SensaCar

Memoria

UNIVERSIDAD
COMPLUTENSE
MADRID

Ignacio Baena Kuroda
Cristóbal Saraiba Torres
Ignacio Vítores Sancho

ÍNDICE

INTRODUCCIÓN:

La aplicación SensaCar fue diseñada en dos partes: una parte la cual gestiona la venta de vehículos y otra la cual maneja las competencias de ciertos empleados distribuidos por departamentos.

ARQUITECTURA DE LA APLICACIÓN:

Para ello se ha integrado el modelo MVC (Modelo Vista Controlador) por lo que se divide la aplicación en esas tres partes de forma muy distinguida.

1. Modelo: Contiene básicamente toda la funcionalidad básica de la aplicación, es decir, es la representación de la información la cual el sistema opera.
2. Vista: Está formado por el conjunto de ventanas o interfaces gráficas las cuales recogen la información de los usuarios y se transmite a través del controlador por la aplicación. finalmente la vista se actualiza en función de la respuesta recibida.
3. Controlador. Es el encargado de unir la vista y el modelo , es su mediador. El controlador recoge la información que la vista envía y se la transmite al modelo. Tras esto recibe la respuesta del modelo que es enviada de nuevo a la vista para que sea capaz de actualizarse.

Hemos aplicado una arquitectura multicapa formada por : Presentación , Negocio, Integración.

1. Presentación: En esta capa se encontrará toda la lógica de representación de interfaces gráficas. Se implementan todos los ActionListener encargados de comunicarse con el Controlador y éste, mandar información al modelo. Todo esto se llevará a cabo a través de una serie de *Comandos* lo cuales se ejecutarán en función del Evento que se envíe en ese momento.
2. Negocio: Es la capa encargada de gestionar toda la lógica de negocio de la aplicación. Recibe la información del controlador y es el encargado de comunicarse con la capa de integración para consultar la base de datos. Incidir en que es el único encargado de gestionar la lógica de negocio. Se utiliza principalmente el patrón SA (Service Application)
3. Integración: En esta capa la aplicación se conecta con la base de datos, es decir, recoge información de ella, la inserta, la actualiza y la elimina (*lógicamente) si fuese necesario. Se utiliza principalmente el patrón DAO (Data Access Object).

Más adelante profundizaremos más en cada patrón mencionado.

* lógicamente: los *delete* son convierte un booleano activo a false, es decir, no se borra de la base de datos, simplemente lo deja inactivo.

Gracias a estos modelos de arquitectura podemos hacerla mucho más mantible de forma que los cambios realizados en distintas capas no afecta al resto de estas. Esto hace que sea más reusable.

Es más fácil de entender el código al ser una arquitectura muy utilizada por lo que cualquier persona con un poco de conocimiento de ingeniería del software podría entenderlo sin mucha dificultad.

DIAGRAMAS DE CLASE:

Estos son los distintos módulos los cuales se han realizado diagramas clases.

Presentación: Cliente, Venta, Modelos - Empleado, Departamento, Competencia.

Negocio: Cliente, Venta, Modelos - Empleado, Departamento, Competencia.

Integración: Cliente, Venta, Modelos - Empleado, Departamento ,Competencia.

DIAGRAMAS DE SECUENCIA:

Aquí están todos los diagramas de secuencia que hemos realizado en el proyecto.

Presentación:

1. VentanaPrincipal.
2. VentanaVentas: actualizarVenta, añadirVenta, eliminarVenta, mostrarTodoVenta, mostrarVentaID.
3. Comandos:
    1. Ventas:abrirVentaCommand, añadirModeloVentaCommand, DevolverVentaCommand,EliminarVentaCommand,MostrarVentaCommand.
    2. Empleados: AñadirEmpleado, EliminarEmpleadoCommand, ActualizarEmpleado, MostrarEmpleado, MostrarTodoEmpleadoCommand
4. Controlador: ejecutar

Negocio:

1. Modelos: queryModelo.
2. Ventas:abrirVenta, añadirModeloVenta, cerrarVenta, devolverVenta, eliminarModeloVenta, eliminarVenta, leerListaVentas, leerVenta, obtenerDetallesVenta
3. Competencia: actualizarCompetenciaDeEmpleado, añadirCompetenciaAEmpleado, EliminarCompentenciaDeEmpleado.

4. Departamento: añadirEmpleadoADepartamento, borrarEmpleadoDeDepartamento, calcularNominaDepartamento.
5. Empleado: actualizarEmpleado, borrarEmpleado, crearEmpleado, leerEmpleado, leerListaEmpleados.

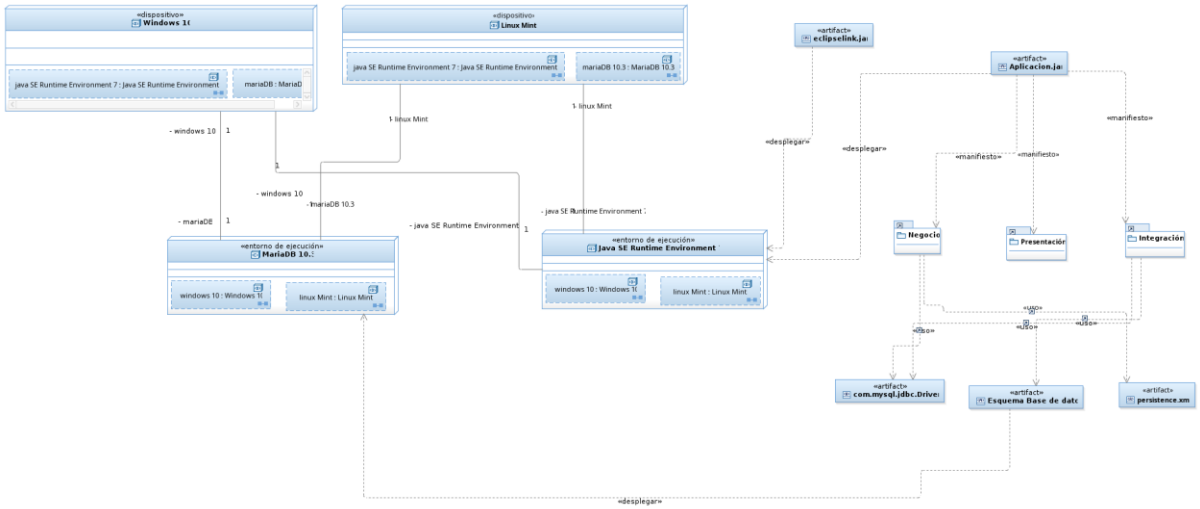
Integración:

1. Clientes : actualizarCliente, añadirCliente, eliminarCliente ,mostrarClientePorID, mostrarListaClientes.
2. Modelos : añadir Modelo, eliminarModelo, actualizarModelo, mostrarModeloPorID, mostrarListaModelo.executeQuery.
3. Ventas: añadirVenta, eliminarVenta, mostrarListaVentas, MostrarVentaPorID, getLineaVentas, updateTablaLineaVentas, insertaLineaVentas.

DIAGRAMA DE DESPLIEGUE:

*Un **Diagrama de Despliegue** modela la arquitectura en tiempo de ejecución de un sistema. Esto muestra la configuración de los elementos de hardware (nodos) y muestra cómo los elementos y artefactos del software se trazan en esos nodos. Un Nodo es un elemento de hardware o software.*

En nuestro caso encontramos Linux Mint y Windows 10 como nodos principales sobre los cuales se va a ejecutar el software creado.



(en el proyecto SensaCarModelo se encuentra el diagrama referenciado)

PATRONES UTILIZADOS:

DATA ACCESS OBJECT (DAO)

Patrón utilizado para independizar el acceso a datos de la BBDD. En la arquitectura multicapa se encuentra en la capa de Integración.

Utilizado en:

Capa de Integración

- DAOVentas
- DAOCliente
- DAOModelo

TRANSFER (T)

Crea una entidad para independizar el intercambio de datos entre las distintas capas del proyecto.

- TVentas
- TDetallesVenta
- TCliente
- TModelo
- TCompetencia
- TEmpleado
- TDepartamento
- TQuery

Se utilizan en torno a toda la aplicación. La presentación crea un T para enviarlo a la capa de integración a través del controlador y la capa de negocio.

TRANSFER OBJECT ASSEMBLER (TOA)

El *Transfer Object Assembler* usa *Transfer Objects* para recuperar los datos de los distintos objetos de negocio y otros objetos que definen el modelo o una parte del modelo.

- TDetallesVenta

En este caso TDetallesVenta posee un TVenta,TCliente,TModelo.

SERVICE APPLICATION (SA)

Patron utilizado para encapsular la lógica de negocio. Unido con la arquitectura multicapa estará conformada en la capa de Negocio

Capa de Negocio:

- SAVentas
- SACliente
- SAModelo
- SAEmpleado
- SACompetencia
- SADepartamento

COMMAND

Encapsular las peticiones de nuestra aplicación en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones.

Utilizado en :

Capa de Presentación para separar la el controlador de la capa de negocio. Hay comandos para todas las funciones del SA y para la creación de Vistas.

CONTEXTO

Se desea evitar utilizar información del sistema específica del protocolo fuera de su contexto relevante.

Utilizado en :

Capa de Presentación para enviar los datos a través de un par de Evento/Dato de la vista al controlador y viceversa.

FACTORÍA ABSTRACTA

Proporcionar una interfaz para crear familias de objetos relacionados.

En nuestro caso :

- Factoría Integración
- Factoría Negocio
- Factoría Command

SINGLETON

Patrón creado para crear una sola instancia en memoria de una clase.

Utilizado en:

- Factoría de Integración
- Factoría de Negocio
- Factoría de Comandos
- Aplicación de Negocio
- Transaction Manager
- Ventanas
    - Ventana Principal
    - Ventana Ventas
    - Ventana Cliente
    - Ventana Modelo
    - Ventana Empleado
    - Ventana Competencia
    - Ventana Departamento

SERVICE TO WORKER

Se desea llevar a cabo el manejo de peticiones y la invocación e lógica del negocio antes de pasar al control a la vista

Utiliza:

- Application Controller
- Command
- Factoria de Commandos (en sustitución del Dispatcher)

ALMACÉN DE DOMINIO

Patrón utilizado para separar la persistencia del modelo de objetos

Utilizado en la capa de integración para gestionar la transaccionalidad

Capa Integración:

- Transaction Manager

OBJETO DE NEGOCIO

Patrón utilizado en la parte de JPA para crear las entidades y poder persistir los datos creados

Utilizado en :

Capa de Negocio:

- Empleado
- EmpleadoTC (tiempo completo)
- EmpleadoTP (tiempo parcial)
- Departamento
- Competencia
- Tiene (tabla de cruce entre empleado y competencia)
- TieneID (id de la tabla Tiene)

QUERY

Patrón utilizada para separar las querys complejas de las clases DAO

Utilizado en Negocio pero implementado en Integración.

- Query cliente
- Query modelo

RECURSOS UTILIZADOS:

En nuestro proyecto hemos utilizado IBM RSA principalmente para la creación del modelo y el código.

A la hora de gestionar las bases de datos hemos utilizado MariaDB.

"MariaDB es un sistema de gestión de bases de datos derivado de MySQL con licencia GPL (General Public License). Es desarrollado por Michael (Monty) Widenius —fundador de MySQL—, la fundación MariaDB y la comunidad de desarrolladores de software libre.1 Introduce dos motores de almacenamiento nuevos, uno llamado Aria —que reemplaza a MyISAM— y otro llamado XtraDB —en sustitución de InnoDB—. Tiene una alta compatibilidad con MySQL ya que posee las mismas órdenes, interfaces, API y bibliotecas, siendo su objetivo poder cambiar un servidor por otro directamente."

Con todo esto utilizamos además DBeaver, una herramienta que nos facilita mucho su manejo gracias a una interfaz muy intuitiva.

"DBeaver es un cliente de SQL y una herramienta de administración de bases de datos. Para bases de datos relacionales utiliza la JDBC API para interactuar con bases de datos vía JDBC Driver"

INCIDENCIAS

La mayor a la incidencia que sufrimos a lo largo del desarrollo de la aplicación fue la disolución del equipo inicial en un equipo de solo 3 integrantes mencionados en la portada.

Esto hizo que la carga de trabajo aumentara, pero en comunicación con el profesor de la asignatura pudimos actualizar la SRS para que fuese más llevadera.

BIBLIOGRAFÍA

-Transparencias de la asignatura

-https://en.wikipedia.org/wiki/DBeaver

-https://es.wikipedia.org/wiki/MariaDB

-http://www.sparxsystems.com.ar/resources/tutorial/uml2_deploymentdiagram.html