DP1 2020-2021

Documento de Diseño del Sistema

Azulejos Gresur

https://github.com/gii-is-DP1/dp1-2020-g3-12

Miembros:

- Atienza Carretero, Carlos
- Galera Barrera, Tomás
- Pérez Ruiz, Lucas
- Rondán Domínguez, Borja
- Santisteban Corchazos, Alejandro

Tutor: Carlos Guillermo Müller Cejas

GRUPO G3-12

Versión <1>

<20/12/2020>

Historial de versiones

Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto

Fecha	Versión	Descripción de los cambios	Sprint
20/12/2020	V1	• Creación del documento	2

Azulejos Gresur
Grupo: G3-12

Contents

Azulejos Gresur
Grupo: G3-12

Introducción

La empresa con la que vamos a realizar nuestro proyecto de DP1, se llama Azulejos Gresur y está situada en la Sierra de Cádiz, concretamente en la localidad de El Gastor. Se dedica principalmente a la venta de materiales de construcción, ya sea a particulares, PYMEs o profesionales y realiza las compras de sus materiales al por mayor. También presta servicios relacionados con los vehículos que tiene, por ejemplo, transporte de sílices, material pesado, alquiler de la grúa por horas…

Al ser una empresa creada en 1997, no hace uso de un sistema informático actualizado. Por ello, y tras reunirnos con el propietario, decidimos realizar un nuevo sistema ajustándose a las nuevas necesidades del propietario.

En primer lugar, el sistema incluirá un sistema de login para cada empleado, según su rol en la empresa, lo cual le permitirá tener acceso o no a ciertas partes del sistema.

En segundo lugar, el sistema a desarrollar contendrá un apartado de facturación, en el cual se incluirá una sección para la generación de facturas (en PDF), otra para ingresos y gastos del negocio y una última para los proveedores a los cuales se les hace la compra de productos.

En tercer lugar, también incluirá, un apartado referente a los empleados que trabajan en la empresa. Como mencionamos antes, gracias al sistema de login, cada empleado podrá ver en esta sección todo lo referente a su información personal, nómina y su ocupación dentro de la empresa. Cabe recordar, que el administrador del negocio tendrá acceso a esta sección de modo que podrá ver y modificar cualquier dato referente de cada empleado.

En cuanto a la logística dividiremos el sistema en 2 apartados:

- **Transporte**: En esta sección, según tu cargo que ejerzas en la empresa, dispondrás de una información u otra. Es decir, el administrador podrá ver toda la información relativa al transporte. Esto incluye, todos los vehículos de la empresa, junto a su límite de carga, los conductores asociados a cada vehículo, los seguros e ITV de los vehículos, los transportes que hay que hacer cada día y verificar si algún vehículo sobrepasa el límite de carga (podrá poner un aviso de advertencia en otro apartado que se explicará posteriormente). Finalmente, los transportistas podrán ver su vehículo asociado, junto a las entregas a realizar ese mismo día y futuros…

- **Stock**: Esta sección será controlada por el administrador y el encargado de almacén. En ella, se podrá comprobar la disponibilidad de la empresa para cualquier producto. Y si se diese el caso de entrar en el stock de seguridad, poner un anuncio pertinente avisando de ello. Además, también podrá comprobar los productos perecederos y avisar de ello (silicona, pintura, pegamento…)

Finalmente, contendrá una sección relacionada con el mantenimiento en la cual, tanto administrador como cualquier trabajador, podrá poner avisos referentes a algo ocurrido, incidencias, informes del día… de modo que otra persona sea informada de cualquier cambio realizado y su posible repercusión en su labor…

El objetivo de este sistema será el de facilitar al propietario ciertas tareas, que actualmente son hechas de forma más rudimentaria y con aplicaciones obsoletas, pudiendo gracias al nuevo sistema hacerlas de manera más eficiente, ahorrando así tiempo.

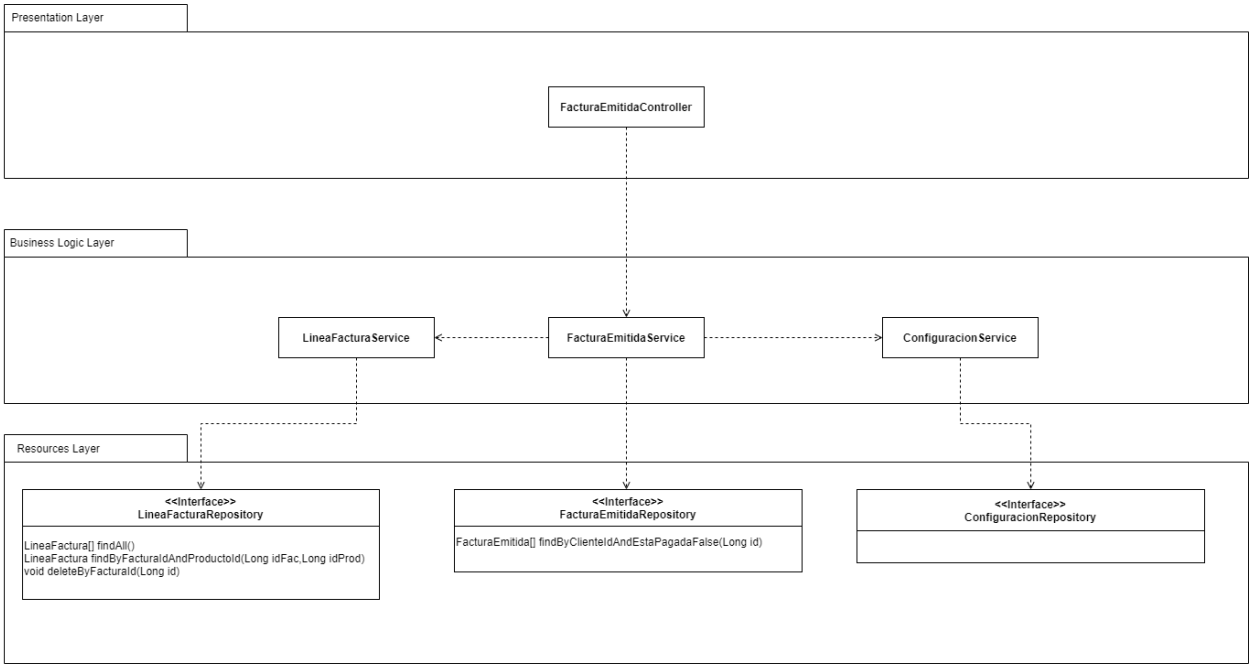
Diagrama(s) UML:

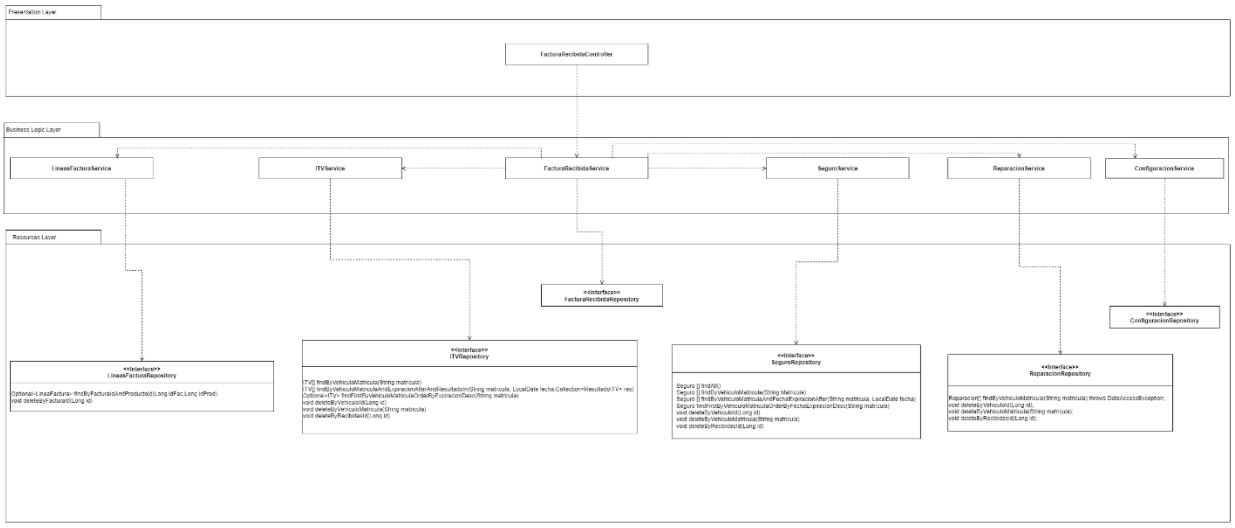
Diagrama de Dominio/Diseño

Ver diagramas de dominio en el Entregable 1.

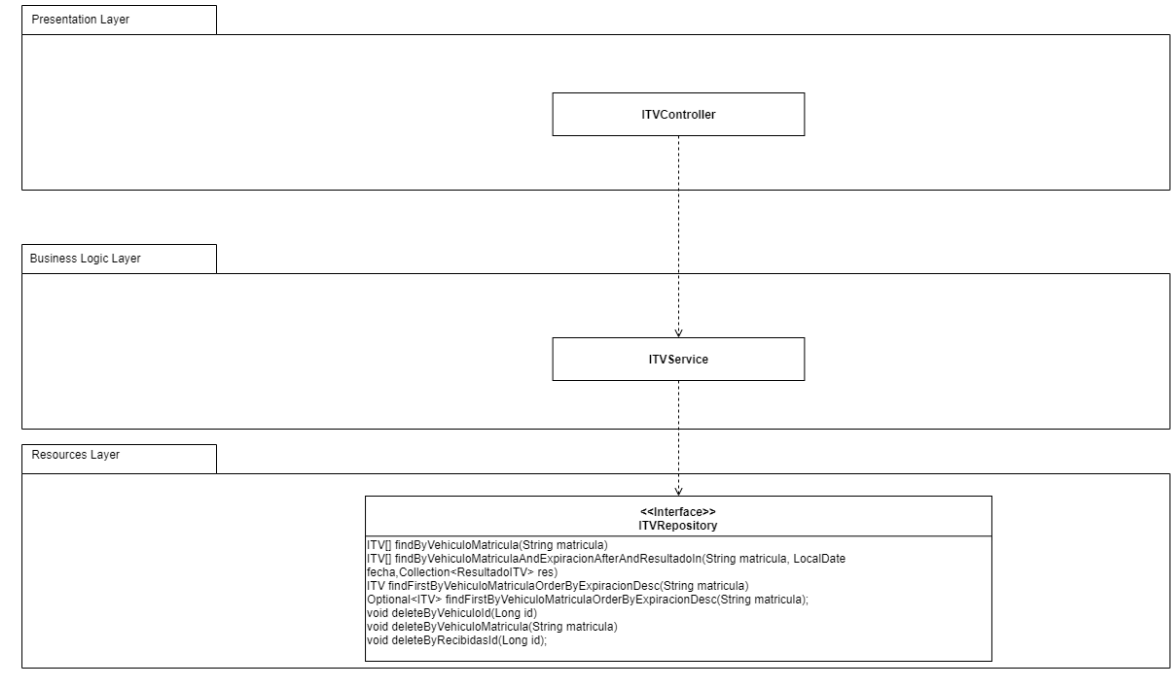
Azulejos Gresur
Grupo: G3-12

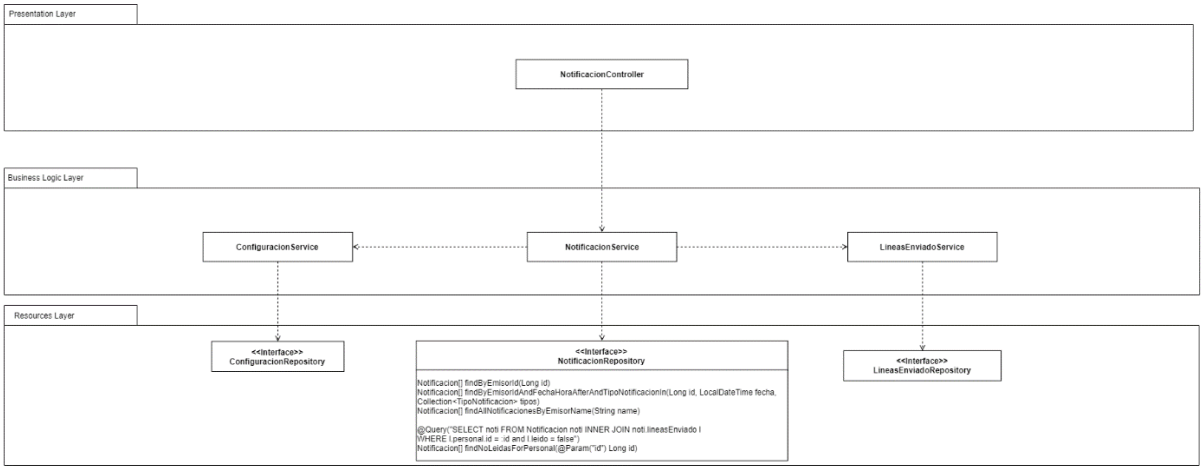
Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

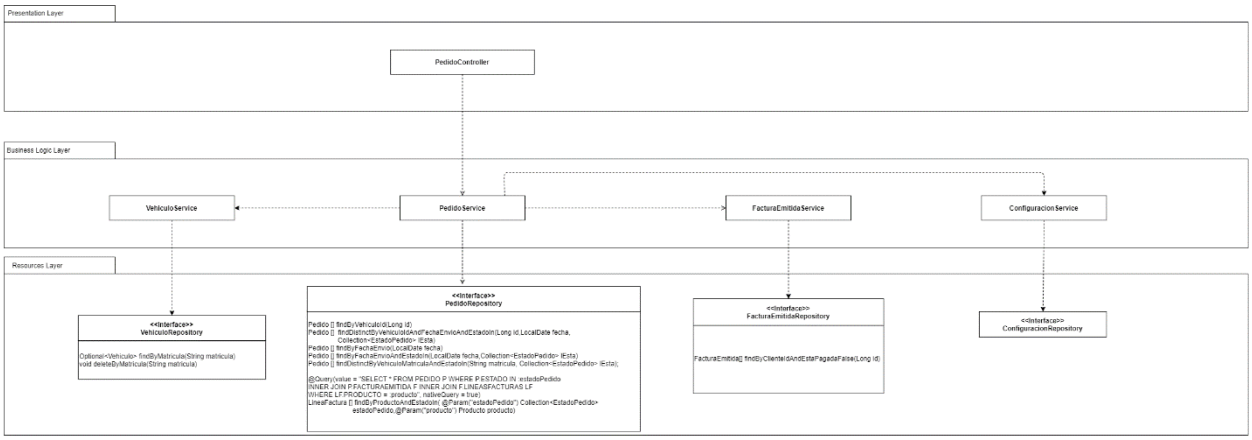




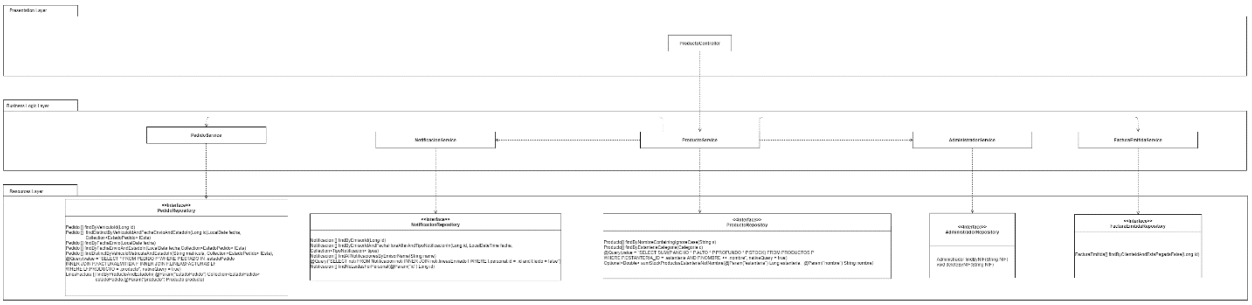
Azulejos Gresur
Grupo: G3-12

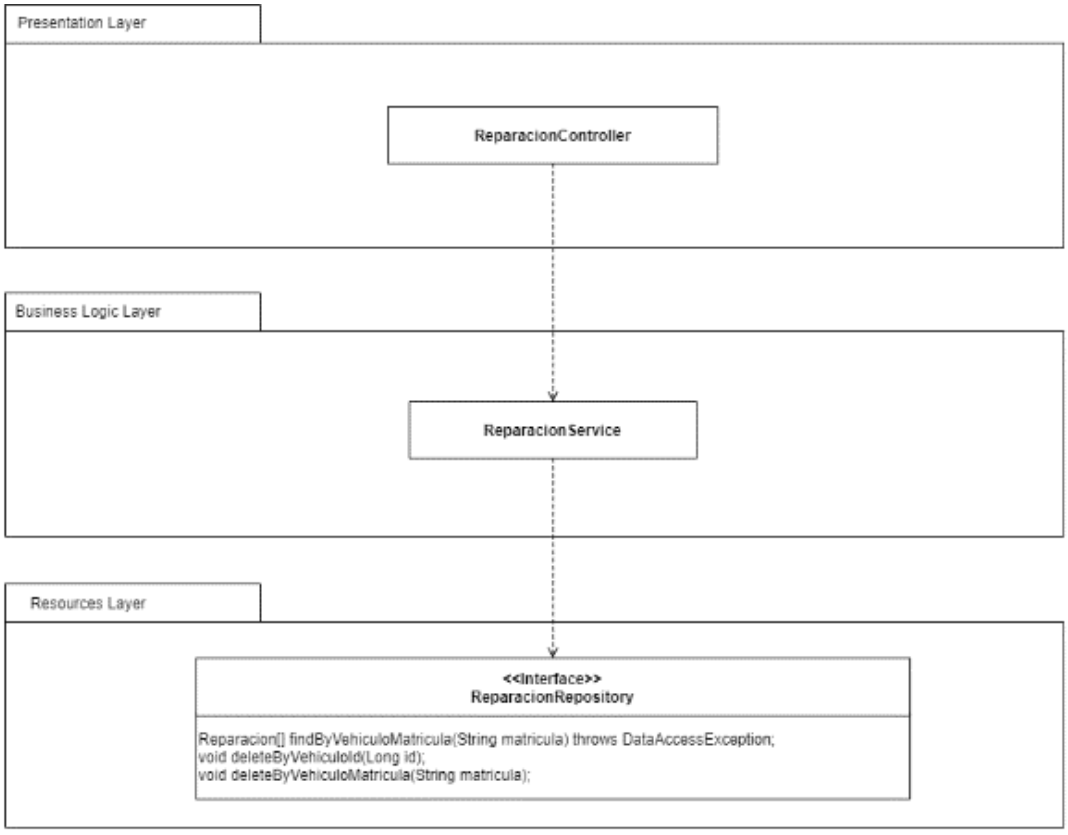


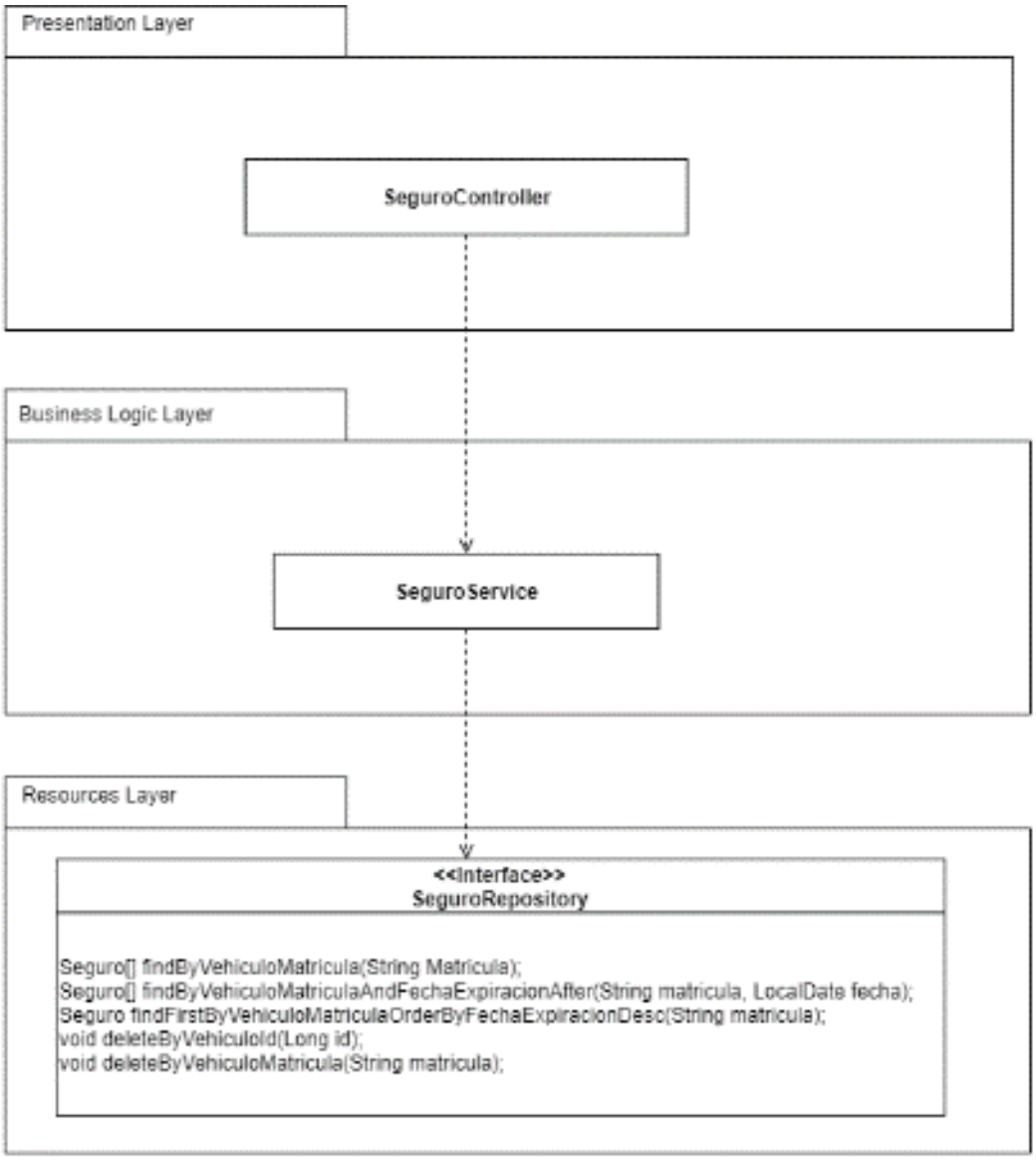


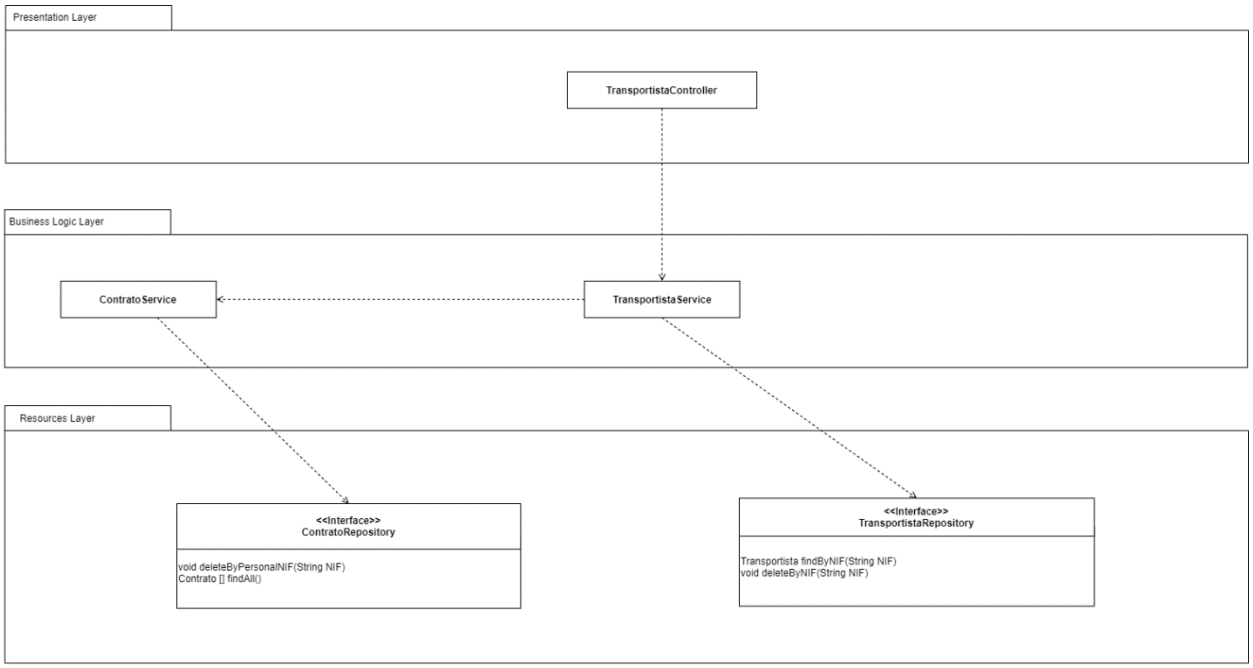


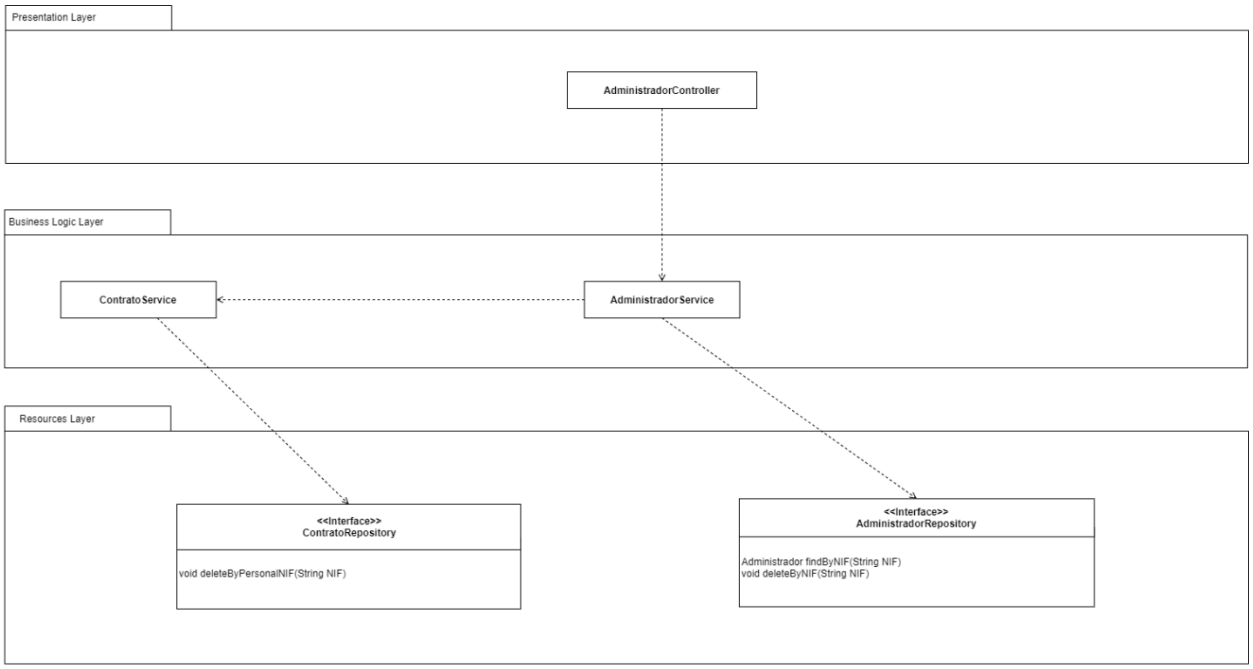
Azulejos Gresur
Grupo: G3-12

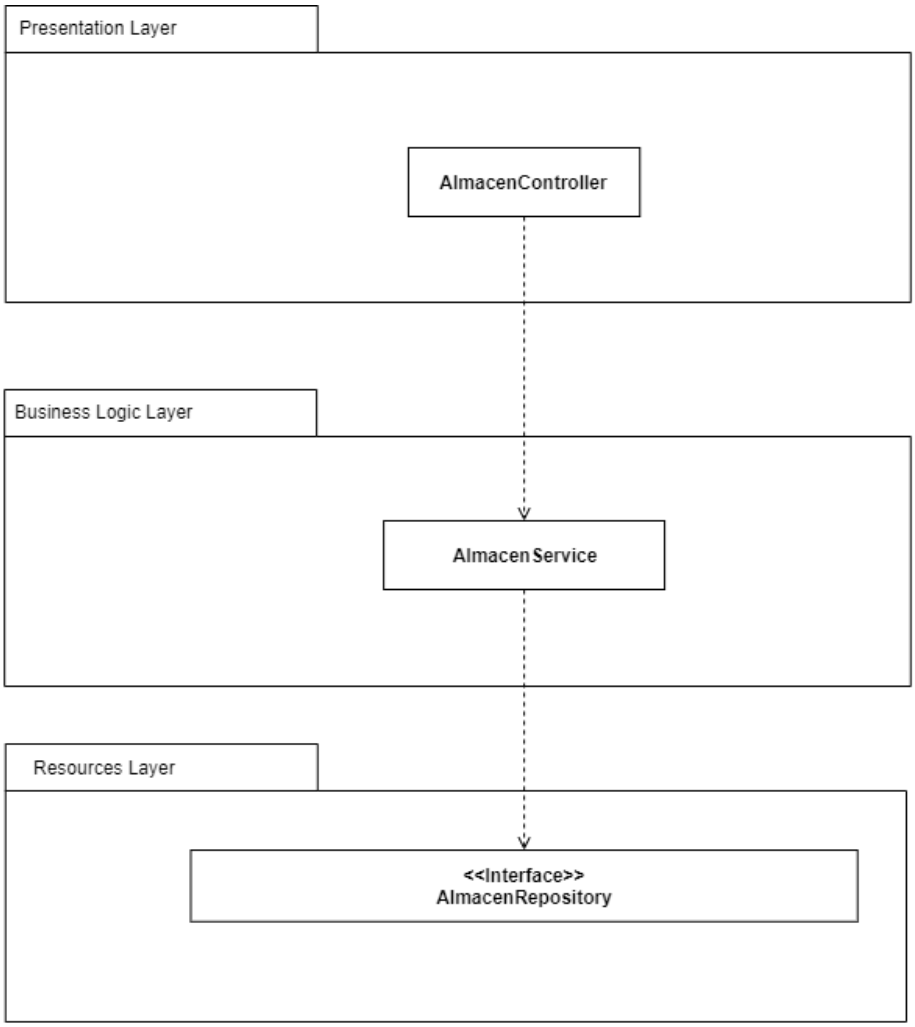




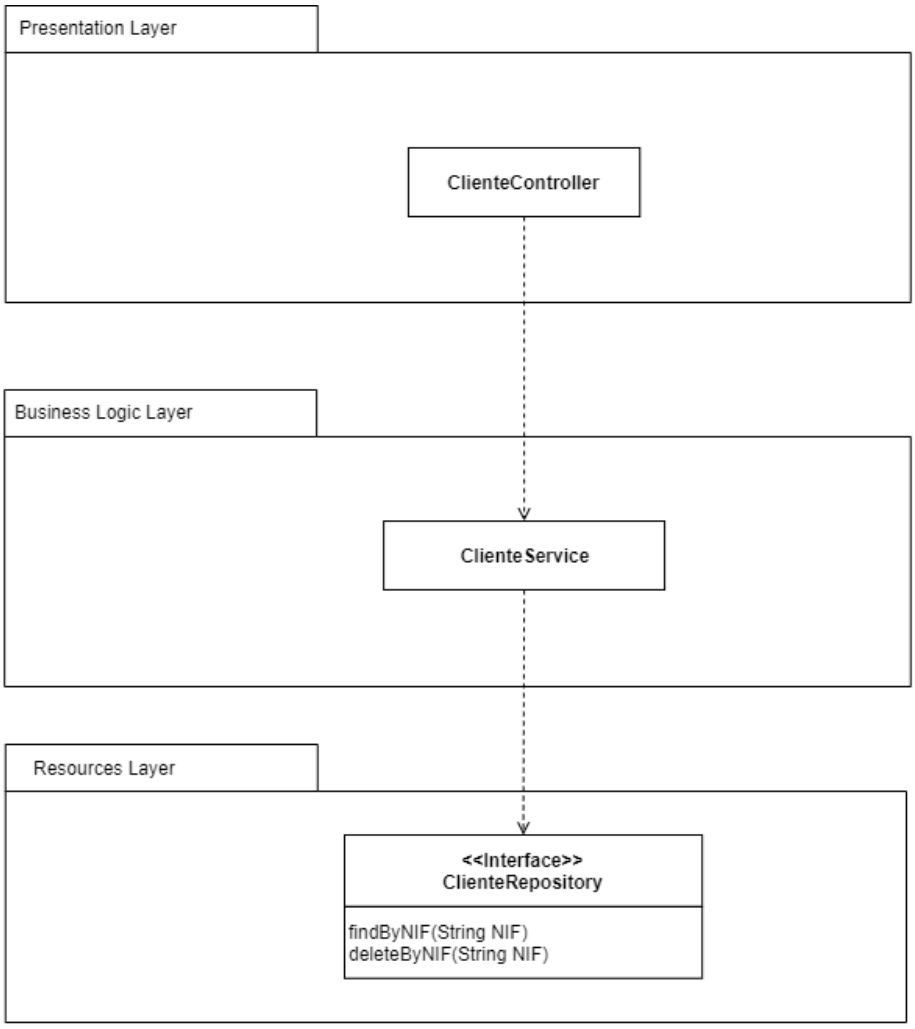

Presentation Layer
SeguroController
Business Logic Layer
SeguroService
Resources Layer
<<Interface>>
SeguroRepository
Seguro[] findByVehiculoMatricula(String Matricula);
Seguro[] findByVehiculoMatriculaAndFechaExpiracionAfter(String matricula, LocalDate fecha);
Seguro findFirstByVehiculoMatriculaOrderByFechaExpiracionDesc(String matricula);
void deleteByVehiculoId(Long id);
void deleteByVehiculoMatricula(String matricula);

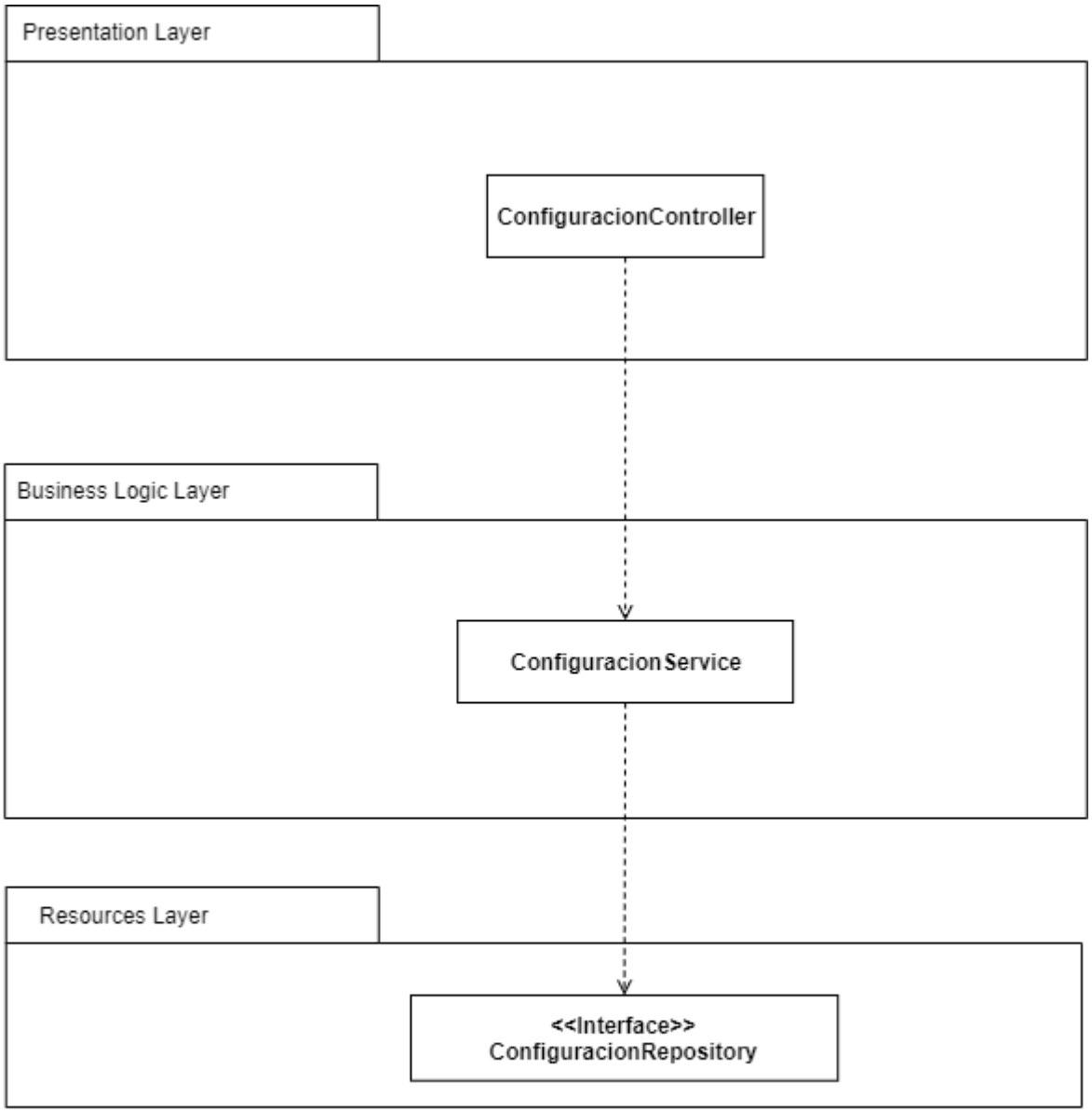

Presentation Layer
TransportistaController
Business Logic Layer
ContratoService
TransportistaService
Resources Layer
<<Interface>>
ContratoRepository
void deleteByPersonalNIF(String NIF)
Contrato [] findAll()
<<Interface>>
TransportistaRepository
Transportista findByNIF(String NIF)
void deleteByNIF(String NIF)

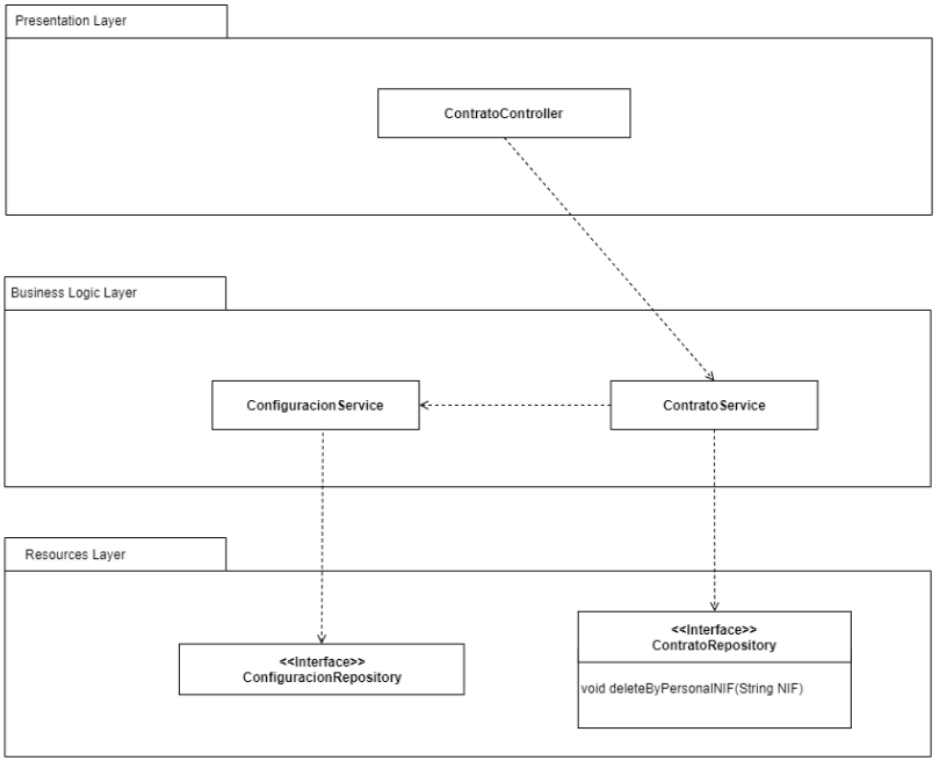

Presentation Layer
AdministradorController
Business Logic Layer
ContratoService
AdministradorService
Resources Layer
<<Interface>>
ContratoRepository
void deleteByPersonalNIF(String NIF)
<<Interface>>
AdministradorRepository
Administrador findByNIF(String NIF)
void deleteByNIF(String NIF)

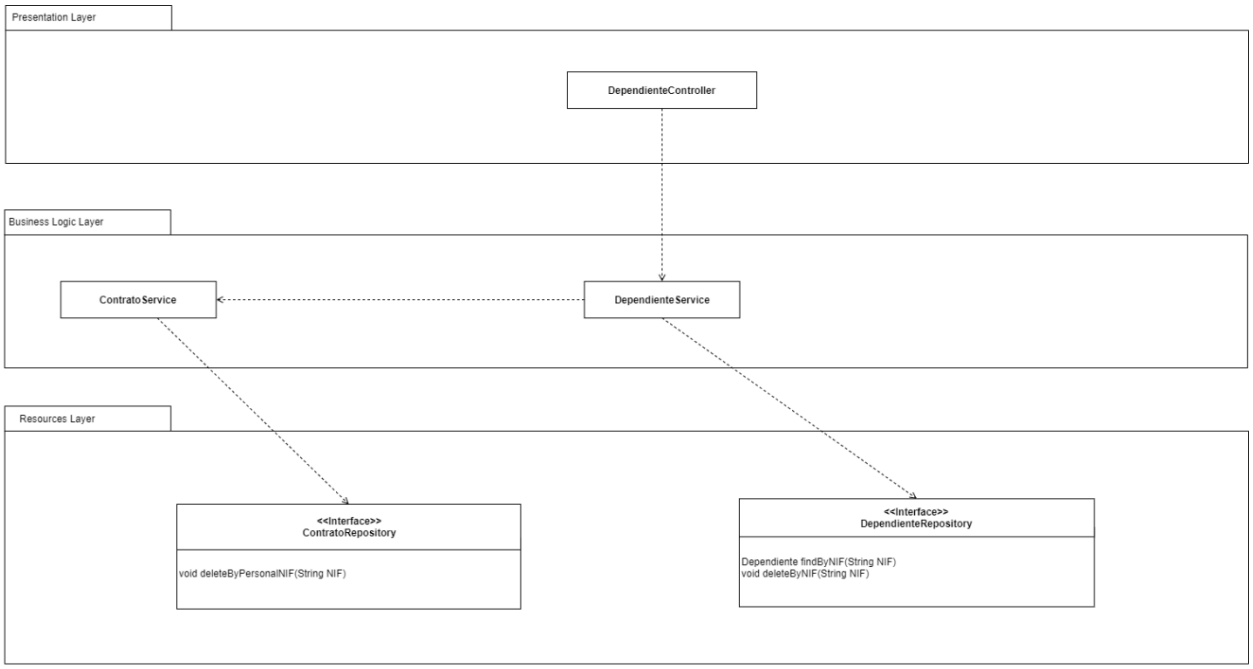

Presentation Layer
AlmacenController
Business Logic Layer
Almacen Service
Resources Layer
<<Interface>>
AlmacenRepository

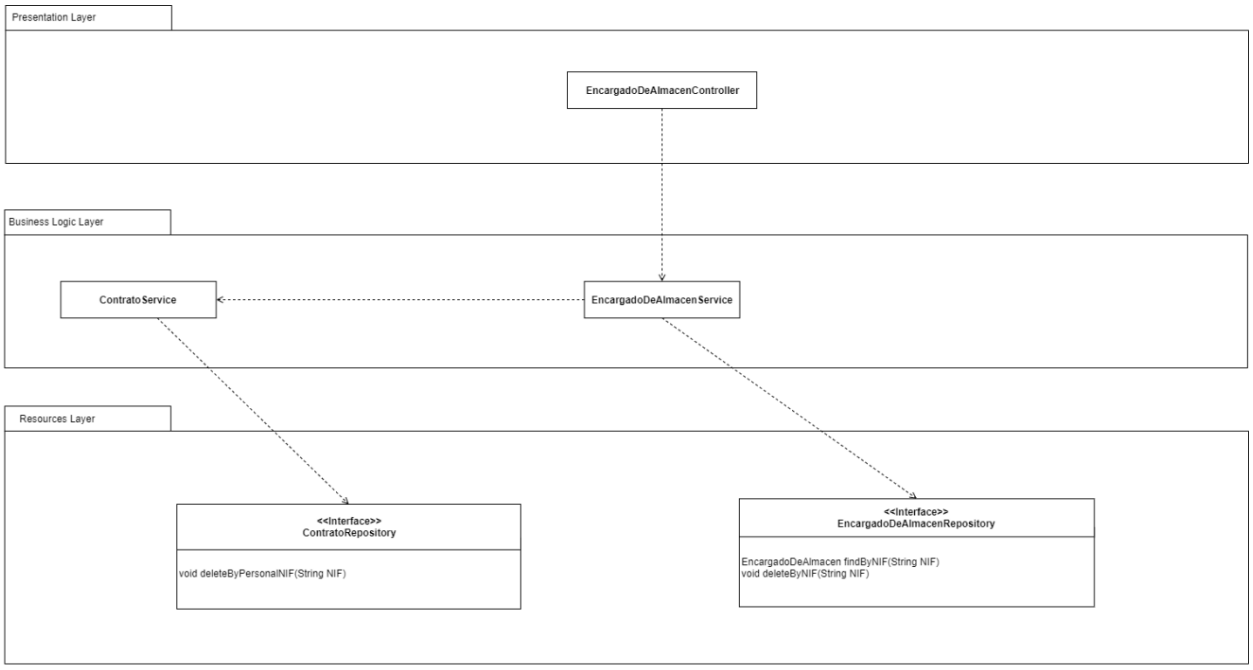
Azulejos Gresur
Grupo: G3-12

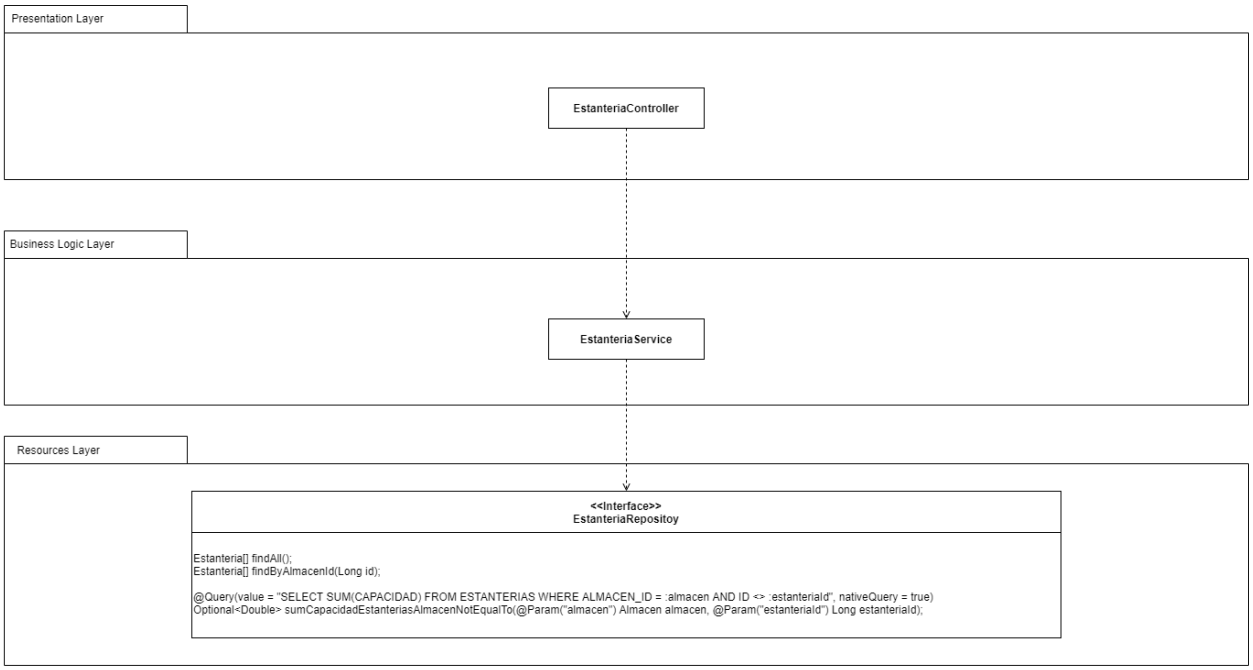



Presentation Layer
ConfiguracionController
Business Logic Layer
ConfiguracionService
Resources Layer
<<Interface>>
ConfiguracionRepository


Presentation Layer
ContratoController
Business Logic Layer
ConfiguracionService
ContratoService
Resources Layer
<<Interface>>
ConfiguracionRepository
<<Interface>>
ContratoRepository
void deleteByPersonalNIF(String NIF)

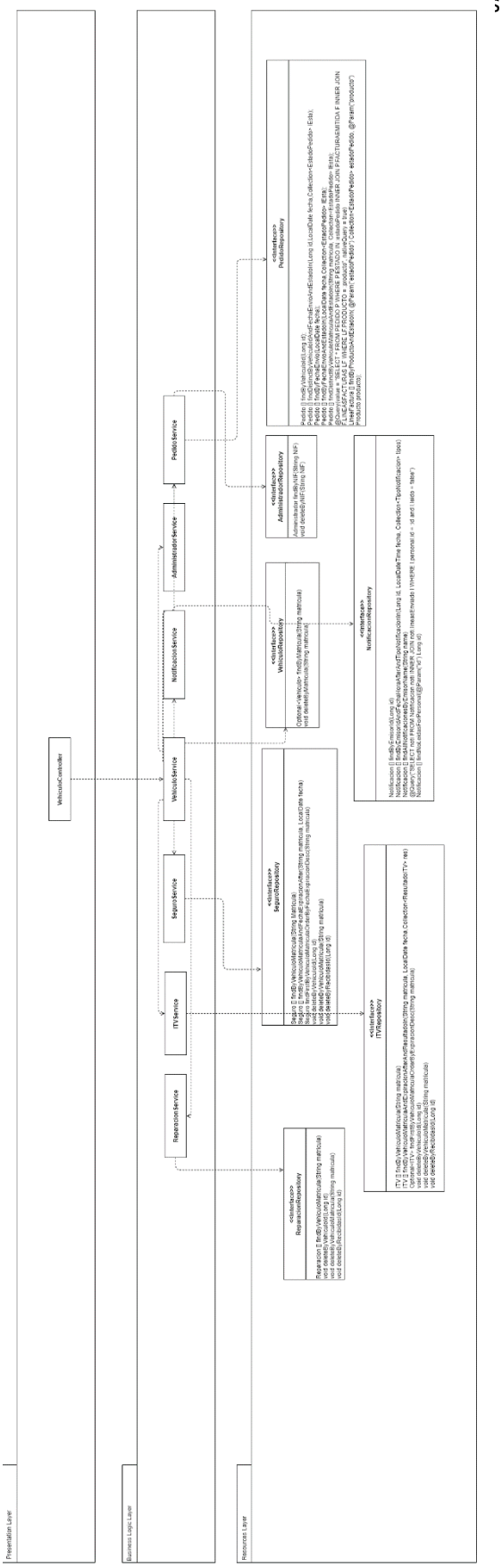

Presentation Layer
DependienteController
Business Logic Layer
ContratoService
DependienteService
Resources Layer
<<Interface>>
ContratoRepository
void deleteByPersonalNIF(String NIF)
<<Interface>>
DependienteRepository
Dependiente findByNIF(String NIF)
void deleteByNIF(String NIF)


Presentation Layer
EncargadoDeAlmacenController
Business Logic Layer
ContratoService
EncargadoDeAlmacenService
Resources Layer
<<Interface>>
ContratoRepository
void deleteByPersonalNIF(String NIF)
<<Interface>>
EncargadoDeAlmacenRepository
EncargadoDeAlmacen findByNIF(String NIF)
void deleteByNIF(String NIF)


Presentation Layer
EstanteriaController
Business Logic Layer
EstanteriaService
Resources Layer
<<Interface>>
EstanteriaRepositoy
Estanteria[] findAll();
Estanteria[] findByAlmacenId(Long id);
@Query(value = "SELECT SUM(CAPACIDAD) FROM ESTANTERIAS WHERE ALMACEN_ID = :almacen AND ID <> :estanteriaId", nativeQuery = true)
Optional<Double> sumCapacidadEstanteriasAlmacenNotEqualTo(@Param("almacen") Almacen almacen, @Param("estanteriaId") Long estanteriaId);

Azulejos Gresur
Grupo: G3-12

;



Azulejos Gresur
Grupo: G3-12

Patrón: MVC (Modelo – Vista – Controlador)

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

El patrón MVC se aplica principalmente al framework de Spring.

La parte de la **vista está implementada con React** (ver fichero /app), una biblioteca Javascript que envía peticiones a los **Controladores REST** (org.springframework.gresur.web) de Spring. A su vez dichos controladores interactuan con los **Servicios** (org.springframework.gresur.service) , los cuales operan en la BD mediante los **Repositorios** (org.springframework.gresur.repository).

Describir las partes de la aplicación donde se ha aplicado el patrón. Si se considera oportuno especificar el paquete donde se han incluido los elementos asociados a la aplicación del patrón.

Clases o paquetes creados

(paquetes mencionados en el apartado anterior)

Ventajas alcanzadas al aplicar el patrón

- En los controladores de Spring se utilizan objetos Java en lugar de Servlets, lo cual **facilita el testeo**.
- Mejor **escalabilidad**, lo que permite dividir el trabajo durante el desarrollo.
- Favorece la **cohesión**, el **bajo acoplamiento** y la **separación de responsabilidades**.
- Facilidad para la realización de **pruebas unitarias** de los componentes.

Patrón: FrontController

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

El patrón arquitectónico FrontController es aplicado en nuestro proyecto, dado que utilizamos el framework Spring, en el cual se aplica con el fin de que todas las peticiones realizadas a través de la web que vamos a crear, las gestiona y envía al controlador apropiado de manera que cada URL del sitio web la asocia a una función de la clase controller correspondiente.

Clases o paquetes creados

Para implementar dicho patrón, hemos creado el paquete "src/main/java/org.springframework.gresur.web" en el cual se encuentran todas las clases relacionadas con los controladores. Estos controladores van a contener las distintas funciones javas que se van a realizar cuando el FrontController reciba la petición de la web y la gestione enviándola al controlador que contenga la URL de dicha petición y por tanto realice la función que está asociada a la URL.

Ventajas alcanzadas al aplicar el patrón

- La implementación del FrontController es provista por el propio framework de Spring, por lo cual su implementación nos resultó sencilla y a la hora de probarla nos funcionaba de manera correcta, por lo que resultó satisfactorio que funcionara correctamente desde un principio.

- La decisión sobre quién va a ser el apropiado administrador de una solicitud puede ser más flexible.

Patrón: SPA (Single Page Application)

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Describir las partes de la aplicación donde se ha aplicado el patrón. Si se considera oportuno especificar el paquete donde se han incluido los elementos asociados a la aplicación del patrón.

La aplicación de este patrón es posible gracias a la implementación de React (Framework JavaScript) el cual adopta los principios de SPA.

Este patrón se aplicará en la capa de presentación concretamente en las vistas. Esto es así, ya que como sabemos con SPA se carga únicamente una página en el navegador y esta no se recarga durante el uso. Al interactuar el usuario con ella, la aplicación irá reescribiendo y actualizando dinámicamente la página en lugar de tenerla que carga nuevamente desde el servidor.

Clases o paquetes creados

Todo lo que se encuentra en la carpeta app (React) forma parte de la aplicación de este patrón.

Ventajas alcanzadas al aplicar el patrón

- Tiempo de respuesta rápido y fluido
- Fácil de construir y depurar
- Transición fluida al desarrollo móvil
- Buena integración con arquitecturas de capas

Patrón: Inyección de Dependencia

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

La aplicación de este patrón es clara al utilizar el Framework Spring.

La idea es que Spring controle la aplicación, implementando nosotros el código que provea la funcionalidad especifica de la aplicación. Spring va a ser quien lance nuestra aplicación…

Clases o paquetes creados

Se ha creado la clase GresurApplication y GresurInitializer en el paquete org.springframework.gresur

Ventajas alcanzadas al aplicar el patrón

- Al controlar Spring la aplicación, solo tenemos que realizar el código de la funcionalidad para que este nos puede aportar la funcionalidad deseada.

Azulejos Gresur
Grupo: G3-12

Patrón: Template view

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Dado que usamos React en este creamos componentes en HTML que funcionan como plantillas en la cual insertamos datos para modificarlos podemos decir que hacemos uso de este patrón.

Clases o paquetes creados

En la carpeta "/app/src/components" se encuentran los distintos componentes reutilizables de React, los cuales contienen la plantilla de un elemento HTML con diferentes atributos.

Ventajas alcanzadas al aplicar el patrón

- Una ventaja de este patrón es el reutilizado de componentes y que no dependemos de los datos en el proceso de representación de estos.

Patrón: Service Layer

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Para la aplicación de este patrón se han creado servicios que componen la capa de servicios. Dicha capa contiene el control de transacciones, la lógica de la aplicación y "checks" de seguridad.

Clases o paquetes creados

Se han creado servicios para cada una de las entidades del proyecto, todos estos servicios se encuentran en el paquete "src/main/java/org.springframework.gresur.service"

Ventajas alcanzadas al aplicar el patrón

- Dado que contamos con múltiples entidades, la aplicación de este patrón nos ha permitido gestionarlas y coordinarlas de la mejor manera posible.
- Puesto que la lógica de la aplicación es compleja, la implementación de una capa de servicios nos ha permitido implementar la lógica de negocio de nuestra aplicación fácilmente.

Patrón: (Meta)Data Mapper

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

En nuestra aplicación usamos Data Mapper puesto que para comunicar los modelos con la base de datos a través de los Repositorios y el uso del Entity Manager, ya que las transacciones de base de datos las realizamos en estos.

Clases o paquetes creados

Todo lo referente a la aplicación de este patrón se encuentra en "src/main/java/org.springframework.gresur. repository", lugar donde se encuentran cada uno de los repositorios de nuestro proyecto.

Ventajas alcanzadas al aplicar el patrón

- El uso de este patrón nos permite la independencia de los objetos respecto a la base de datos permitiéndonos un mayor control y personalización.

Patrón: Identity Field

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

La aplicación de este patrón podemos verlo claramente en la clase BaseEntity, con el atributo Id al ponerle el @Id + @GeneratedValue. Esto hará que se asocie con la clave primaria en la base de datos y además incrementará automáticamente la secuencia.

Clases o paquetes creados

En la clase BaseEntity de la que extienden la mayoría de nuestras entidades.

Ventajas alcanzadas al aplicar el patrón

- Fácil identificación de cada entidad mediante el Id.
- Autoincremento de la secuencia al crear una nueva entidad.

Patrón: Layer Supertype

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Con el fin de evitar duplicar código, se agrupan los atributos comunes a varias clases en una sola clase padre, de manera que todas las clases hijas tienen acceso a eso atributos. Este es el caso de BaseEntity, que es una superclase de la cual heredan todas las demás, esta superclase contiene el atributo “id”. También se ha aplicado este patrón en las clases Personal y Factura, aunque con un tipo de herencia distinto para el mapeo en base de datos.

Clases o paquetes creados

Se han creado las clases BaseEntity, Personal y Factura. Estas clases contienen atributos comunes de las clases hijas.

Ventajas alcanzadas al aplicar el patrón

- Se ha conseguido reducir el código de las entidades al crear la superclase que contiene atributos comunes.

Patrón: RepositoryPattern

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Como utilizamos el patrón de Modelo-Vista-Controlador haciendo uso de Spring, haremos uso de RepositoryPattern, ya que vamos a necesitar un repositorio por cada entidad que tengamos en la Base de Datos, de manera que estas clases encapsulen la lógica requerida para acceder a estos datos.

Azulejos Gresur
Grupo: G3-12

Clases o paquetes creados

Todo lo referente a la aplicación de este patrón se encuentra en "src/main/java/org.springframework.gresur.repository", lugar donde se encuentran cada uno de los repositorios de nuestro proyecto.

Ventajas alcanzadas al aplicar el patrón

- Da la impresión de que estamos accediendo a una colección de objetos que tenemos en memoria en lugar de que estés haciendo operaciones con la base de datos, de manera que nos facilita el acceso a los mismos.

Patrón: Lazy Loading

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Aplicamos este patrón en nuestro proyecto de manera estándar, ya que en las relaciones "OneToMany" y "ManyToMany" que existen entre entidades, no hemos indicado un parámetro como tal para que se use el patrón de diseño "Eager", por lo que estamos haciendo uso de este de manera por defecto.

Clases o paquetes creados

Este patrón se encontrará en los modelos que son en los cuales vamos a encontrar las relaciones en la cuales podemos elegir entre "Lazy" o "Eager", la ruta es "src/main/java/org.springframework.gresur.model"

Ventajas alcanzadas al aplicar el patrón

- Más rapidez en los tiempos iniciales de carga.
- Menor consumo de memoria.

Patrón: Pagination

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

Dado que tenemos un gran número de productos a mostrar, cargar todos directamente sería muy costoso por lo que hacemos uso del patrón Pagination para dividir la carga en partes más pequeñas y fácil de manejar.

Clases o paquetes creados

Para hacer uso de este patrón usamos la opción que nos permite el PagingAndSortingRepository para crear paginación en las funciones de búsqueda de los productos. Esto lo podemos ver en "src/main/java/org.springframework.gresur.repository.productoRepository" donde se encuentra las distintas llamadas JPQL referentes a los productos.

Ventajas alcanzadas al aplicar el patrón

- La mejor representación de los datos de los productos.
- Mejor rendimiento debido a reducir la carga de datos usada en cada momento.

Azulejos Gresur
Grupo: G3-12

Decisiones de diseño

En esta sección describiremos las decisiones de diseño que se han tomado a lo largo del desarrollo de la aplicación que vayan más allá de la mera aplicación de patrones de diseño o arquitectónicos.

Decisión I

Descripción del problema:

En la asignatura se propone realizar la vista de nuestra aplicación con tecnología JSP, sin embargo, la carga de la interfaz es más lenta que la de otras tecnologías. Además, el código resulta poco legible en vistas más complejas.

Alternativas de solución evaluadas:

Se barajaron distintos frameworks y librerías como: Angular, React o Vue.js. No obstante, ninguno de los miembros del grupo había utilizado anteriormente con ninguna de estas tecnologías.

Justificación de la solución adoptada

Finalmente se optó por usar React, pues es más fácil y rápido de aprender según mencionaban algunos foros para desarrolladores. Además, React facilita la aplicación del patrón SPA en nuestro proyecto, pues uno de nuestros objetivos principales era ofrecer una experiencia de usuario lo más fluida posible y con bajos tiempos de carga/actualización.

Decisión II

Descripción del problema:

El problema que nos surgió fue que nuestro proyecto tiene dos relaciones “especiales” la primera que sería entre Personal y Contrato y la segunda de Personal a Notificación. Como la clase Personal está marcada con @MappedSuperClass, no es persistente en BBDD por lo que la relaciones no podrían estar señalando a la tabla en cuestión y tendría que crearse una relación individual entre las dos tablas (Notificación y Contrato) dichas con cada una de las clases heredadas de la entidad Personal.

Alternativas de solución evaluadas:

- La primera y más básica alternativa que nos planteamos fue la de no tener clases heredadas lo cual provocaría que perdamos información de distinción entre los distintos usuarios que el sistema registra.
- La segunda alternativa sería crear todas las relaciones entre las clases Contrato y Notificación con las cuatro clases hijas, lo cual esto incrementaría la complejidad y se generarían tablas innecesarias en BBDD.
- La tercera y última alternativa es utilizar la herencia “Table per Class” y el ID en “GeneratedValue(Strategy=GenerationType.Table)” con el fin de generar una tabla para cada una de las entidades incluida la super.

Justificación de la solución adoptada

Aplicamos la tercera alternativa ya que de esta forma el ID que se genera en todas las tablas hijas va a ser único, es decir, que no se va a repetir entre ninguna de las tablas, de igual forma esta solución nos ayudaba a que si añadíamos un nuevo empleado seguía la secuencia del ID de la clase padre “Personal” y se almacenaba en la clase hija correspondiente, haciendo así que en la clase padre no se añada información reiterada. Esto, a su vez nos permitía que al hacer un “findById” en la clase padre

"Personal" nos devolviera el objeto igualmente, aunque no esté persistente en la superclase. Gracias a esta solución podemos hacer relaciones con personal, manteniendo así la independencia entre las clases que heredan de él.

Decisión III

Descripción del problema:

Durante la concepción del sistema de facturas que iba a incluir el proyecto, pensamos en el hecho de editar una factura dado que ocurriese un error a la hora de introducir los datos en el pedido asociada a ella como, por ejemplo, se introduce una cantidad errónea de un producto o dato erróneo del cliente.

Alternativas de solución evaluadas:

- La primera alternativa era editar los atributos que se deseaban de la factura en cuestión, pero leyendo en la Agencia Tributaria esto era ilegal, por lo que desechamos esta alternativa cuando conocimos esta información
- La segunda alternativa sería crear una factura rectificativa de la factura la cual se pretende modificar, de manera que esto se traduce en la aparición de una relación nueva, en la cual una factura puede tener una rectificación "0..1" .

Justificación de la solución adoptada

Como estamos desarrollando este proyecto para un cliente real, queremos ajustarnos a necesidades reales del cliente, por eso nos decidimos por la segunda alternativa, ya que era la opción correcta y legal de modificar una factura, de manera que se ajusta así a una necesidad real que pueda tener el usuario final de esta aplicación.

Decisión IV

Descripción del problema:

En nuestra primera versión del UML teníamos que el **Transportista** estaba relacionado con el **Vehículo**, y que el Vehículo estaba relacionado con los **Pedidos** que llevaba dicho vehículo. El principal problema de este diseño era su rigidez, pues cada transportista tenia asociado un único vehículo y no se podía modificar. Además, en caso de que hubiese algún incidente con un pedido, sería más complejo poder encontrar al transportista de dicho pedido.

Alternativas de solución evaluadas:

La principal alternativa que se nos ocurrió fue asociar al **Transportista** directamente al **Pedido**, y asociar el **Pedido** al **Vehículo** en el que era enviado.

Justificación de la solución adoptada

Esta fue la mejor solución, pues el sistema es más flexible a la hora de asociar los transportistas a los vehículos (pudiendo un mismo transportista usar varios vehículos) y también resolvía el problema de encontrar al repartidor en caso de incidencia.

Azulejos Gresur
Grupo: G3-12

Decisión V

Descripción del problema:

Durante la concepción del sistema de facturas que iba a incluir el proyecto, pensamos que iban a ser necesarios tener en BBDD los diferentes conceptos que podría tener una factura recibida, como bien es el caso de compra de material, gastos de vehículos… para así diferenciar una de otras.

Alternativas de solución evaluadas:

- La primera alternativa sería crear una de herencia a partir de la tabla padre Factura Recibida, en la que deberíamos tener tantos hijos como conceptos hubiese, por lo que esta alternativa nuevamente haría que existiesen varias tablas en la BBDD de uso excepcional.
- La segunda alternativa sería crear un tipo Enumerado con los diferentes conceptos que pueda tener una factura recibida y de esta forma únicamente solo tendríamos la tabla Factura Recibida.

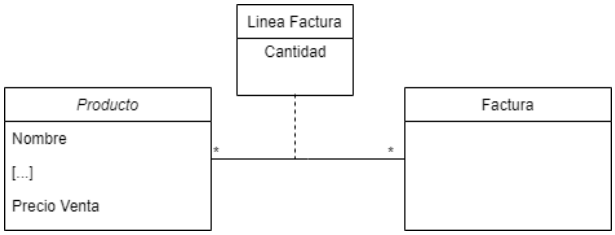
Justificación de la solución adoptada

Como no buscamos crear en BBDD multitud de tablas que o bien son innecesarias o bien van a tener un uso muy exclusivo nos decantamos por la segunda alternativa, además de que esta era la opción mas simple a la hora de llevarla a cabo a la implementación.

Decisión VI

Descripción del problema:

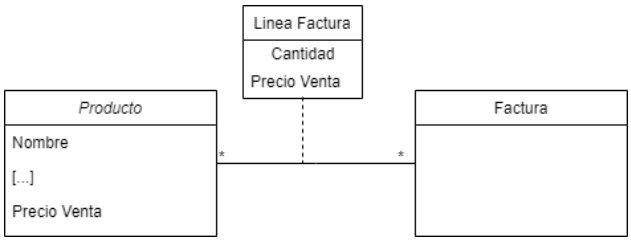
En nuestra primera versión del UML teníamos una relación entre estas entidades tal que así:



Una relación n-n entre Producto y Factura, donde el atributo Línea Factura contenía la cantidad de productos de dicha factura. Sin embargo, nos dimos cuenta de que, si se modifica el precio de un producto, este afectará al importe de las facturas anteriores.

Alternativas de solución evaluadas:

La principal solución que encontramos fue añadir otro atributo a línea de factura que fuese “Precio Venta”.



Justificación de la solución adoptada

De esta forma tendríamos el atributo “Precio Venta” del Producto (donde se almacena el precio actual) y “Precio Venta” de la Línea Factura, donde se almacena el precio de venta de un producto en el momento en el que se realiza la factura.