Patrón Especificación

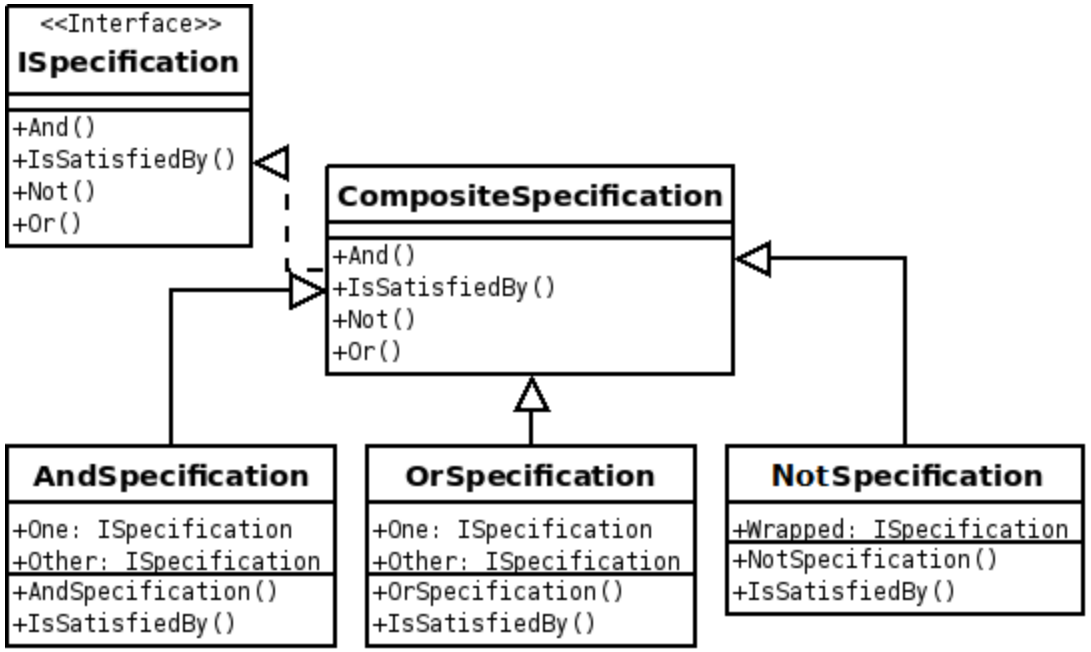
Patrón Especificación y Repositorio Java

Aplicaciones Distribuidas

Curso 2025/26

Introducción

- El patrón Especificación ofrece un mecanismo para encapsular reglas de negocio que se pueden combinar para establecer filtros de consulta o validaciones.
- Se suele usar con el patrón Repositrio en la metodología *Domain Driver Design*.



Patrón Especificación en Java

Definición basada en predicados (`java.util.function.Predicate`).

```
public class Especificacion<T> {
    private Predicate<T> reglas;
    public Especificacion(Predicate<T> inicial){
        reglas = inicial;
    }
    public Especificacion<T> and(Predicate<T> rule){
        reglas = reglas.and(rule);
        return this;
    }
    public Especificacion<T> or(Predicate<T> rule){
        reglas = reglas.or(rule);
        return this;
    }
    public Especificacion<T> not(Predicate<T> rule){
        reglas = reglas.and(rule).negate();
        return this;
    }
    public boolean isSatisfiedBy(T object){
        return reglas.test(object);
        }
    }
```

Reglas de negocio

```
public class ReglasEncuesta {

        public static Predicate<Encuesta> activas() {
                LocalDateTime ahora = LocalDateTime.now();
                return encuesta -> encuesta.getApertura().isBefore(ahora) &&
                                   encuesta.getCierre().isAfter(ahora);
        }

        public static Predicate<Encuesta> opcionesSuficientes() {
                return encuesta -> encuesta.getNumeroOpciones() > 1;
        }

        public static Predicate<Encuesta> sinVotos() {
                LocalDateTime ahora = LocalDateTime.now();
                return encuesta -> encuesta.getCierre().isAfter(ahora) &&
                                   encuesta.getOpciones().stream()
                .map(Opcion::getNumeroVotos)
                .reduce(0, Integer::sum) == 0;
        }
}
```

Definiendo una especificación

Ejemplo de especificación que combina dos reglas de negocio:

```
Especificacion<Encuesta> spec =
    new Especificacion<Encuesta>(
        ReglasEncuesta.opcionesSuficientes())
        .and(ReglasEncuesta.sinVotos());
```

Extensión del patrón repositorio

```
public interfce Repositorio <T, K> {
    // ...
    List<T> getAll() throws RepositorioException;

        List<K> getIds()throws RepositorioException;

    // Selección con el Patrón especificación
        List<T> getByEspecificacion(Especificacion<T> spec)
    throws RepositorioException;
}
```

Implementación en la interfaz genérica

La implementación de la operación `getByEspecificacion` se podría apoyar en el método `getAll` y quedar implementada en un método *default* de la interfaz `Repositorio`.

```
        default List<T> getByEspecificacion(Especificacion<T> spec)
        throws RepositorioException {
                    return getAll().stream()
                .filter(obj -> spec.isSatisfiedBy(obj))
                .collect(Collectors.toList());
        }
```

Inconvenientes de la implementación genérica

La implementación es **ineficiente** al estar apoyada en la consulta `getAll`.

Mejora:

- En cada uno de los sistemas de persistencia habría que **redefinir** ese método para aprovechar las capacidades de consulta del sistema de almacenamiento (ej. SQL).
- Por tanto, habría que transformar la especificación abstracta en sentencias del lenguaje de consulta concreto. Esta tarea es **costosa**.

Aproximación pragmática

Alternativa: Extender el repositorio genérico para cada entidad incluyendo métodos *ad hoc* de consulta.

```
public interface RepositorioAdHocEncuestas
    extends RepositorioString<Encuesta> {

    public List<Encuesta> getByActivas()
        throws RepositorioException;

    public List<Encuesta> getBySinVotos()
         throws RepositorioException;

    public List<Encuesta> getByNumeroOpcionesMayorQue(int numero)
         throws RepositorioException;

        // ...

}
```

Aproximación pragmática

La interfaz *ad hoc* del repositorio podría incluir una implementación por defecto de las operaciones basada en el método `getAll`.

```
public interface RepositorioAdHocEncuestas
    extends RepositorioString<Encuesta> {
//...
    public default List<Encuesta> getByActivas()
        throws RepositorioException {
                LocalDateTime ahora = LocalDateTime.now();
                return getAll().stream()
            .filter(encuesta -> encuesta.getApertura().isBefore(ahora) &&
                               encuesta.getCierre().isAfter(ahora))
                    .collect(Collectors.toList());
        }
```

De este modo, se ofrece una implementación operativa para cualquier repositorio dejando opcional la redefinición para mejorar la eficiencia.

Aproximación pragmática

La implementación concreta de la interfaz para un sistema de persistencia extiende la implementación genérica (por ejemplo RepositorioJSON) e implementa las operaciones:

```
public class RepositorioEncuestasJSON
    extends RepositorioJSON<Encuesta>
    implements RepositorioAdHocEncuestas {

    @Override
    public List<Encuesta> getByActivas()
        throws RepositorioException{
            // ... consulta optimizada para JSON (JSONPath)
    }

    public List<Encuesta> getBySinVotos()
         throws RepositorioException{
            // ...
    }
        // ...
}
```

Aproximación pragmática

Con la definición *ad hoc* del repositorio, al obtener la implementación de la factoría tendríamos que asignarlo al repositorio ad hoc de la entidad (interfaz `RepositorioAdHocEncuestas`).

```
RepositorioAdHocEncuestas repositorioConcreto =
    FactoriaRepositorios.getRepositorio(Encuesta.class);
List<Encuesta> resultado = repositorioConcreto.getByActivas();
```

En el fichero `repositorio.properties` se configura:

```
encuestas.modelo.Encuesta=encuestas.repositorio.RepositorioEncuestasJSON
```