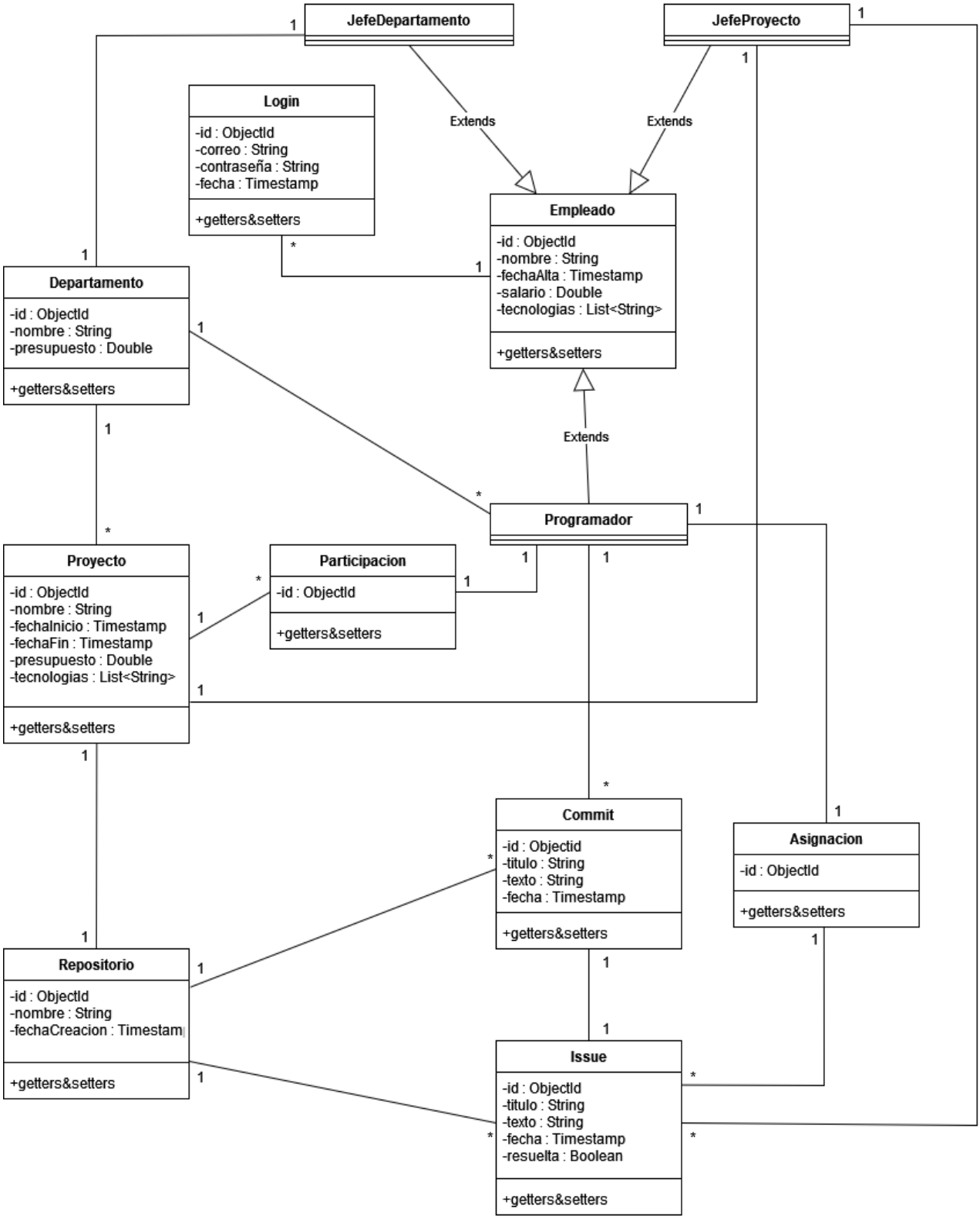
PRÁCTICA ACCESS DATA NO SQL

ACCESO A DATOS

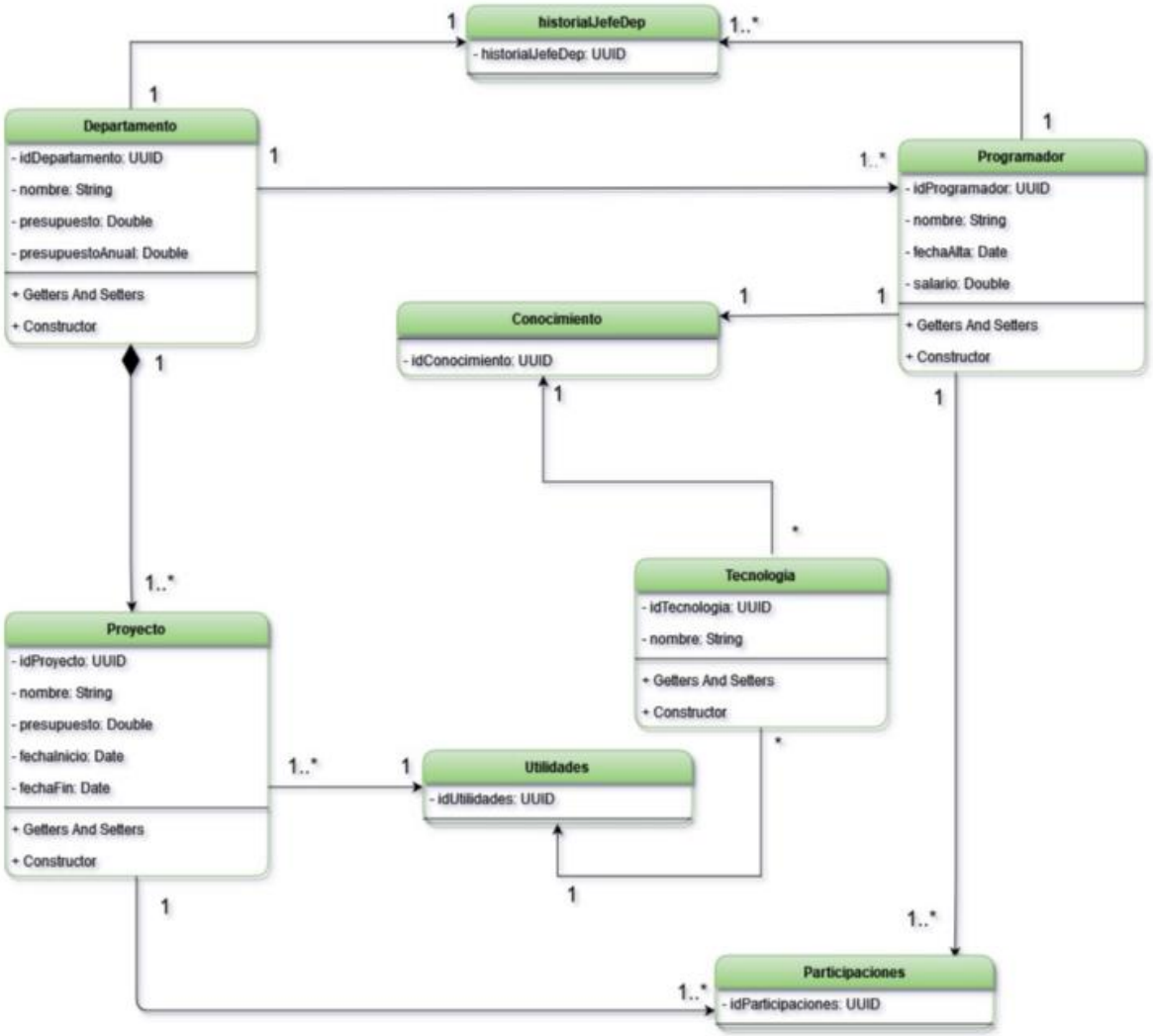
DYLAN HURTADO LÓPEZ & EMILIO LÓPEZ NOVILLO

- Diagrama de clases:



- **Explicación E/R**

Tomamos como base el diagrama entidad relación de la práctica anterior. En el cual encontramos un problema a la hora de relacionar con JPA las entidades.



Este problema consistía en que JPA no puede tener varios atributos de una misma Entidad dentro de una Entidad esto pasaba en Departamento con Programador. Programador era una única Entidad que se usaba para los jefes también por lo que en Departamento teníamos varios atributos de tipo Programador.

La solución que se nos ocurrió fue crear una clase padre Empleado de la que heredarán los programadores y los jefes para así diferenciarlos con JPA y hacer uso de la herencia de una forma óptima, teniendo en Empleado los atributos comunes de los programadores y los jefes. Esto planteó otro problema con las entidades de JPA para referenciar al id de la entidad. Conseguimos solucionarlo mediante el uso de la anotación @MapperSupperClass.

Otro problema que encontramos fue al intentar almacenar entidades embebidas para hacer menos llamadas a la base de datos. El error consiste en que mongo no deja embeber una entidad que tenga id para almacenarla. Nosotros teníamos la intención de embeber JefeDepartamento en Departamento ya que solo se usa al buscar información sobre Departamento. Pero debido a este error pensamos que quitar los id a los jefes no sería una opción correcta y decidimos que es mejor almacenarlos en colecciones diferentes. Para embeberlos posteriormente en los DTO.

- ## Cardinalidad:

Departamento contendrá un JefeDepartamento, varios Programadores y varios Proyectos.

Proyecto contendrá un Departamento, un JefeProyecto, varias Participaciones y un Repositorio.

Programador contendrá un Departamento, una Asignación, una Participación y varios Commits.

JefeDepartamento contendrá un Departamento.

JefeProyecto contendrá un Proyecto.

Repositorio contendrá un Proyecto, varios commits e issues.

Commit contendrá un Programador, Repositorio e Issue.

Issue contendrá un Repositorio, JefeProyecto, Asignación y Commit.

Para romper las relaciones ManyToMany creamos las tablas Participación e Asignación.

Login contendrá un Empleado y un Empleado contendrá varios Login.

- ## Navegabilidad

En cuanto a la navegabilidad de nuestro diagrama, tenemos bidireccionalidad en todas las entidades ya que aunque no se identifique en el diagrama JPA lo hará por nosotros. Nosotros solo tendremos que limitar la bidireccionalidad en los toString de las diferentes Entidades para no entrar en bucles infinitos.

Por lo tanto tenemos bidireccionalidad.

De estas listas en la base de datos no se almacenan columnas(@JoinColumn). Aunque eso no quita que sean accesibles desde el código.

- ## Cascada

La cascada es utilizada para en caso de que se borre, se modifique o se inserte una entidad. JPA maneja las relaciones en base a eso. Así conseguimos que las entidades que contienen una lista que está relacionada con otra entidad, no se quede desactualizada.

- ## Fetch:

Para evitar problemas a la hora de manejar los objetos que sacamos de la base de datos utilizamos EAGER. Para nuestro caso de uso EAGER es lo más óptimo pero es más pesado que LAZY así que no siempre hay que usar EAGER. EAGER carga todos los atributos del objeto al sacarlo de la base de datos y LAZY lo saca con atributos sin inicializar. Por eso hay que tener en cuenta que casos de uso tenemos y cuál nos conviene usar.

- ## Análisis del Modelo de Negocio y Funcionamiento

Analizando el modelo de negocio y su funcionamiento en profundidad nos encontramos con diferentes funcionalidades y servicios:

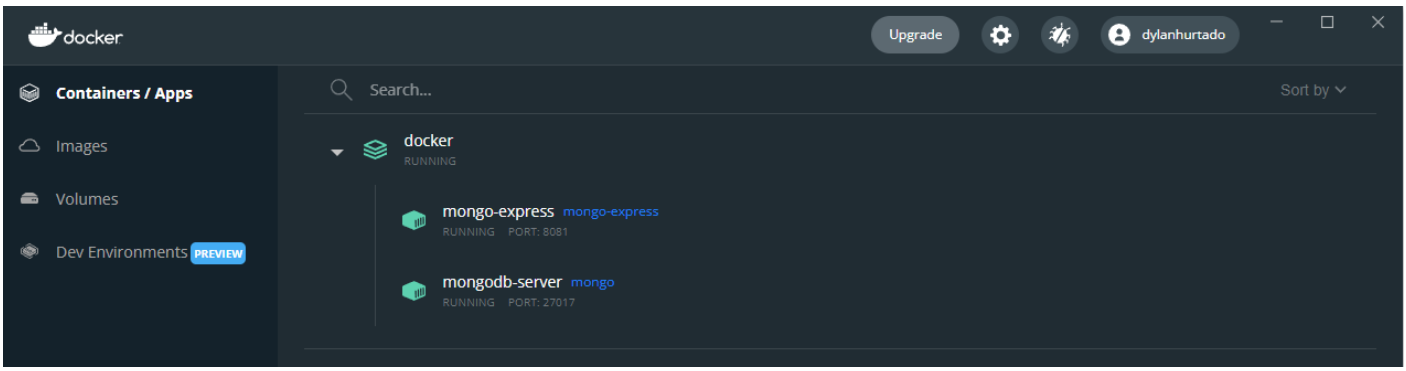
El primero en esta práctica es una base de datos no relacional o noSQL que se llama MongoDB para acceder y manejar los distintos datos pedidos.

Dicha base de datos se lanza a través de contenedor docker-compose el cual contiene la base de datos MongoDB y Mongo Express que se abre en el localhost en el puerto 8081 de esta manera tenemos fácil acceso a través de cualquier navegador web.

: > Escritorio > AD-Practica04-AccessData-noSQL > docker > ! mongodb.yml

```yaml
# Mis servicios
# Iniciamos los servicios
services:
  # MONGO DB
  mongodb-server:
    image: mongo
    container_name: mongodb-server
    ports:
      - 27017:27017
    expose:
      - 27017 #Puerto de mongo
    environment:
      # Puedes cambiar los datos que quieras
      MONGO_INITDB_ROOT_USERNAME: mongoadmin
      MONGO_INITDB_ROOT_PASSWORD: mongopass
      MONGO_INITDB_DATABASE: mongodb
      # ME_CONFIG_MONGODB_URL: mongodb://root:example@mongo:27017/
      command: --auth

    volumes:
      - mongodb-volume:/data/db
      #Init db automaticamente
      - ./init:/docker-entrypoint-initdb.d

    networks:
      - mongo-network
    # restart: always

  # MONGO EXPRESS
  mongo-express:
    image: mongo-express
    container_name: mongo-express
    # restart: always
    ports:
      - 8081:8081
    networks:
      - mongo-network
    depends_on:
      - mongodb-server

    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: mongoadmin
      ME_CONFIG_MONGODB_ADMINPASSWORD: mongopass
      ME_CONFIG_MONGODB_SERVER: mongodb-server
    restart: unless-stopped

# Mi volumenes de datos compartidos
volumes:
  mongodb-volume:

# Si queremos que tengan una red propia a otros contenedores
networks:
  mongo-network:
    driver: bridge
```

El cual nos permite comprobar, modificar y visualizar la base de datos y sus diferentes documentos.



Una vez que se ha levantado el docker y tenemos la base de datos lista. Se inicia el programa y se ejecuta el método initDatabase() que está situado en Facade.java , el cual se encarga de meter los datos de ejemplo a la base de datos.

Lo hace apoyándose en una clase llamada HibernateController.java . Esta clase es la mediadora entre la base de datos y las entidades de JPA, la que usamos para crear el CRUD de los pojos gestionados con JPA .

Cada vez que se ejecuta el programa este método llamado initDatabase() llama a remove Database() al principio y así conseguimos que en cada ejecución se inicializan los mismos datos.
Seguidamente le pide al usuario por consola ,a través de un scanner de Java, que introduzca un email y seguidamente una contraseña si esta es incorrecta. Se vuelve a solicitar, en caso contrario, se pasa a un menú por consola en el que puedes elegir varias opciones a través de números.

```
    ---   MENU    ---
1.- Información departamento completo.
2.- Lista de issues abiertas por proyecto.
3.- Programador por proyecto ordenados por n° commits.
4.- Programadores y su productividad completa.
5.- Los 3 proyectos más caros y salarios.
6.- Proyectos con información completa.
7.- Login completos y ordenados por programador.
8.- Salir.
    ELIGA UNA OPCIÓN:
8
Saliste con éxito.
```

En función del número que ponga hará unas operaciones u otras y devolverá el resultado en JSON. Si se introduce un número fuera de las opciones (ejemplo el 10). Te dirá que la opción no es correcta y mostrará el menú de nuevo.

En el paquete repositorio encontramos una interfaz llamada CrudRepository la cual sirve de base para todas las demás clases esta clase sirve para estandarizar unos métodos y tener más ordenadas las clases.

```java
package repository;

import ...

public interface CrudRepository<T, ID> {

    // Operaciones CRUD

    // Obtiene todos
    Optional<List<T>> getAll() throws SQLException;

    // Obtiene por ID
    Optional<T> getById(ID id) throws SQLException;

    // Salva
    Optional<T> save(T t) throws SQLException;

    // Actualiza
    Optional<T> update(T t) throws SQLException;

    // Elimina
    Optional<T> delete(T t) throws SQLException;

}
```

En dichas clases se realizan unas operaciones denominadas **CRUD (Create, Read, Update, Delete)**. Las cuales en pocas palabras son las operaciones básicas, específicamente en el programa se encuentran:

- **GetAll():** obtienes todos los objetos referidos a la misma clase.
- **GetById(id):** obtienes el objeto coincidente con el atributo denominado id. Buscándolo y comparándolo a través del parámetro que se otorga en dicha función.
- **Save(Clase objeto):** Método que sirve para insertar los datos provenientes de un objeto a la base de datos.
- **Update(Clase objeto):** Método que sirve para modificar un documento de esa colección en la base de datos a través de los datos incluidos en el objeto pasado por parámetro.
- **Delete(Clase objeto):** Método que sirve para borrar un documento de esa colección en la base datos a través de los datos incluidos en el objeto pasado por parámetro.

Dichas operaciones se llaman en una clase abstracta BaseService que servirá para las demás clases en el paquete services.
En las clases de services se hace una inyección de dependencias en el constructor. Lo que quiere decir que el servicio necesitará al repositorio.

Inyección de dependencias:

```
public class CommitService extends BaseService<Commit, Long, RepoCommit> {


    CommitMapper mapper = new CommitMapper();

    // Inyección de dependencias en el constructor. El servicio necesita este repositorio
    public CommitService(RepoCommit repository) { super(repository); }
```

Se crea e inicializa un objeto de la clase del paquete mapper.
El cual tiene dos métodos **fromDTO** y **toDTO**. En la clase de service en este caso se utiliza dentro de cada operación CRUD el mapper con el método toDTO que se encarga de mapear un objeto de una clase a un objeto de la claseDTO. Y esto sirve para no tocar directamente con los objetos que se comunican con la base de datos y también para poder mapear como quieras la salida.

En resumen el mapper pasa una clase de modelo del paquete dao a una clase dto como se ve en ejemplo de esta clase mapper:

```java
public class IssueMapper extends BaseMapper<Issue, IssueDTO> {
    @Override
    public Issue fromDTO(IssueDTO item) {
        return new Issue(item.getId(),item.getTexto(),
                item.getTexto(),item.getFecha(),item.getResuelta(),
                item.getJefe(),item.getProgramadores(),
                item.getRepositorio(),item.getCommit());
    }

    @Override
    public IssueDTO toDTO(Issue item) {
        return IssueDTO.builder()
                .id(item.getId())
                .titulo(item.getTexto())
                .texto(item.getTexto())
                .fecha(item.getFecha())
                .resuelta(item.getResuelta())
                .programadores(item.getProgramadores())
                .jefe(item.getJefe())
                .commit(item.getCommit())
                .repositorio(item.getRepositorio())
                .build();
    }
```

Una vez mapeados los objetos de la operación CRUD utilizada se envía a una serie de clases las cuales ya trabajan con las clases DTO. Estas clases Controllers dan una salida en JSON, ayudados por los métodos toString() que tienen las clases DTO. Los cuales son importantes para delimitar las llamadas recursivas.

Ejemplo de toString():

```java
@Override
public String toString(){
    return "Departamento{id="+this.id
            +", nombre="+this.nombre
            +", jefe="+this.jefeDepartamento
            +", presupuesto="+this.presupuesto
            +", presupuesto_anual="+this.presupuestoAnual
            +", proyectos_desarrollo="+proyDesarrollo.stream().map(Proyecto::getId).collect(Collectors.toList())
            +", proyectos_finalizados="+proyFinalizados.stream().map(Proyecto::getId).collect(Collectors.toList())
            +", programadores="+ programadores.stream().map(Programador::getId).collect(Collectors.toList())
            +"}";
}
```

Y por último para crear las opciones del menú mencionadas anteriormente. Hacemos uso de las clases Controller y con ayuda de la API Stream de Java sacamos el resultado deseado tal cual lo plantea el enunciado.

Ejemplo de código de consulta:

```
private void proyectosMasCaros() {
    System.out.println("Los 3 proyectos mas caros y el salario de sus programadores.");
    ProyectoController controller = ProyectoController.getInstance();
    controller.getAllProyectos().stream().sorted(Comparator.comparing(ProyectoDTO::getPresupuesto))
            .limit(3).collect(Collectors.toList()).forEach(p -> System.out.println(p.programadoresSalario()));
}
```

En cuanto al login mencionado al principio cuando se introduce el email y la contraseña. Esta contraseña es cifrada con SHA-256(*SHA-256* es un hash de 64 dígitos hexadecimales)

Clase Cifrador:

```
public class Cifrador {
    public String toSHA256(String password) {
        MessageDigest md = null;
        try {
            md = MessageDigest.getInstance("SHA-256");
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }

        byte[] hash = md.digest(password.getBytes());
        StringBuffer sb = new StringBuffer();

        for (byte b : hash) {
            sb.append(String.format("%02x", b));
        }

        return sb.toString();
    }

}
```

Se comparan los datos con los de los programadores. Y en el caso de que coincida con la de alguno. Se guarda en la base de datos la sesión y el usuario accedería al menú.

Patrones

Singleton:

Hemos utilizado más de una vez este patrón para asegurarnos de que una clase tenga una única instancia a la vez a pesar de que esta proporcione un punto de acceso global.

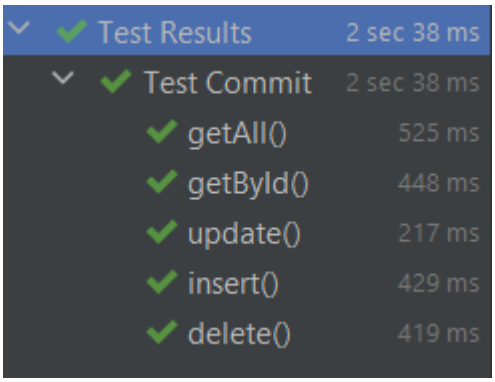
Facade:

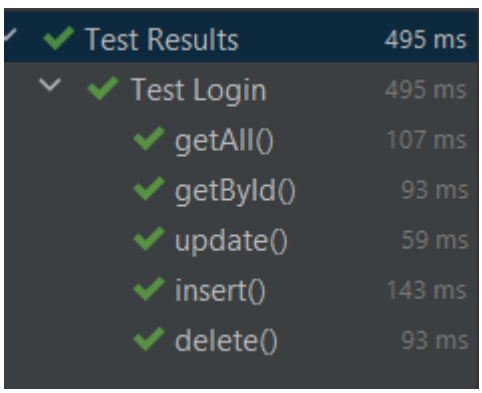
El propósito principal del uso de este patrón es organizar la información de cara a uso de la misma en el Main (app). Desde esta hacemos una especie de interfaz o biblioteca con todos los métodos finales del acceso a los datos para mostrarlos al usuario.

- **Test:**

A la hora de hacer los test se presentaron diferentes problemas debido a métodos y atributos static de clases inferiores las cuales mockito no puede simular. Intentamos solucionar estos errores con diferentes dependencias de mockito que soportaban el uso de static, pero seguía dando el mismo problema por lo que decidimos realizar test de integración a partir de los controladores. Si hubiéramos conseguido implementar mockito correctamente hubieramos podido probar más y diferentes casuísticas.

A la hora de ejecutar los test se ejecutan de manera correcta pero encontramos un error en algunas clases que no entendemos el motivo por el que aparece.
Los test que se ejecutan bien al 100% son Commit y Login. Commit con Hibernate y Login sin Hibernate.





En el resto de clases encontramos el siguiente error a pesar de tener la misma estructura de test que Commit.

```
org.opentest4j.AssertionFailedError: expected: dto.DepartamentoDTO@5e763c07<Departamento{id=2, n
Expected :Departamento{id=2, nombre=dep2, jefe=null, presupuesto=50000.0, presupuesto_anual=1000
Actual   :Departamento{id=2, nombre=dep2, jefe=null, presupuesto=50000.0, presupuesto_anual=1000
<Click to see difference>

 <5 internal lines>
    at test_integracion.DepartamentoTest.delete(DepartamentoTest.java:93) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1541) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1541) <23 internal lines>
```

Al parecer nos indica que los objetos son diferentes pero analizamos los toString que se muestran por pantalla y son iguales hasta el último atributo.

Si entramos en Click to see difference:

Podemos apreciar que también nos indica que el contenido de ambos objetos es el mismo y que no tiene diferencias.
Analizando más el error mostrado por pantalla, parece indicarnos que la posición de memoria de ambos objetos son distintas y obviamente para comparar dos objetos tienen que tener diferentes posiciones de memoria sino no tendría sentido hacer ese test.

La conclusión es que los test están bien pero desconocemos el motivo de este error y porque sucede.