom(res, numStop, r);
    }
}
```

Código Refactorizado:

```
private void getStopsAfterFromAndDestination(String from, List<String> res, int numStop, List<Ride> rides1) {
    for(Ride r: rides1) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }

        numStop = searchStop(from, numStop, r);
        getStopsAfterFrom(res, numStop, r);
    }
}

private int searchStop(String from, int numStop, Ride r) {
    for(Stop s: r.getStops()) {
        if(s.getName().equals(from)) {
            numStop = s.getNumStop();
        }
    }
    return numStop;
}
```

Explicación Refactorización:

Para solucionar el '*Bad Smell*', se ha optado por separar el segundo *for* del método en otro método diferente. La refactorización del método se ha creado utilizando la siguiente función de eclipse: ***Refactor > Extract Method***. Así, se ha conseguido que cada método cuente con únicamente dos ramificaciones, haciéndolos más mantenibles y legibles.

3.2. Bad Smell 4 (Iker López)

En el siguiente fragmento se puede apreciar un *bad smell* que ocurre cuando un método es demasiado largo y mezcla varias tareas en un mismo bloque de código. Esto hace que sea más difícil de entender, mantener y probar con el tiempo. A continuación, se muestra el código original donde se detecta este problema:

Código inicial:

```
public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail, List<Vector<String>> sl) 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.getTransaction().begin();

        Driver driver = db.find(Driver.class, driverEmail);
        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

        if(!sl.isEmpty()) {
            for(Vector<String> vs : sl) {
                int id = 0;
                while (db.find(Stop.class, id) != null) {
                    id++;
                }
                Stop s = new Stop(id, Integer.parseInt(vs.get(1).toString()),ride, vs.get(0),Float.parseFloat(vs.get(2).toString()),ride.getDate());
                ride.getStops().add(s);
                db.persist(s);
            }
        }

        db.persist(driver);
        db.getTransaction().commit();

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

Código Refactorizado:

```

public Ride createRide(String from, String to, Date date, int nPlaces, float price, String driverEmail, List<Vector<String>> sl) throws RideAlreadyExistException, RideMustBeLaterThanTodayException {
    logger.info("=> DataAccess: createRide= from= "+from+" to= "+to+" drivers="+driverEmail+" date "+date);
    try {
        validateRideDate(date);
        db.getTransaction().begin();

        Driver driver = db.find(Driver.class, driverEmail);
        checkExistingRide(from, to, date, driver);
        Ride ride = driver.addRide(from, to, date, nPlaces, price);

        if(!sl.isEmpty()) {
            addStopsToRide(sl, ride);
        }

        db.persist(driver);
        db.getTransaction().commit();

        return ride;
    } catch (NullPointerException e) {
        db.getTransaction().commit();
        return null;
    }
}

```

```

private void addStopsToRide(List<Vector<String>> sl, Ride ride) {
    for(Vector<String> vs : sl) {
        generateNewStopId(ride, vs);
    }
}

private void generateNewStopId(Ride ride, Vector<String> vs) {
    int id = 0;
    while (db.find(Stop.class, id) != null) {
        id++;
    }
    Stop s = new Stop(id, Integer.parseInt(vs.get(1).toString()), ride, vs.get(0), Float.parseFloat(vs.get(2).toString()), ride.getDate());
    ride.getStops().add(s);
    db.persist(s);
}

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

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


```

Explicación Refactorización:

Para que el código sea más claro y fácil de mantener, se decidió dividir el método en varias partes más pequeñas, cada una encargada de una tarea concreta. Esto ayudó a reducir la complejidad y a que la lógica sea más sencilla de seguir. Además, al separar la funcionalidad, ahora resulta mucho más fácil localizar errores, hacer cambios o añadir mejoras en el futuro sin afectar al resto del método.

4. Duplicate code

Para esta parte del proyecto, nos deberemos centrar en otra clase que no sea DataAccess, ya que según los resultados de SonarCloud, esta cuenta con un 0.0% de líneas duplicadas.

	Duplicated Lines (%)	Duplicated Lines
 DataAccess.java	0.0%	0

Por lo que nos centraremos en las clases de la GUI, ya que estas cuentan con varias duplicaciones en su implementación.

4.1. Bad Smell 5 (Mikel Pallin)

En este apartado nos vamos a centrar en la clase MainUserGUI. Como se puede apreciar en la siguiente imagen, el código está repetido varias veces, por lo que se ha tenido que refactorizar.

Código inicial:

```
private void paintAgain() {
    jLabelSelectOption.setText(ResourceBundle.getBundle("Etiquetas").getString("MainUserGUI.SelectOption"));
    jButtonQueryQueries.setText(ResourceBundle.getBundle("Etiquetas").getString("MainUserGUI.QueryRides"));
    jButtonBookQuery.setText(ResourceBundle.getBundle("Etiquetas").getString("MainUserGUI.BookRide"));
    this.setTitle(ResourceBundle.getBundle("Etiquetas").getString("MainUserGUI.MainTitle")+ " - User :"+user.getUsername());
}
```

Código Refactorizado:

```
private void paintAgain() {
    ResourceBundle bundle = ResourceBundle.getBundle("Etiquetas");

    jLabelSelectOption.setText(bundle.getString("MainUserGUI.SelectOption"));
    jButtonQueryQueries.setText(bundle.getString("MainUserGUI.QueryRides"));
    jButtonBookQuery.setText(bundle.getString("MainUserGUI.BookRide"));
    this.setTitle(bundle.getString("MainUserGUI.MainTitle")+ " - User :"+user.getUsername());
}
```

Explicación Refactorización:

Para solucionar el 'Bad Smell', se ha añadido una variable que contenga la repetición del código. La refactorización se ha creado utilizando la siguiente función de eclipse: **Refactor > Extract Local Variable**. Así, se ha conseguido suprimir la repetición de código que había en este método.

4.2. Bad Smell 6 (Iker López)

En la versión original, cada radio button de idioma se declaraba y se le asignaba un *ActionListener* de forma independiente, repitiendo la misma lógica tres veces, lo que genera duplicidad de código.

Código inicial:

```
rdbtnNewRadioButton = new JRadioButton("English");
rdbtnNewRadioButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Locale.setDefault(new Locale("en"));
        System.out.println("Locale: " + Locale.getDefault());
        paintAgain();
    }
});
buttonGroup.add(rdbtnNewRadioButton);

rdbtnNewRadioButton_1 = new JRadioButton("Euskara");
rdbtnNewRadioButton_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        Locale.setDefault(new Locale("eus"));
        System.out.println("Locale: " + Locale.getDefault());
        paintAgain();
    }
});
buttonGroup.add(rdbtnNewRadioButton_1);

rdbtnNewRadioButton_2 = new JRadioButton("Castellano");
rdbtnNewRadioButton_2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Locale.setDefault(new Locale("es"));
        System.out.println("Locale: " + Locale.getDefault());
        paintAgain();
    }
});
buttonGroup.add(rdbtnNewRadioButton_2);

panel = new JPanel();
panel.add(rdbtnNewRadioButton_1);
panel.add(rdbtnNewRadioButton_2);
panel.add(rdbtnNewRadioButton);
```


Código Refactorizado:

```
rdbtnNewRadioButton = createLanguageButton("English", "en");
rdbtnNewRadioButton_1 = createLanguageButton("Euskara", "eus");
rdbtnNewRadioButton_2 = createLanguageButton("Castellano", "es");

panel = new JPanel();
panel.add(rdbtnNewRadioButton_1);
panel.add(rdbtnNewRadioButton_2);
panel.add(rdbtnNewRadioButton);

private JRadioButton createLanguageButton (String label, String localeCode) {
    JRadioButton button = new JRadioButton(label);
    button.addActionListener(e -> {
        Locale.setDefault(new Locale(localeCode));
        System.out.println("Locale: " + Locale.getDefault());
        paintAgain();
    });
    buttonGroup.add(button);
    return button;
}
```

Explicación Refactorización:

Se ha creado un método auxiliar *createLanguageButton* que recibe el texto del botón y el código de idioma, y se encarga de crear el radio button, asignarle el *ActionListener* y añadirlo al *ButtonGroup*. De esta manera, la creación de los tres botones queda más limpia y evita duplicaciones.

5. Keep unit interfaces small

5.1. Bad Smell 7 (Mikel Pallin)

Para este último *'Bad Smell'* nos volveremos a centrar en el método *getStopsAndDestinations()* ya que este método cuenta 6 variables dentro de su código, superando las 4 variables como máximo impuestas por el libro *'Building Maintainable Software'*.

Código inicial:

```
public List<String> getStopsAndDestinations(String from){
    List<String> res = new ArrayList<String>();
    int numStop = 0;

    TypedQuery<Ride> query1 = db.createQuery("SELECT r FROM Ride r WHERE r.stops.name=?1", Ride.class);
    TypedQuery<Ride> query2 = db.createQuery("SELECT r FROM Ride r WHERE r.from=?2", Ride.class);
    query1.setParameter(1, from);
    query2.setParameter(2, from);

    List<Ride> rides1 = query1.getResultList();
    List<Ride> rides2 = query2.getResultList();

    getStopsAfterFromAndDestination(from, res, numStop, rides1);
    getAllStopsAndDestination(res, rides2);

    return res;
}
```

Código Refactorizado:

```
public List<String> getStopsAndDestinations(String from){
    List<String> res = new ArrayList<String>();
    int numStop = 0;

    getStopsAfterFromAndDestination(from, res, numStop);
    getAllStopsAndDestination(from, res);

    return res;
}

private void getStopsAfterFromAndDestination(String from, List<String> res, int numStop) {
    TypedQuery<Ride> query1 = db.createQuery("SELECT r FROM Ride r WHERE r.stops.name=?1", Ride.class);
    query1.setParameter(1, from);
    List<Ride> rides1 = query1.getResultList();

    for(Ride r: rides1) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }
        numStop = searchStop(from, numStop, r);
        getStopsAfterFrom(res, numStop, r);
    }
}

private void getAllStopsAndDestination(String from, List<String> res) {
    TypedQuery<Ride> query2 = db.createQuery("SELECT r FROM Ride r WHERE r.from=?2", Ride.class);
    query2.setParameter(2, from);
    List<Ride> rides2 = query2.getResultList();

    for (Ride r: rides2) {
        if(!res.contains(r.getTo())) {
            res.add(r.getTo());
        }
        for(Stop s: r.getStops()) {
            if(!res.contains(s.getName())) {
                res.add(s.getName());
            }
        }
    }
}
```

Explicación Refactorización:

Para solucionar el '*Bad Smell*', las queries y todas sus variables se han pasado al método del bucle que les corresponde, bajando así el número de variables con las que cuenta el método principal, contando ahora con únicamente dos variables en total.

5.2. Bad Smell 8 (Iker López)

Para este último Bad Smell nos centraremos en el método `addBooking()`, ya que cuenta con 7 parámetros, superando el límite de 4 indicado por el libro *Building Maintainable Software*, lo que dificulta su comprensión y mantenimiento.

Código inicial:

```
public boolean addBooking(User user, Ride ride, String from, String to, Integer numSeats, Double prize, Stop stop) {
    db.getTransaction().begin();
    User u = db.find(User.class, user.getEmail());
    int id = 0;
    while (db.find(Booking.class, id) != null) {
        id++;
    }

    Booking b = new Booking(id, ride, user, ride.getDriver(), from, to, numSeats, prize);

    Driver d = db.find(Driver.class, ride.getDriver().getEmail());

    if (stop != null) {
        b.setStop(stop);
    }

    u.getBooked().add(b);
    d.getBookings().add(b);

    db.persist(u);
    db.persist(d);
    db.persist(b);
    db.getTransaction().commit();
    return true;
}
```

Código Refactorizado:

```
public boolean addBooking(User user, Ride ride, Integer numSeats) {
    db.getTransaction().begin();

    User u = db.find(User.class, user.getEmail());
    Driver d = db.find(Driver.class, ride.getDriver().getEmail());

    int id = 0;
    while (db.find(Booking.class, id) != null) {
        id++;
    }

    String from = ride.getFrom();
    String to = ride.getTo();
    Double prize = (double) ride.getPrice();
    Stop stop = (ride.getStops() != null && !ride.getStops().isEmpty()) ? ride.getStops().get(0) : null;

    Booking b = new Booking(id, ride, user, ride.getDriver(), from, to, numSeats, prize);

    if (stop != null) {
        b.setStop(stop);
    }

    u.getBooked().add(b);
    d.getBookings().add(b);

    db.persist(u);
    db.persist(d);
    db.persist(b);
    db.getTransaction().commit();
    return true;
}
```

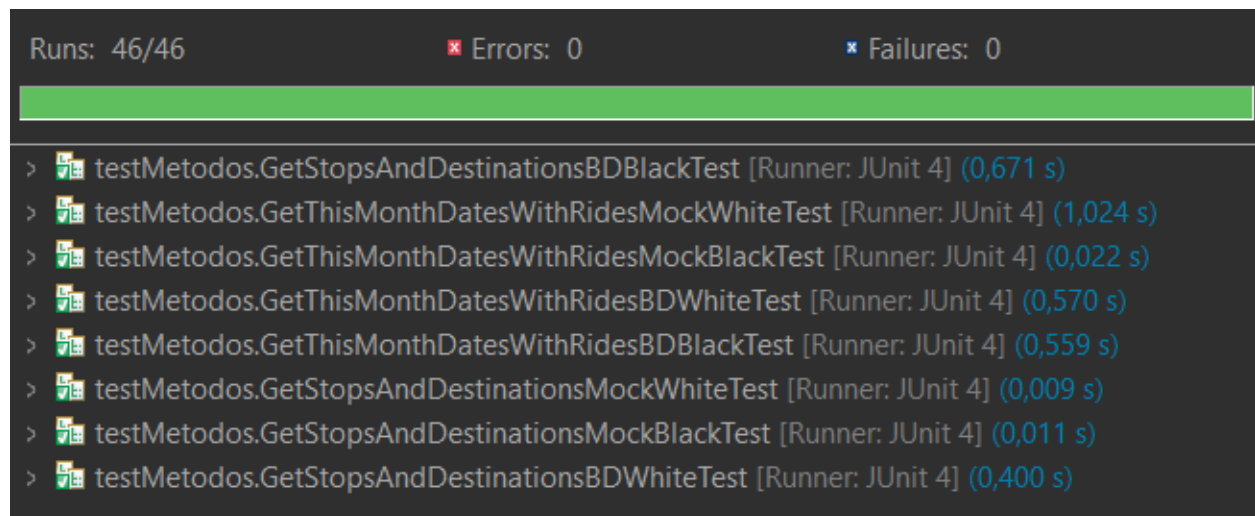
Explicación Refactorización:

Para solucionar el Bad Smell, se agruparon los parámetros relacionados en objetos existentes como *User* y *Ride*, reduciendo así el número de parámetros del método. Esto hace que el código sea más legible y fácil de mantener.

Código Refactorizado:

6. Test después de Refactorización

Después de las refactorizaciones hechas para este proyecto, los test creados en el proyecto anterior deberían de funcionar exactamente igual. Para comprobar esto, simplemente se han vuelto a ejecutar los test de JUnit, devolviendo un resultado exitoso.



The screenshot displays a JUnit test runner interface with a dark background. At the top, it shows 'Runs: 46/46', 'Errors: 0' (with a red 'x' icon), and 'Failures: 0' (with a blue 'x' icon). Below this is a solid green progress bar. The main area lists eight test cases, each preceded by a green checkmark icon and a right-pointing arrow. The test names and their execution times are as follows:

- > testMetodos.GetStopsAndDestinationsBDBlackTest [Runner: JUnit 4] (0,671 s)
- > testMetodos.GetThisMonthDatesWithRidesMockWhiteTest [Runner: JUnit 4] (1,024 s)
- > testMetodos.GetThisMonthDatesWithRidesMockBlackTest [Runner: JUnit 4] (0,022 s)
- > testMetodos.GetThisMonthDatesWithRidesBDWhiteTest [Runner: JUnit 4] (0,570 s)
- > testMetodos.GetThisMonthDatesWithRidesBDBlackTest [Runner: JUnit 4] (0,559 s)
- > testMetodos.GetStopsAndDestinationsMockWhiteTest [Runner: JUnit 4] (0,009 s)
- > testMetodos.GetStopsAndDestinationsMockBlackTest [Runner: JUnit 4] (0,011 s)
- > testMetodos.GetStopsAndDestinationsBDWhiteTest [Runner: JUnit 4] (0,400 s)