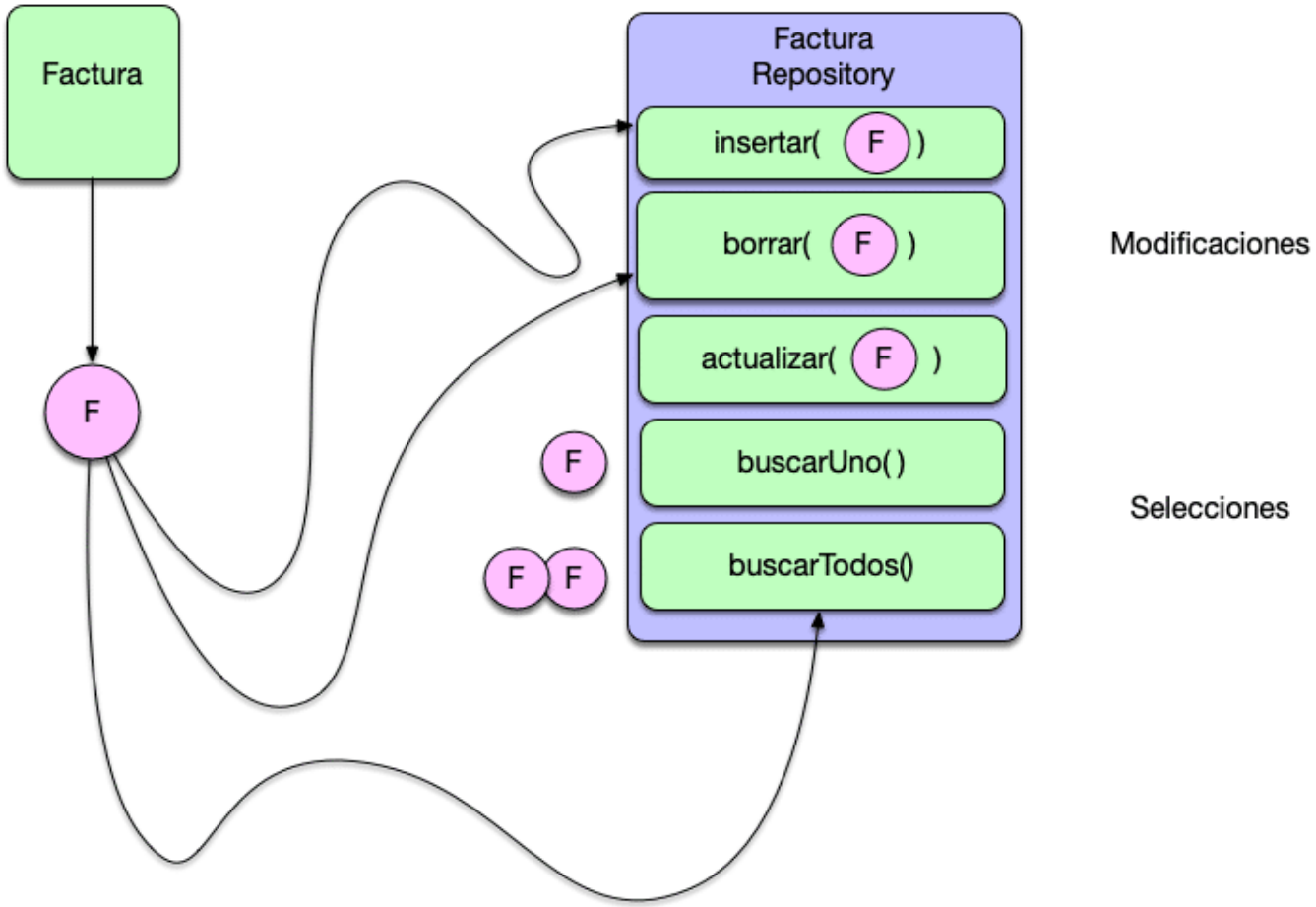
Tabla de Contenidos

El patron Repository es uno de los patrones más clásicos en Enterprise Design Pattern. Este patrón hace referencia a como persistir un objeto en una base de datos . Por el ejemplo si disponemos de la clase Factura como podemos persistir esta de forma sencilla . Para ello el patrón de diseño se encarga de agrupar todas las operaciones en una única clase que las gestiona.



- [ihc-hide-content ihc_mb_type=”show” ihc_mb_who=”4″ ihc_mb_template=”1″]
  Este patrón es bastante habitual construirlo con JPA , así que vamos a ver un ejemplo básico

de como abordarlo:

```
package com.arquitecturajava.rest;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;


import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class FacturaRepository {
        @PersistenceContext
        private EntityManager em;

        public List<Factura> buscarTodas() {

                return em.createQuery("select f from Factura f",
Factura.class).getResultList();
        }

        public Factura buscarUna(int numero) {

                return em.find(Factura.class, numero);
        }
        @Transactional
        public void borrar(Factura factura) {
```

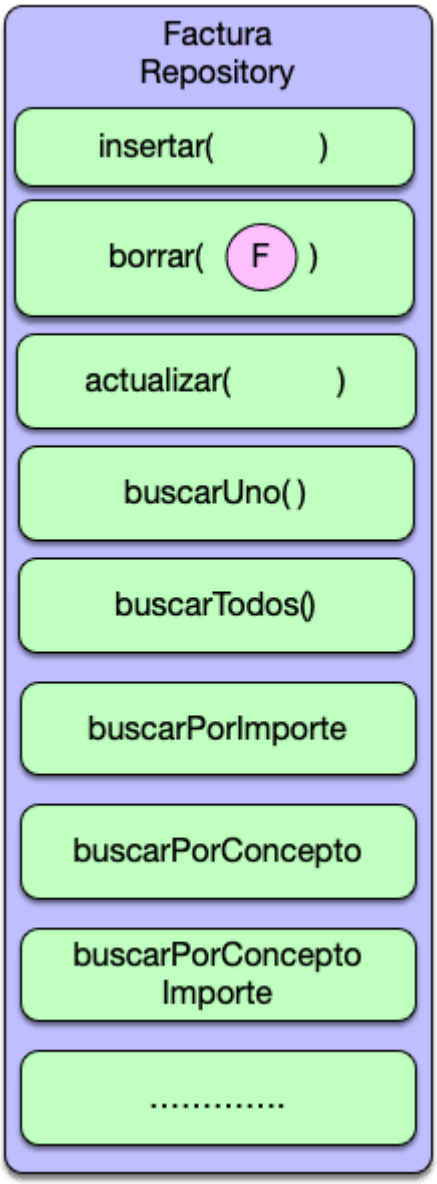
```
                em.remove(em.merge(factura));
        }
        @Transactional
        public void insertar(Factura factura) {

                em.persist(factura);
        }
        @Transactional
        public void actualizar(Factura factura) {

                em.merge(factura);
        }

}
```

Este código es bastante funcional y nos aborda todas las necesidades que nosotros tenemos a la hora de gestionar la persistencia elemental de una Factura. Ahora bien según vaya el programa evolucionando nos encontraremos con que los métodos de búsquedas se incrementan.

```
public List<Factura> buscarPorConcepto(String concepto) {

                TypedQuery<Factura> consulta = em.createQuery("select
f from Factura f where f.concepto= :concepto",
                                Factura.class);
                consulta.setParameter("concepto", concepto);
                return consulta.getResultList();
        }

        public List<Factura> buscarPorImporte(double importe) {

                TypedQuery<Factura> consulta = em.createQuery("select
```

```
f from Factura f where f.importe= :importe",
                                Factura.class);
                consulta.setParameter("importe", importe);
                return consulta.getResultList();
        }

        public List<Factura> buscarPorConceptoyImporte(String
concepto,double importe) {

                TypedQuery<Factura> consulta = em.createQuery("select
f from Factura f where f.importe= :importe and f.concepto= :concepto",
                                Factura.class);
                consulta.setParameter("concepto", concepto);
                consulta.setParameter("importe", importe);
                return consulta.getResultList();
        }
```

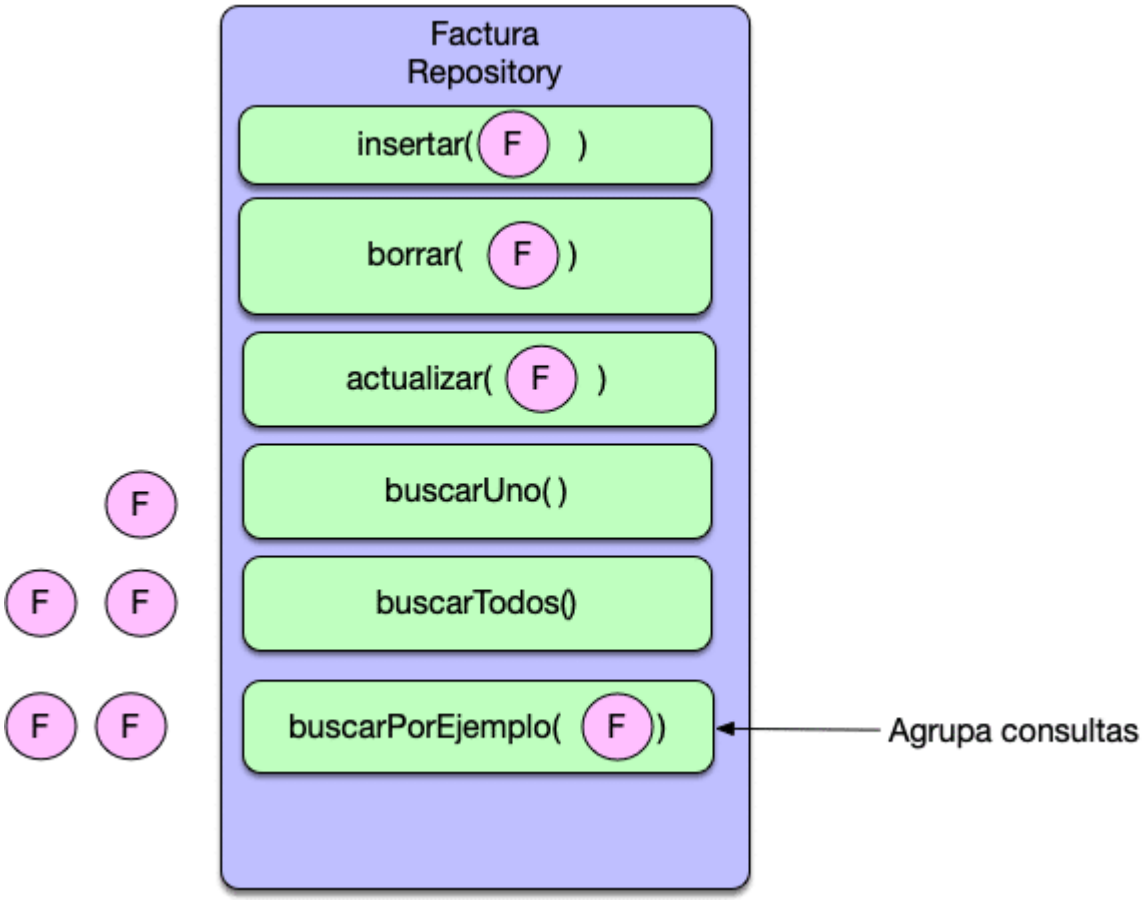
En este caso hemos añadido tres métodos diferentes uno que nos busca por concepto , otro que nos busca por importe y finalmente uno que busca por ambos campos. Todo es correcto , pero tenemos que empezar a darnos cuenta de que quizá no sea la mejor solución cuando el número de propiedades crezca .¿Porque? Porque a mayor número de propiedades mayor combinación de búsquedas aparece . Podemos buscar por fecha , por numero por fecha y numero por importe por importe y numero etc.



Esto es lo que se suele denominar la explosión de repositorios . Tenemos repositorios que disponen de un número de métodos muy muy amplio. Al final nos encontraremos con un problema ya que los repositorios serán cada vez más difíciles de manejar al tener el interface tantos y tantos métodos . Generando fragilidad a la hora de hacer cambios.

API de Criteria y Query By Example

Una solución a este problema es apoyarnos por ejemplo a nivel de JPA en el API de Criteria y hacer que todos estos métodos se agrupen en un único método que se denomine busquedaPorEjemplo (Factura f) y este método reciba una Factura como parámetro .



El método a nivel interno se encargará de decidir que propiedades disponen de valor no nulo y con ellas generar una consulta dinámica que nos devuelva el filtrado requerido.

```
public List<Factura> buscarPorEjemplo(Factura ejemplo) {

    CriteriaBuilder cb = em.getCriteriaBuilder();
```

```
                CriteriaQuery<Factura> cc =
cb.createQuery(Factura.class);
                // consulta raiz
                Root<Factura> facturas = cc.from(Factura.class);
                List<Predicate> predicados = new
ArrayList<Predicate>();

                if (ejemplo != null) {

                        // una instruccion de return con el filtrado
                        if (ejemplo.getNumero() != 0) {

predicados.add(cb.equal(facturas.get("numero"), ejemplo.getNumero()));

                        }

                        if (ejemplo.getConcepto() != null) {

predicados.add(cb.equal(facturas.get("concepto"),
ejemplo.getConcepto()));
                        }

                        if (ejemplo.getImporte() != 0) {

predicados.add(cb.equal(facturas.get("importe"),
ejemplo.getImporte()));
                        }
                }
                // convertir una lista en un array
                cc.select(facturas).where(predicados.toArray(new
Predicate[] {}));
```

```
            TypedQuery<Factura> consulta=em.createQuery(cc);
            return consulta.getResultList();
    }
```

Patron Repository Conclusiones

De esta manera conseguimos reducir el número de métodos del repository y dejar un interface mucho más compacto. Acabamos de ver un Ejemplo del patron Repository y como utilizar métodos orientados a ejemplos.

Otros artículos relacionados

- ¿Que es el patron BFF?
- El patrón de inyección de dependencia y su utilidad
- Usando el patron factory

[/ihc-hide-content]