



Escuela Técnica Superior de
Ingeniería Informática

Universidad de Sevilla

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DISEÑO Y PRUEBAS II

REFACTORING

Autores:

Francisco Manuel Cordero Vela

Alejandro Fuentes Gómez

José Manuel González Mancilla

Miguel Ponce Melero

María Teresa Monge Zambrano

Pedro Padilla Molina

Grupo:

G3-11

Lunes 5 de Junio de 2020

1. Introducción

El objetivo de este documento es analizar el código fuente para conseguir que este sea más legible y entendible a simple vista. Para ello, el equipo se ha detenido profundamente en los controladores y servicios, ya que estas clases son el lugar donde reside casi al completo la lógica de negocio.

El equipo de trabajo ha acordado hacer 5 refactorizaciones, escogiendo las partes del código que más necesitaban una mejora, basandonos en las técnicas aprendidas en la asignatura.

Como viene siendo habitual, el equipo ha trabajado en parejas. La asignación de parejas y las respectivas tareas son las siguientes:

- Pareja 1: Formada por María Teresa Monge Zambrano y Francisco Manuel Cordero Vela. Las tareas asignadas a esta pareja han sido refactorización de métodos en las clases DashboardController y VetService.
- Pareja 2: Formada por José Manuel González Mancilla y Alejandro Fuentes Gómez. Las tareas asignadas a esta pareja ha sido la refactorización en la clase PetController.
- Pareja 3 : Formada por Miguel Ponce Melero y Pedro Padilla Molina. Las tareas asignadas a esta pareja ha sido la refactorización en la clase PetController.

2. Refactorizaciones

2.1. Refactorización 1: PetController.

La refactorización 1 consiste en mejorar la entendibilidad del método `ProcessDelete` que reside en la clase `PetController`. Como se puede intuir, este método consiste en borrar una mascota. Sin embargo, existe un escenario negativo que en determinadas ocasiones no permite la eliminación de la mascota. A este escenario se llega cuando la mascota que se quiere eliminar tiene residencias activas. Por ello, el código propuesto inicialmente tenía muchas líneas de código debido a la aparente magnitud de todas las comparaciones que tiene que hacer el método. Se puede observar en la siguiente imagen el método antes de realizar la refactorización:

```
@GetMapping(value = "/pets/{petId}/delete")
public String processDelete(@PathVariable("ownerId") final int ownerId, @PathVariable("petId") final int petId, final Mc
    Pet pet = this.petService.findPetById(petId);
    Owner owner = this.ownerService.findOwnerById(ownerId);
    Collection<Diagnosis> diagnosis = this.petService.findAllDiagnosisByPetId(petId);
    int count = 0;
    for(Diagnosis d: diagnosis) {
        if(d.getResidence() != null) {
            if(LocalDate.now().isBefore(d.getResidence().getDateEnd())) {
                count++;
            }
        }
    }

    if(count == 0) {
        for(Diagnosis di: diagnosis) {
            pet.removeDiagnosis(di);
            this.diagnosisService.deleteDiagnosis(di);
        }

        owner.removePet(pet);
        this.petService.deletePetById(petId);

        return "redirect:/owners/{ownerId}";
    } else {
        return "redirect:/errorDelete";
    }
}
```

Analizando el código del método, se puede observar que se tiene dos bucles en un mismo método. Por lo tanto, esto provoca que el código posea un "Bad Smell", ya que a simple vista, una persona que viera el código, debería invertir un periodo considerable de tiempo analizándolo para conocer cuál es la función que cumple el método completo. Como consecuencia, se puede mejorar sacando dichos bucles del método principal creando dos métodos privados auxiliares. A los métodos citados poseen un nombre totalmente identificativos con la función que realizan para una correcta interpretación del código. La refactorización comentada, se puede observar en las siguientes imágenes:

```
@GetMapping(value = "/pets/{petId}/delete")
public String processDelete(@PathVariable("ownerId") final int ownerId, @PathVariable("petId") final int petId,
    final ModelMap model) {
    Pet pet = this.petService.findPetById(petId);
    Owner owner = this.ownerService.findOwnerById(ownerId);
    Collection<Diagnosis> diagnosis = this.petService.findAllDiagnosisByPetId(petId);
    if (areThereAnyResidence(diagnosis)) {
        deleteDiagnosisFromPet(diagnosis, pet);
        owner.removePet(pet);
        this.petService.deletePetById(petId);
        return "redirect:/owners/{ownerId}";
    } else {
        return "redirect:/errorDelete";
    }
}
```

Método `processDelete` refactorizado.

```

private Boolean areThereAnyResidence(Collection<Diagnosis> diagnosis) {
    int count = 0;
    Boolean res = true;
    for (Diagnosis d : diagnosis) {
        if (d.getResidence() != null) {
            if (LocalDate.now().isBefore(d.getResidence().getDateEnd())) {
                count++;
            }
        }
    }
    if (count >= 1) {
        res = false;
    }
    return res;
}



private void deleteDiagnosisFromPet(Collection<Diagnosis> diagnosis, Pet pet) {
    for (Diagnosis di : diagnosis) {
        pet.removeDiagnosis(di);
        this.diagnosisService.deleteDiagnosis(di);
    }
}

```

Métodos auxiliares.

Como se puede observar, con estos métodos auxiliares, se puede conocer si una mascota tiene al menos una residencia activa de una forma clara y concisa. Además, si no la posee, se borran primero todos los diagnósticos asociados a esa mascota y posteriormente se elimina la mascota de la base de datos.

Finalmente, el equipo de trabajo ha comprobado que no existe ninguna anomalía con respecto al funcionamiento de la aplicación. Podemos corroborar esta información debido a que los test de integración del PetController continúan funcionando sin que hayan sufrido ninguna modificación.

 testProcessDeleteFormError() (0,092 s)
 testProcessDeleteFormSuccess() (0,101 s)

2.2. Refactorización 2: ResidenceService.

La refactorización 2 consiste en mejorar la clase ResidenceService. Esta clase tiene varios métodos que realizan la misma función. Por ello, debemos dejar solo uno de estos métodos al que además le pondremos la propiedad readOnly que no estaba añadida:

```
@Transactional(rollbackFor = NoElementException.class)
@Cacheable("activeResidences")
public Collection<Pet> findAllResidenceNotEnded() throws DataAccessException, NoElementException{
    Collection<Pet> res = new ArrayList<Pet>();
    res = this.residenceRepository.findAllNotEnded();
    if(res.isEmpty()) {
        throw new NoElementException();
    }
    return res;
}

//Este no se usa
public Set<Residence> findActiveResidences(){
    Set<Residence> residences = new HashSet<Residence>();
    for(Residence r :this.residenceRepository.findAll()) {
        if(r.getDateEnd().isAfter(LocalDate.now())) {
            residences.add(r);
        }
    }
    return residences;
}
```

Al mirar el código comprobamos como ambos métodos realizan la misma función, siendo el segundo método más complejo y menos entendible. Por lo tanto, esto provoca que el código posea varios "Bad Smell", ya que, tenemos métodos duplicados, además de no tener la propiedad readOnly. Como consecuencia, hemos de eliminar el segundo método y añadir la etiqueta readOnly al primero:

```
@Transactional(readOnly= true, rollbackFor = NoElementException.class)
@Cacheable("activeResidences")
public Collection<Pet> findAllResidenceNotEnded() throws DataAccessException, NoElementException{
    Collection<Pet> res = new ArrayList<Pet>();
    res = this.residenceRepository.findAllNotEnded();
    if(res.isEmpty()) {
        throw new NoElementException();
    }
    return res;
}
```

Como se puede observar, nos hemos ahorrado un método duplicado además de añadir una propiedad fundamental para que no se puedan hacer modificaciones a lo que devuelve el método.

Para terminar, hemos cambiado todas las funciones que usaban el método ya eliminado para que ahora utilicen el mostrado en la captura de arriba, sin encontrar ningún problema en la aplicación.

2.3. Refactorización 3: findVetWithMostVisit.

La refactorización 3 consiste en mejorar la calidad, legibilidad y eficiencia del código del método findVetWithMostVisit, que se encuentra en el VetService. Dicho método se encontraba al inicio como vemos a continuación:

```

@Transactional(readonly = true)
public Vet findVetWithMostVisit() {
    Collection<Visit> visits = this.vetRepository.findAllVisits();
    Collection<Vet> vets = this.findVets();
    Vet result = new Vet();
    Integer res = 0;

    for(Vet v: vets) {
        Integer count = 0;
        for(Visit visit: visits) {
            if(visit.getVet().getId().equals(v.getId())) {
                count++;
            }
        }
        if(count>res) {
            res = count;
            result = v;
        }
    }
    return result;
}

```

Como se puede apreciar con solo observar el método, es muy poco eficiente, puesto que nos estamos trayendo por un lado todas las visitas y por otro todos los veterinarios, y estamos recorriendo con un doble for, las visitas dentro de los veterinarios, hasta que ambos coinciden y tienen el mismo id, tanto el veterinario que recorremos, como el veterinario de la visita que también recorremos, sumando 1 en un contador, cuando se da este caso. Nos guardamos este contador en una variable, para ir comparando en cada iteración y , de esta forma quedarnos con el veterinario con más visitas siempre.

Evidentemente, esto se puede mejorar, si nos traemos directamente de base de datos el veterinario con más visitas, y nos ahorramos toda esta parafernalia difícil de entender y poco eficiente, pero que en su día fue la forma más rápida con la que conseguimos que funcionara.

```

@Query("SELECT max(v.vet), count(v.vet) FROM Visit v GROUP BY v.vet ORDER BY count(v.vet) DESC")
Collection<Vet> findVetMostVisit() throws DataAccessException;

```

Query findVetMostVisit creada.

```

@Transactional(readonly = true)
public Vet findVetWithMostVisit() {

    Vet vet = this.vetRepository.findVetMostVisit().stream().findFirst().get();

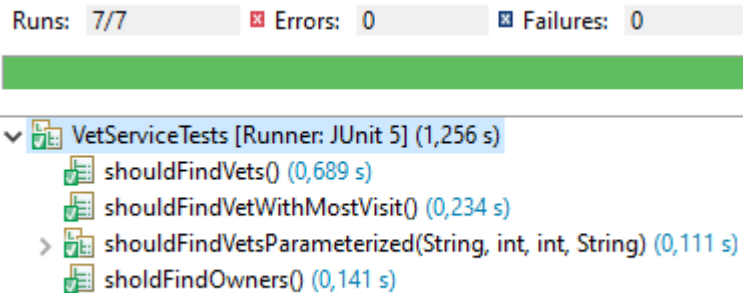
    return vet;
}

```

Método findVetWithMostVisit refactorizado.

Con esa query hacemos lo mismo que estamos haciendo anteriormente en el método pero con lenguaje sql, ahorrandonos bastante coste de computación y dejando el método mucho más limpio.

Finalmente, el equipo de trabajo ha comprobado que no existe ninguna anomalía con respecto al funcionamiento de la aplicación. Podemos corroborar esta información debido a que los test de integración del VetService continúan funcionando sin que hayan sufrido ninguna modificación.



2.4. Refactorización 4: Repositories y SpringDataRepositories.

La refactorización número 4, consiste en ordenar de forma correcta los métodos usados en los repositorios y en los repositorios SpringData. Puesto que por lo general, las queries propias de métodos que creabamos nosotros, se deben crear en los SpringDataRepositories y desde los Reposirories se les hace una llamada a esos métodos creados. Sin embargo, eso no lo teníamos ordenado de esta forma en todos los repositorios, y se ha hecho ahora.

```
public interface PetRepository {
    List<PetType> findPetTypes() throws DataAccessException;

    Pet findById(int id) throws DataAccessException;

    void save(Pet pet) throws DataAccessException;

    void deletePetById(int petId) throws DataAccessException;

    Collection<Pet> findAll();
    @Query("Select p.diagnosis from Pet p where p.id =:id")
    Collection<Diagnosis> findPetDiagnosis(@Param("id") int petId);

    @Query("SELECT max(p.type), COUNT( p.type ) AS total FROM Pet p where p.visits IS NOT EMPTY GROUP BY p.type ORDER BY total DESC")
    Collection<PetType> findFrequentPet();
}
```

Como se puede ver, se estaban realizando las queries en el propio PetRepository. Sin embargo, se han movido a SpringDataPetRepository, de la siguiente forma:

```
public interface SpringDataPetRepository extends PetRepository, Repository<Pet, Integer> {
    @Override
    @Query("SELECT ptype FROM PetType ptype ORDER BY ptype.name")
    List<PetType> findPetTypes() throws DataAccessException;

    @Override
    @Query("Select p.diagnosis from Pet p where p.id =:id")
    Collection<Diagnosis> findPetDiagnosis(@Param("id") int petId);

    @Override
    @Query("SELECT max(p.type), COUNT( p.type ) AS total FROM Pet p where p.visits IS NOT EMPTY GROUP BY p.type ORDER BY total DESC")
    Collection<PetType> findFrequentPet();
}
```

```
public interface PetRepository {
    List<PetType> findPetTypes() throws DataAccessException;

    Pet findById(int id) throws DataAccessException;

    void save(Pet pet) throws DataAccessException;

    void deletePetById(int petId) throws DataAccessException;

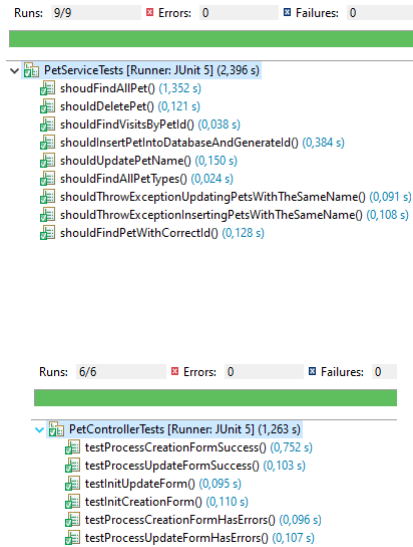
    Collection<Pet> findAll();

    Collection<Diagnosis> findPetDiagnosis(int petId) throws DataAccessException;

    Collection<PetType> findFrequentPet() throws DataAccessException;
}
```

PetRepository refactorizado.

Como vemos, se han movido estas queries y se han llamado desde el repositorio. Además, se ha comprobado que la aplicación sigue funcionando sin ningún problema. Para demostrarlo, hemos lanzado los test correspondientes a dicha parte de la aplicación y siguen funcionando sin problema, a parte de probar la aplicación.



2.5. Refactorización 5: DashboardController.

La refactorización número 5, consiste en mejorar legibilidad del Dashboard, ya que cada método estaba ubicado en un controlador distinto y era muy difícil. Por ejemplo, el método para obtener el tipo de mascota que frecuenta más la clínica veterinaria estaba en DiagnosisController y para poder buscarlo era muy complicado.

```

135
136 //Admin Dashboard
137 @GetMapping(value = "admin/diagnosis")
138 public String processAdminDiagnosisList(ModelMap model) {
139     Collection<Diagnosis> diagnosis = new ArrayList<Diagnosis>();
140     try {
141         diagnosis = this.diagnosisService.listDiagnosis();
142     }
143
144     catch(NoElementOnListException ex){
145         return "redirect:errorNoElement";
146     }
147
148     model.put("diagnosis", diagnosis);
149     return "diagnosis/diagnosislist";
150 }
151
152
153 @GetMapping("/pets/frequentPet")
154 public String listFrequentPet(ModelMap modelMap) {
155     PetType pet= petService.findFrequentPet().stream().findFirst().get();
156     modelMap.addAttribute("frequentPet", pet);
157     return "/pets/frequentPetList";
158 }
159

```

Ejemplo de método mal ubicado.

Para mejorar todo ello y tener todo ubicado en un mismo sitio, se ha movido todo lo que tiene que ver con Dashboard a DashboardController y por lo tanto también se han tenido que cambiar los tests de lugar a su correspondiente a Dashboard.


```

@Autowired
public DashboardController(ResidenceService residenceService, DiagnosisService diagnosisService, PetService petService) {
    this.diagnosisService = diagnosisService;
    this.petService = petService;
    this.vetService = vetService;
    this.residenceService = residenceService;
}

@GetMapping(value = "/admin")
public String dashboard(Map<String, Object> model) {
    String vetWithMostVisit = getVetWithMostVisits();
    model.put("vetWithMostVisits", vetWithMostVisit);
    return "admin/dashboard";
}

@GetMapping("/pets/frequentPet")
public String listFrequentPet(ModelMap modelMap) {
    PetType pet = petService.findFrequentPet().stream().findFirst().get();
    modelMap.addAttribute("frequentPet", pet);
    return "/pets/frequentPetList";
}

//Admin Dashboard
@GetMapping(value = "admin/diagnosis")

@Test
public void testProcessfindAllResidencesNotEndedSuccess() throws Exception {
    ModelMap model = new ModelMap();
    String view = dashboardController.findAllResidencesNotEnded(model);

    assertEquals(view, "residences/adminResidenceList");
}

@Test
public void testChooseVetList() throws Exception {
    Map<String, Object> model = new HashMap<String, Object>();

    String view = dashboardController.chooseVetList(model);

    assertEquals(view, "vets/chooseVet");
}

```

Una vez realizado todo ello, se ha comprobado que funcione bien la aplicación (sin añadir nueva funcionalidad) y que los tests funcionen, como por ejemplo DashboardIntegrationTests.

```

Runs: 5/5   Errors: 0   Failures: 0

```

```

✓ DashboardControllerIntegrationTests [Runner: JUnit 5] (0,880 s)
  ✓ testProcessfindAllResidencesNotEndedSuccess() (0,519 s)
  ✓ testProcessAdminDiagnosisListSuccess() (0,132 s)
  ✓ testChooseVetList() (0,024 s)
  ✓ testGetVetWithMostVisits() (0,080 s)
  ✓ testProcessAdminFrequentPet() (0,125 s)

```