CQRS o Command Query Responsability Segregation es un patrón de diseño enterprise. No es un patrón excesivamente complejo pero sí es un patrón que cuesta bastante entender su utilidad . Para ello vamos a partir de un ejemplo bastante sencillo en el que tenemos las clase Persona y Dirección:

[ihc-hide-content ihc_mb_type=”show” ihc_mb_who=”4″ ihc_mb_template=”1″]

```
package com.arquitecturajava.cqrs;

public class Persona {

  private String nombre;
  private int edad;
  private Direccion direccion;
  public Direccion getDireccion() {
    return direccion;
  }
  public void setDireccion(Direccion direccion) {
    this.direccion = direccion;
  }
  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
  public int getEdad() {
```

```
    return edad;
  }
  public void setEdad(int edad) {
    this.edad = edad;
  }
  public Persona(String nombre, int edad) {
    super();
    this.nombre = nombre;
    this.edad = edad;
  }
}
```

```
package com.arquitecturajava.cqrs;

public class Direccion {

  private String calle;
  private String numero;
  private Persona persona;
  public Persona getPersona() {
    return persona;
  }
  public void setPersona(Persona persona) {
    this.persona = persona;
  }
  public String getCalle() {
    return calle;
  }
  public void setCalle(String calle) {
    this.calle = calle;
  }
  public String getNumero() {
```

```
    return numero;
  }
  public void setNumero(String numero) {
    this.numero = numero;
  }
}
```

CQRS y Repositorios

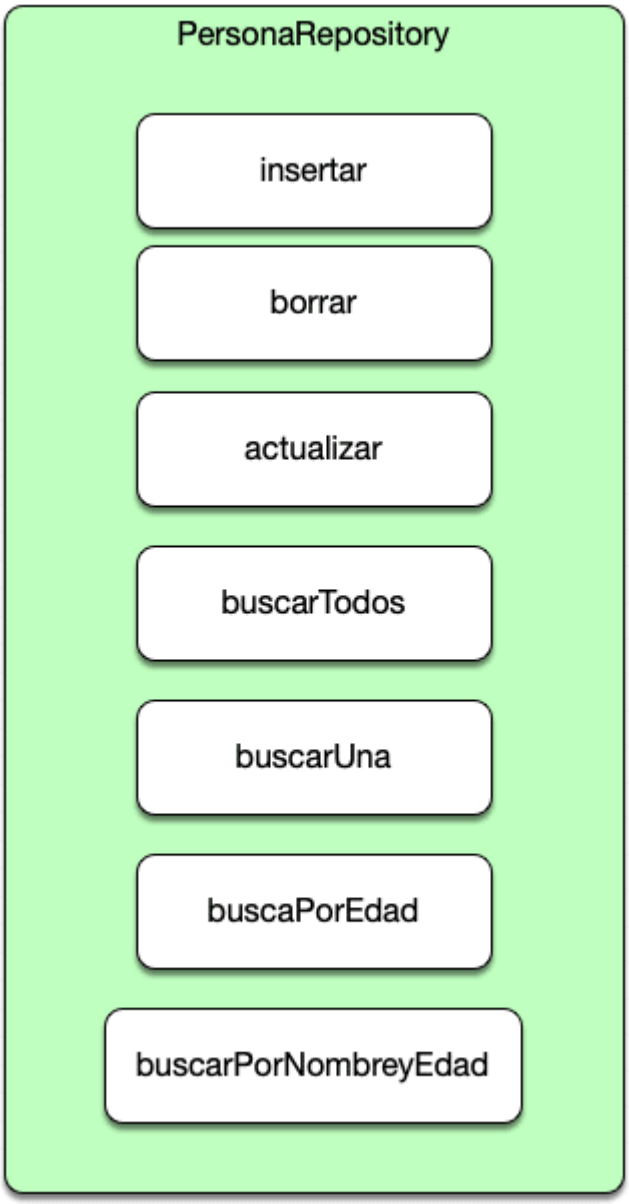
Lo más habitual es cuando diseñamos las clases comenzar a diseñar también las clases de Repositorio ya que vamos a salvar esta información en la base de datos . En este ejemplo vamos a generar un repositorio que simula las operaciones elementales en memoria con Java 8.

```
package com.arquitecturajava.cqrs;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class PersonaRepositoryMemoria implements PersonaRepository {

  private List<Persona> lista;

  public PersonaRepositoryMemoria(List<Persona> lista) {
    super();
    this.lista = lista;
  }

  @Override
  public void insertar(Persona p) {
    lista.add(p);
```

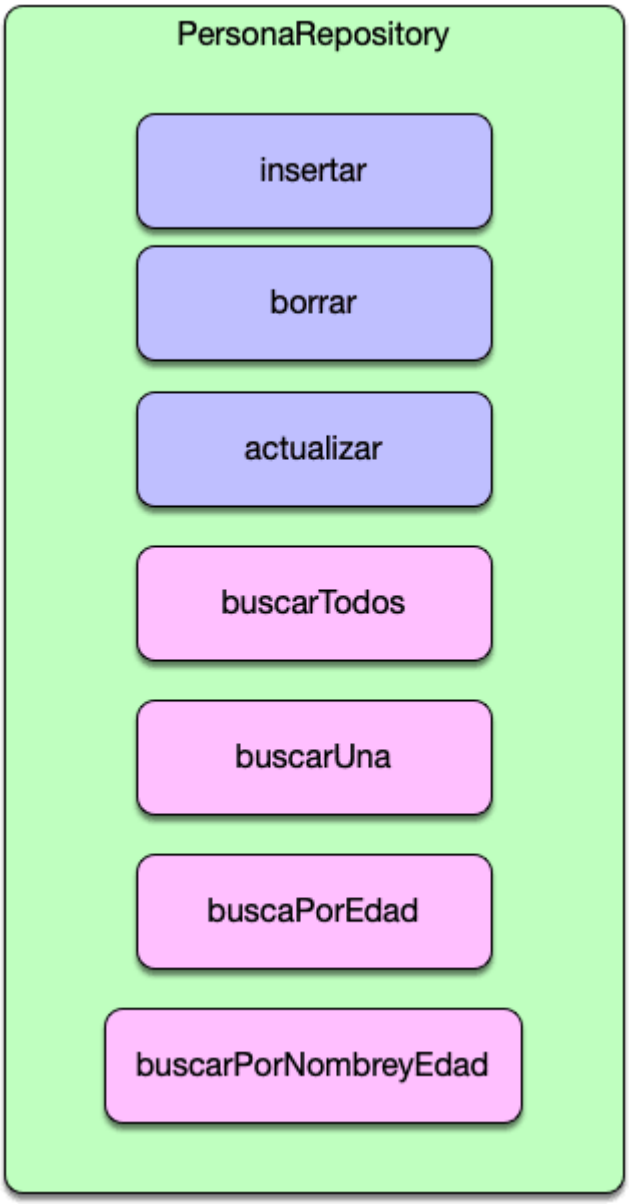
```
  }

  @Override
  public void borrar(Persona p) {
    lista.remove(p);
  }

  @Override
  public void actualizar(Persona p) {

    Optional<Persona> elegida = lista.stream().filter(persona ->
persona.getNombre().equals(p)).findFirst();
    if (elegida.isPresent()) {
      elegida.get().setEdad(p.getEdad());
    }

  }

  @Override
  public List<Persona> buscarPorEdad(int edad) {

    List<Persona> personas = lista.stream().filter(persona ->
persona.getEdad() == edad)
        .collect(Collectors.toList());

    return personas;
  }

  @Override
  public List<Persona> buscarPorNombreyEdad(String nombre, int edad) {
```

```
    List<Persona> personas = lista.stream().filter(persona ->
persona.getEdad() == edad)
        .filter(persona ->
persona.getNombre().equals(nombre)).collect(Collectors.toList());
    return personas;
  }

  @Override
  public Optional<Persona> buscarUna(String nombre) {

    Optional<Persona> elegida = lista.stream().filter(persona ->
persona.getNombre().equals(nombre)).findFirst();

    return elegida;
  }

  @Override
  public List<Persona> buscarTodos() {

    return lista;
  }
}
```

No se trata de un repositorio muy complejo pero si que acumula ya bastantes métodos:



Hay situaciones en las que los repositorios comienzan a agrupar una responsabilidad y comenzamos a tener demasiados métodos en ellos. En principio no es una situación excesivamente dramática pero si poco a poco se van sumando métodos y más métodos nos encontraremos con que hay muchos métodos de búsqueda y muy pocos que realicen modificaciones ,borrados o inserciones . Es como si hubiera dos responsabilidades agrupadas en el mismo repositorio.

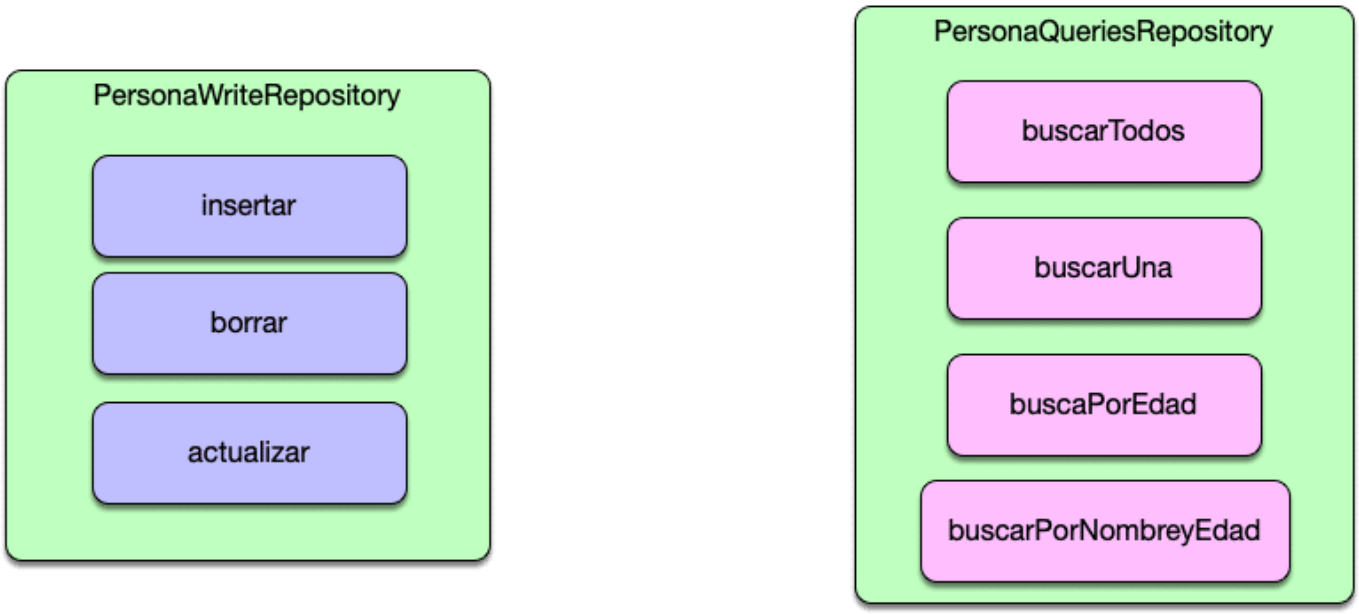


Son responsabilidades que comienzan a verse como distintas y que podrían ser separadas . Para tomar una decisión sobre cómo estas responsabilidades deben tratarse debemos hacernos algunas preguntas adicionales.

- ¿La aplicación tiene un enfoque claramente diferente para la parte de consultas que la parte de inserciones/borrados? . Es decir podemos estar ante una aplicación que probablemente tiene decenas o cientos de pantallas con consultas y unas pocas pantallas

para borrar o insertar registros.

- ¿Necesitamos una mayor escalabilidad en la parte de consultas por parte de la aplicación? . Es decir en el futuro podemos tener cientos de usuarios concurrentes realizando consultas mientras que tenemos unos pocos usuarios introduciendo datos de forma más puntual.
- ¿Podemos desear construir un modelo diferente de dominio para la parte de consultas que para la parte de inserciones ya que estas son más puntuales?
  Si la respuesta es SI a todas estas preguntas . Probablemente estamos ante una situación en la que queramos utilizar este patron CQRS (Command Query Responsability Segregation)



Este patrón de diseño se encarga de dividir nuestros repositorios en dos clases diferentes . La primera clase se encarga de realizar las consultas Queries y la segunda clase se encarga de las modificaciones.

```
package com.arquitecturajava.cqrs2;

import java.util.List;
```

```
import java.util.Optional;
import java.util.stream.Collectors;

import com.arquitecturajava.cqrs.Persona;

public class PersonaRepositoryQueriesMemoria implements
PersonaRepositoryQueries {

  private List<Persona> lista;

  public PersonaRepositoryQueriesMemoria(List<Persona> lista) {
    super();
    this.lista = lista;
  }

  @Override
  public List<Persona> buscarPorEdad(int edad) {

    List<Persona> personas = lista.stream().filter(persona ->
persona.getEdad() == edad)
        .collect(Collectors.toList());

    return personas;
  }

  @Override
  public List<Persona> buscarPorNombreyEdad(String nombre, int edad) {

    List<Persona> personas = lista.stream().filter(persona ->
persona.getEdad() == edad)
        .filter(persona ->
```

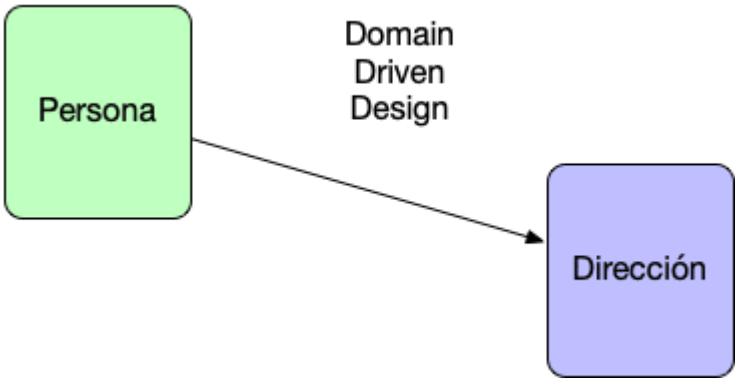
```
persona.getNombre().equals(nombre)).collect(Collectors.toList());
    return personas;
  }

  @Override
  public Optional<Persona> buscarUna(String nombre) {

    Optional<Persona> elegida = lista.stream().filter(persona ->
persona.getNombre().equals(nombre)).findFirst();

    return elegida;
  }

  @Override
  public List<Persona> buscarTodos() {

    return lista;
  }

}
```

```
package com.arquitecturajava.cqrs3;

import java.util.List;
import java.util.Optional;



public class PersonaRepositoryWriteMemoria implements
PersonaRepositoryWrite {

  private List<Persona> lista;
```

```
  public PersonaRepositoryWriteMemoria(List<Persona> lista) {
    super();
    this.lista = lista;
  }

  @Override
  public void insertar(Persona p) {
    lista.add(p);
  }

  @Override
  public void borrar(Persona persona) {
    lista.remove(persona);
  }

  @Override
  public void actualizar(Persona p) {

    Optional<Persona> elegida = lista.stream().filter(persona ->
persona.getNombre().equals(p)).findFirst();
    if (elegida.isPresent()) {
      Persona persona= elegida.get();
      persona.setEdad(p.getEdad());
    }

  }
}
```

En principio podría valernos a nivel inicial con una mera separación de responsabilidades por un lado las consultas y por el otro lado las modificaciones . Ahora bien el patrón de diseño se denomina CQRS (Command Query Responsability Segregation). No es una simple separación de responsabilidades sino que suele implicar cambios más profundos. Cuando

nosotros modificamos información es muy muy común usar los objetos de negocio como Persona o Dirección para salvar los cambios . Por lo tanto el modelo de escritura esta muy orientado a Domain Driven Design y a relaciones entre clases:



En cambio esto no tiene porque ser así en el modelo de Queries ya que estas necesitan muchas veces afinar las consultas y devolver unos datos muy concretos y compactos . Por ejemplo a nivel del Repositorio de Queries es muy habitual usar DTOS o Data Transfer Objects.

```
package com.arquitecturajava.cqrs2;

public class PersonaDTO {

  private String nombre;
  private int edad;
  private String calle;
  private String numero;
  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
```

```
  public int getEdad() {
    return edad;
  }
  public void setEdad(int edad) {
    this.edad = edad;
  }
  public String getCalle() {
    return calle;
  }
  public void setCalle(String calle) {
    this.calle = calle;
  }
  public String getNumero() {
    return numero;
  }
  public void setNumero(String numero) {
    this.numero = numero;
  }
  public PersonaDTO(String nombre, int edad, String calle, String
numero) {
    super();
    this.nombre = nombre;
    this.edad = edad;
    this.calle = calle;
    this.numero = numero;
  }
}
```

Por lo tanto podemos modificar algunos de los métodos a nivel del interface de Queries para gestionar DTOS en este caso usando PersonaDTO

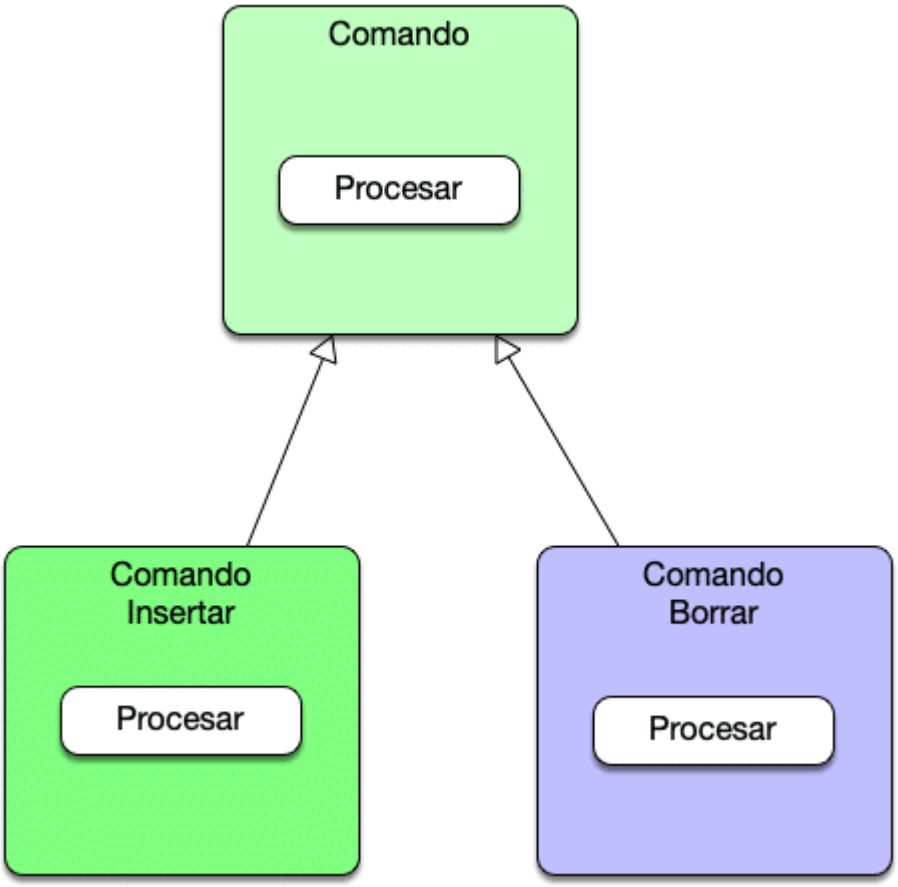
```
@Override
```

```
  public Optional<PersonaDTO> buscarUna(String nombre) {
    PersonaDTO dto = null;
    Optional<Persona> elegida = lista.stream().filter(persona ->
persona.getNombre().equals(nombre)).findFirst();
    if (elegida.isPresent()) {
      Persona p= elegida.get();
      dto = new
PersonaDTO(p.getNombre(),p.getEdad(),p.getDireccion().getCalle(),p.get
Direccion().getNumero());
    }
    return Optional.of(dto);
  }
```

Acabamos de realizar un cambio a nivel del interface de Queries para que puede retornar un DTO que incluya la Persona y su Dirección de forma directa. Esto es bastante común en el api de consultas y mucho menos común en el api de comandos. Nos queda de construir el api de comandos.

CQRS y Comandos

¿ Dónde tenemos aquí una estructura de Comandos? . La realidad es que no los tenemos estamos usando . En CQRS no siempre es obligatorio tener una propia jerarquía de comandos. En muchas ocasiones es suficiente con quedarnos donde nos hemos quedado ahora . Dos interfaces uno de consulta y otro de comandos de modificación . Ahora bien hay situaciones sobre todo aquellas situaciones muy ligadas a MicroServicios y BoundedContext en donde construir una jerarquía de comandos aporta. Para ello debemos crearnos un interface Comando y varias clases que lo implementen:



Veamos su código :

```
package com.arquitecturajava.cqrs3;

public interface Comando {

  public void procesar();
}
```

Cada una de las operaciones que antes se encontraban en el interface PersonaRepositoryWrite van a ser envueltas por un comando:

```
package com.arquitecturajava.cqrs3;

import java.util.List;

public class ComandoInsertar implements Comando {

  private Persona persona;
  private PersonaRepositoryWrite repositorio;
  public ComandoInsertar(Persona persona, List<Persona> lista) {
    repositorio= new PersonaRepositoryWriteMemoria(lista);
    this.persona=persona;
  }


  @Override
  public void procesar() {
    repositorio.insertar(persona);
  }
}
```

```
package com.arquitecturajava.cqrs3;

import java.util.List;

public class ComandoBorrar implements Comando {

  private String nombre;
  private PersonaRepositoryWrite repositorio;
  public ComandoBorrar(String nombre, List<Persona> lista) {
    repositorio= new PersonaRepositoryWriteMemoria(lista);
    this.nombre= nombre;
  }
```

```
  @Override
  public void procesar() {
    repositorio.borrar(new Persona(nombre));
  }
}
```

De esta manera la parte de escritura queda todavía mucho más organizada con una jerarquía de comandos concreta. Muchas personas no ven con claridad para qué añadir una jerarquía de comandos ya que aumenta la complejidad . Eso es cierto y podríamos haber llamado directamente desde nuestro controlador al repositorio de persistencia. Sin embargo muchas veces los comandos al hacer referencia a operaciones que son de modificación aportan una estructura para “extender” y añadir a futuro código adicional que arquitecturas más complejas necesiten. Por ejemplo podríamos necesitar que se genere un evento cada vez que se realiza una operación de insercion. Ese código se podría generar a nivel del comando en el método procesar:

```
@Override
public void procesar() {
  repositorio.insertar(persona);
  System.out.println("evento de insercion");
}
```

Conclusiones

Construir una Arquitectura basada en CQRS no es sencillo y supone añadir complejidad pero hay situaciones en las que puede ser muy necesaria. El enfoque que he implementado aquí es muy muy generalista luego cada uno tendrá que adaptarlo a sus frameworks . Pero he preferido simplemente explicar la división de responsabilidades. Por ejemplo en mi caso los comandos reciben una lista para mantener una lista de objetos en memoria. En una implementación real este parámetro sobraría ya que nos conectaríamos a una base de datos.

Otros artículos relacionados

1. ¿Qué es Spring Boot?
2. Spring @ComponentScan y configuración
3. ¿Qué es Spring WebFlux?
4. Curso Spring Boot

[/ihc-hide-content]