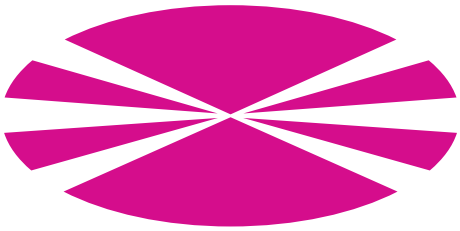

Facultad de Informática de la Universidad de A Coruña
Departamento de Computación

PROYECTO DE FIN DE CARRERA
INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Aplicación de gestión de un taller mecánico de reparaciones

Alumno: Santiago Regueiro Ovelleiro
Director: Marcos Ortega Hortas
Fecha: 21 de enero de 2013

Dr. Marcos Ortega Hortas
Profesor contratado Doctor en Facultad de Informática
Departamento de Computación
Universidad de A Coruña

CERTIFICA: Que la memoria titulada *"Aplicación de gestión de un taller mecánico de reparaciones"* ha sido realizada por Santiago Regueiro Ovelleiro bajo mi dirección y constituye su Proyecto de Fin de Carrera de Ingeniería Técnica en Informática de Gestión.

En A Coruña, a 21 de enero de 2013

Dr. Marcos Ortega Hortas
Director del proyecto

Resumen:

En este proyecto se ha realizado el diseño y desarrollo de una aplicación multiplataforma de gestión de un taller de reparación de vehículos. El objetivo de la aplicación es proporcionar una herramienta sencilla, eficiente y asequible orientada a pequeñas y medianas empresas que se introduzcan por primera vez en la informatización de sus sistemas o que deseen sustituir sistemas obsoletos e ineficientes.

Los empleados de la empresa podrán ser registrados en el sistema y ser proporcionados un nombre de usuario que les permitirá el registro de clientes con sus vehículos, así como de las compañías que aseguren a dichos vehículos. Podrán crear reparaciones con sus presupuestos, que necesitarán del uso de mano de obra y de ciertas piezas, también registrados en el sistema. Les será posible realizar pedidos de piezas a proveedores y consultar su stock actual. Toda operación comercial será facturable e impresa para la entrega al cliente.

Para su desarrollo se ha seguido una metodología de desarrollo iterativa. Se ha procedido al diseño del modelo de datos en UML que posteriormente ha sido traducido a una base de datos relacional en MySQL y a su definición en lenguaje Java. En cada etapa se ha realizado el diseño de un interfaz gráfico en Java Swing, apoyado por la Plataforma Netbeans para facilitar la integración de las distintas vistas de los datos, que cumplirán los casos de uso diseñados con anterioridad. El acceso a los datos almacenados en MySQL desde la aplicación Java ha sido realizado utilizando Hibernate y los informes y facturas han sido diseñados con JasperReports y DynamicReports.

El resultado ha sido una aplicación altamente modular que permite con facilidad llevar el día a día de una empresa que se dedique a la reparación de todo tipo de vehículos. La aplicación permite la conexión de múltiples usuarios concurrentes y la introducción y modificación de datos sin tener que preocuparse por la existencia de conflictos, que el sistema detectará y avisará para que no se produzcan inconsistencias en los datos. El uso de una base de datos externa en MySQL permite el acceso a los datos producidos por la aplicación desde cualquier otro sistema perteneciente a la empresa facilitando así la integración con otros programas en uso por la compañía.

Lista de palabras clave:

Taller, Reparaciones, Java, Mysql, Hibernate.

A mis padres

Agradecimientos

A mi tutor, Dr. Marcos Ortega Hortas por su ayuda durante la realización de este proyecto.

A mis padres, por su apoyo en todo momento y su ayuda durante el desarrollo y las pruebas del trabajo.

Hardware y software utilizado:

- **Sistema Operativo:** Windows 8 Pro 64 bits.
- **Máquina:** Intel C2D E8500 @ 3.15Ghz, 4GB RAM DDR2, 1Tb HDD.
- **Entorno de desarrollo:** Netbeans IDE. Versiones 7.0 a 7.2.
- **Entorno Java:** Java Development Kit 6 y 7.
- **SGBD:** Mysql 5.5 y Mysql Workbench 5.2.
- **Procesador de textos:** Texmaker 3.5.2 + MiKTeX 2.9.
- **Creación de diagramas:** Enterprise Architect 8, Microsoft Visio 2013, Pencil 2.0.3.
- **Gestión de cambios:** msysGit + TortoiseGit 1.7.
- **Otros:** Sublime Text 2, BitBucket.

Índice general

Introducción

El sector de la reparación de automóviles, como muchos otros, está dividido en dos grandes grupos de empresas. Por un lado están las franquicias y los grandes talleres oficiales y por otro los talleres de barrio o de pequeñas ciudades. Las grandes franquicias y talleres oficiales se encuentran altamente informatizados, poseyendo bases de datos y programas hechos a medida que integran todas las operaciones que la empresa puede llegar a realizar. Sin embargo, los pequeños talleres de barrio en muchos casos se resisten a dar el paso por el gran coste que puede llegar a tener este cambio o la falta de soluciones software apropiadas.

El gran avance en el campo de la informática en la última década, que ha permitido que los ordenadores sean cada vez más potentes y baratos, ha hecho asequible la presencia de al menos un equipo en buena parte de las empresas del país, pero la falta de soluciones apropiadas ha relegado su uso en muchos casos a la simple cumplimentación de facturas y al uso de otros programas no pensados para la gestión de un taller de reparaciones.

Por ello se ha decido el desarrollo de una herramienta software de gestión que permita generar una base de datos de clientes y sus vehículos, las reparaciones que se realizan y el stock de piezas disponibles para dichas reparaciones.

Objetivos

El objetivo de este proyecto es el análisis, diseño e implementación de una aplicación software de gestión empresarial para escritorio enfocada al sector de reparaciones de vehículos. Su finalidad será la de permitir gestionar con sencillez la información derivada de la realización de reparaciones, como pueden ser los detalles de esta, el cliente y el vehículo involucrado o las piezas que se han utilizado.

El punto de entrada de la herramienta será una interfaz gráfica que nos permita realizar los siguientes objetivos:

- **Conexión de usuarios:** Los empleados que hagan uso del sistema tendrán un nombre de usuario y contraseña asignados que utilizarán para obtener acceso a la

aplicación.

- **Alta, baja y modificación de registros:** El sistema permitirá dar de alta clientes, vehículos, aseguradoras, reparaciones, piezas, mano de obra, pedidos, proveedores, almacenes, empleados y usuarios, así como modificarlos y eliminarlos.

- **Generación de documentos:** Se permitirá la creación de listados de registros o de vistas detalladas de ellos, así como la generación de facturas, presupuestos u ordenes de reparación.

Estructura de la memoria

La memoria se ha dividido en las siguientes secciones:

Fundamentos teóricos. Se describe el sector al que va dirigida la aplicación, indicando el trabajo que realiza y las necesidades que pueda tener. Se explica el significado de un sistema de gestión y los tipos conocidos y los requisitos que una empresa del sector podría tener de este sistema.

Análisis de antecedentes o alternativas. Se investigan otras aplicaciones que realicen una tarea similar, que son descritas indicando datos como precio, disponibilidad o sector principal al que se enfoca, junto con sus ventajas e inconvenientes.

Fundamentos tecnológicos. Se explican las soluciones tecnológicas disponibles para la realización de la aplicación y se decide cuáles de ellas van a ser usadas.

Metodología. Describe en profundidad el proceso de desarrollo del software. Se analiza el problema a resolver y se propone un diseño del sistema, indicando cómo se ha implementado cada subsistema y su funcionalidad.

Prestaciones destacadas. Resume las funcionalidades principales que la aplicación ofrece una vez terminado su desarrollo.

Pruebas realizadas. Se describen los distintos tipos de pruebas que se han usado para evaluar la completud y corrección del software desarrollado.

Planificación y evaluación de costes. Se detalla la división en pequeñas tareas de desarrollo y el tiempo que se espera que tome cada una, así como el tiempo que ha llevado en realidad y el gasto que habría supuesto.

Conclusiones. Se comentan los resultados de la implementación y los problemas que se han encontrado durante el trabajo.

Líneas Futuras. Se describen posibles ampliaciones que se podrían realizar al sistema si se desease añadir funcionalidades o mejorar las ya presentes.

Anexos. Se incluye un pequeño manual de instalación y uso que detalla los pasos necesarios para obtener un sistema completamente funcional y cómo comenzar a utilizarlo.

Capítulo 1

Fundamentos teóricos

1.1. Contexto

El sector del automóvil en España es uno de los que más han sufrido los efectos de la crisis económica mundial ocurrida en los últimos años. Las caídas continuas en las ventas, llegando en Septiembre de 2012 a mínimos históricos [4] y no esperándose que sea posible una recuperación a corto plazo gracias a los planes del Gobierno [11], han repercutido también en el número de reparaciones que los españoles realizan a sus vehículos. Aunque la tendencia actual lleve más a reparar el vehículo que ya se posee en vez de comprar uno nuevo, la realidad es que en 2011 cerraron o cambiaron de titular 2500 talleres de reparaciones, con más de 4800 temiendo por su supervivencia a corto y largo plazo[6].

Todo esto nos lleva a asegurar la necesidad de una reducción de costes y una mayor eficiencia en el desarrollo de las actividades de reparación de automóviles. Para ello las empresas tienen a su disposición sistemas de gestión empresarial, que les facilitará el desarrollo de las operaciones del día a día y les podría permitir descubrir que secciones necesitarían reducir sus gastos o cuales son las actividades que proporcionan un mayor beneficio a la empresa.

1.2. Taller de reparaciones

Un taller mecánico es un lugar dónde se realizan reparaciones a automóviles o motos. El taller de reparaciones puede ser especializado, como aquellos que se dedican únicamente al mantenimiento de la electrónica, al trabajo con lunas y cristales o a la reparación de chapa y pintura, mientras que otros serán de propósito más general, pudiendo limitarse a la reparación de sólo una marca o un tipo de vehículo.

Diariamente los clientes acuden a estos establecimientos con la necesidad de realizar alguna reparación a sus vehículos, proceso que normalmente lleva a intervenir más de una persona y que conlleva la gestión de los arreglos que se realizan y del tiempo y coste asociado a ellos. La empresa deberá llevar al menos una contabilidad de las reparaciones que realice, indicando las piezas que ha gastado o vendido en el proceso para poder administrar el stock restante en sus almacenes, y facturando todos sus gastos e ingresos para que les sea posible pagar todos los impuestos necesarios y conocer con exactitud el dinero que tienen disponible para continuar realizando sus actividades.

La realización de una reparación conlleva la generación de un gran cantidad de información. Primero es necesario conocer la marca y modelo del vehículo en cuestión, para poder decidir, una vez se conozcan las necesidades de la reparación, si las piezas que tenemos disponibles son compatibles o si será necesario realizar un pedido. Dicha reparación afectará al menos a un mecánico que se encargará de reconocer el estado del vehículo e identificar los pasos necesarios para revertir el problema con el que se les han presentado, y una vez informado al cliente, y con su consentimiento, realizar la reparación. Este trabajo llevará cierta cantidad de horas, que deberán ser contabilizadas y normalmente repercutidas en la factura final que será entregada al cliente y que deberá indicar detalladamente todas las modificaciones realizadas en el vehículo y su precio asociado y que habrá de pagar para poder retirar su automóvil. Como se puede apreciar, este proceso nos ha llevado, por ejemplo, a tener que averiguar los datos del cliente y su vehículo, a conocer el precio por hora de los distintos trabajos o el coste de las piezas utilizadas.

El taller también habrá de realizar la gestión de las piezas disponibles en el almacén. Semanalmente o al menos mensualmente tendrá que hacer un recuento de los productos disponibles para poder identificar aquellos que es necesario reponer para el funcionamiento normal de la empresa durante el siguiente periodo. Estos pedidos serán hechos a proveedores con los que la empresa normalmente ha firmado un contrato y que se comprometen a realizar la entrega de las piezas necesarias por el taller.

Tradicionalmente la información de la empresa se almacenaba en grandes archivos, que estaban separados en fichas y necesitaban ser organizados manualmente. Para realizar sus trabajos el personal debía consultar las fichas de clientes, de vehículos, o los libros del almacén y proveedores. Con la irrupción de la informática todo este proceso esta siendo movido a sistemas de información conocidos como Sistemas de Gestión Empresarial.

1.3. Sistemas de gestión empresarial

En informática, un sistema de información es cualquier sistema o subsistema de equipo de telecomunicaciones o computacional interconectados y que se utilicen para obtener,

almacenar, manipular, administrar, mover, controlar, desplegar, intercambiar, transmitir o recibir voz y/o datos, e incluye tanto los programas de computación ("softwarez "firmware") como el equipo de cómputo.

En teoría de sistemas, un sistema de información es un sistema, automatizado o manual, que abarca personas, máquinas, y/o métodos organizados de recolección de datos, procesamiento, transmisión y diseminación de datos que representa información para el usuario.

En geografía y cartografía, un Sistema de Información Geográfica (SIG) se utiliza para integrar, almacenar, editar, analizar, compartir y desplegar información georeferenciada. Existen muchas aplicaciones de SIG, desde ecología y geología, hasta las ciencias sociales.

En sociología los sistemas de información son sistemas sociales cuyo comportamiento está fuertemente influenciado por los objetivos, valores y creencias de los individuos y grupos, así como por el desempeño de la tecnología.

Sin embargo, generalmente el término Sistema de Información se asocia con su aplicación en el campo de la Gestión de Empresas, debido a la gran importancia que han adquirido estos sistemas dentro de este campo. Se puede decir que durante los últimos años, los sistemas de información constituyen uno de los principales ámbitos de estudio en el área de organización de empresas. El entorno donde las compañías desarrollan sus actividades se vuelve cada vez más complejo. La creciente globalización, el proceso de internacionalización de la empresa, el incremento de la competencia en los mercados de bienes y servicios, la rapidez en el desarrollo de las tecnologías de información, el aumento de la incertidumbre en el entorno y la reducción de los ciclos de vida de los productos originan que la información se convierta en un elemento clave para la gestión, así como para la supervivencia y crecimiento de la organización empresarial.

Una vez centrados los Sistemas de Información en este campo, la gestión de empresas, existe un amplio abanico de definiciones para este concepto. Tal vez la más precisa y comúnmente aceptada sea la que define un Sistema de Información como -conjunto formal de procesos que, operando sobre una colección de datos estructurada de acuerdo a las necesidades de la empresa, recopila, elabora y distribuyen selectivamente la información necesaria para la operación de dicha empresa y para las actividades de dirección y control correspondientes, apoyando, al menos en parte, los procesos de toma de decisiones necesarios para desempeñar funciones de negocio de la empresa de acuerdo con su estrategia. Otra definición de sistema de información también aceptada comúnmente es la que indica que -sistema de información es aquel conjunto de componentes interrelacionados que capturan, almacenan, procesan y distribuyen la información para apoyar la toma de decisiones, el control, análisis y visión de una organización- [8].

1.3.1. Tipos de sistemas de información

Desde un punto de vista empresarial, según la función a la que vayan destinados o el tipo de usuario final del mismo, los SI pueden clasificarse en [8]:

- Sistema de procesamiento de transacciones (TPS).- Gestiona la información referente a las transacciones producidas en una empresa u organización.
- Sistemas de información gerencial (MIS).- Orientados a solucionar problemas empresariales en general.
- Sistemas de soporte a decisiones (DSS).- Herramienta para realizar el análisis de las diferentes variables de negocio con la finalidad de apoyar el proceso de toma de decisiones.
- Sistemas de información ejecutiva (EIS).- Herramienta orientada a usuarios de nivel gerencial, que permite monitorizar el estado de las variables de un área o unidad de la empresa a partir de información interna y externa a la misma.
- Sistemas de automatización de oficinas (OAS).- Aplicaciones destinadas a ayudar al trabajo diario del administrativo de una empresa u organización.
- Sistema experto (SE).- Emulan el comportamiento de un experto en un dominio concreto.
- Sistema de Planificación de Recursos (ERP).- Integran la información y los procesos de una organización en un solo sistema.

1.3.2. Evolución Histórica

Desde la aparición de las primeras computadoras, los sistemas de información se han ido introduciendo en las empresas como una potente herramienta para optimizar y mejorar su gestión. Esta introducción de los sistemas de información, ha sido progresiva, evolucionando los sistemas de información en función de su área de aplicación en la empresa y de la tecnología existente en cada momento. Por ello, los sistemas de información para la gestión en la empresa han pasado por diferentes fases, que se detallan a continuación[2].

1.3.2.1. Software de Gestión contable

Las primeras aplicaciones de los sistemas informáticos en la empresa fueron introducidas alrededor de 1960, en el área de la Gestión Contable. La contabilidad es una ciencia claramente definida, mediante leyes, normas y reglas que deben ser seguidas por las empresas,

independientemente de su naturaleza y el sector al que pertenezcan. Por ello, el diseño e implantación de un software para su gestión es mucho más sencillo que para otras áreas de la empresa, ya que el *análisis de requisitos* ya viene dado desde las administraciones de los diferentes países, y es exactamente el mismo para todas las empresas que se rigen por su legislación.

1.3.2.2. Gestión administrativa

Inmediatamente después de la aparición del software contable, surgió la necesidad de gestionar también el área administrativa. De esta manera se desarrollaron sistemas de información que podrían gestionar las facturas, los pagos y los cobros., los cuales quedaban almacenados en un sistema informático, para su posterior consulta o estudio. Si además este sistema estaba integrado con el de gestión contable y financiera, el ahorro de trabajo y el aumento de la productividad estaba garantizados.

1.3.2.3. Control de Stocks

Tras esto, el desarrollo de software para la empresa centró sus esfuerzos de investigación y desarrollo en el área de control de Stocks, apareciendo así los sistemas ICS, con los cuales se podía conocer el número de unidades de un producto existente en almacén, los consumos realizados en los diferentes periodos, y por supuesto, su valor. De nuevo, la integración con el resto de aplicaciones de la empresa se revelaron como un factor muy importante en la optimización de los procesos.

1.3.2.4. Planificación de materiales

Entre finales de los años sesenta y principios de los setenta, aparecen los primeros sistemas MRP. Estos sistemas, surgen como evolución natural de los sistemas de gestión de stocks, con la inclusión de las partidas de material llamadas BOM. La característica fundamental de estos sistemas es la aplicación de un enfoque jerárquico a la gestión de inventarios, permitiendo básicamente la elaboración del plan de uso de materiales a partir de tres elementos fundamentales:

- El Programa Maestro de Producción (PMP).
- La lista de materiales (BOM).
- El fichero de registro de inventarios (IRF).

Los sistemas MRP producen notables avances, entre los que destacan la reducción de inventarios, del tiempo de proceso y suministro y el incremento de la eficiencia. Sin em-

bargo, para alcanzar estos beneficios es necesaria una gran exactitud en la gestión de la producción. Su principal problema es que pasa por alto las restricciones de capacidad y las técnicas de gestión de talleres.

1.3.2.5. Planificación de fabricación

Los sistemas MRP II son una evolución natural de los Sistemas MRP que surgen durante los años 80. En esta nueva fase en la evolución se comienzan a tener en cuenta tanto las necesidades de gestión y planificación del uso de material, como la de recursos y capacidades necesaria para la fabricación. Este tipo de sistemas, están centrados en el área de producción, si bien es cierto, que tratan de integrarse con otras aplicaciones que gestionen otras áreas de la empresa.

1.4. Requerimientos del sector de reparaciones

Un sistema de información que desee enfocar su uso al sector de las reparaciones de vehículos deberá cumplir una serie de requisitos propios del sector que intentan representar.

1.4.1. Gestión de Clientes y Vehículos

La empresa necesita mantener una lista actualizable de clientes, con todos los datos personales necesarios para identificarlos y contactar con ellos en caso de necesidad, así como de sus vehículos y sus características técnicas.

1.4.2. Gestión de aseguradoras

En ciertos casos las empresas aseguradoras deberán hacerse cargo de las reparaciones realizadas al vehículo y para ello la empresa deberá guardar la información necesaria para ponerse en contacto con ellas.

1.4.3. Gestión de reparaciones

Las reparaciones comprenden el uso de piezas, mecánicos y horas de trabajo, que la empresa deberá conocer con exactitud para poder facturar correctamente. También tendrá la necesidad de mantener un listado de precios de los distintos trabajos que realizan sus empleados.

1.4.4. Gestión de almacén

Debido al amplio número de piezas implicadas en los trabajos a realizar en un taller, la empresa deberá llevar un listado de los productos disponibles, su precio y la cantidad disponible en todo momento.

1.4.5. Gestión de pedidos

La empresa deberá llevar un control de los productos pedidos a sus proveedores, guardando los datos de estos para poder comunicarse con ellos.

1.4.6. Impresión de informes y facturas

El sistema deberá permitir imprimir las facturas derivadas de la realización de reparaciones o pedidos e informes como listas de clientes o históricos de trabajos.

1.4.7. Otras características técnicas deseables

Los sistemas de gestión empresarial, además de tener que permitir la edición y consulta de la información necesaria para la realización de las operaciones de la empresa, suelen añadir otras características técnicas que pueden ser más o menos útiles dependiendo de las necesidades de la empresa.

Soporte multi-usuario Debe permitir la existencia de distintos usuarios en el sistema y la posibilidad de que estos realicen uso de los datos almacenados al mismo tiempo, para no ralentizar la producción.

Posibilidad de edición de permisos Los permisos de los que dispone un usuario han de poder se editados, permitiendo así limitar las funciones que puede hacer este dependiendo de su puesto de trabajo y añadiendo un nivel más de seguridad al programa.

Soporte multi-equipo Posibilidad de conectarse al sistema desde cualquier equipo que lo tenga instalado, obteniendo los mismos datos y experiencia que si lo hiciera desde su máquina habitual en la empresa.

Capítulo 2

Estudio o análisis de antecedentes o alternativas

Debido a la informatización empresarial que ha ocurrido durante los últimos años, existen multitud de programas disponibles para realizar la gestión de un taller de reparaciones o de una tienda en general. Desgraciadamente, muchos de estos programas han ido quedando obsoletos con el paso de los años debido a la falta de actualizaciones y puede llegar a ser difícil encontrar una solución adecuada que se adapte correctamente a todos los trabajos que realiza la empresa.

2.1. Aplicaciones de gestión de un taller de reparaciones

A continuación se detallan algunas de las soluciones existentes en la actualidad, divididas entre si están especializadas en la gestión de talleres de reparaciones o si están pensadas para un uso más general para cualquier empresa que se dedique a la prestación de un servicio o a la venta de bienes.

2.1.1. Comerciales

Sistemas a medida. Existen multitud de empresas especializadas en el desarrollo y mantenimiento de sistemas de gestión empresarial a medida. Normalmente se compone de un producto base que puede ya ser suficiente para la gestión de las tareas de la empresa, pero que puede ser modificado para adaptarse a las particularidades del negocio concreto. Suelen tratarse de sistemas caros, disponibles desde los 300€ a las decenas de miles de euros.

Autosoft Taller. Actualmente en su versión 4, aparecida en 2011 y desarrollado en EEUU, posee versión de evaluación por 30 días y su precio es de 50€ si se decide utilizar la edición básica para pequeñas empresas o 219€ si se prefiere la edición completa. Es posible realizar un alquiler de la licencia de esta versión por 75€ anuales. Disponible en `http://www.autosofttaller.com/`.

Aswin. Es un DMS estándar destinado a los reparadores autorizados, agentes y concesionarios de las redes de distribución, con un alto grado de integración con los procesos de las Marcas. Más información en `http://www.adpdsi.es/es/soluciones/sistemas-de-gestion-de-concesionarios/aswin.asp`.

Prosicar Taller. Otra herramienta de gestión de talleres con un precio de 99€ en su versión más básica o 181€ si se desea la experiencia completa. Disponible en `http://www.doscar.com`.

Otras soluciones encontradas incluyen programas tales como **Bayta Mecano**, **Taller7**, **KitCar**, **Gestplus Business Automoción** o **ITACTIL Taller Mecánico**, disponibles desde 60€ el más económico a 390€ el más caro.

2.2. Gestión de stocks y facturación

Los siguientes programas no se centran en la gestión de un tipo concreto de empresa, sino en la gestión general de cualquier tipo de negocio dedicado a la venta de productos o a la oferta de servicios.

2.2.1. Comerciales

SQL Pyme. Creado en La Coruña por una empresa de desarrollo y comercialización de software empresarial. Es un ERP estándar fruto de más de 17 años de desarrollo y mejora continua. Provee de módulos que pueden adaptar el programa a las necesidades de una empresa de fabricación, venta o reparación, asi como conexión con dispositivos móviles. Disponible en `http://www.distritok.com/distritok/SQL-Pyme/erp_pyme_introduccion.asp`.

WinPyme. Desarrollado en España, es un programa de gestión empresarial completo que permite llevar la gestión de productos, facturación, mano de obra, ingresos, gastos o pedidos y trabajos. Tiene un coste anual de 84€ para su versión mas básica, que no permite realizar pedidos o trabajos, subiendo hasta 169€ por su versión completa con

todas las características disponibles, y existiendo 4 versiones intermedias que varían en precio y características. Si de desease una licencia indefinida el precio varía entre 257€ y 665€. Es una aplicación monousuario y monopuesto. Más información en `http://www.winpyme.com`.

inFlow. Procedente de Canadá, es otra solución empresarial de propósito general que permite llevar una gestión de productos, su stock, pedidos y ventas. En su versión más básica, disponible gratuitamente para hasta 100 productos o clientes, no permite la conexión de múltiples usuarios ni la gestión de los permisos que éstos tienen. Existe una versión disponible por 226€ que elimina la limitación en el número de productos y añade múltiples usuarios y por último, la versión completa por 377€, que permite la edición de los permisos de cada usuario así como la generación de un mayor número de informes que los disponibles en las otras dos ediciones anteriores. A la venta en `http://inflowinventory.com/`.

2.2.2. Gratuítas

inFlow Como ya se ha expuesto anteriormente, gratuíto para hasta 100 productos y clientes y para instalaciones monousuario.

Abanq Software libre de tipo ERP orientado a la administración, gestión comercial, producción o comercio electrónico, entre otras aplicaciones. Disponible en `http://abanq.org/`.

2.3. Ventajas, inconvenientes y conclusión

El mayor inconveniente contra el uso de alguna de estas alternativas es su precio. Aunque alguna soluciones pueden ser adquirida por tan poco como 60€, en muchos casos se tratan de ediciones limitadas de software que no ha sido actualizado en mucho tiempo. Las soluciones de calidad han de ser desarrolladas a medida o adaptando una aplicación de propósito general mediante el uso de los módulos que ponen a disposición sus creadores y que hace aumentar el precio total.

Otro de los problemas observados durante la prueba de estas aplicaciones es su incompletud. En algunos casos se ha observado que no es posible realizar una gestión de pedidos o empleados, mientras que en otras se ha desarrollado una gestión avanzada del stock descuidando la gestión de los clientes y vehículos. La aplicaciones en las que se ha observado este defecto no pueden ser ampliadas mediante el uso de módulos, lo que hace que

dependiendo del uso que queramos darle puedan llegar a ser inapropiadas, generando más costes en el futuro cuando necesitemos nuevas funcionalidades y haya que trasladar los datos a un nuevo sistema.

También se ha observado que las aplicaciones con un coste menor no dan soporte al uso de múltiples usuarios y están pensadas para su uso en un único ordenador, impidiendo compartir información entre distintos terminales, lo cual ralentizaría la producción en cuanto dos empleados quieran acceder al sistema al mismo tiempo.

Con el desarrollo de esta aplicación se pretende hacer disponible una herramienta libre, gratuita y funcional que permita cubrir todas las necesidades de una empresa de reparaciones y que permita tanto el acceso de manera estándar a los datos almacenados como la posibilidad de ampliar sus funcionalidades de una manera sencilla.

Concluimos que la mejor manera de prestar este servicio es construir una aplicación sencilla, enfocada a pequeñas y medianas empresas con un número de usuarios y carga de trabajo media y que necesite una solución adaptable que no requiera un gran desembolso económico previo.

Capítulo 3

Estudio de viabilidad o fundamentos tecnológicos

3.1. Lenguaje de programación

Un lenguaje de programación es un idioma artificial desarrollado para facilitar la comunicación hombre-máquina en los sistemas informáticos. Existen cientos de lenguajes de programación, organizados en decenas de paradigmas diferentes. Para el desarrollo de este proyecto se ha decido hacer uso de un lenguaje de programación orientado a objetos, por su mejor representación de los datos de los conceptos del mundo real y su gran abstracción en su modelado como objetos.

Los lenguajes de programación orientados a objetos se caracterizan por estar orientados a los datos y a sus posibles comportamientos en vez de a las funciones. Su gran ventaja es la naturalidad en el modelado de datos del mundo real como objetos dentro de la estructura del programa. El mayor inconveniente es un menor rendimiento y un mayor consumo de recursos, mitigado en gran parte por la potencia actual de los ordenadores disponibles.

Dentro de los lenguajes de programación orientados a objetos, destacan C++, C#, Java, Perl, Python, Ruby o Visual Basic y de entre ellos se ha elegido Java por su disponibilidad en múltiples sistemas operativos y por incluir entre sus librerías estándar la biblioteca gráfica Swing que permite realizar una interfaz gráfica compatible con todos los sistemas en los que Java esté disponible.

El lenguaje de programación Java fue introducido por primera vez en 1995 por Sun Microsistems. Derivado de lenguajes como C y C++, Java fue diseñado para ser más intuitivo y sencillo de usar que los antiguos lenguajes, especialmente debido a su simplista modelo orientado a objetos y facilidades automatizadas como la gestión de memoria. En su tiempo, Java atrajo el interés de los desarrolladores por su orientación a objetos, arquitectura

concurrente, excelente seguridad y escalabilidad, y porque aplicaciones desarrolladas en el lenguaje Java podrían ser ejecutadas en cualquier sistema operativo que contuviera una Máquina Virtual de Java (JVM). Desde sus comienzos, Java fue descrito con un lenguaje que permite a los desarrolladores "escribir una vez, ejecutar en todas partes" ya que el código es compilado en archivos class que contiene bytecode y los archivos resultantes pueden ser ejecutados en cualquier JVM compatible. Este concepto ha hecho de Java un éxito inmediato para el desarrollo de aplicaciones de escritorio, que luego se bifurcó en diferentes soluciones tecnológicas con el paso de los años, incluyendo el desarrollo de aplicaciones basadas en web y aplicaciones de internet enriquecidas (RIA). Hoy, Java está implementado en un amplio rango de dispositivos, incluyendo teléfonos móviles, impresoras, dispositivos médicos, reproductores de Blu-ray, etc.

La plataforma Java consiste en una jerarquía de componentes, comenzando con el Kit de Desarrollo Java (JDK), que está compuesto por el Entorno en Tiempo de ejecución Java (JRE), el lenguaje de programación Java, y las herramientas necesarias para desarrollar y ejecutar aplicaciones Java. El JRE contiene la Máquina Virtual (JVM), además de las interfaces de programación en Java (APIs) y las librerías que asisten en el desarrollo de aplicaciones.

La JVM es la responsable de interpretar los archivos Java class y ejecutar su código. Todo sistema operativo que es capaz de ejecutar código Java tiene su propia versión de la máquina virtual. Para ello, el JRE ha de estar instalado en cualquier sistema que vaya a ejecutar aplicaciones Java. Oracle proporciona implementaciones del JRE para la gran mayoría de los principales sistemas operativos. Cada sistema tiene su propia versión, por ejemplo, los dispositivos móviles pueden ejecutar una versión reducida de la máquina virtual que esta optimizada para ejecutar aplicaciones desarrolladas con Java Mobile Edition (ME).

Las APIs de Java son una colección de clases predefinidas que son usadas por todas las aplicaciones Java. Cualquier programa Java que sea ejecutado en la máquina virtual hace uso de las Java APIs y librerías. Esto permite a las aplicaciones hacer uso de funcionalidades que han sido predefinidas y cargadas en la JVM y deja a los desarrolladores con más tiempo para preocuparse de los detalles de su aplicación específica. Las clases que comprenden las APIs y librerías permiten a las aplicaciones Java comunicarse con el sistema operativo. De tal manera, la plataforma Java se ocupa de interpretar las instrucciones provistas por la aplicación Java y convertirlas en comandos del sistema operativo en el que se está ejecutando la máquina virtual. La última revisión estable del lenguaje y la plataforma Java es la versión 7, introducida por primera vez de manera estable en Junio de 2011; y estando ya en revisión el borrador de la versión 8 la cual se espera que esté disponible a mediados de 2013. La revisión más extendida y probada actualmente sigue siendo la versión 6, disponible desde 2006, y es la elegida para la realización de este

proyecto por su compromiso entre compatibilidad y características.

Gracias a la compatibilidad hacia atrás de las máquinas virtuales, es posible ejecutar código compilado para la versión 6 del lenguaje en máquinas cuya JVM se encuentra en la versión 7 u 8. Esto permite la implementación del programa en todo tipo de sistemas, desde los menos potentes o más preocupados por la seguridad, que presuntamente se encontrarían usando la versión 6 hasta los sistemas de última tecnología que tendrán las últimas revisiones de la máquina virtual. [3] [13]

3.2. Bases de datos

Toda aplicación que realice un uso intensivo de datos necesitará alguna forma de almacenar y operar con dichos datos. El estándar para el almacenamiento de gran cantidad de datos son los sistemas de gestión de bases de datos. Por base de datos entendemos una colección de datos relacionales, siendo dichos datos hechos conocidos que pueden registrarse y que tienen un significado implícito. Esta definición abarcaría desde el catálogo manual a base de fichas de una biblioteca al catálogo detallado automatizado de una tienda online.

Coloquialmente se usa el término Base de datos (BD) para referirse en realidad a los sistemas de gestión de bases de datos (SGBD). Un SGBD es una colección de programas que permite a los usuarios crear y mantener una base de datos. Un SGBD es por tanto un sistema que permite la definición, construcción y manipulación de bases de datos para distintas aplicaciones.

La definición de la base de datos consiste en especificar el tipo de los datos, las estructuras que los almacenarán así como sus restricciones. La construcción es entonces el proceso de almacenar los datos en algún medio controlado por el SGBD. Por último la manipulación de la base de datos incluye funciones para consultar, actualizar o eliminar datos así como generar informes a partir de ellos. Un sistema de gestión deberá proporcionar también fiabilidad en el almacenamiento de los datos, consistencia y seguridad en su acceso.

En la actualidad, los principales sistemas de gestión de bases de datos se pueden resumir en:

- Relacionales: Basadas en el modelo relacional, se caracterizan por el almacenamiento de los datos en tablas definidas funcionalmente por los datos y relacionadas entre sí. Generalmente usan el lenguaje de definición de datos SQL.

- Jerárquicas: Su principal característica es la organización de los datos en árboles, generando relaciones Padre-Hijo, en las que el padre puede tener múltiples hijos pero el hijo sólo un padre.

- En Red: Organiza los datos como registros, conectados entre sí por medio de enlaces. Es similar a la estructura jerárquica, pero añadiendo la posibilidad de que los hijos tengan múltiples padres.

- Orientadas a objetos: Surgidas tras la popularización de la programación orientada a objetos. Tratan de almacenar el objeto completo, incluyendo sus datos, estado actual y comportamiento. Las últimas revisiones del lenguaje SQL soportan este nuevo estándar.

- NoSQL: Se identifican por el no uso del lenguaje SQL. Fueron desarrolladas para el manejo de grandes cantidades de datos en sistemas distribuidos y ha sido largamente adoptadas por las grandes compañías de Internet, como Google o Facebook.

Los SGBD basados en el modelo relacional son los mas extendidos y conocidos y por ello se ha decido su uso. La organización de los datos en un sistema externo al programa en cuestión proporcionará mayor interoperabilidad con otros componentes de la empresa y un posible acceso a los datos generados por nuestro sistema a través de algún otro componente del sistema informático de la empresa.

Dentro de los sistemas relacionales, podemos dividirlos según su condición de sistemas comerciales o libres. Sistemas comerciales:

- Microsoft Access: Aparecida por primera vez en 1992, se encuentra disponible para Windows a través de la suite de aplicaciones Microsoft Office. Su mayor ventaja es la interoperabilidad con otros sistemas y lenguajes de Microsoft. La principal contra es que no está disponible para otros sistemas operativos.

- IMB DB2: Disponible desde 1983 y desarrollado por IBM para gran cantidad de sistemas operativos. Junto con Oracle y MS SQL son uno de los SGBD más utilizados.

- Oracle: Actual líder en gestión de bases de datos. Desarrollado desde 1979, está disponible para Windows y distintas variantes de Unix.

- Microsoft SQL Server: Desarrollado por Microsoft desde 1989, es el SGBD comerciales más usado en sistemas Windows.

Sistemas de código abierto

- MySQL: Su facilidad de uso y rapidez lo han convertido en el sistema libre más popular, con versiones para los SO Windows, Linux, Mac en incluso para móviles Symbian. Junto con PHP y Apache conforma el paquete de desarrollo web más utilizado.

- PostgreSQL: Principal competidor de MySQL. Disponible para gran cantidad de sistemas operativos. Preferido en un principio por desarrolladores provenientes de Oracle y SQL Server, su principal característica era la robustez, pecando un poco de lentitud en sus operaciones. Sus últimas versiones lo han puesto a la altura de sus competidores en este aspecto.

- SQLite: Sistema consistente en una pequeña librería embebible el código fuente de cualquier aplicación. Su principal característica es su pequeño tamaño y por ello es ampliamente usada en sistemas embebidos o como almacén de configuraciones en un gran número de aplicaciones.

Como los datos almacenados por la aplicación pueden ser útiles para otros sistemas de la empresa, se ha descartado el uso de SQLite, quedando PostgreSQL y MySQL como principales opciones. Se ha decidido utilizar esta última por haber sido utilizada con éxito con anterioridad. Las características de ambas son similares por lo que esta elección queda muchas veces reducida a preferencias personales. [7]

3.3. Acceso a la base de datos y persistencia

Para poder acceder a una base de datos desde un lenguaje de programación es necesaria la utilización de un driver específico para el sistema de bases de datos que estemos utilizando, suministrado por el vendedor e incompatible con los demás sistemas existentes, que proporciona un conjunto de funciones que permiten la conexión y manipulación de los datos almacenado. Para facilitar esta labor Sun desarrolló en 1996 la Java Database Connectivity (JDBC), una API para conectar aplicaciones escritas en Java a las bases de datos más populares. Esta API permite la conexión de manera estándar a cualquier base de datos soportada, significando que no sería ya necesario rescribir el código de acceso si se decide cambiar de sistema en mitad del desarrollo o incluso una vez implementado. También permite la ejecución de sentencias en SQL estándar y el proceso de los resultados.

Casi todas las aplicaciones requieren datos persistentes. La persistencia es uno de los conceptos fundamentales en el desarrollo de aplicaciones. Si un sistema de información no guardase sus datos cuando este es apagado, de poca utilidad sería su uso. Para poder guardar la información contenida en los objetos, es necesario realizar una conversión de Java a SQL y en sentido contrario cuando se quiere recuperar un objeto. Esta tarea puede llegar a complicarse demasiado con JDBC si tenemos muchos objetos o su construcción es compleja, debido a la incompatibilidad entre el paradigma de la programación orientada a objetos y el de las bases de datos relacionales. Para facilitarla se desarrollaron sistemas de mapeo objeto-relacional (ORM) que realizan la traducción entre la representación lógica del objeto y su forma atomizada capaz de ser, de forma automatizada usando metadatos

que describen la relación entre los objetos y las tablas de la base de datos. Estos sistemas tienen como ventaja la simplificación del acceso y modificación de los datos almacenados y su traducción a objetos pero traen también inconvenientes como un menor rendimiento y la posibilidad de acabar trabajando con bases de datos mal diseñadas si se les deja que realicen toda la traducción automáticamente.

Existen multitud de sistemas ORM, pero los más populares para Java son Hibernate, EclipseLink y Java Persistence API (JPA).

Java Persistence API es la solución estándar provista por Java y es usada como base para la implementación de la mayoría de frameworks ORM existentes en la actualidad. Desarrollado más tarde que sus competidores, sus características fueron basadas principalmente en aquellas de Hibernate, TopLink (precursor de EclipseLink) y Java Data Objects (JDO), pero una aplicación que utilice JPA no necesita la utilización de otras herramientas externas como Hibernate y se puede considerar tan independiente como cuando se realizaban las mismas operaciones gracias la interfaz JDBC. JPA especifica el lenguaje de consultas Java Persistence Query Language (JPQL), similar al SQL y es el cual permite realizar las consultas a la base de datos.

EclipseLink nació en 2007 gracias a la donación por parte de Oracle del código fuente de TopLink. TopLink fue creada 1990 como solución ORM para el lenguaje de programación Smalltalk y fue posteriormente portado a Java en 1998. Proporciona características similares a las de JPA, como un lenguaje de consultas, mapas objeto-relacionales o caché de objetos en memoria. En 2008, Sun lo utilizó como base para realizar la especificación de su Java Persistence API 2.0.

Hibernate, desarrollado en la actualidad por Jboss Inc, compañía perteneciente a Red Hat Enterprises, creadores de la famosa distribución Red Hat Linux, fue también uno de los primeros sistemas en aparecer. Su misión era mejorar la solución de persistencia provista por Java con sus Enterprise Java Beans (EJB).

De entre estos sistemas se decidió utilizar Hibernate por ser la solución propuesta más popular, con mayor cantidad de documentación y por dar comúnmente menos problemas que la plataforma nativa de Java. [16]

3.4. Aplicaciones distribuidas

Definimos una aplicación o sistema distribuido como un conjunto de máquinas autónomas interconectadas por una red que están equipadas con el software adecuado para coordinar la compartición de sus recursos. Los recursos compartidos pueden tratarse tanto de información como de recursos computacionales, como memoria o procesador, y pueden ser distribuídos y compartidos de diferentes maneras[10].

Aunque normalmente estén implicadas varias máquinas, en la actualidad es común también el diseño de aplicaciones dividas en varias capas pero cuya ejecución se limita al uso de una sola máquina. En este caso la división se realiza para obtener sistemas menos cohesionados, con capas intercambiables o fácilmente actualizables, en vez de buscar una mayor rapidez y capacidad de procesamiento.

De entre los tipos que se detallan a continuación se ha elegido el modelo cliente-servidor para permitir un desarrollo más sencillo y un despliegue más fácil en el entorno de la empresa.

3.4.1. Tipos de arquitectura de aplicaciones distribuidas

3.4.1.1. Modelo cliente-servidor

El modelo más frecuente en sistemas distribuidos es el modelo cliente-servidor, implementado en dos capas. En este modelo intervienen dos partes, una máquina cliente en la que el usuario solicita información y una segunda máquina que actúa como servidor, almacenando los datos y procesando el resultado de las solicitudes.

El cliente puede ocuparse de realizar transformaciones sobre los datos o puede limitarse simplemente a mostrarlos por pantalla, siendo el servidor el que se ocupa de realizar el procesado. La elección entre estas dos posibilidades se realiza según las características y limitaciones de la red o la capacidad de procesado del servidor.

El resto de sistemas distribuidos pueden considerarse una extensión del modelo cliente-servidor básico

3.4.1.2. Arquitectura de n-capas y n-niveles

Una primera extensión consiste en el modelo de 3 capas, ampliamente utilizado en la computación distribuida actual. Esta pensada para proporcionar un mejor soporte ante los cambios constantes de requisitos de los procesos de negocios actuales.

El cliente realiza una petición al servidor intermedio, el cual actúa también como cliente para trasladar esa petición al servidor final. Cuando éste devuelve los datos requeridos, el servidor intermedio realiza el procesado, teniendo en cuenta la lógica del negocio, y devuelve el resultado al cliente, que solamente se encargaría de mostrarlo.

Si durante el uso de la aplicación se produjese un cambio en el modelo de negocio y no se modificasen los casos de uso, sólo sería necesario realizar cambios en el nivel intermedio sin tener que tocar ni el modo de almacenamiento de los datos ni la presentación. Lo mismo ocurre si deseamos cambiar alguna de las otras capas, pudiendo hacerlo de manera

independiente.

Este modelo es ampliable al número de capas que deseemos, especializando cada una de ellas en un tipo de proceso en concreto, como pueden ser la lógica de negocio, lógica de presentación, acceso a los datos o peticiones a otros sistemas.

Junto con la división en capas, los sistemas pueden ser divididos en niveles, correspondientes normalmente a una maquina que se ocupa de la ejecución de las capas que le asignemos. Así, podríamos tener una aplicación de 3 capas y un nivel, que se ejecutaría en un único ordenador y un sistema de 4 capas y dos niveles en el que el cliente sólo mostraría los datos y el servidor se ocuparía del resto de capas. Es posible también realizar la división de cada nivel en un número de máquinas mayor a uno, obteniendo así una mayor redundancia en el proceso y permitiendo seguir funcionando si alguna de estas máquinas no se encuentra disponible en algún momento.

3.4.1.3. Entornos Grid

Los entornos grid surgieron de la necesidad de las empresas de realizar tareas altamente costosas computacionalmente, como podría ser el procesado de una enorme cantidad de datos, y para el que se necesitaría la adquisición y mantenimiento de potentes máquinas que serían infrautilizadas el resto del tiempo.

Para responder a esta necesidad surgieron compañías que se ocupan de alquilar sus sistemas a las compañías que necesiten hacer uso de ellos. Estos sistemas consisten en una red de ordenadores que trabajan juntos y que se pueden considerar como un único sistema altamente integrado y conocido como *cluster*. La unión de varias máquinas en un único *cluster* produce como resultado un sistema de mayor rendimiento y disponibilidad que si se usaran los ordenadores separadamente, al mismo tiempo que resultan más económicos que adquirir una máquina que ofrezca un rendimiento similar.

3.4.1.4. Peer to Peer

El término Peer to Peer (P2P) define una red de ordenadores en los que ninguno de ellos actúa únicamente como cliente o servidor. Cada nodo de la red realiza ambas tareas a la vez, actuando como cliente al solicitar servicios a otros nodos y como servidor, proporcionando el mismo servicio.

La no existencia de un nodo u organización central ha hecho que estos modelos sean ampliamente utilizados para crear redes libres, escalables, autoorganizadas y anónimas para el intercambio de información. Su mayor defecto es la calidad del servicio, ya que es posible que la máquina a la que queramos solicitar el servicio no se encuentre disponible

o que los datos que estemos buscando no estén siendo proporcionados por ninguno de los nodos en ese momento.

3.5. Interfaces de Usuario

La interfaz gráfica de usuario consiste en una capa de una aplicación que ofrece una represanción visual de información y las acciones disponibles para realizar sobre ellos, utilizando un conjunto de imágenes y objetos gráficos.

Surgen como evolución de las interfaces de texto de línea de comandos con la intención de facilitar la interacción del usuario con el ordenador.

El lenguaje Java ofrece entre sus librerías por defecto la librería gráfica Swing, encargada de proporcionar una interfaz de programación (API) para el desarrollo de interfaces gráficas de usuario que hacen uso del patrón modelo-vista-controlador, el cual separa conceptualmente la visión de lo datos de su almacenamiento y procesado.

Swing fue desarrollado como evolución del Advanced Window Toolkit (AWT) para permitir el desarrollo de grandes aplicaciones empresariales con un gran número de componentes y proporciona un diseño nativo integrado con todos los sistemas operativos soportados sin necesidad de escribir código específico para ellos.

Los componentes necesarios para el desarrollo de una interfaz de usuario, como pueden ser etiquetas, botones o tablas son proporcionados por la propia librería, con un comportamiento estándar pero configurable para permitir refinar su funcionamiento y adaptarlos a las necesidades de cualquier diseño. También proporciona paneles y ventanas en los que colocar dichos componentes y formar el diseño final de la interfaz.

Junto con Swing existen otras librerías que ofrecen funcionalidades similares como puede ser el Standard Widget Toolkit (SWT), desarrollado en sus inicios por IBM y mantenido en la actualidad por la comunidad Eclipse y pensado como un conjunto de librerías complementarias a AWT, o el Google Web Toolkit (GWT), que proporciona herramientas para desarrollar interfaces en código Javascript para aplicaciones realizadas en Java.

Los propios desarrolladores del lenguaje Java y de Swing han creado también la librería JavaFX, especializada en la creación de interfaces para aplicaciones web que funcionen tanto en un navegador como en un sistema de escritorio.

Sin embargo Swing se mantiene como estándar en el desarrollo de aplicaciones de escritorio multisistema por su facilidad y disponibilidad. SWT ha ido evolucionando hasta obtener funcionalidades y rendimiento similar al de Swing pero se ha especializado en permitir un mayor control y complejidad en los componentes del sistema antes de una mayor automatización en el desarrollo y la falta de necesidad de código específico para los

sistemas en los que queremos que nuestro sistema esté disponible.

Se escogió la librería Swing por su facilidad de diseño visual haciendo uso del editor integrado en Netbeans, su capacidad de funcionamiento en distintos sistemas operativos y estar pensada para el desarrollo de aplicaciones de escritorio. [5]

Capítulo 4

Metodología

4.1. El proceso de desarrollo del software

Un proceso se define como la colección de actividades de trabajo, acciones y tareas que se realizan cuando va a crearse algún producto terminado. Una estructura general para la ingeniería del software se puede dividir en cinco actividades principales: *comunicación*, *planificación*, *modelado*, *construcción* y *despliegue*. Además, a lo largo del todo el proceso se realizarán también actividades complementarias o *sombrilla*, como pueden ser el seguimiento del proyecto, administración de riesgos, revisiones técnicas o administración de configuraciones. [14]

Tradicionalmente, el desarrollo del software seguía un patrón de codificación directa, consistente en programar y arreglar continuamente, sin seguir ningún orden. Para intentar ordenar el caos en el que consistía este proceso, se propusieron una serie de modelos de desarrollo o ciclos de vida que pretenden organizar las distintas etapas del desarrollo, indicando qué se debería hacer en cada momento.

Los principales modelos de proceso son:

Modelo en cascada Conocido también como el ciclo de vida clásico, sugiere un enfoque secuencial para el desarrollo del software, comenzando con la especificación de requisitos y continuando con la planificación, modelado, construcción y despliegue. Hace énfasis en la especificación temprana de requisitos y la completud de cada paso antes de pasar al siguiente, siguiendo un modelo prácticamente lineal.

Modelo incremental Divide el programa en una serie de trozos o incrementos, que engloban funcionalidades completas y podrían ser susceptibles de ser entregados como programa final. Se comienza desarrollando el producto fundamental, que contendrá los

requisitos básicos, y una vez terminado, se comprueba el resultado y se diseña un plan para desarrollar el siguiente incremento. Cada incremento puede ser desarrollado siguiendo cualquier otra metodología de proceso, aunque es común que se aplique el ciclo de vida en cascada. Este proceso se repite hasta que se termine el producto final.

Modelo evolutivo También conocido como prototipado, basa su existencia en la noción de que el software, como todo sistema complejo, evoluciona con el tiempo. Es frecuente que los requisitos cambien a lo largo del desarrollo y puede no ser realista trazar una trayectoria directa hacia el producto final. El modelo se caracteriza por desarrollar de manera rápida versiones cada vez más completas del software, que serán evaluadas por el cliente una vez terminadas para identificar el cumplimiento de los requisitos planteados y la identificación de nuevas necesidades, al contrario del modelo incremental, en el que ya se conocía con anterioridad lo que se va a realizar en la siguiente fase.

Modelo en espiral Es una variante del modelo evolutivo. Desarrolla el software en una serie de entregas evolutivas, que comienzan durante las primeras iteraciones como modelos o prototipos, para posteriormente convertirse en versiones cada vez más completas del producto final.

Modelos concurrentes Realiza todas las actividades del desarrollo del software al mismo tiempo, no necesitando terminar su actividad actual para cambiar a otro de los estados del desarrollo. Esto hace posible que se pueda volver a comunicar con el cliente o realizar una prueba de la implementación en mitad de un ciclo de modelado sin necesidad de esperar a que se hayan cumplido todas las tareas a realizar, retomándolo de nuevo en el mismo punto en el que se había abandonado.

Modelos ágiles Se caracterizan por realizar etapas muy cortas de desarrollo, entregando con frecuencia software funcional y promoviendo en todo momento la comunicación entre todos los miembros del equipo, lo que favorece la retroalimentación de todas las etapas del proceso. Pretende obtener un mayor rendimiento del equipo, centrando el trabajo en pequeños objetivos realizables en un corto espacio de tiempo.

Para el desarrollo de este proyecto se ha decidido seguir una metodología de desarrollo iterativa, siguiendo un modelo incremental y aplicando el ciclo de vida en cascada para la realización de cada etapa, como la que se puede apreciar en la Figura 4.1.

Una vez obtenidos los requisitos del sistema se ha procedido a dividir su implementación en etapas. Tras las primeras etapas, en la que se definirán los modelos básicos del programa,

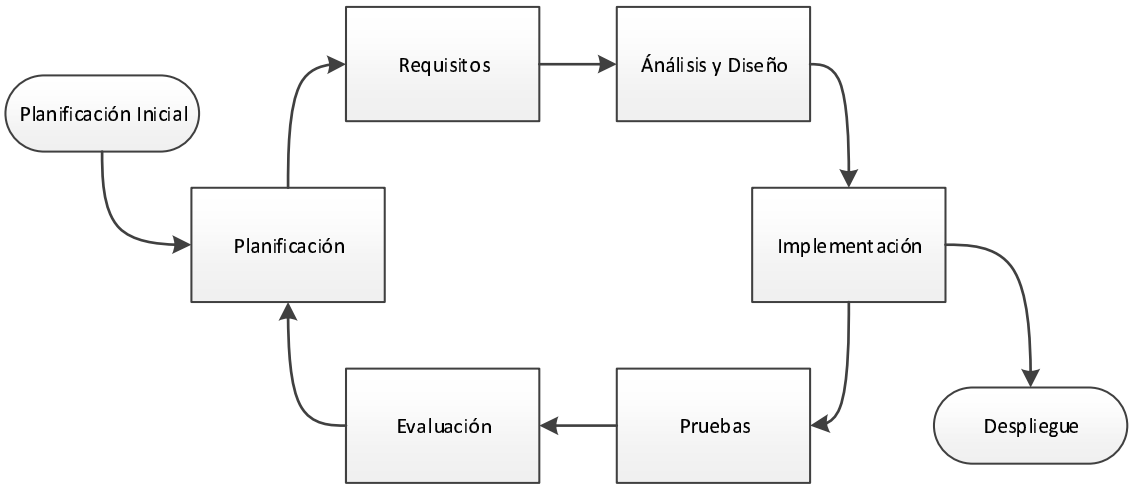

Figura 4.1: Metodología de desarrollo iterativa

se pasará a desarrollar un nuevo caso de uso, modelándolo y diseñando una nueva interfaz que permita realizarlo de manera gráfica. Una vez terminada cada etapa se ha probado su funcionamiento y pasado a evaluar si esta cumplía con los requisitos establecidos, volviendo a iterar sobre el mismo caso si no los cumpliera. Una vez terminado, se pasa a la siguiente etapa hasta que se complete la implementación de todos los casos de uso y el sistema pueda ser desplegado.

4.2. Planificación Inicial

4.2.1. Análisis de requisitos

Los requisitos funcionales del sistema deberán derivarse de las actividades que realice la empresa de reparación de vehículos en su día a día. Por ello, podemos afirmar que a la empresa acuden cada día *clientes*, que poseen *vehículos* y que habrán de ser evaluados para averiguar si es necesario realizar alguna *reparación* y en ese caso, *presupuestarla*. Estas reparaciones necesitarán de *piezas* que habrán de ser *pedidas* a los distintos *proveedores* de la empresa y guardadas en un *almacén*. Las reparaciones serán realizadas por *empleados* pertenecientes al taller y abonadas por el cliente o la *aseguradora* del vehículo, a quienes se les presentará una *factura* que detalle los trabajos realizados, así como el precio de la *mano de obra* y cualquier otra eventualidad.

Una vez averiguados todos los datos que habrá de almacenar el sistema, se necesita saber qué otras cualidades serán necesarias para su funcionamiento. Así pues se decide que el entorno habrá de permitir la conexión de *usuarios*, que tendrán determinados *permisos* que limitarán las acciones que puedan realizar dentro de la aplicación y que serán agrupados

en *roles*.

A continuación se detallan todos los requisitos averiguados hasta ahora junto con las cualidades deseables del software

4.2.1.1. Requisitos funcionales

Indica las posibilidades que deberá ofrecer la aplicación a sus usuarios.

- Almacenamiento y modificación de información: El sistema habrá de almacenar información de clientes, vehículos, aseguradoras, las reparaciones que se les realice y sus presupuestos, junto con la mano de obra y las piezas a utilizar. También se necesitaran guardar los pedidos y proveedores a los que se realizan, los almacenes donde se guardan, así como los empleados a cargo de la empresa y las facturas producidas por las reparaciones y pedidos.

- Creación y conexión de usuarios: La aplicación deberá permitir la definición y conexión de usuarios al sistema, indicado una serie de acciones o permisos que podrán realizar en el sistema.

- Conexión de múltiples usuarios: El sistema deberá permitir el trabajo concurrente de varios usuarios, que podrán realizar modificaciones a los mismos datos, ocupándose el sistema de averiguar si se produce algún conflicto e intentando resolverlo.

- Impresión de documentos: Se deberá ofrecer la posibilidad de imprimir la información mostrada en pantalla.

4.2.1.2. Requisitos no-funcionales

Otras cualidades necesarias del software que no influyen en las posibilidades que este ofrece.

- Soporte multiplataforma: El sistema podrá desplegarse en la mayor parte de sistemas operativos disponibles en la actualidad.

- Facilidad de uso: El sistema podrá ser usado con facilidad por cualquier empleado sin necesidad de un gran periodo de adaptación.

- Interfaz gráfico: Se ha de desarrollar un interfaz de usuario que permita la introducción y modificación de toda la información detallada anteriormente.

4.2.1.3. Reglas de negocio

Particularidades de trabajo de la empresa, que deberán ser reflejados en el desarrollo del sistema.

- Los clientes y vehículos habrán de estar registrados en el sistema antes de comenzar una reparación.
- Los presupuestos estarán ligados a una orden de reparación, pudiendo existir únicamente uno por orden.
- Las facturas irán enlazadas con sus reparaciones o pedidos, existiendo solamente una por cada pedido o reparación realizada.
- Los empleados recibirán un nombre de usuario que les permitirá conectarse al sistema y realizar el trabajo que se les haya asignado.
- Toda información almacenada en el sistema podrá ser desactivada, manteniéndose guardada pero indicando que su estado no es correcto.
- Toda transacción monetaria podrá recibir descuentos o recargos.
- Será posible editar la moneda utilizada, así como los impuestos aplicables a las transacciones.

4.2.2. Definición de casos de uso

Una vez averiguados todos los requisitos y cualidades que habrá de proporcionar el sistema, el siguiente paso en el desarrollo del proyecto es identificar las interacciones y necesidades del usuario.

Como ya hemos visto, el sistema tendrá que permitir almacenar y modificar información, por lo que tendremos un caso de uso que consistirá en *dar de alta un registro en el sistema*, otro que permita *modificar dicho registro* y por último, también *eliminarlo*. Habrá de ser posible también la *consulta de la información almacenada*. Al existir usuarios, el sistema habrá de permitir la *conexión de usuarios* y, por tanto, su *desconexión*.

Tras haberlos identificado, se puede consultar la lista completa de casos de uso en el Cuadro 4.1.

Grupo	Clase	Acción	Identificador
Empleados	Empleado	Alta	CU.01
		Baja	CU.02
		Modificación	CU.03
		Consulta	CU.04
Clientes	Cliente	Alta	CU.05
		Baja	CU.06
		Modificación	CU.07
		Consulta	CU.08
	Vehículo	Alta	CU.09
		Baja	CU.10
		Modificación	CU.11
		Consulta	CU.12
	Aseguradora	Alta	CU.13
		Baja	CU.14
		Modificación	CU.15
		Consulta	CU.16
Piezas	Pieza	Alta	CU.17
		Baja	CU.18
		Modificación	CU.19
		Consulta	CU.20
	Stock	Alta	CU.21
		Baja	CU.22
		Modificación	CU.23
		Consulta	CU.24
	Proveedores	Alta	CU.25
		Baja	CU.26
		Modificación	CU.27
		Consulta	CU.28
	Pedidos	Alta	CU.29
		Baja	CU.30
		Modificación	CU.31
		Consulta	CU.32
	Factura de pedidos	Alta	CU.33
		Baja	CU.34
		Modificación	CU.35
		Consulta	CU.36

	Almacén	Alta	CU.37
		Baja	CU.38
		Modificación	CU.39
		Consulta	CU.40
Reparaciones	Reparación	Alta	CU.41
		Baja	CU.42
		Modificación	CU.43
		Consulta	CU.44
	Presupuesto	Alta	CU.45
		Baja	CU.46
		Modificación	CU.47
		Consulta	CU.48
	Factura de reparación	Alta	CU.49
		Baja	CU.50
		Modificación	CU.51
		Consulta	CU.52
Usuarios	Usuario	Conectar	CU.53
		Desconectar	CU.54
		Alta	CU.55
		Baja	CU.56
		Modificación	CU.57
		Consulta	CU.58
	Rol	Alta	CU.59
		Baja	CU.60
		Modificación	CU.61
		Consulta	CU.62

Cuadro 4.1: Casos de uso

4.2.2.1. Actores

Los actores del sistema serán los **empleados** de la empresa. Ellos se encargarán de introducir los datos en el sistema y de realizar cualquier modificación que sea necesaria.

A modo de ejemplo se puede suponer que la empresa consista de al menos un **recepcionista** o secretario, un **mecánico**, un **encargado del almacén** y un **administrador** del sistema.

Se habrá de considerar al **cliente** como un actor entre bastidores, puesto que aunque no llegue a introducir directamente sus datos, interviene en el proceso aportando su do-

cumentación.

También tendremos que tener en cuenta a los **proveedores**, que deberán proporcionar su información de contacto para que sea posible realizar los pedidos.

4.2.3. Descripción de casos de uso

4.2.3.1. Conectar usuario

Identificador CU53.

Descripción Un usuario inicia sesión en el sistema.

Actores Administrador, recepcionista, mecánico o encargado del almacén.

Precondiciones El usuario existe en el sistema.

Disparador Se ha arrancado el programa o un usuario ha iniciado la acción de cambiar de usuario.

Escenario

- El actor introduce su nombre y contraseña.
- El sistema comprueba su validez y le permite conectarse.

Excepciones

- El nombre de usuario no existe o la contraseña no es correcta: el sistema indica al actor que los datos proporcionados no son correctos y no le deja conectarse.

Postcondiciones El usuario queda conectado en el sistema.

Notas Una vez conectado, el sistema habrá de comprobar sus permisos y permitirle iniciar solamente los casos de uso que su rol pueda realizar.

El sistema no permitirá realizar ningún caso de uso si no hay un usuario conectado.

4.2.3.2. Desconectar usuario

Identificador CU54

Descripción Un usuario finaliza su sesión en el sistema.

Actores Administrador, recepcionista, mecánico o encargado del almacén.

Precondiciones El usuario a desconectar está conectado al sistema.

Disparador El actor le pide al sistema que finalice su sesión.

Escenario

- El sistema comprueba que no exista cambios sin guardar.
- El sistema desconecta al usuario.

Excepciones

- El actor ha hecho cambios en el sistema pero no los ha guardado: El sistema pregunta si se desean guardar los cambios pendientes o descartarlos antes de desconectar al usuario.

Postcondiciones El usuario ha sido desconectado del sistema.

4.2.3.3. Alta de un registro

Aplicable a la acción de dar de alta un nuevo *cliente*, *vehículo*, *aseguradora*, *reparación*, *pieza*, *mano de obra*, *pedido*, *proveedor*,*almacén*, *empleado*, *usuario* o *rol*.

Identificador CU1, CU5, CU9, CU13, CU17, CU21, CU25, CU29, CU33, CU37, CU41, CU45, CU49, CU55, CU59.

Descripción Un usuario del sistema da de alta un nuevo registro en el sistema.

Actores Cliente, proveedor, administrador, recepcionista, mecánico o encargado del almacén.

Precondiciones El actor tiene un usuario conectado al sistema y tiene permiso para crear registros del tipo deseado.

Disparador El actor accede al navegador correspondiente al tipo de registro a añadir y pulsa en nuevo registro.

Escenario

- El actor introduce los datos requeridos por la aplicación.
- El sistema valida los datos y da de alta el nuevo registro.

Excepciones

- Algún dato requerido no ha sido cubierto: el sistema le indica al actor que ha dejado datos obligatorios sin cubrir y no da de alta el registro.
- Algún dato es incorrecto: el sistema le indica al actor que debe verificar y corregir los datos equivocados y no da de alta el registro.
- El sistema detecta que el registro ya estaba dado de alta: el sistema le indica al actor que el registro que está intentando dar de alta ya existe y no lo da de alta.

Postcondiciones El nuevo registro queda guardado en el sistema.

Notas Los datos a guardar dependerán de la naturaleza de cada tipo de registro.

Si el registro tiene un número de identificación público, el sistema podrá asignarle uno automáticamente si el actor no ha introducido uno.

4.2.3.4. Modificar registro

Aplicable a la acción de modificar un *cliente*, *vehículo*, *aseguradora*, *reparación*, *presupuesto*, *factura de reparación*, *pieza*, *mano de obra*, *pedido*, *factura de pedido*, *proveedor*,*almacén*, *empleado*, *usuario* o *rol*.

Identificador CU3, CU7, CU11, CU15, CU19, CU23, CU27, CU31, CU35, CU39, CU43, CU47, CU51, CU57, CU61.

Descripción Un usuario del sistema modifica datos existentes en el sistema.

Actores Cliente, proveedor, administrador, recepcionista, mecánico o encargado del almacén.

Precondiciones El actor tiene un usuario conectado al sistema y tiene permiso para modificar el registro deseado. El registro a modificar existe en el sistema.

Disparador El actor accede al navegador correspondiente al tipo de registro que desea modificar.

El actor busca el registro mediante el uso del navegador.

El actor selecciona el registro a modificar y pulsa sobre el botón editar.

Escenario

- El actor modifica los datos deseados.
- El sistema valida los datos y actualiza el registro.

Excepciones

- Algún dato requerido no ha sido cubierto: el sistema le indica al actor que ha dejado datos obligatorios sin cubrir y no da de alta el registro.
- Algún dato es incorrecto: el sistema le indica al actor que debe verificar y corregir los datos equivocados y no da de alta el registro.
- El sistema detecta que el registro ya estaba dado de alta: el sistema le indica al actor que el registro que está intentando dar de alta ya existe y no lo da de alta.
- El sistema detecta que otro usuario ha realizado cambios en el registro después de que este se haya leído: el sistema le indica al actor que otro usuario ha hecho modificaciones y le pregunta si desea sobrescribir o descartar los cambios.

Postcondiciones Los cambios en los datos del registro quedan reflejados en el sistema.

4.2.3.5. Baja de un registro

Aplicable a la acción de borrar un *cliente*, *vehículo*, *aseguradora*, *reparación*, *pieza*, *mano de obra*, *pedido*, *proveedor*,*almacén*, *empleado*, *usuario* o *rol*.

Identificador CU2, CU6, CU10, CU14, CU18, CU22, CU26, CU30, CU34, CU38, CU42, CU46, CU50, CU56, CU60.

Descripción Un usuario elimina un registro del sistema.

Actores Cliente, proveedor, administrador, recepcionista, mecánico o encargado del almacén.

Precondiciones El actor tiene un usuario conectado al sistema y tiene permiso para modificar el registro deseado. El registro a dar de baja existe en el sistema.

Disparador El actor accede al navegador correspondiente al tipo de registro que desea modificar.

El actor busca el registro mediante el uso del navegador.

El actor selecciona el registro a eliminar y pulsa sobre el botón borrar.

Escenario

- El sistema pide confirmación de la acción.
- El actor confirma.
- El sistema borra al registro.

Excepciones

- El actor decide no borrar el registro. No confirma la acción y el sistema no realiza ningún cambio.
- Si el registro está referido en el sistema por cualquier otro registro, el sistema indicará que no se puede borrar sin haber eliminado antes los conflictos.
- Si el registro posee otros registros subordinados que no están referidos por ningún otro elemento del sistema, serán también eliminados.

Postcondiciones El registro ha sido borrado del sistema.

4.2.4. División en etapas

Una vez descritos y definidos todos los casos de usos principales, se dividen en grupos y a cada grupo se le asigna una etapa del desarrollo. Así en la primera etapa se desarrollará la arquitectura del sistema y el modelo básico de datos, para continuar en la segunda con la definición de la base de datos y su integración con el modelo. En la siguiente etapa se construirá la interfaz de usuario básica y se implementarán los casos de uso relacionados con los empleados, así como todas las funciones necesarias para almacenarlos en la base de datos. La siguiente consistiría en realizar la interfaz de usuario para los clientes y así sucesivamente con el resto de grupos como se puede apreciar en la Figura 4.2.

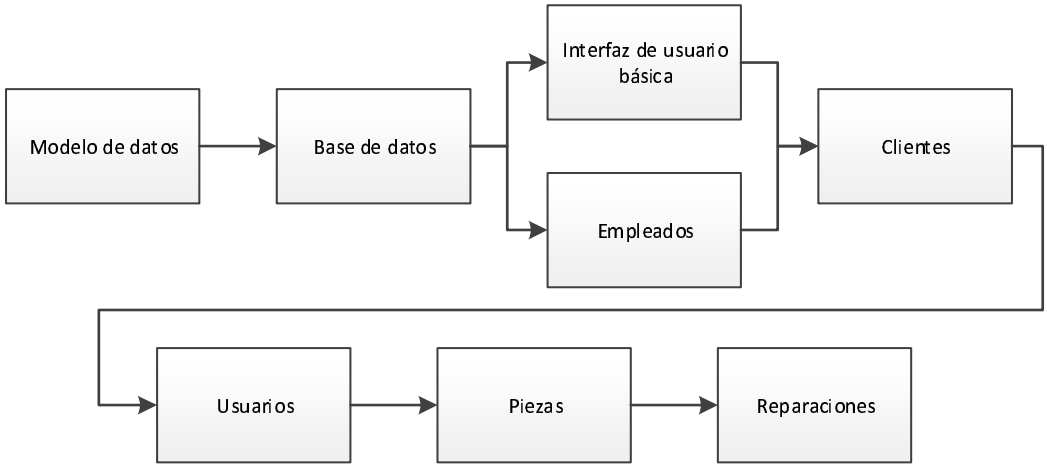

Figura 4.2: Etapas del desarrollo

4.3. Primera etapa: Arquitectura y modelo de datos

4.3.1. Requisitos

El modelo ha de almacenar usuarios, roles y permisos; empleados; clientes, vehículos y aseguradoras; piezas, stock, proveedores, pedidos y facturas; reparaciones, presupuestos y facturas.

4.3.2. Definición del modelo conceptual

El primer paso en la definición de un nuevo modelo de datos consiste en desarrollar el modelo conceptual del sistema. Con este modelo se realiza una descripción de la realidad con un alto nivel de abstracción, que permite entender con facilidad el sistema y proporciona una manera sencilla de identificar y organizar las clases que son relevantes para los

requisitos de uso del producto, así como sus asociaciones y dependencias[14].

4.3.2.1. Esquema conceptual

El esquema conceptual del sistema se puede consultar en la Figura 4.3. Consiste en la definición de los elementos que componen el modelo de trabajo de un taller y cómo se relacionan entre ellos. Nos da una idea global del personal involucrado y la información que intercambian y que necesitaremos almacenar.

4.3.3. Arquitectura de la aplicación

Como se ha explicado anteriormente, se ha decido realizar la aplicación con arquitectura cliente-servidor 2-tier, utilizando MySQL como servidor de bases de datos y realizando la aplicación en código Java bajo la Netbeans Platform.

En la Figura 4.4, se pueden apreciar las distintas partes que conformarán el sistema final, que consistirá en un servidor de bases de datos y la aplicación, que se dividirá en una serie de módulos definidos en la sección 4.3.3.1.

4.3.3.1. Diagrama de módulos y paquetes

Como se describe en el apartado 4.5.1, uno de los beneficios del uso de la *Plataforma Netbeans* es la facilidad de división de un programa en pequeños módulos poco acoplados entre sí que proporcionan una mayor cohesión al sistema.

Ya que numerosos procesos harán uso de Hibernate, se ve necesaria la creación de un módulo que contendrá todas las librerías necesarias para el correcto funcionamiento de Hibernate y los usuarios, contraseñas y configuración necesaria para el correcto acceso a la base de datos. Este módulo será el llamado **Persistency** y de él dependerán todos los DAMs (*Módulo de Acceso a Datos*) del sistema.

De manera similar, durante el desarrollo de la aplicación se apreció que varios módulos compartirían la necesidad de utilizar librerías auxiliares para la realización de ciertas tareas, como la validación de números de teléfono, Números de Identificación Fiscal o la impresión de documentos. Por tanto se decide crear el módulo **SharedLibraries** (*Librerías Compartidas*) que las agrupe y el módulo **Reports** (*Informes*) que proporcionará métodos de generación, visualización e impresión de documentos.

Así mismo, para la gestión del login de usuarios es necesaria la creación de un módulo que agrupe los métodos a utilizar en la comprobación de credenciales de usuario. Este módulo se llamará **Login**, será el primero en ejecutarse al iniciar la aplicación y de él

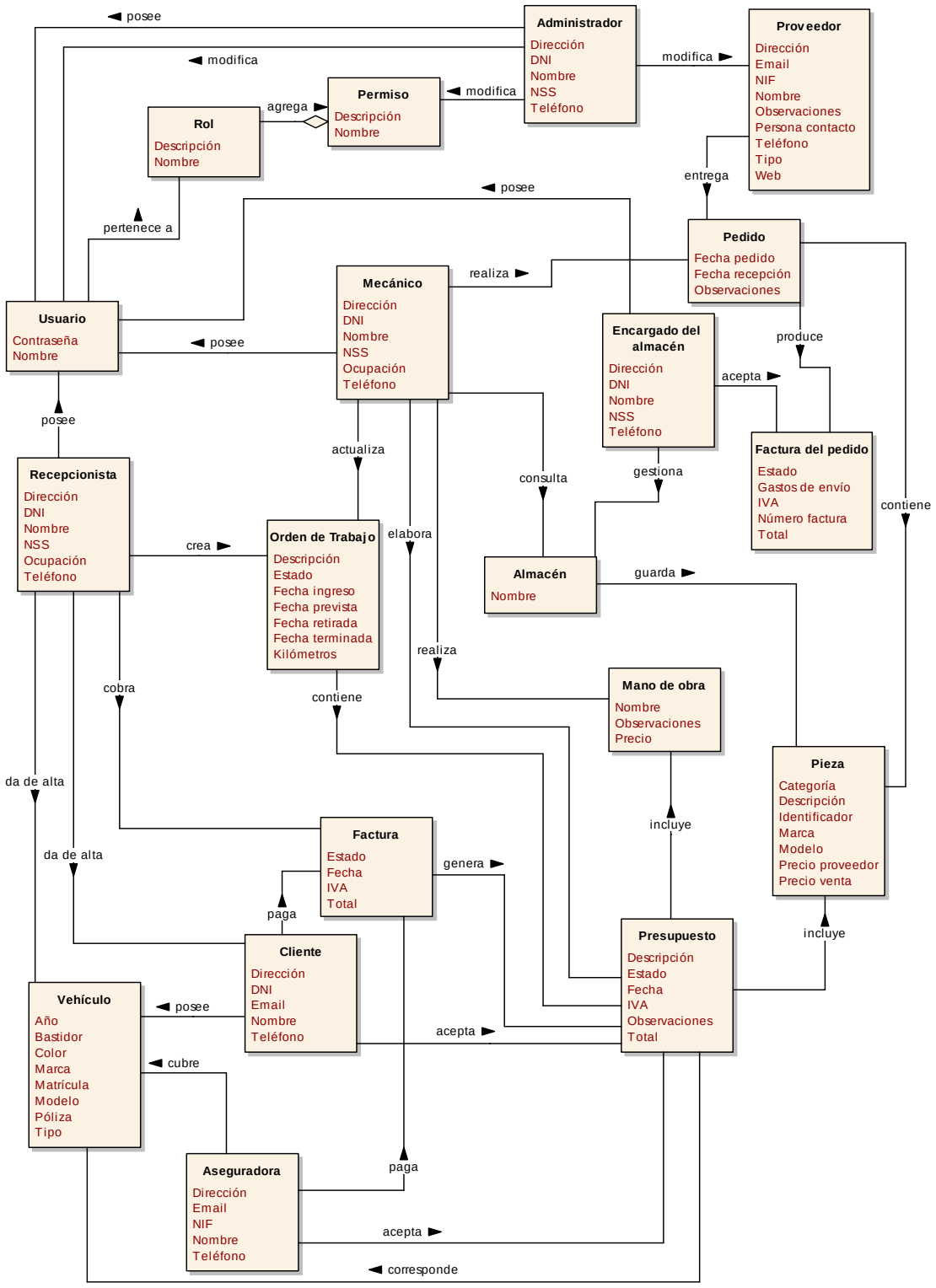

Figura 4.3: Modelo conceptual

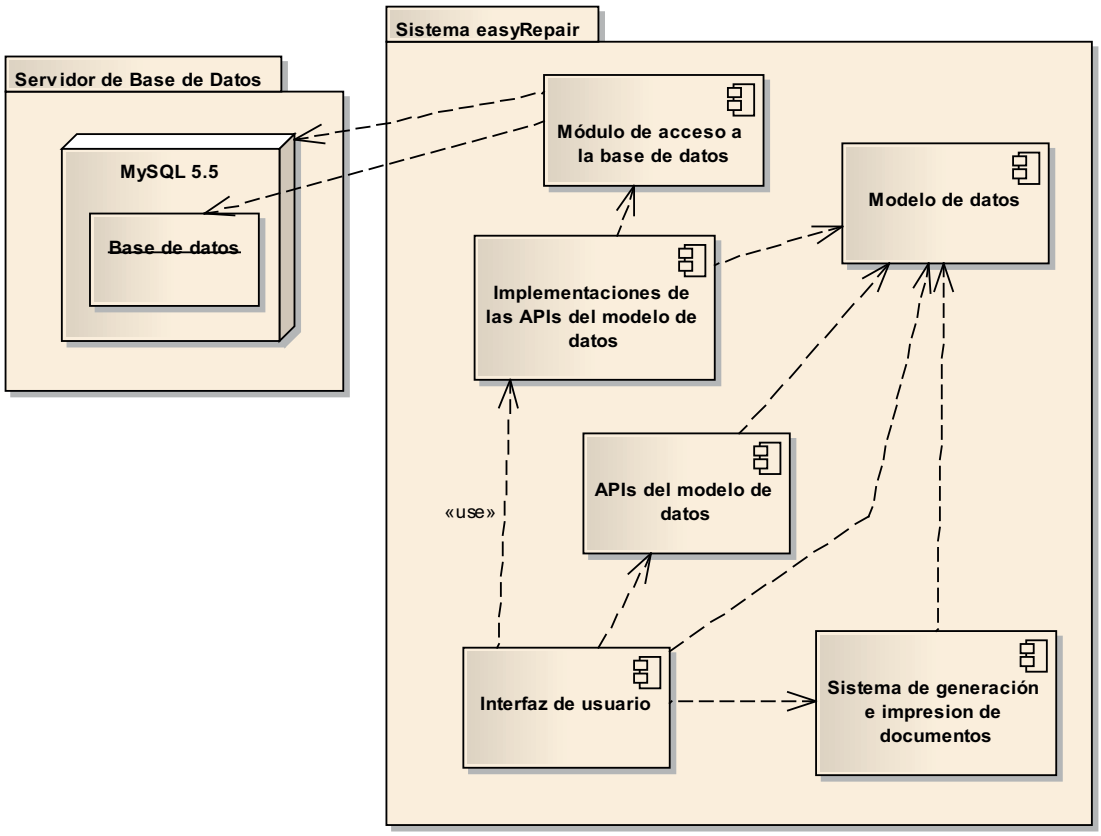

Figura 4.4: Arquitectura del sistema

deprenderán todas las interfaces de usuario

Aprovechando la infraestructura que provee Netbeans para la prestación de servicios, se decide crear un módulo que provea solamente de la interfaz de acceso a datos, sin ninguna implementación concreta de sus funcionalidades, y del que dependerán los módulos que necesiten realizar el acceso a estos datos, llamado en el caso de los empleados, **EmployeeAPI** y por otro lado un módulo que proveerá la implementación concreta para el acceso a los datos almacenados en una base de datos MySQL a través de Hibernate y llamado **EmployeeHibernateDAM**, que será proporcionado automáticamente por Netbeans a los módulos que pidan acceso a los datos de dichos empleados. Así, si en un futuro decidiéramos cambiar de gestor de bases de datos, de su modo de acceso o incluso se quisieran almacenar los datos directamente en archivos de texto, sólo sería necesario desarrollar un nuevo módulo y hacerle saber al sistema que esta disponible, mediante un método que Netbeans llama *registro de módulos*, sin necesidad de cambiar la API, el modelo de datos o los módulos de la interfaz de usuario ni sus dependecias.

Por último, será necesaria la creación de una interfaz de usuario que agrupe las funcionalidades necesarias para el acceso y modificación de los datos y que en el caso de los

empleados se llamaría **EmployeeUI**

En la Figura 4.5 se puede apreciar la división en módulos del sistema y a continuación, un pequeño resumen de lo definido hasta ahora:

- **DataModel**, se corresponde con el modelo de datos del dominio. Incluye todas las entidades de las que se guardará información en el sistema.
- **SharedLibraries**, agrupa las librerías open-source utilizadas por buena parte de los demás módulos y la gestión de opciones del programa.
- **Reports**, contiene las librerías necesarias para la creación e impresión de documentos, así como las clases requeridas para hacer uso de ellas desde otros módulos.
- **Login**, se ocupa de mostrar el cuadro de diálogo de conexión de usuarios y de validar los datos introducidos en él, además de la gestión de permisos del usuario que haya iniciado sesión.
- **Persistency**, proporciona la conexión a la base de datos MySQL, haciendo uso de las librerías de Hibernate.
- **ClientAPI**, **EmployeeAPI**, **RepairAPI**, **StockAPI** y **UserAPI**, indican la interfaz que deberán seguir los módulos que realizarán el guardado y posterior lectura de los datos. Gracias a la Netbeans Platform, los módulos de la interfaz de usuario podrán depender directamente de las API y no de la implementación concreta de los sistemas.
- **ClientHibernateDAM**, **EmployeeHibernateDAM**, **RepairHibernateDAM**, **StockHibernateDAM** y **UserHibernateDAM**, constituyen una implementación de las interfaces anteriores que permite la conexión con bases de datos MySQL a través de Hibernate y el almacenamiento y lectura de la información del sistema.
- **ClientUI**, **EmployeeUI**, **RepairUI**, **StockUI** y **UserUI**, contienen la interfaz de usuario, realizada en Swing.

4.3.4. Modelo de datos

Del estudio del esquema conceptual se pueden extraer las entidades que el sistema deberá conocer y las relaciones entre ambas. Después, se habrán de organizar en grupos que se convertirán en los paquetes Java de la aplicación.

Este modelo de datos será almacenado en un módulo de Netbeans, que constituirá una de las bases de nuestro sistema y del que dependerán todos aquellos módulos que necesiten hacer uso directo de los datos.

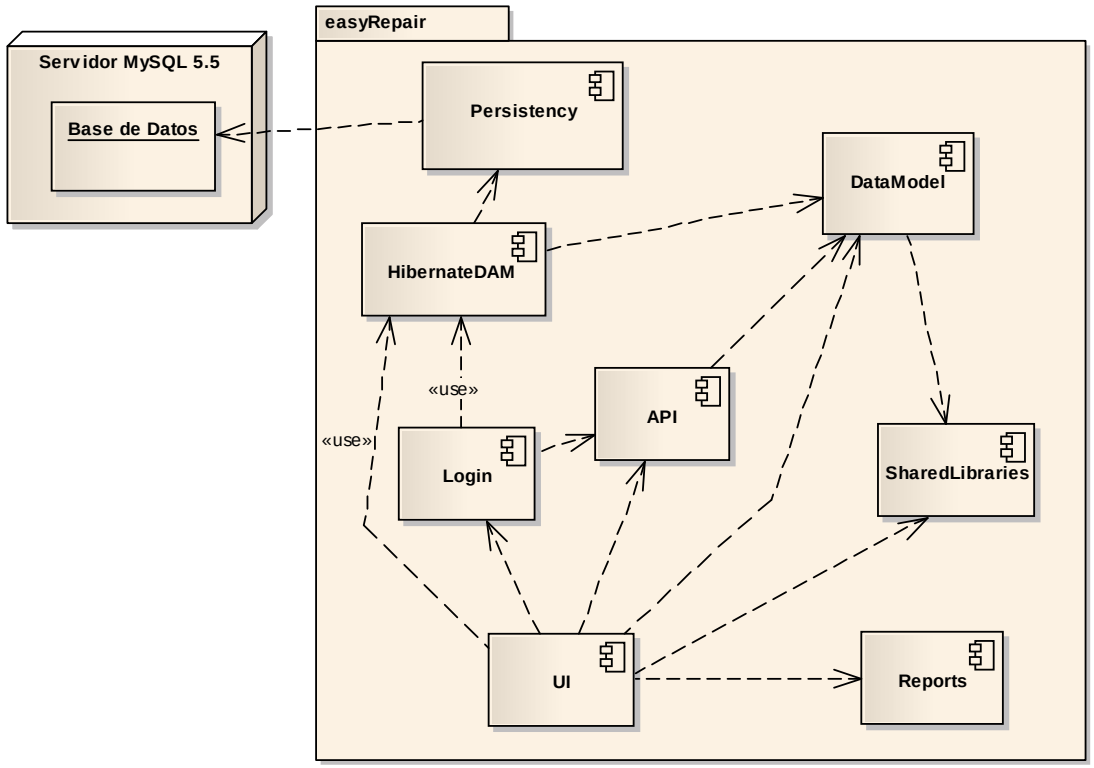

Figura 4.5: Diagrama de módulos

Las clases básicas fueron llamadas clases compartidas y consisten en un grupo de clases que serán agregadas para construir las entidades principales del sistema. Por ejemplo, un empleado que definiremos luego sería una persona, con dirección, teléfono y email.

Clases compartidas (Figura 4.6), almacenadas en el paquete **es.regueiro.easyrepair.model.shared**:

- **Contact:** Un contacto, compuesto como una entidad con *NIF*, *Teléfono*, *Email* y *Dirección*.
- **Company:** Una compañía o empresa en general. Es un *Contacto* con nombre.
- **Person:** Una persona. Consiste en un *Contacto* con nombre y apellidos.
- **Address:** Una dirección postal. Se almacenan todos los campos que definen una dirección, junto con una etiqueta identificadora y posibles notas.
- **Email:** Una dirección de email. Junto con la dirección se almacenan una etiqueta y notas.
- **Phone:** Un número de teléfono, además de la etiqueta y notas.
- **NIF:** Un Número de Identificación Fiscal español.

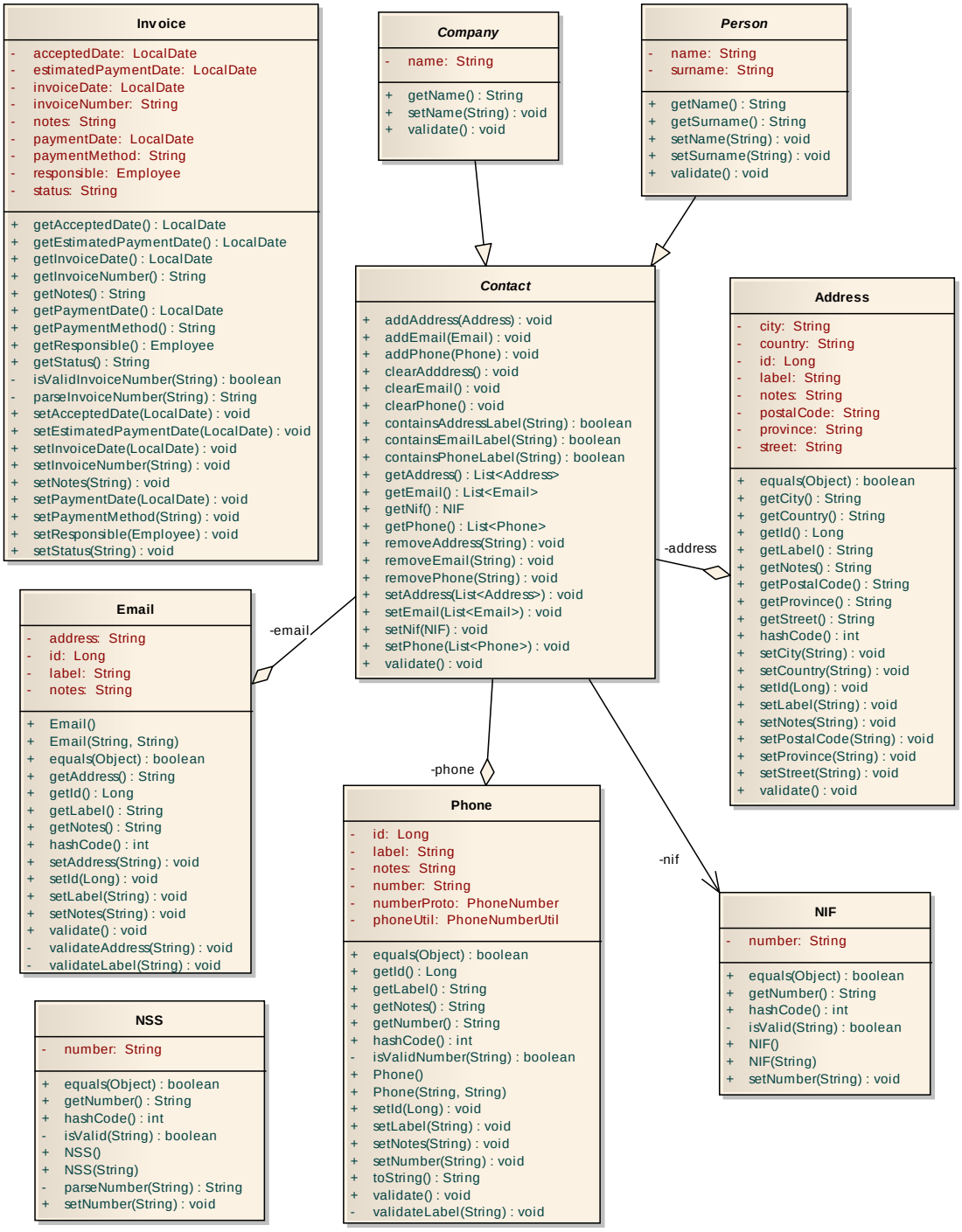

Figura 4.6: Diagrama de clases compartidas

- **NSS:** Un Número de la Seguridad Social española.
- **Invoice:** Una factura. Almacenaremos las fechas de emisión, aceptación, pago estimado y pago definitivo, además de un número identificador, el responsable de cobro, el método de pago, el estado de la factura y las notas pertinentes.

El resto de clases a diseñar serán las entidades de las que querremos almacenar datos. Las agruparemos según si están relacionadas con los clientes, con los empleados, con las reparaciones, con los pedidos y con los usuarios.

Cliente, almacenadas en **es.regueiro.easyrepair.model.client**: (Figura 4.7)

- **Client:** Un cliente que vendrá a la empresa y querrá hacer una reparación. Será una *Persona* con *Vehículos* junto con un número de cliente y notas.
- **Vehicle:** Un vehículo que posee el cliente. Sólo puede tener un dueño y no puede existir sin un cliente asociado. Se almacenarán todos sus datos identificativos, la compañía que lo asegura y su dueño.
- **IsuranceCompany:** La compañía de seguros encargada de asegurar el vehículo, si tuviera. Es una *Compañía* con página web y notas.

Empleado, almacenado en el paquete **es.regueiro.easyrepair.model.employee**: (Figura 4.8)

- **Employee:** Un empleado del taller, consistente en un *Contacto* con *Número de la Seguridad Social*, ocupación y número de identificación, junto con las notas.

Reparaciones, almacenadas en **es.regueiro.easyrepair.model.repair**: (Figura 4.9)

- **RepairOrder:** Una orden de reparación. En ella se almacenan el *Vehículo* a reparar, el *Empleado* responsable de la reparación, las *Piezas* y *Mano de Obra* necesarias, la *Factura* y el *Presupuesto* del coste, junto con las fechas de inicio, fin, estimada y entrega, los kilometros y el nivel del tanque de gasolina con los que llegó el coche, una descripción del trabajo a realizar, su estado actual y notas si procede, así como un número identificador para el taller.
- **Estimate:** El presupuesto de la reparación. Ya que la lista de piezas y mano de obra se incluye en la orden de reparación, sólo tendremos que almacenar las fechas de creación y aceptación del presupuesto, el *Empleado* responsable de su gestión, su estado actual y las notas, junto con un posible descuento si se quisiera hacer y el número identificador. También enlazaremos la *Orden de reparación* a la que se refiere.

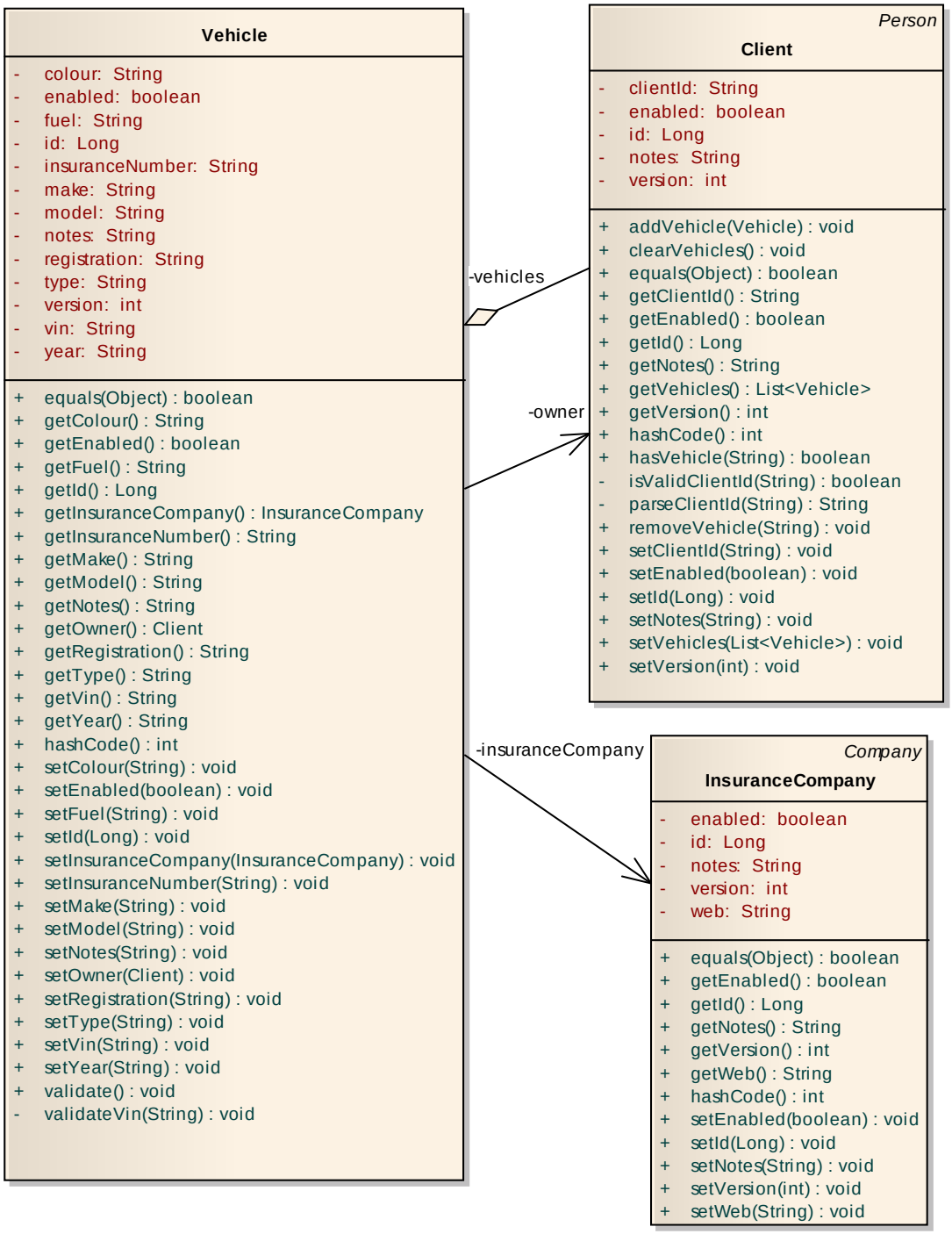

Figura 4.7: Diagrama de clases de Clientes

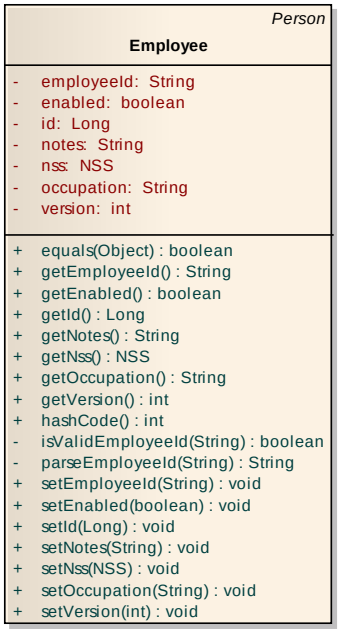

Figura 4.8: Diagrama de clases de Empleados

- **RepairInvoice:** La factura de la reparación. Es una entidad *Factura* a la que se le añade un responsable del pago y un enlace a la *Orden de reparación*.
- **Labour:** Mano de obra a realizar en las reparaciones. Guardamos su nombre, descripción, precio y notas.
- **LabourLine:** Datos adicionales de la mano de obra incluida en cierta reparación. Indicamos la *Mano de Obra* relacionada, las horas que toma y un descuento si se desease.

Stock, almacenados en **es.regueiro.easyrepair.model.stock**: (Figura 4.10)

- **Part:** Una pieza. Guardaremos su marca, modelo, categoría de producto, precio y notas, así como una lista de su *Stock* en nuestros almacenes.
- **PartOrder:** Un pedido de piezas a un proveedor. Almacena el *Proveedor* encargado de enviarlas, el *Almacén* al que llegarán, el *Empleado* responsable del pedido, la lista de *Partes* implicadas y su *Factura*, junto con las fechas de realización, llegada estimada y llegada, los costes de envío y otros costes que puedan surgir, un posible descuento, el estado actual, las notas y un número identificador.
- **PartOrderInvoice:** La factura del pedido de piezas. Es una simple *Factura* con un enlace al *Pedido*

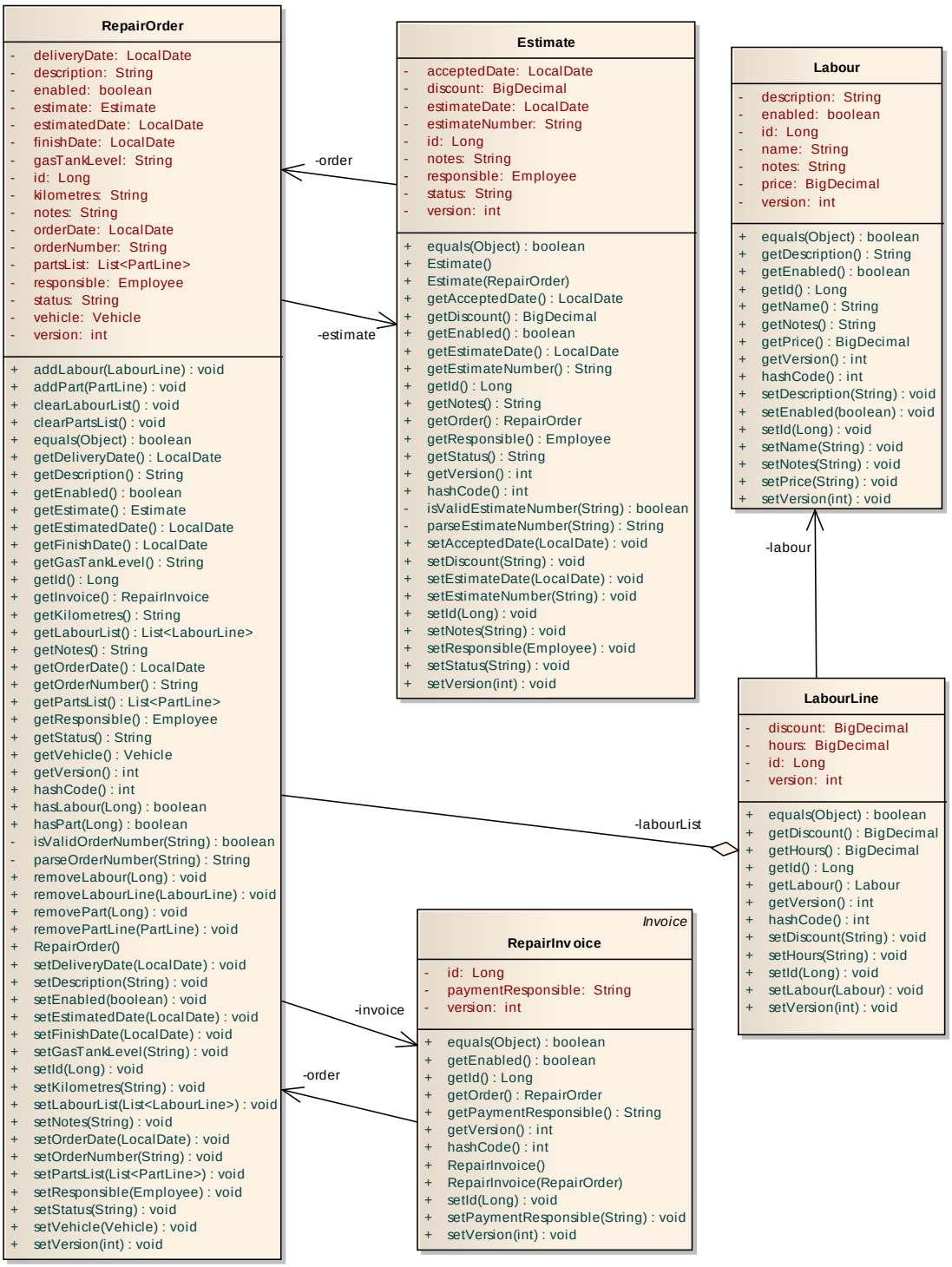

Figura 4.9: Diagrama de clases de Reparaciones

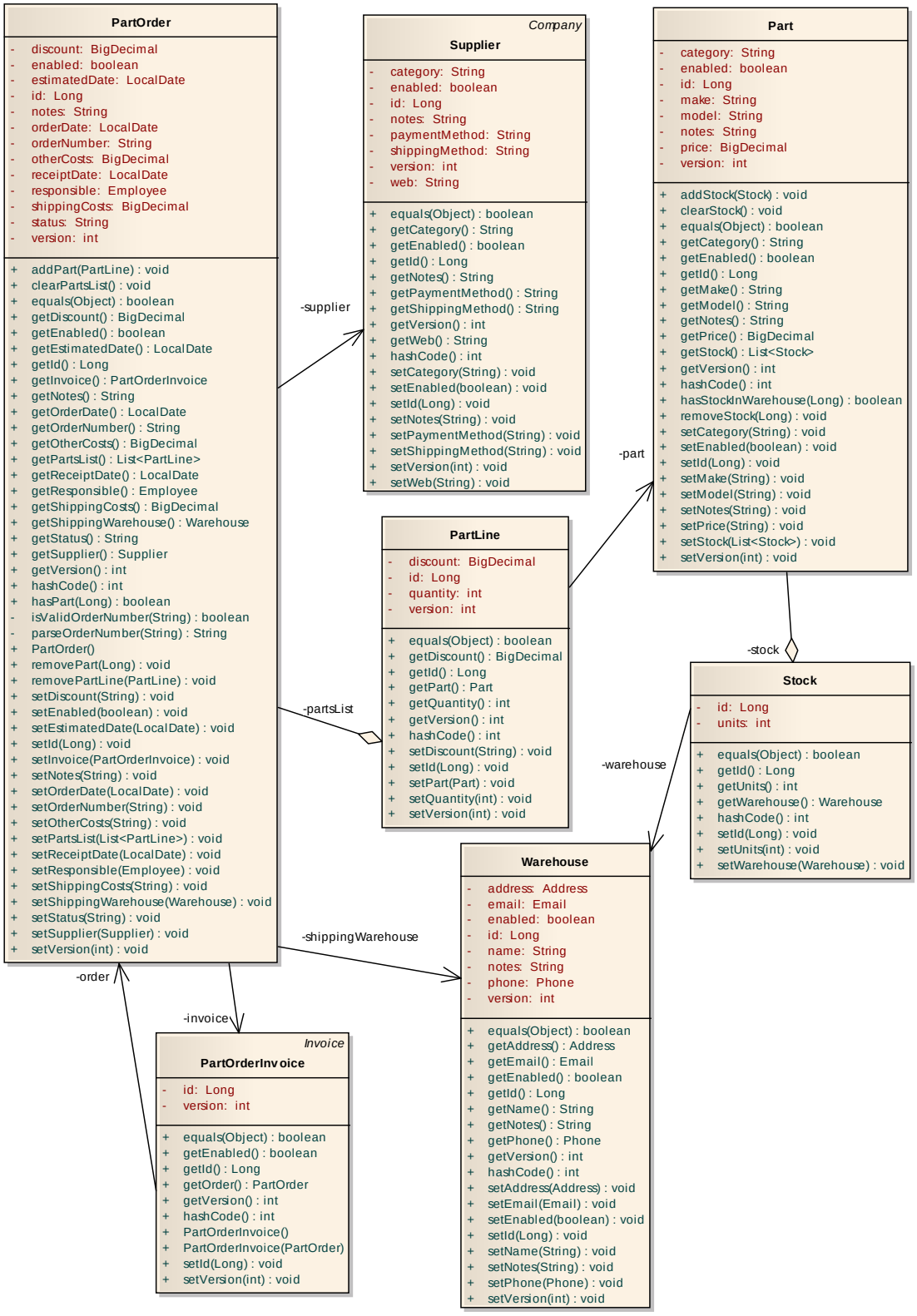

Figura 4.10: Diagrama de clases de Stock

- **PartLine:** Datos adicionales de las piezas incluidas en cierta reparación. Incluye, junto con la *Parte*, la cantidad y el descuento aplicado.
- **Stock:** Unidades disponibles en un almacén. Almacena las unidades y el *Almacén*
- **Supplier:** Un proveedor. Es una *Compañía* con categoría, web, método de pago, método de envío y notas.
- **Warehouse:** Un almacén. Guarda, junto con el nombre y notas, una *Dirección*, *Teléfono* y *Email*.

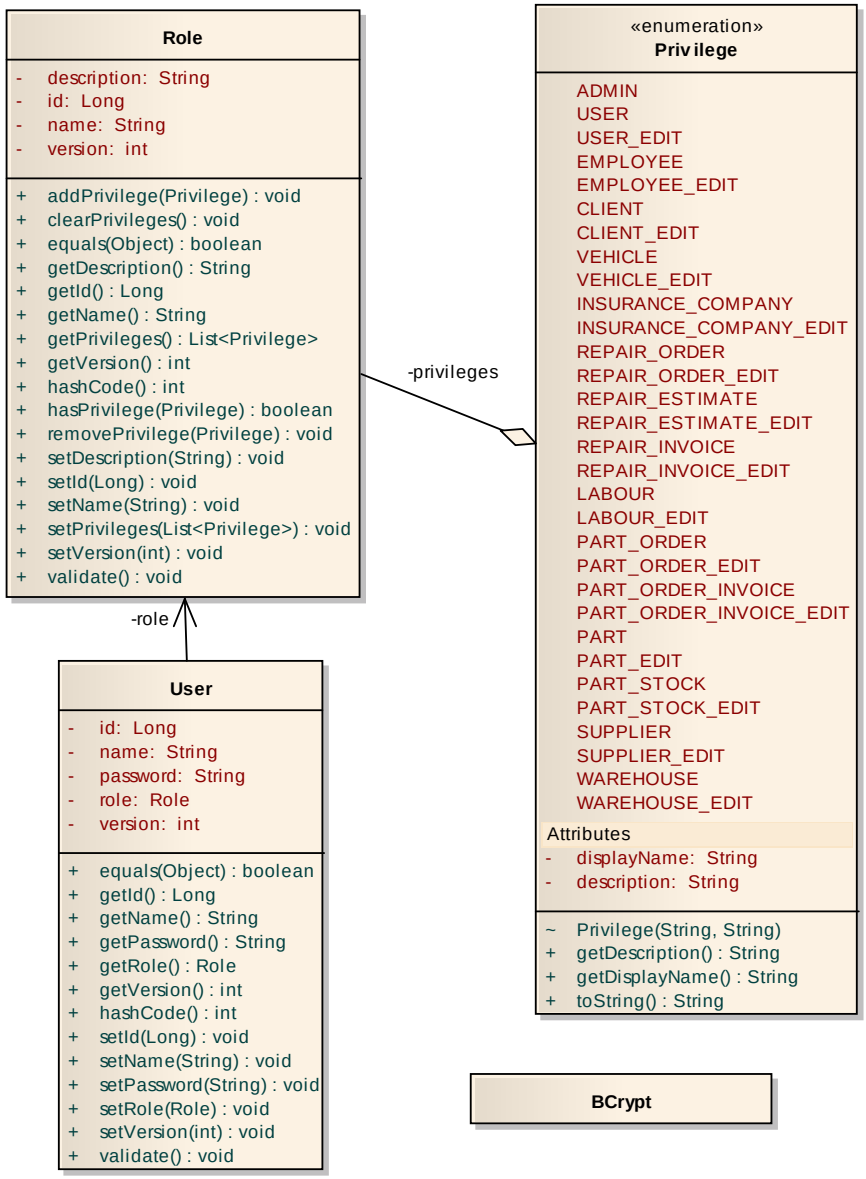

Figura 4.11: Diagrama de clases de Usuarios

Usuarios:, pertenecientes al paquete **es.regueiro.easyrepair.model.user**: (Figura 4.11)

- **User:** Un usuario del sistema. Guardamos su nombre, hash de la contraseña y su *Rol*.
- **Role:** Un rol de usuario. Consiste en una lista de *Privilegios*, junto con su nombre y descripción.
- **Privilege:** Un permiso de usuario. Definidos como una enumeración con un nombre y descripción.
- **BCrypt:** Librería de cifrado auxiliar utilizada para generar los *hashes* de las contraseñas del usuario y realizar las comprobaciones de igualdad de contraseña.

El diagrama completo del modelo de datos, con la relaciones entre las distintas entidades se puede apreciar en la Figura 4.12.

Para facilitar la integración con Hibernate u otros Sistemas de Mapeado Objeto-Relacional, las clases que serán almacenadas en la Base de Datos contienen, además de los datos descritos anteriormente, un atributo llamado *id*, utilizado para almacenar el identificador que le asigna el Sistema de Gestión de Bases de Datos, otro llamado *version*, que identifica el número de revisión de los datos en el momento que son leídos o guardados. Este número es usado por Hibernate para comprobar si el registro que estamos modificando ha recibido algún cambio por parte de otro usuario después de que nosotros lo hayamos leído. Por último, para permitir ocultar registros sin necesidad de eliminarlos, se ha añadido el atributo *enabled* que indicará si el objeto consta como activado o desactivado en la Base de Datos.

4.4. Segunda etapa: Diseño de la base de datos y conexión con el modelo

4.4.1. Diseño de la base de datos

Una vez definidas todas las entidades del sistema, es necesario realizar el diseño de una base de datos que permita guardar con seguridad toda la información. En la Figura 4.13 se pueden apreciar las tablas que la conformarán y sus relaciones, mientras que en las Figuras 4.14, 4.15, 4.16, 4.17 y 4.18 se puede ver la definición de los atributos que se guardarán en cada tabla.

Cada tabla contiene los atributos pertinentes a cada Entidad, definidos con el tipo de datos y límite de caracteres apropiado al atributo de la entidad que representan.

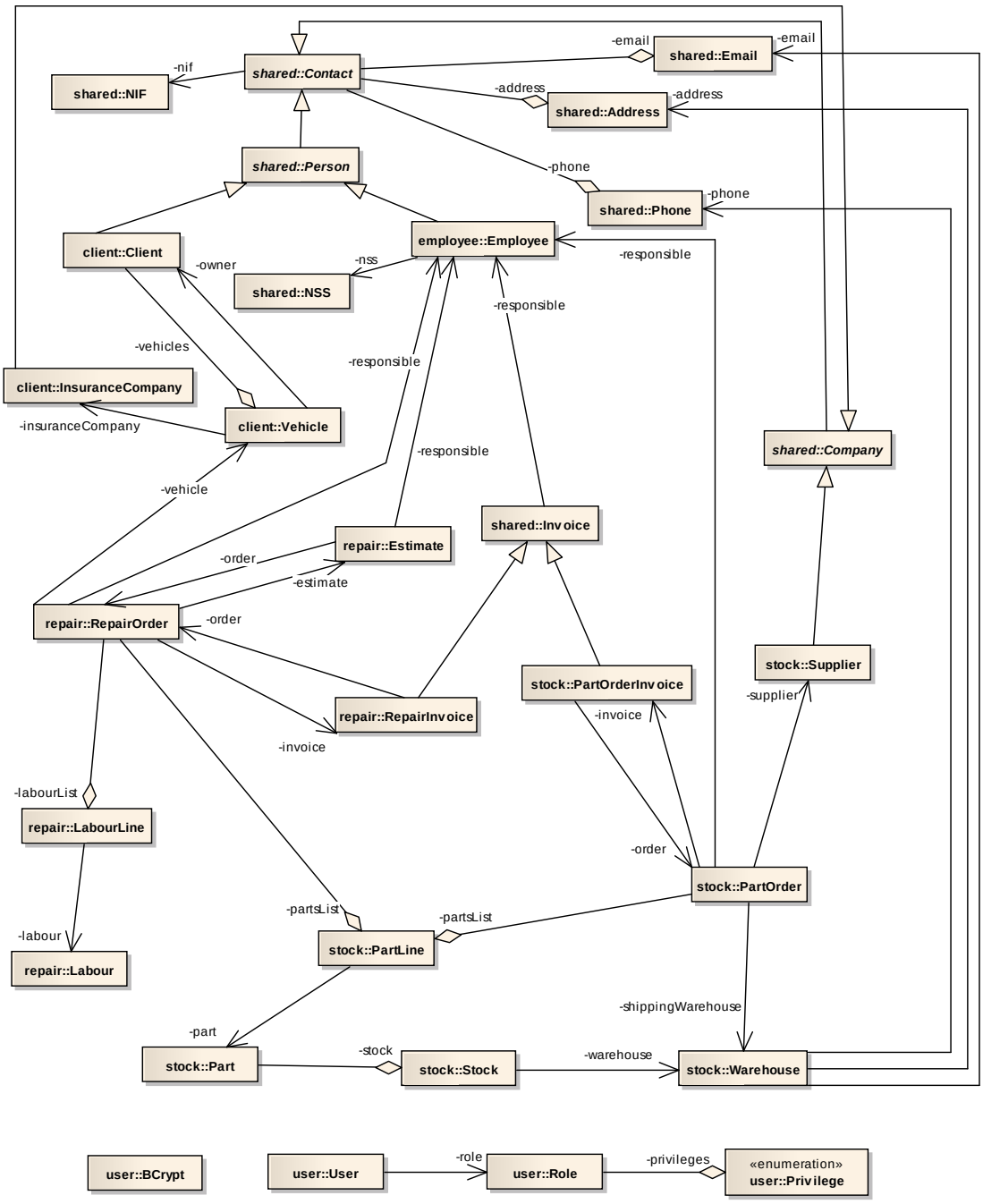

Figura 4.12: Modelo de datos

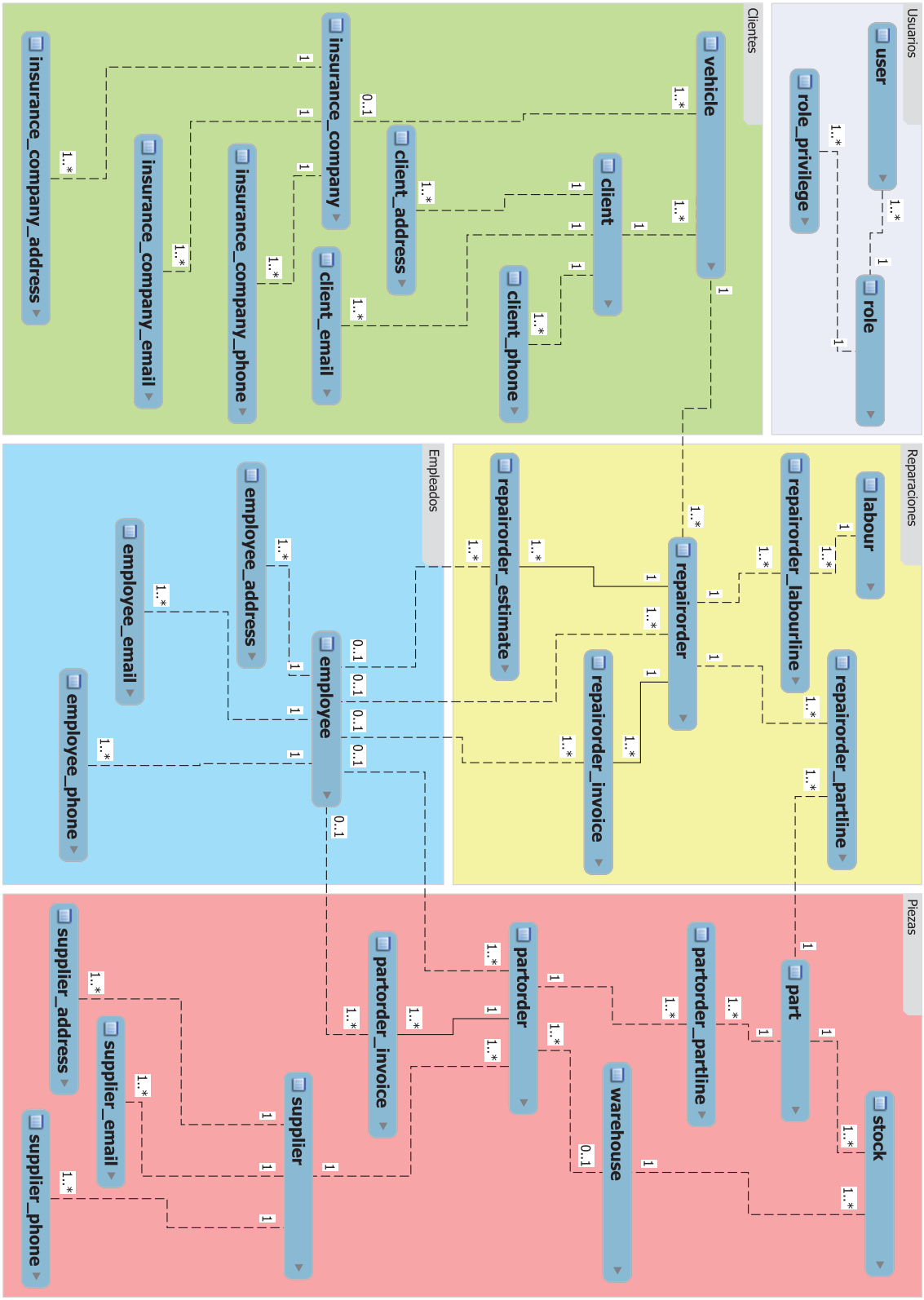

Figura 4.13: Modelo de la Base de Datos

Cuando se trata de un atributo de texto normal se les definió como *VARCHAR*, mientras que para las notas o textos posiblemente largos como las descripciones, se usó *LONGTEXT*.

Los números de identificación se almacenan como *INT* o *BIGINT* dependiendo de su longitud, y los booleanos como *TINYINT* de 1 dígito de longitud.

Por último, las fechas se almacenen con formato *DATE* y las cantidades que puedan tener cifras decimales como *DECIMAL*

Todas las tablas, excepto *role_privilege*, tienen como **clave primaria** un *número* de 20 dígitos auto-incrementado. Aquellas entidades que pueden ser deshabilitadas, contienen también el atributo **enabled**, y generalmente también se incluye el atributo **version**, un *número* de 11 dígitos que indica el número de revisión del registro. No es necesario incluirlo en todas las tablas puesto que si se modifica alguna tabla débilmente relacionada con otra, Hibernate aumenta automáticamente el número de versión del padre.

El primer diseño de la base de datos fue hecho aprovechándose de la propiedad *hbm2dll* de la configuración de Hibernate, que permite generar automáticamente las sentencias de creación de la base de datos a partir de los archivos de configuración detallados en la sección 4.4.2.

Una vez obtenido el diseño básico de la base de datos, este fue optimizado para las necesidades de nuestro sistema y los ficheros de configuración de Hibernate fueron cambiados apropiadamente. Con esto obtenemos el diseño final que se puede ver en las figuras indicadas anteriormente.

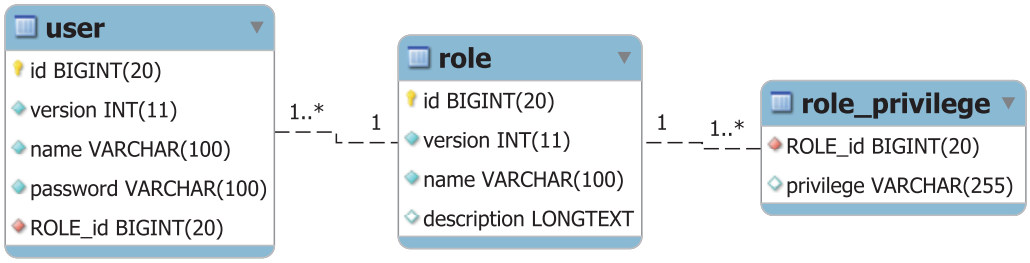

Figura 4.14: Tablas de Usuarios

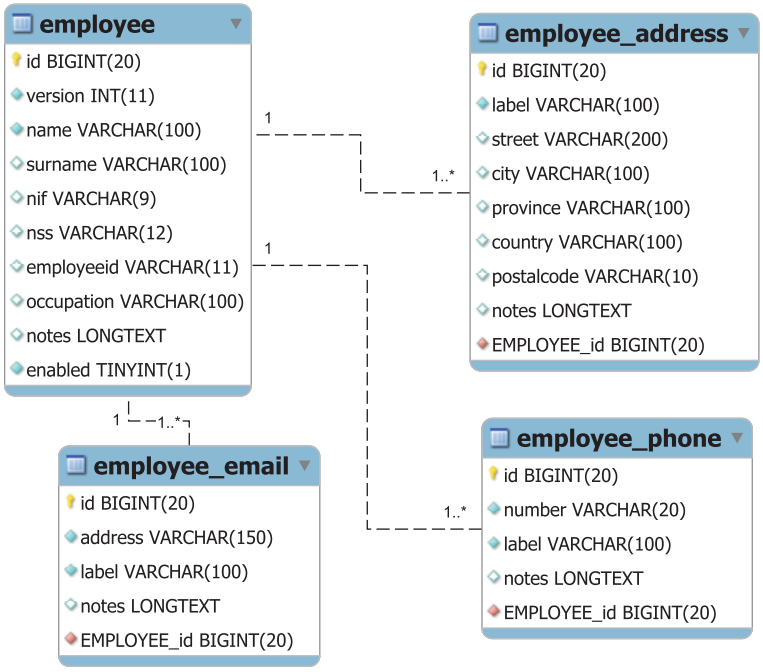

Figura 4.15: Tablas de Empleados

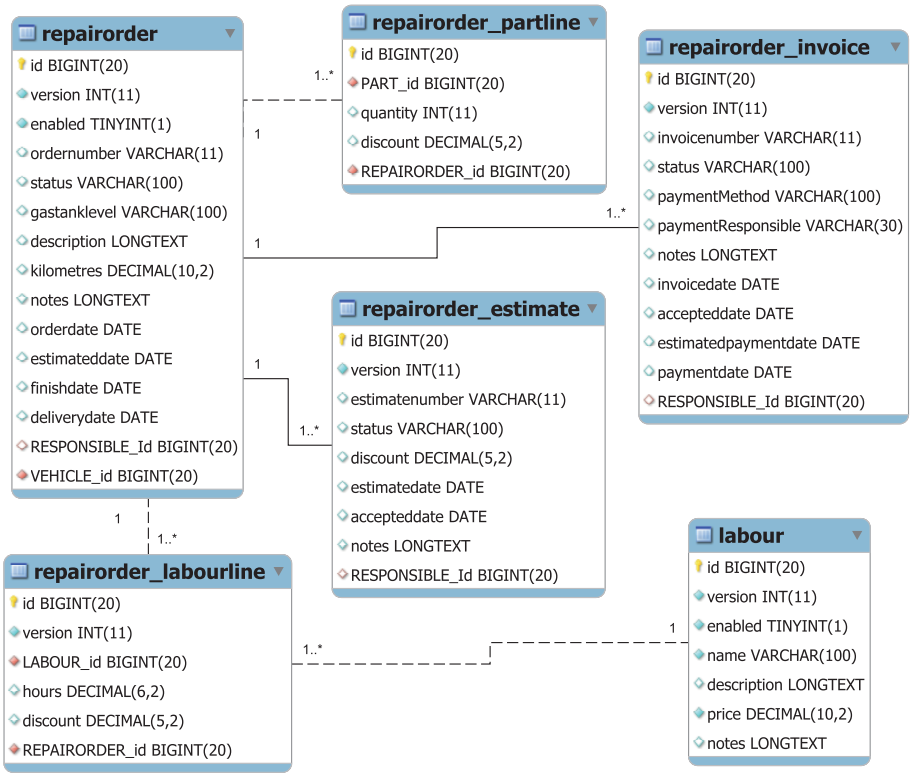

Figura 4.16: Tablas de Reparaciones

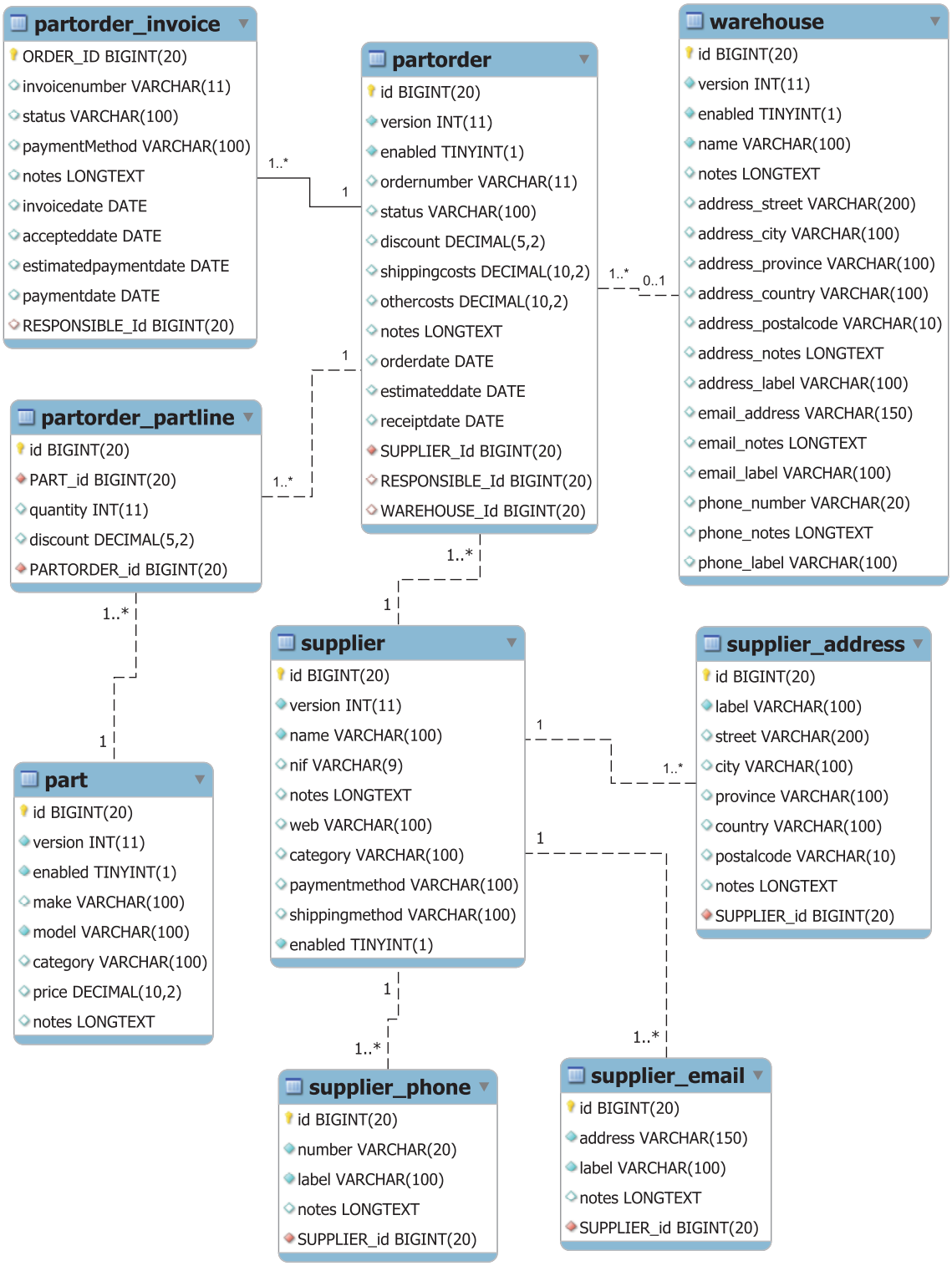

Figura 4.17: Tablas de Piezas

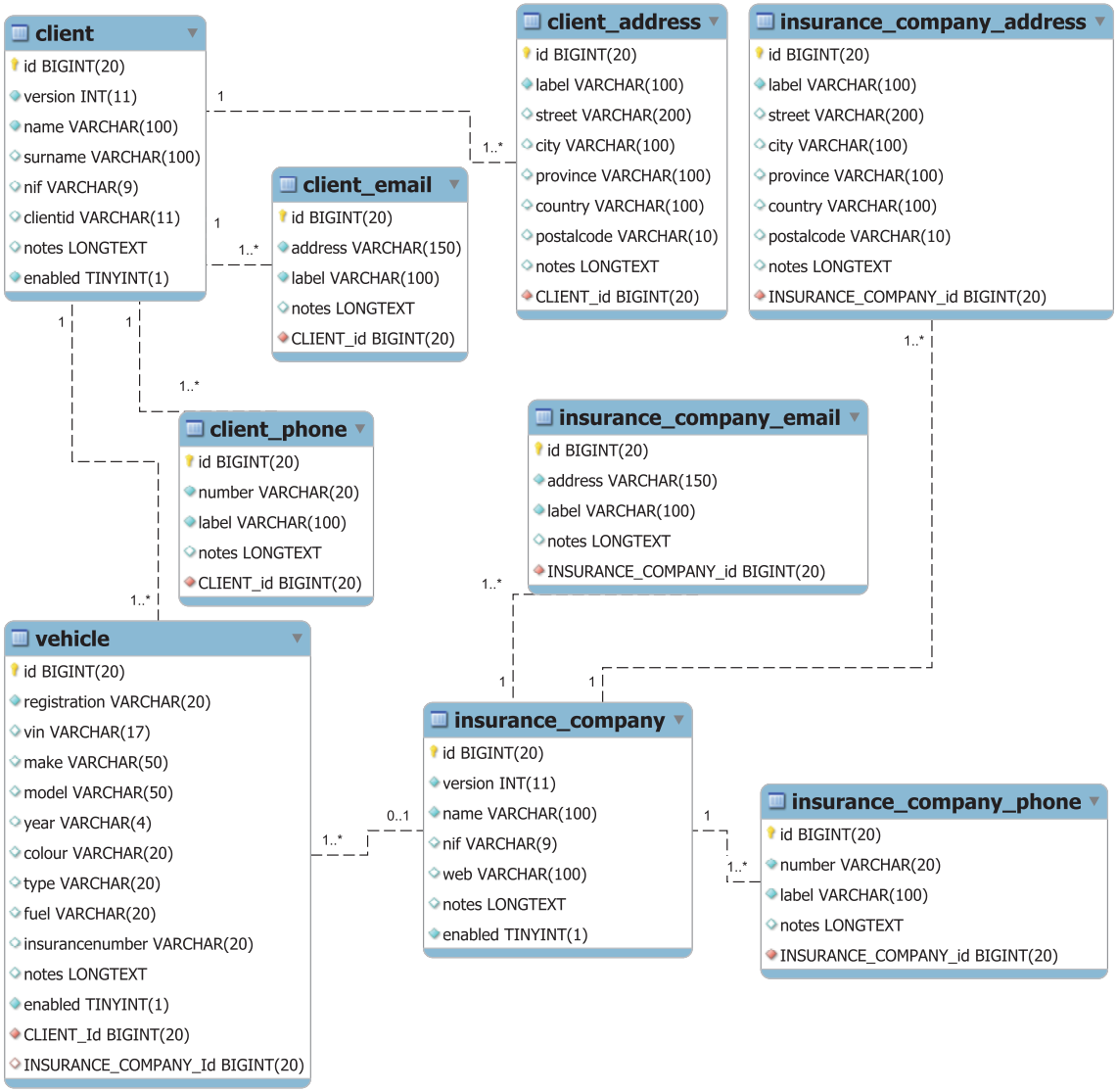

Figura 4.18: Tablas de Clientes

4.4.2. Conexión del modelo con la base de datos: Hibernate

Para enlazar la base de datos con sus correspondientes entidades en el modelo se decide hacer uso de **Hibernate**. Como ya se ha explicado, Hibernate es una herramienta de mapeo objeto-relacional (ORM) que facilita la conexión entre los atributos de un objeto y su representación en la base de datos, mediante el uso de archivos XML o de anotaciones en las clases.

Dichas anotaciones se almacenar en los módulos *XHibernateDAM*, siendo *X* un grupo de entidad, utilizando archivos de texto con la extensión *hbm.xml*. Así, por ejemplo, el archivo correspondiente a *Client.java* sería almacenado en *Client.hbm.xml* y contendría el siguiente texto:

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0
3  //EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4  <hibernate-mapping default-lazy="false" >
5     <class name="es.regueiro.easyrepair.model.client.Client" table="CLIENT">
6          <id name="id" type="long">
7              <column name="id" length="11"/>
8              <generator class="identity" />
9          </id>
10         <version name="version" column="version" access="field" />
11         <property name="name" type="string" >
12             <column name="name" length="100" not-null="true"/>
13         </property>
14         <property name="surname" type="string">
15             <column name="surname" length="100" />
16         </property>
17         <component name="nif" unique="true">
18             <property name="number" type="string" column="nif" length="9"/>
19         </component >
20         <bag name="vehicles" table="CLIENT_VEHICLE" access="field"
21           inverse="true" cascade="all-delete-orphan">
22             <key column="CLIENT_id"/>
23             <one-to-many
24                 class="es.regueiro.easyrepair.model.client.Vehicle"/>
25         </bag>
26         <property name="clientId" type="string" >
27             <column name="clientid" length="11" unique="true"/>
```

```
28          </property>
29          <property name="notes" type="text">
30              <column name="notes"/>
31          </property>
32          <property name="enabled" type="boolean">
33              <column name="enabled" default="1" not-null="true"/>
34          </property>
35          <bag name="address" table="CLIENT_ADDRESS" access="field"
36             cascade="all-delete-orphan">
37              <key column="CLIENT_id" not-null="true"/>
38              <one-to-many
39                  class="es.regueiro.easyrepair.model.shared.Address" />
40          </bag>
41          <bag name="email" table="CLIENT_EMAIL" access="field"
42             cascade="all-delete-orphan">
43              <key column="CLIENT_id" not-null="true"/>
44              <one-to-many class="es.regueiro.easyrepair.model.shared.Email"/>
45          </bag>
46          <bag name="phone" table="CLIENT_PHONE" access="field"
47             cascade="all-delete-orphan" >
48              <key column="CLIENT_id" not-null="true"/>
49              <one-to-many class="es.regueiro.easyrepair.model.shared.Phone"/>
50          </bag>
51      </class>
52  </hibernate-mapping>
```

Como se puede ver en el código anterior, los mapas consisten en un archivo *XML*, siguiendo un *Doctype* definido por Hibernate y contenido entre las etiquetas *hibernate-mapping*.

Las clases son definidas con la etiqueta *class*, acompañada del nombre de la clase y el nombre de la tabla que le corresponde.

Los atributos serán indicados con la etiqueta *property*, seguido de su nombre y el tipo, longitud, unicidad o nulidad, si procede. Para indicar la columna de la tabla donde se almacenarán se usa la etiqueta *column*, junto con el nombre, longitud, nulidad o unicidad, si fuese necesario.

La clave primaria es definida con la etiqueta *id*, seguida de su nombre y tipo. *column*, seguida de su longitud. Si, como es el caso, la columna dónde se almacena es auto-

incrementable se ha de indicar mediante la etiqueta *generator* acompañada de la case de identificador que se desee.

Si se desea poder comprobar si los datos han sufrido modificaciones mientras la sesión permanecía abierta, se ha de usar la etiqueta *version*, junto con el nombre de atributo que la almacenará en la clase y el nombre de columna. En este caso se añade la propiedad *access*, que indica el tipo de acceso que Hibernate realizará al dato, siendo por defecto el uso de los getters y setters que hayamos definido, o en el caso actual, leyendo directamente la variable privada mediante reflectividad.

Para indicar que el tipo de un atributo se corresponde con otra entidad definida en nuestro modelo se utiliza la etiqueta *component*.

Si este componente es múltiple, se usan las etiquetas *bag*, *set* o *list*, correspondientes con listas sin ordenar, conjuntos y listas ordenadas respectivamente.

Para indicar una relación con otra entidad, se utilizan las etiquetas *one-to-many*, *many-to-one*, *one-to-one* o *many-to-many*, según corresponda, indicando la clave externa de la tabla en la que se almacenará con la etiqueta *key*. En el caso que nos corresponde, se puede apreciar que la lista de vehículos se almacena en la tabla *CLIENT_VEHICLE* y se accede a través del uso directo de la variable. También se indica la propiedad *inverse="true"*, que es utilizada para hacer saber a Hibernate quién es responsable de la relación y se utiliza la etiqueta *cascade* para definir lo que ocurre cuando se elimina un registro con asociaciones.

4.5. Tercera etapa: Diseño de la interfaz de usuario y del módulo de empleados

4.5.1. Diseño de la interfaz de usuario

Para facilitar la gestión de las numerosas ventanas necesarias en una sistema de gestión de información se decidió hacer uso de la*Plataforma Netbeans* ''`http://netbeans.org/features/platform/`'', un framework de desarrollo de aplicaciones de escritorio en Java, que proporciona las siguientes características [12]:

- **Sistema de módulos:** La naturaleza modular de la *Plataforma Netbeans* te da la posibilidad de cumplir complejos requisitos combinando módulo pequeños, simples y fácilmente probables que encapsulan características similares de la aplicación.
- **Manejo del ciclo de vida:** Al igual que los servidores de aplicaciones web, como *GlassFish* o *WebLogic*, proveen servicios de gestión del ciclo de vida a aplicaciones web, el *contenedor en tiempo de ejecución de Netbeans* provee servicios de gestión del ciclo de vida a aplicaciones Java de escritorio.

- **Conectabilidad, Infraestructura de Servicios y Sistema de Archivos:** Los usuarios finales de la aplicación se beneficiaran de la posibilidad de instalar, desinstalar o activar y desactivar módulos en tiempo de ejecución.

  La *Plataforma Netbeans* provee una infraestructura para registrar y recuperar implementaciones de los servicios permitiendo minimizar las dependencias directas entre módulos individuales y consiguiendo una arquitectura poco acoplada y muy cohesionada.

  Así mismo, la *Plataforma Netbeans* provee de un servicio de sistema de archivos virtual, el cual es un registro jerárquico en el que se pueden almacenar configuraciones del usuario, parecido al Registro de Windows.

- **Sistema de Ventanas y Herramientas de Interfaz Estandarizadas:** La mayoría de las aplicaciones serias necesitan más de una ventana. Codificar una buena interacción entre múltiples ventanas no es una tarea trivial. El sistema de ventanas de Netbeans permite maximizar/minimizar, acoplar/desacoplar y arrastrar y soltar ventanas, sin necesitar de escribir ningún código extra.

  Swing y JavaFX son herramientas estándar para el desarrollo de interfaces de usuario y pueden ser utilizadas junto con la *Plataforma Netbeans*. Sus beneficios incluyen la posibilidad de cambiar la apariencia de la aplicación, la portabilidad de los componentes a través de todos los sistemas operativos y la posibilidad de añadir muchos componentes gratuitos de terceras partes.

  En conjunto con el sistema de ventanas, Netbeans provee muchos otros componentes relacionados con la interfaz, como hojas de propiedades, un paleta de elementos, una ventana de salida de datos, un gestor de plugins o complejos componentes Swing para presentar datos.

Junto con estas características, es necesario indicar que para el desarrollo de grandes interfaces de usuario en *Swing*, es recomendable la utilización de un diseñador gráfico que ayude a realizar esta tarea con rapidez. El IDE Netbeans incluye el diseñador *Matisse*, que cuenta con muchos años de desarrollo a sus espaldas y permite generar las distintas ventanas de manera gráfica y sencilla.

Otros entornos de desarrollo, como *Eclipse* `''http://www.eclipse.org/''` o *IntelliJ IDEA* `''http://www.jetbrains.com/idea/''` incluyen ya desde hace un par de años diseñadores de interfaces de usuario, pero todavía no han llegado al nivel de *Netbeans*.

Para la introducción de datos en nuestro sistema tendremos que crear un editor que permita tanto añadir información como modificar la existente. También será necesario un buscador que permita encontrar datos en nuestra aplicación y para presentar la información existente cuando no es necesario editarla se ha decidido crear un visor detallado. Tenemos

por lo tanto la necesidad de crear tres ventanas por cada grupo de datos que deseemos ofrecer.

Gracias a la Plataforma Netbeans, no es necesario el uso de ningún código especial para gestionar las ventanas; simplemente tendremos que indicar de que tipo són y él mismo se ocupa de colocarlas y darle sus propiedades básicas. Los tipos de ventana en Netbeans y su colocación se pueden apreciar en la Figura 4.19.

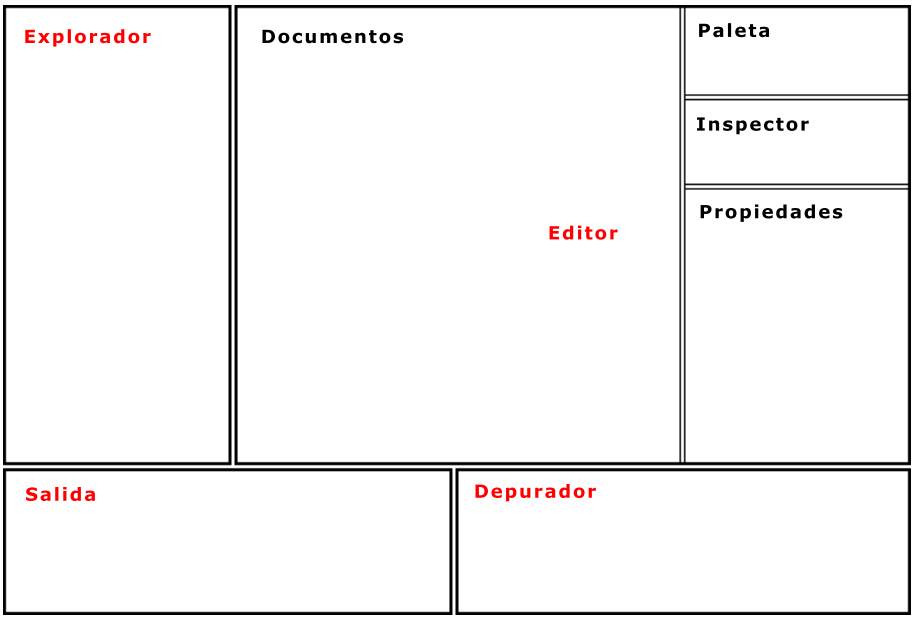

Figura 4.19: Sistema de ventanas de Netbeans

Para las ventanas del editor y el buscador se ha optado por colocarlas por defecto en la posición marcada como *Documentos* en la Figura 4.19 y la vista detallada se colocará en la posición *Explorador*. Gracias a la gestión de ventanas de Netbeans, esta colocación podrá ser editada fácilmente por el usuario arrastrando y soltando las ventanas en la posición que más le interese.

4.5.2. Diseño del módulo de empleados

Una vez averiguadas las necesidades de la interfaz, podemos pasar al desarrollo de los módulos que le darán sus características.

Hasta ahora tenemos desarrollado el modelo de datos, la base de datos y la conexión entre ellos usando Hibernate. También hemos definido la necesidad de un módulo de Login, Informes, Persistencia y otro de Librerías Compartidas. Ahora tendremos que desarrollar un módulo que contenga la interfaz gráfica de gestión de los **Empleados** y un módulo de acceso a los datos de estos.

El módulo de acceso a los datos deberá permitir buscar empleados existentes, utilizando distintos criterios como su nombre u ocupación, crear nuevos empleados, guardar cambios en empleados existentes y por último, la posibilidad de eliminarlos. Como los datos pueden llegar a estar altamente relacionados entre ellos, y no queremos vernos en la necesidad de borrar una factura porque el empleado responsable de ella fue despedido y lo queremos borrar del sistema, se decide también añadir la posibilidad de activar y desactivar registros.

Como ya se ha explicado en la sección 4.3.3.1, este módulo de acceso se dividirá en el **EmployeeAPI** y **EmployeeHibernateDAM**.

La interfaz de usuario deberá proporcionar un buscador, un editor y una vista detallada y será almacenada en el módulo **EmployeeUI**

4.5.2.1. EmployeeAPI

Para distinguir entre la acción de buscar y editar, la API de empleados se podría dividir en una clase buscador, llamada *EmployeeFinder*, una clase *EmployeeSaver* que se ocupe de guardar, borrar y desactivar, y un controlador que proporcione acceso a las operaciones que ambas realizan, el *EmployeeController*.

En la Figura 4.20 se puede comprobar la definición de la interfaz de acceso a los datos a través de estas tres clases.

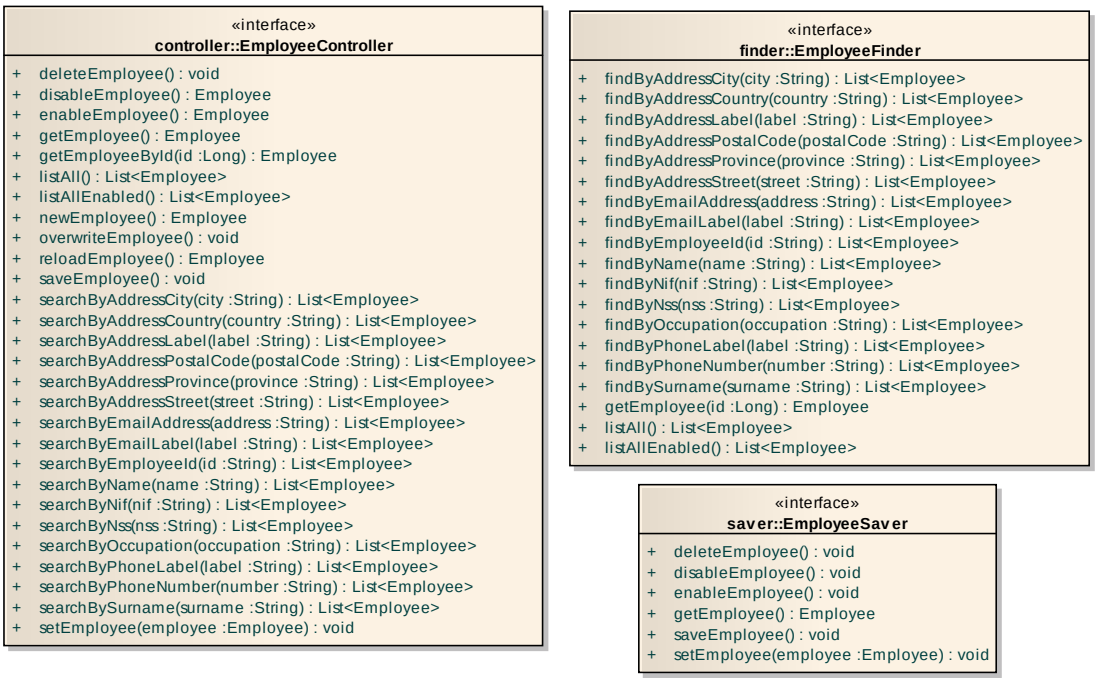

Figura 4.20: Diagrama de clases de Employee API

En la clase **EmployeeFinder** tendríamos la definición de los métodos de búsqueda de empleados:

- **findByX(String)**: Son métodos que indican que podremos buscar empleados por su dirección postal, email, nombre, NIF, número de la seguridad social, ocupación y número de teléfono. Devuelven una lista que podría contener múltiples, uno o ningún empleado.
- **getEmployee(Long)**: Define un método para obtener un empleado directamente a través de su número de identificación
- **listAll()** y **listAllEnabled()**: Indican métodos para listar a todos los empleados y a todos los que están marcados como activados.

Por otro lado, en **EmployeeSaver** tenemos los siguientes métodos:

- **setEmployee(Employee)**: Se utilizará para indicar qué empleado se modificará.
- **deleteEmployee()**: Eliminaría del sistema el empleado.
- **disableEmployee()** y **enableEmployee()**: Desactivan o activan el empleado
- **saveEmployee()**: Guarda en el sistema las modificaciones realizadas.
- **getEmployee()**: Devuelve el objeto del empleado en su estado actual.

Y por último en **EmployeeController** tenemos, además de todos los métodos disponibles el la clase de búsqueda y en la de guardado, los siguientes métodos:

- **getEmployeeById()**: Buscaría al empleado haciendo uso de su ID único. Se corresponde con un enlace directo al método *getEmployee(id : Long)* de *EmployeeFinder*
- **newEmployee()**: Crea un nuevo empleado y devuelve su objeto.
- **overwriteEmployee()**: Cuando un empleado ha sido modificado por otro usuario durante el tiempo que hemos tardado en realizar nuestros cambios, el sistema lo detecta y ofrece la posibilidad de sobrescribir las modificaciones realizadas por el otro usuario.
- **reloadEmployee()**: Vuelve a leer un empleado del sistema. Se utilizará para descartar cambios realizados y volver al estado anterior de un empleado.

4.5.2.2. EmployeeHibernateDAM

Este módulo se ocupa de implementar las interfaces definidas en el módulo *EmployeeAPI* haciendo uso de Hibernate para acceder a la base de datos.

La Figura 4.21 muestra la definición de las clases de este módulo.

Además de la implementación de los métodos definidos en el apartado 4.5.2.1, se puede apreciar la aparición de una clase nueva llamada **Installer**. Esta clase es llamada por el sistema de módulos de Netbeans al activar o desactivar un módulo y además es utilizado por las clases de búsqueda y guardado para administrar sesiones.

Hibernate entiende una sesión como la abstracción de la noción de un servicio persistente [9]. Su ciclo de vida está comprendido entre el inicio y el fin de una transacción, que se entiende como una unidad atómica de modificación de datos.

Cuando el sistema desea interactuar con la base de datos, llama al método *createSession()* de la clase *Installer*, que devuelve una nueva sesión preparada para realizar la conexión. A continuación se inicia una transacción, seguida de los métodos de interacción con la base de datos. Por último, si todo el proceso se ha realizado correctamente, se finaliza la transacción y la sesión y se devuelven los datos pedidos, si procede.

En los métodos de búsqueda, se utiliza el método *createCriteria(Class)* de la sesión, que se ocupa de crear un criterio de búsqueda de elementos de la clase que hayamos seleccionado, con las restricciones que deseemos.

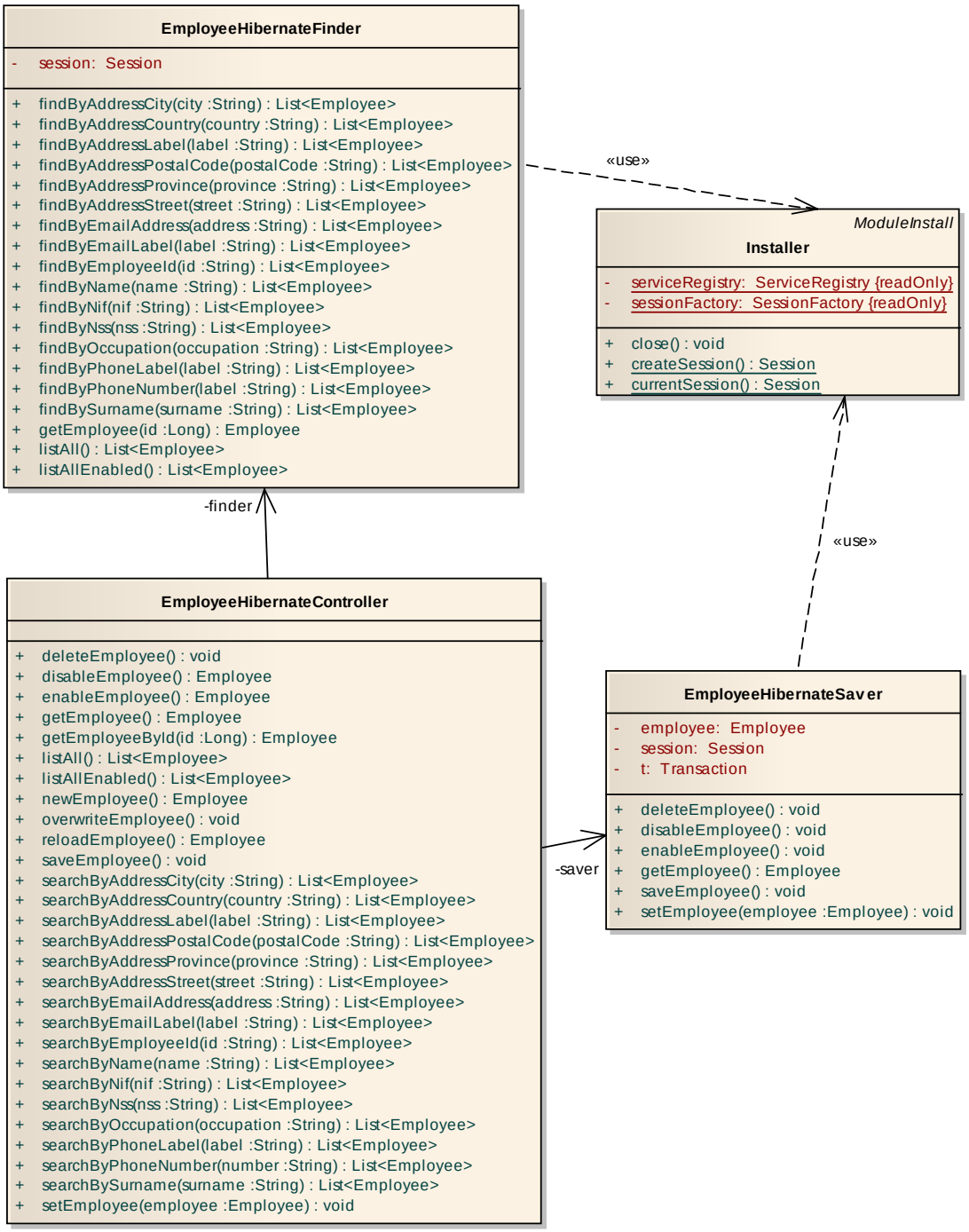

Figura 4.21: Diagrama de clases de Employee Hibernate DAM

Un ejemplo de búsqueda de un empleado por nombre sería:

```
1  public List<Employee> findByName(String name) {
2         try {
3             session = Installer.createSession();
4             Transaction transaction = session.beginTransaction();
5             List<Employee> list =
6                  (List<Employee>) session.createCriteria(Employee.class)
7                     .add(Restrictions.isNotNull("name"))
8                     .add(Restrictions.like("name", "%" + name + "%"))
9                     .list();
10            transaction.commit();
11
12            if (session.isOpen()) {
13                session.close();
14            }
15            return list;
16        } catch (ResourceClosedException e) {
17            return findByName(name);
18        } catch (SessionException e) {
19            return findByName(name);
20        }
21    }
```

Para el guardado de objetos, el proceso es similar, cambiando la llamada al método *createCriteria()* por llamadas a *saveOrUpdate()* o *delete()*, dependiendo de si se desea actualizar el objeto o borrarlo.

4.5.2.3. EmployeeUI

Como hemos definido en el principio de la sección, la interfaz de usuario se compondrá de un buscador, un editor y una vista detallada.

La interfaz será construida utilizando la librería Swing de Java y apoyándose en la gestión de ventanas de la *Plataforma Netbeans*. Siguiendo la guía de estilo de Netbeans, las ventanas se denominarán TopComponent.

Cada TopComponent se almacena como una clase nueva, por lo que tendremos que crear al menos tres clases:

- **EmployeeBrowserTopComponent:** Será la clase que almacenará el buscador de empleados.
- **EmployeeEditorTopComponent:** Correspondiente al editor de empleados.
- **EmployeeDetailedViewTopComponent:** Que definirá la vista detallada.

Para cada ventana, se realiza un prototipo del diseño utilizando herramientas de realización de diagramas, obteniendo finalmente los mockups apreciables en la Figura 4.22.

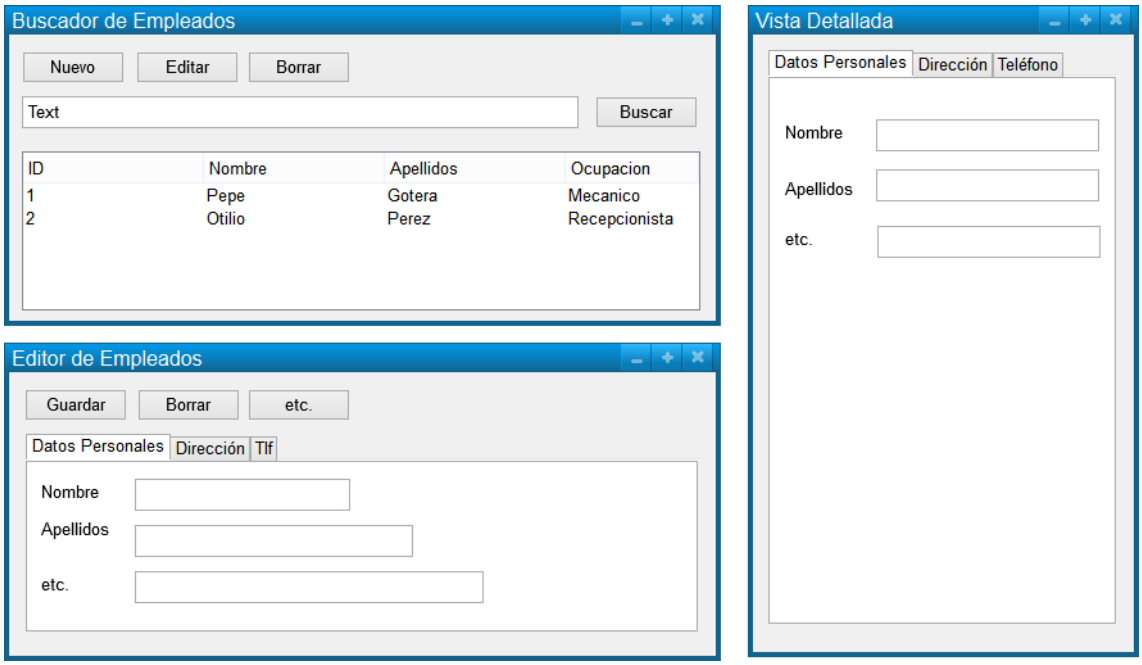

Figura 4.22: Prototipo de diseño de las ventanas del módulo de empleados

Posteriormente se realiza el diseño de las ventanas utilizando la GUI Matisse de Netbeans y a continuación se realiza la programación de las acciones que podrán ser realizadas desde la interfaz.

Para poder acceder a las ventanas deseadas, se incluirá un icono identificativo de cada buscador a la barra de herramientas superior del programa y un menú con enlaces a dicho buscador para cada tipo de dato del sistema. Los buscadores serán los puntos de entrada a todas las funcionalidades pertinentes a un tipo de dato.

Una vez accedido al buscador, se nos presentará la ventana, en la que podremos apreciar una barra superior con botones que nos permitirán realizar ciertas acciones, un cuadro de texto en el que se puede introducir el texto a buscar, un botón de búsqueda y una tabla en la que se mostrará el resultado. Además de lo ya mostrado en los *mockups* antes vistos, se añade también una *combobox* que permitirá elegir sobre qué campo del empleado se

realiza la búsqueda y un botón que permita listar a todos los empleados, junto con una marca que indique si queremos excluir los empleados desactivados.

El diseño final realizado en Swing se puede ver en la Figura 4.23.

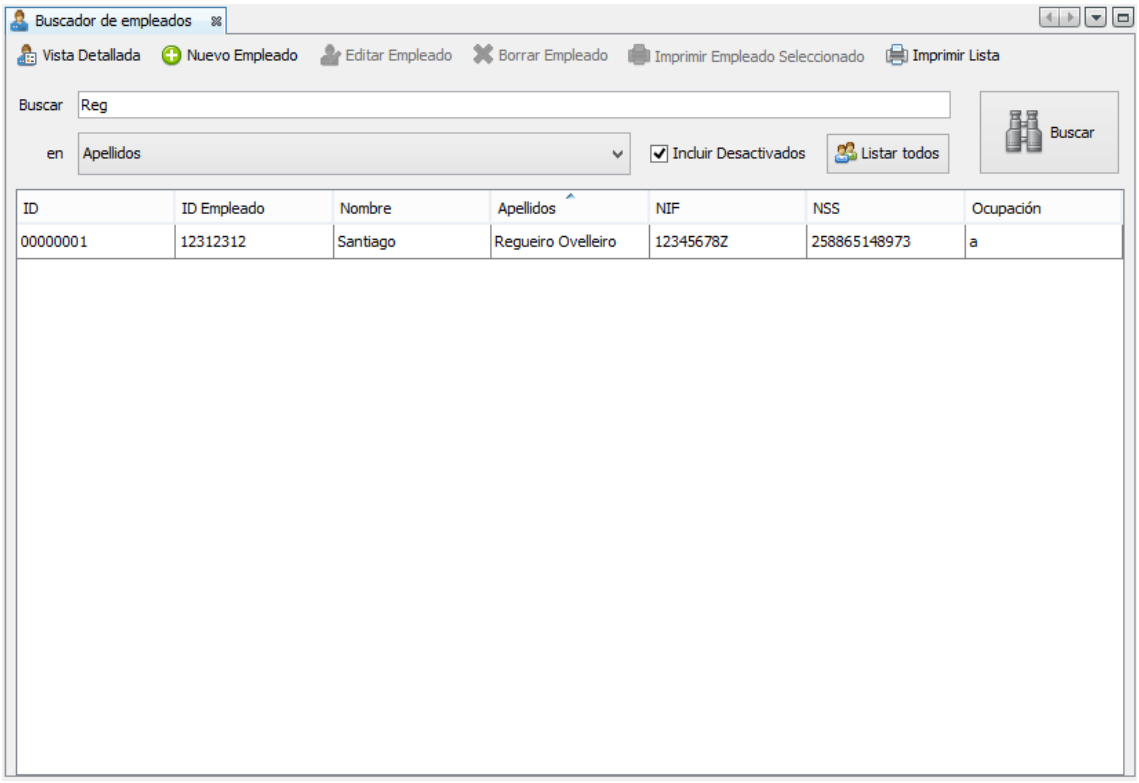

Figura 4.23: Diseño final del buscador de empleados

La primera funcionalidad del buscador es, obviamente, realizar búsquedas. Para ello, una vez se ha pulsado el boton buscar, lo primero que se hace es comprobar qué opción está seleccionada en la *combobox* y con ello decidir qué debemos buscar. A continuación se lee el texto introducido en el campo de texto y se llama al método correspondiente al tipo de búsqueda del EmployeeAPI. Netbeans ahora busca la implementación por defecto de la API de empleados, que en nuestro caso será la de Hibernate, y llama al método concreto, que devolverá una lista de empleados y esta será mostrada en la tabla de la ventana del buscador.

Otra posibilidad sería utilizar el botón de *Listar Todos*, el cual, dependiendo de si la marca de *Incluir Desactivados* colocada a su izquierda está activada o no, llamará a los métodos *listAll()* o *listAllEnabled()* de la API de empleados y todos los almacenados en la base de datos serán mostrados en la tabla.

Por último tenemos la barra de iconos superior, en la que se incluyen los botones para abrir la vista detallada, crear un nuevo empleado, editar uno existente, borrarlo, activar

o desactivarlo y para imprimir o bien un empleado o la lista completa mostrada en la tabla. Los botones de edición, borrado, activación o impresión de un sólo empleado se mantendrán desactivados hasta que sea seleccionado uno en la tabla de resultados. El botón de imprimir lista se activará cuando se realice una búsqueda.

Cuando se selecciona un empleado, Swing envía al controlador de la ventana el mensaje de que el valor seleccionado ha cambiado. Una vez se recibe este mensaje, el controlador y como respuesta obtiene el empleado seleccionado y lo incluye en el **EmployeeLookup**. Un *Lookup* es una construcción de la Plataforma Netbeans, usada principalmente para la comunicación entre clases independientes y el registro de datos en un lugar al que pueden acceder todas las clases que deseemos. Consiste en una clase pública que sigue el patrón de diseño *singleton* a la que se le añaden los métodos que queremos sean accesibles desde otras clases. Así en este caso tendremos un método para almacenar un empleado y otro para recuperarlo. Al mismo tiempo que se incluye en el *Lookup*, los botones de edición se activan.

Si pulsásemos en el botón de imprimir, se abriría una ventana en la que se mostraría un documento que contendría los datos del empleado o la lista de todos los encontrados como resultado de la búsqueda. Esta ventana y el documento son creados por la librería **JasperReports**, un framework de diseño de documentos que permite generar dinámicamente cualquier tipo de documento con los datos almacenados en nuestra aplicación. Junto con *JasperReports*, se utiliza la librería *DinamicReports* que permite realizar los diseños mediante el uso de sentencias de programación sencillas.

Otra posibilidad sería pulsar el botón de activar o desactivar, el cual dependiendo del estado actual del empleado seleccionado realizaría la activación o desactivación mediante el uso de la API de empleados.

Finalmente, si se pulsase el botón de eliminar, se nos mostraría un cuadro de diálogo en le que deberemos confirmar que se desea realizar dicha acción y que de ser confirmada eliminaría al empleado del sistema. Si este empleado estuviese relacionado con algún otro elemento del sistema (podría ser el responsable de una orden de reparación o una factura), se mostraría un diálogo indicando que el empleado no se puede eliminar actualmente.

Si habiendo seleccionado un empleado, pulsásemos en el botón de editar, se abriría la ventana de edición de empleados, la cual se puede apreciar en la Figura 4.24.

La ventana de edición de empleados se puede abrir de dos maneras: pulsando el botón de edición con un empleado seleccionado en el buscador o pulsando el botón de nuevo empleado.

En el caso de que se pulsase el botón de creación de un nuevo empleado, la ventana se mostraría con todos los campos en blanco y lista para comenzar a introducir datos.

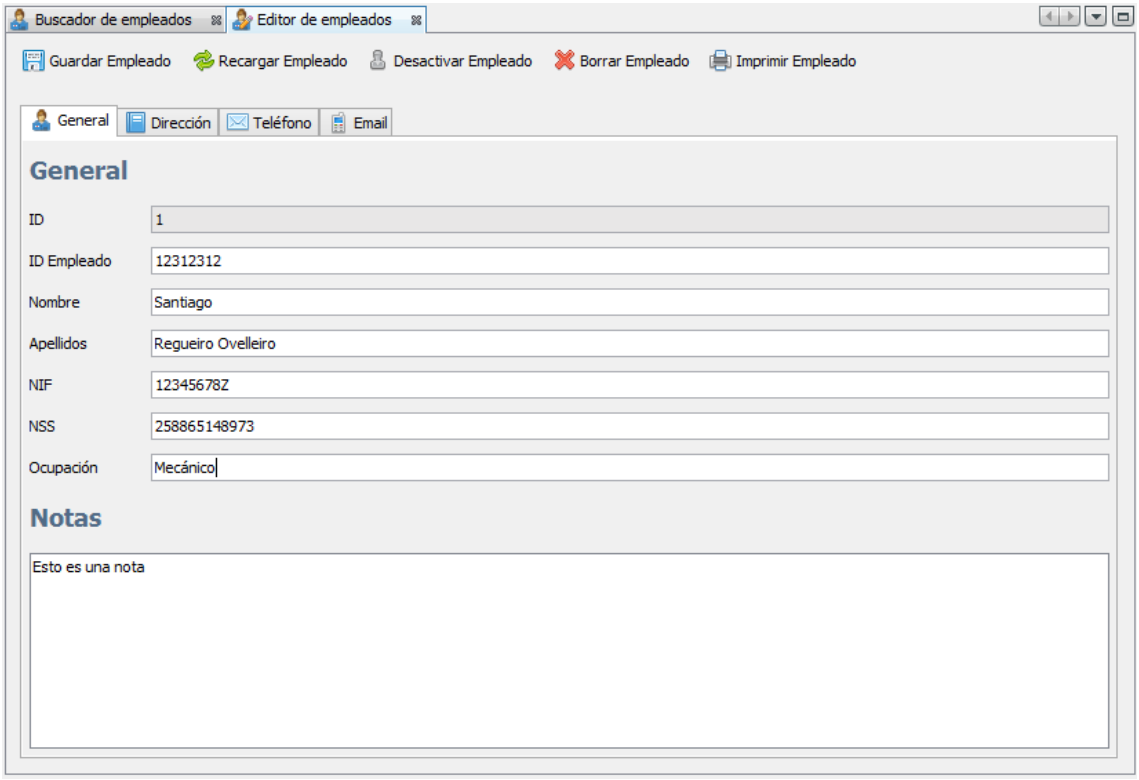

Figura 4.24: Diseño final del editor de empleados

Si en cambio se usase el botón de editar, la ventana tendrá que mostrar los datos actuales del empleado. Para ello lo primero que realiza al abrirse es obtener el empleado guardado en el Lookup, que en el caso de que se hubiera presionado *Nuevo*, sería nulo, y carga todos sus datos en los campos correspondientes.

Los cuadros de texto serán luego editados por el usuario y se podrán guardar cuando este esté listo. Si por cualquier motivo se quisiese recuperar el usuario sin cambios almacenado en el sistema, se podría hacer pulsando en el botón de *Recargar Empleado* y todos los cambios realizados se perderían. También podríamos activarlo o desactivarlo o eliminarlo directamente desde el editor. De nuevo, si hubiera algún problema se nos mostraría un cuadro de diálogo indicándolo.

Si en el tiempo que pasa entre que abrimos el empleado y guardamos los cambios realizados otro usuario ha modificado al empleado y lo ha podido guardar, el sistema nos alertará de que se ha realizado dichos cambios y nos presentara dos opciones; o bien descartar nuestros cambios o sobrescribir los cambios realizados por el otro usuario. Por desgracia Hibernate no es capaz de saber qué campos fueron modificados por cada usuario y no es posible la realización de una agregación de los cambios o incluso una selección de entre ellos. Esto sería posible si el sistema se hubiera realizado con una arquitectura de tres niveles y el servidor conociera en todo momento los cambios que se han realizado en

los objetos o si se mantuviese una base de datos de cambios y usuarios con la arquitectura actual, pero se descartó por la gran complejidad añadida de este sistema y las pocas posibilidades de ediciones concurrentes debido al enfoque del programa a pequeñas y medianas empresas.

Por último es posible acceder a la ventana de Vista Detallada pulsando el botón correspondiente del buscador. Su diseño puede apreciarse en la Figura 4.25.

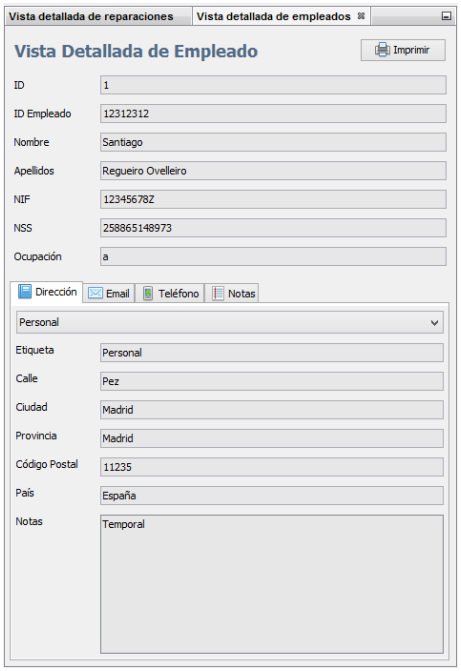

Figura 4.25: Diseño final de la vista detallada de empleados

Esta ventana consiste en prácticamente una copia del editor con la posibilidad de hacer cambios eliminada, y realiza la misma acción al abrirse que dicho editor; busca al empleado en el *Lookup* y si existe, muestra sus datos.

La principal diferencia con el editor, además de la imposibilidad de realizar cambios, consiste en la vigilancia continua del *Lookup*. Si en el buscador seleccionásemos otro empleado distinto, éste sería almacenado en el Lookup y la ventana de Vista Detallada se daría cuenta, pasando a mostrar ahora sus datos.

También incluye un botón de impresión del actual empleado.

4.6. Cuarta etapa: Diseño del módulo de clientes

El módulo de Clientes sigue el mismo diseño definido anteriormente en el módulo de empleados pero con la diferencia de que en este caso tendremos que trabajar con tres

entidades: **Cliente**, **Vehículo** y **Compañía de Seguros**.

4.6.1. ClientAPI

Como en la etapa anterior, tendremos que crear un buscador, un grabador y un controlador. Gracias al diseño del modelo de datos sabemos que los vehículos no pueden existir a menos que pertenezcan a un Cliente, por lo que no será necesario crear una clase que se ocupe de grabar Vehículos, ya que de esto se ocupará la clase que realice esa tarea con los clientes.

Así pues, tendremos que crear las siguientes clases:

- **ClientFinder**, **ClientSaver** y **ClientController:** Las ocupadas de buscar y grabar a los *Clientes*.
- **VehicleFinder** y **VehicleController**: Para buscar *Vehículos*.
- **InsuranceCompanyFinder**, **InsuranceCompanySaver** e **InsuranceCompanyController:** Las clases dedicadas a las *Companías de Seguros*.

En ellas definiremos los mismos métodos ya descritos en la tercera etapa. En la Figura 4.26 podemos consultar la lista completa.

4.6.2. ClientHibernateDAM

Al igual que con la API, tendremos que crear las clases correspondientes a la búsqueda y grabación de *Clientes*, *Vehículos* y *Compañías de Seguros*.

Su funcionamiento es el mismo que el definido para los empleados con la única diferencia de que si queremos almacenar cambios en un *Vehículo*, este deberá estar asignado a un *Cliente* y lo que debemos hacer una vez realizadas las modificaciones es almacenar el Cliente haciendo uso del *ClientHibernateController*.

4.6.3. ClientUI

La interfaz de clientes ha de seguir el mismo diseño que el del módulo de empleados, con la diferencia de la edición de Vehículos. Para este caso se ha decidido añadir una pestaña al editor de clientes que permita añadir, editar y borrar los vehículos asociados a dicho cliente. Su prototipo se puede apreciar en la figura 4.27.

Por tanto finalmente tendremos que diseñar las siguientes ventanas:

Figura 4.26: Diagrama de clases de Client API

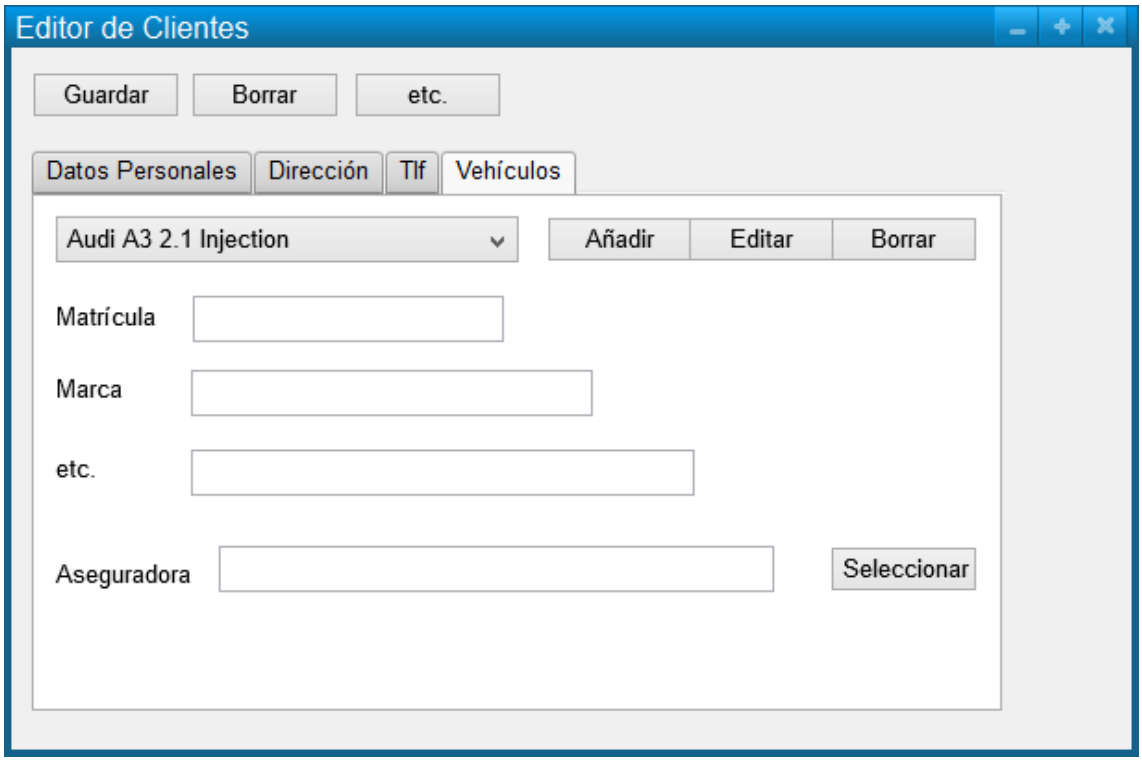

Figura 4.27: Prototipo de diseño del editor de clientes y vehículos

- **ClientBrowser**, **ClientEditor** y **ClientDetailedView:** Dedicadas a la búsqueda, edición y visualización de *Clientes*.
- **VehicleBrowser** y **VehicleDetailedView**: Para buscar y ver *Vehículos*.
- **InsuranceCompanyBrowser**, **InsuranceCompanyEditor** e **InsuranceCompanyDetailedView:** Ventanas dedicadas a las *Companías de Seguros*.

Su funcionamiento es muy similar a aquel visto cuando hemos hecho el diseño de los empleados. La única diferencia es la no existencia del editor de Vehículos. Si quisiésemos editar un vehículo, tendremos que buscarlo o bien usando el buscador de clientes o el de vehículos, dependiendo de los datos que conozcamos, para a continuación abrir el editor de clientes. Si hiciéramos uso del buscador de vehículos, al pulsar en editar se nos mostraría directamente la pestaña dedicada al vehículo, mientras que desde el buscador de clientes nos mostraría primero la pestaña de datos personales.

En la Figura 4.28 se puede apreciar el diseño final del editor de vehículos dentro del editor de clientes.

Otra pequeña diferencia apreciable es la inclusión de un buscador de compañías de seguros dentro del editor de clientes. Ya que para cada vehículo podemos almacenar la compañía que se encarga de asegurarlo, es necesaria la utilización del buscador para encontrar en

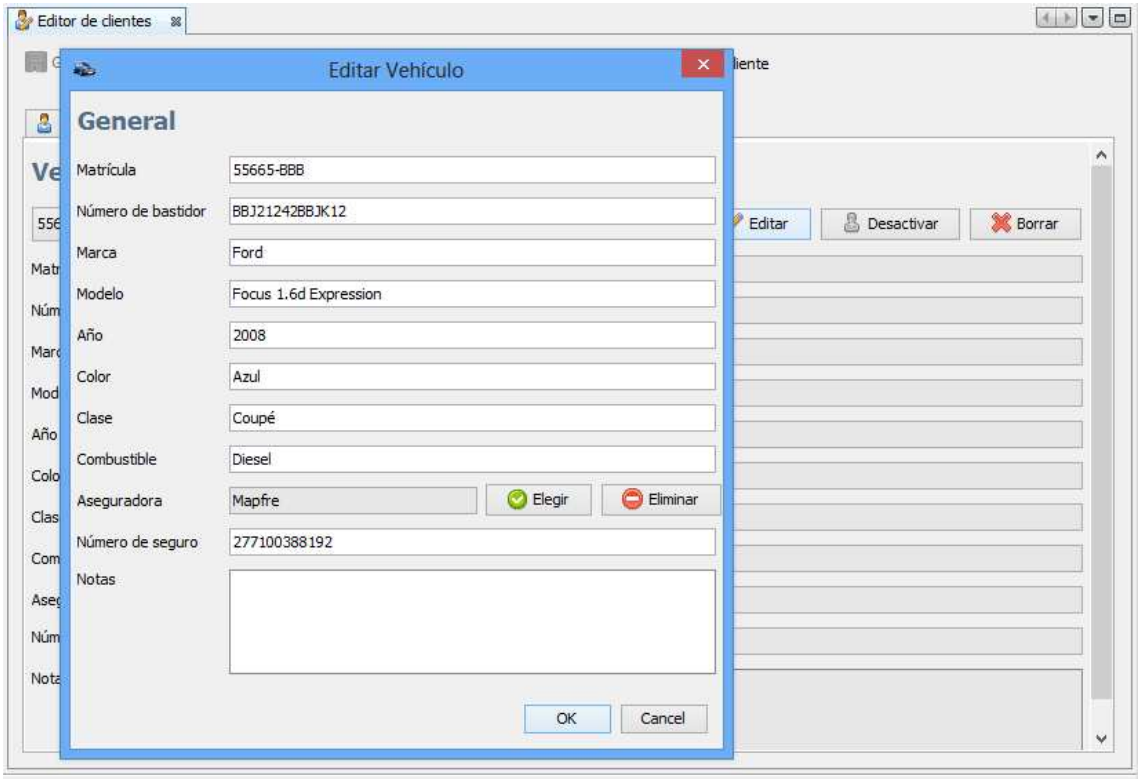

Figura 4.28: Diseño final del editor de vehículos

nuestro sistema la compañía concreta.

Si estando en la ventana de edición de un vehículo pulsásemos en el botón de elegir compañía, nos aparecería un cuadro de diálogo mostrando el buscador de compañías de seguros, junto con los botones de aceptar y cancelar. Si seleccionamos una compañía de entre los resultados obtenidos al buscar, el botón de aceptar se activará y nos permitirá realizar la asignación de la aseguradora.

Su diseño lo podemos ver en la Figura 4.29.

4.7. Quinta etapa: Diseño del módulo de usuarios y gestión de sus permisos

Este grupo de módulos sigue el mismo diseño definido en los anteriores casos, con la inclusión de la gestión de los permisos de los usuarios.

En el modelo de datos se definió que habrá un *Permiso* para buscar cada tipo de dato y otro para poder editarlo. Así tendremos permisos como el *PART_ORDER_EDIT*, que le permitiría editar pedidos de piezas o *INSURANCE_COMPANY* que permite buscar Ase-

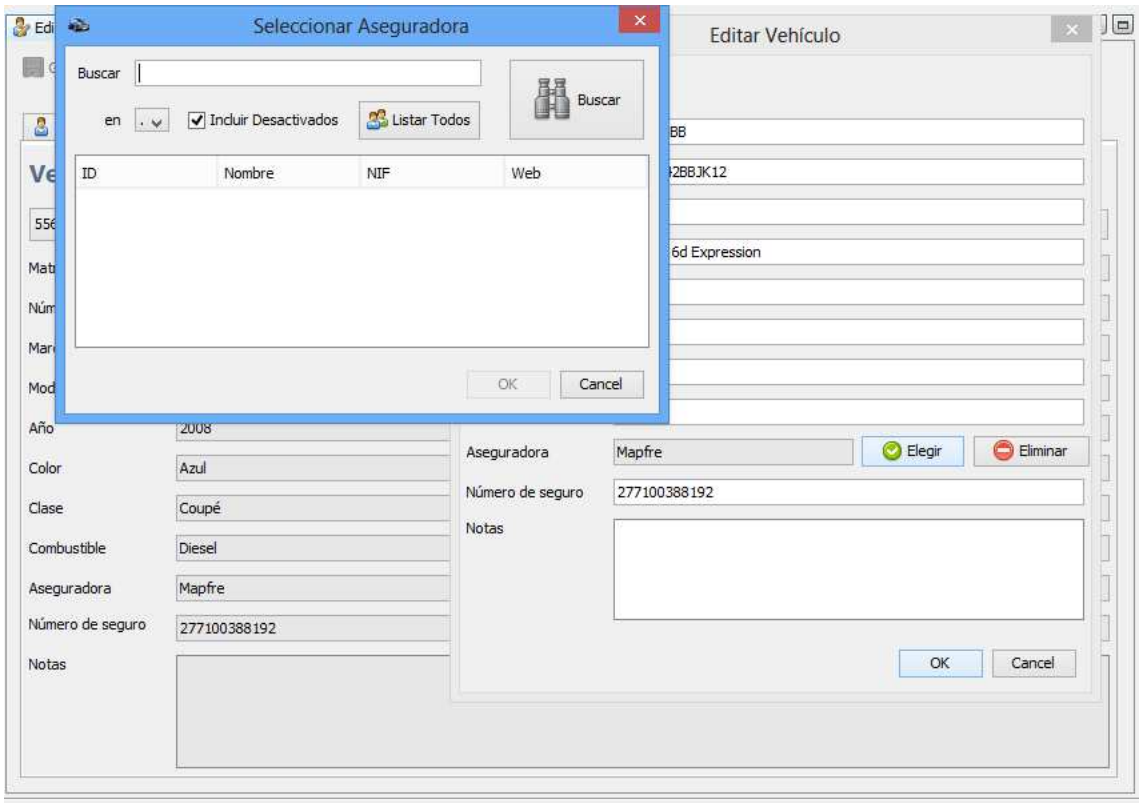

Figura 4.29: Diseño final del selector de aseguradoras

guradoras. Esta división de los privilegios nos permite configurar exactamente a qué puede acceder cada usuario.

4.7.1. Diseño del módulo de Login

El módulo de Login contendrá la definición de los métodos necesarios para confirmar las credenciales de un usuario y permitirle la entrada en la aplicación.

Para esta tarea nos ayudamos de una de los servicios de la Plataforma Netbeans; el instalador de módulos. Cuando Netbeans carga un módulo de nuestra aplicación por primera vez, llama al instalador, el cual realizará las operaciones que nosotros hayamos definido en él y al terminar devolverá el control al gestor de módulos.

Sobrescribiendo el método **restored()** de la clase *Installer* podemos indicar a Netbeans qué queremos realizar cuando este módulo sea instalado. Como necesitamos comprobar las credenciales del usuario que quiere entrar en la aplicación, lo primero que debemos hacer es mostrarle una ventana de Login, como la que se puede ver en la Figura 4.30.

Como se puede apreciar, la ventana tiene un diseño muy simple, con campos para la introducción del nombre de usuario y la contraseña, botones para confirmar o cancelar la

Figura 4.30: Diseño de la ventana de login

acción y un pequeño espacio para indicar errores.

Cuando el usuario introduce sus datos y pulsa en Aceptar, se llamada al método **login()** del gestor de conexión o **LoginHandler**, el cual se encargará de hacer las comprobaciones necesarias y permitir la entrada al usuario si sus datos son correctos.

En la Figura 4.31 se puede ver el diseño completo del módulo.

Como se puede comprobar, la clase *LoginHandler* es el centro del funcionamiento del módulo. Una vez recibe los datos del usuario gracias al *LoginPanel*, llama al método **login(user,password)** de la clase SecurityManager, que se encarga de comprobar, haciendo uso de la *UserAPI* tanto si el usuario existe como que su password es correcto. Una vez hecha esta comprobación, lee los permisos del usuario y le asigna los **PermissionGroup** correspondientes.

Los *PermissionGroup* son una enumeración que almacena los cambios que se le deben hacer a la aplicación si un usuario tiene o no tiene un determinado permiso. Así, si el usuario no tiene permiso para buscar clientes, se cargaría una configuración que le dice a la plataforma Netbeans que debe ocultar la ventana de búsqueda de clientes. Si posteriormente se conectase un usuario que sí tiene este permiso, se debería descargar la anterior configuración y se abriría una que volvería a hacer disponible la ventana.

Esta configuración se realiza con las clases *PermissionGroup*, *PermissionGroupFileSystem* y *PermissionGroupModuleSystem*, encargadas de definir las acciones a realizar, cargar archivos XML en los que se almacenan las acciones e instalar o desinstalar módulos, respectivamente.

Finalmente, si todo ha salido correctamente, el *SecurityManager* le indica al *LoginHandler* que los datos son correctos. El manejador del login ahora ejecuta las acciones necesarias para la configuración de la aplicación, definidas en las tres clases anteriores y coloca al usuario en el **LoginLookup**, un *Lookup* como los utilizados anteriormente que permitirá a cualquier módulo descubrir qué usuario está conectado y cuáles son sus permisos.

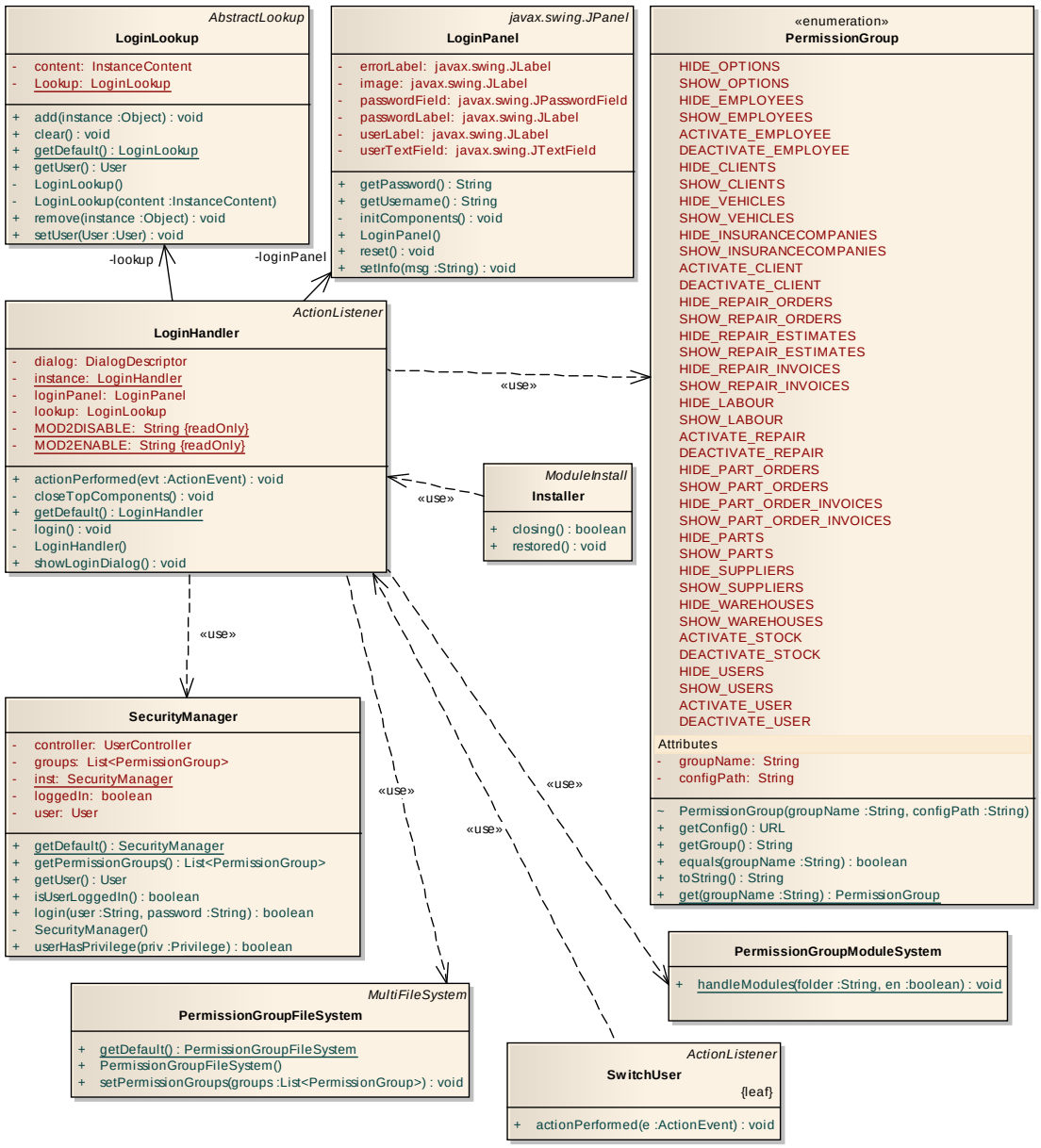

Figura 4.31: Diseño del módulo de login

Por último cierra el cuadro de diálogo de conexión y se muestra la ventana principal de la aplicación.

Este diseño sigue el definido por *Heiko Böck* en *The Definitive Guide to Netbeans Platform 7*[3].

4.7.2. UserAPI

Seguimos el mismo diseño de los módulos anteriores. En este caso tendremos que manejar las clases **User** y **Role**, correspondientes a los *Usuarios* y sus *Roles* respectivamente. Del mismo modo que antes, crearemos un controlador, un buscador y un grabador para cada tipo de dato, quedándonos con las clases apreciables en la Figura 4.32.

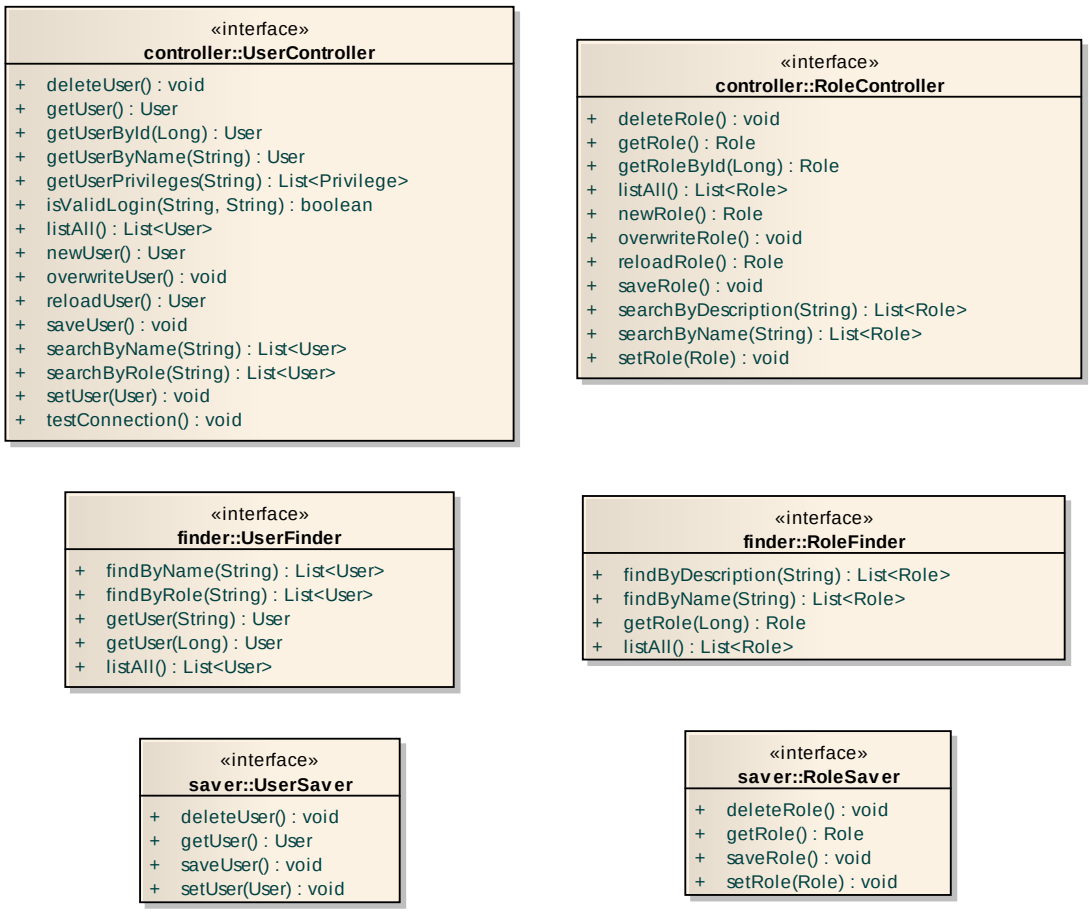

Figura 4.32: Diagrama de clases de User API

4.7.3. UserHibernateDAM

De nuevo seguimos el mismo diseño anterior. Creamos una clase que nos permitirá buscar a un usuario por su nombre o rol, grabarlo y un controlador para acceder a estas funciones y lo mismo para los roles, permitiendo buscarlos por nombre o descripción.

4.7.4. UserUI

Una vez llegados al diseño de la interfaz de gestión de usuarios tenemos una novedad, la existencia de permisos.

En este momento se ha de añadir a los buscadores, editores y vistas detalladas un método que compruebe si el usuario que está conectado al sistema tiene permiso para hacer ciertas acciones.

Como vimos al realizar la definición del módulo de login, cuando se comprueban los permisos se realiza la configuración automática de la interfaz de usuario de la plataforma Netbeans, ocultando los buscadores y las vistas detalladas para los que los usuarios no tienen permiso de uso, por lo que por esa parte no es necesario hacer ningún cambio.

Si un usuario puede acceder a un buscador, pero no se le quiere permitir editar esa información, debemos impedírselo desde la interfaz. Cuando se abre una ventana, Netbeans llama al método *componentActivated()* y este método llamará a uno definido por nosotros y llamado *checkPermissions()*, el que se ocupa de conectar con el *SecurityManager* y preguntarle si hay un usuario conectado y si tiene permiso para editar el tipo de dato del que se ocupa el buscador.

Si la respuesta es no, el botón de *Nuevo X* es desactivado. Otro cambio necesario es la desactivación de los botones de editar y borrar, que se realizan en las funciones que se ocupan de activarlos cuando se selecciona un resultado de la tabla.

Por último es necesario realizar también un pequeño cambio en los editores que manejen varios tipos de dato a la vez, como puede ser el editor de clientes, que permite editar también vehículos. El método *checkPermissions()* deberá comprobar que tenga permisos de edición para ambos tipos de datos o en caso de que sólo tenga de uno, impedir la modificación de los datos del otro.

El siguiente paso en el desarrollo del módulo es la creación de las ventanas de gestión de los usuarios. Seguimos el mismo diseño anteriormente definido, con la salvedad de que no crearemos vistas detallas para *Usuarios* o *Roles*, por ser datos de uso por la aplicación y no de consulta general. Así pues, crearemos los dos buscadores al uso, con la salvedad de ignorar el botón de muestra de la vista detallada y los dos editores, cuyo diseño final se puede apreciar en la Figura 4.33. Se puede apreciar que el editor de roles contiene

una tabla con la lista de permisos, en la cual se podrá realizar una selección múltiple de aquellos que queremos asignar al rol a editar.

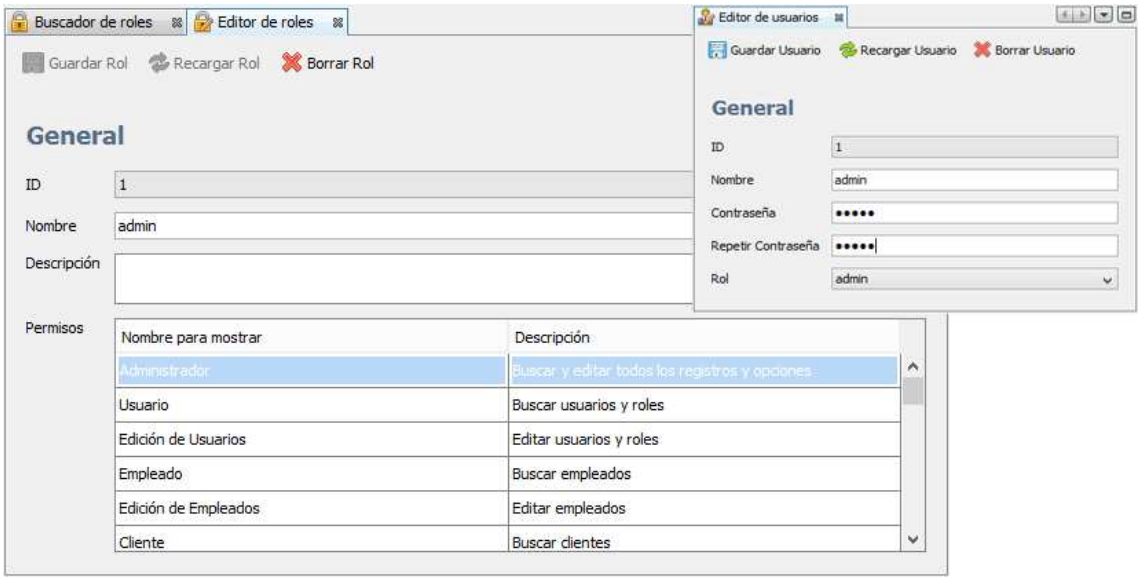

Figura 4.33: Diseño de los editores de Usuarios y Roles

4.8. Sexta etapa: Diseño del módulo de gestión de piezas

Seguimos utilizando el mismo diseño realizado anteriormente. En este caso deberemos almacenar *Piezas*, *Proveedores*, *Almacenes*, *Pedidos de piezas* y *Facturas de Piezas*.

4.8.1. StockAPI

De nuevo seguimos el mismo esquema, con la salvedad de que no necesitaremos un grabador de facturas ya que estas están necesariamente ligadas a la existencia de un pedido y serán modificadas utilizando el grabador de pedidos.

Su diseño completo se puede ver en las Figuras 4.34 y 4.35

4.8.2. StockHibernateDAM

Nos encontramos en el mismo caso de los módulos anteriores. Implementamos las clases y los métodos definidos en la *StockAPI*.

Figura 4.34: Diagrama de clases de los controladores de Stock API

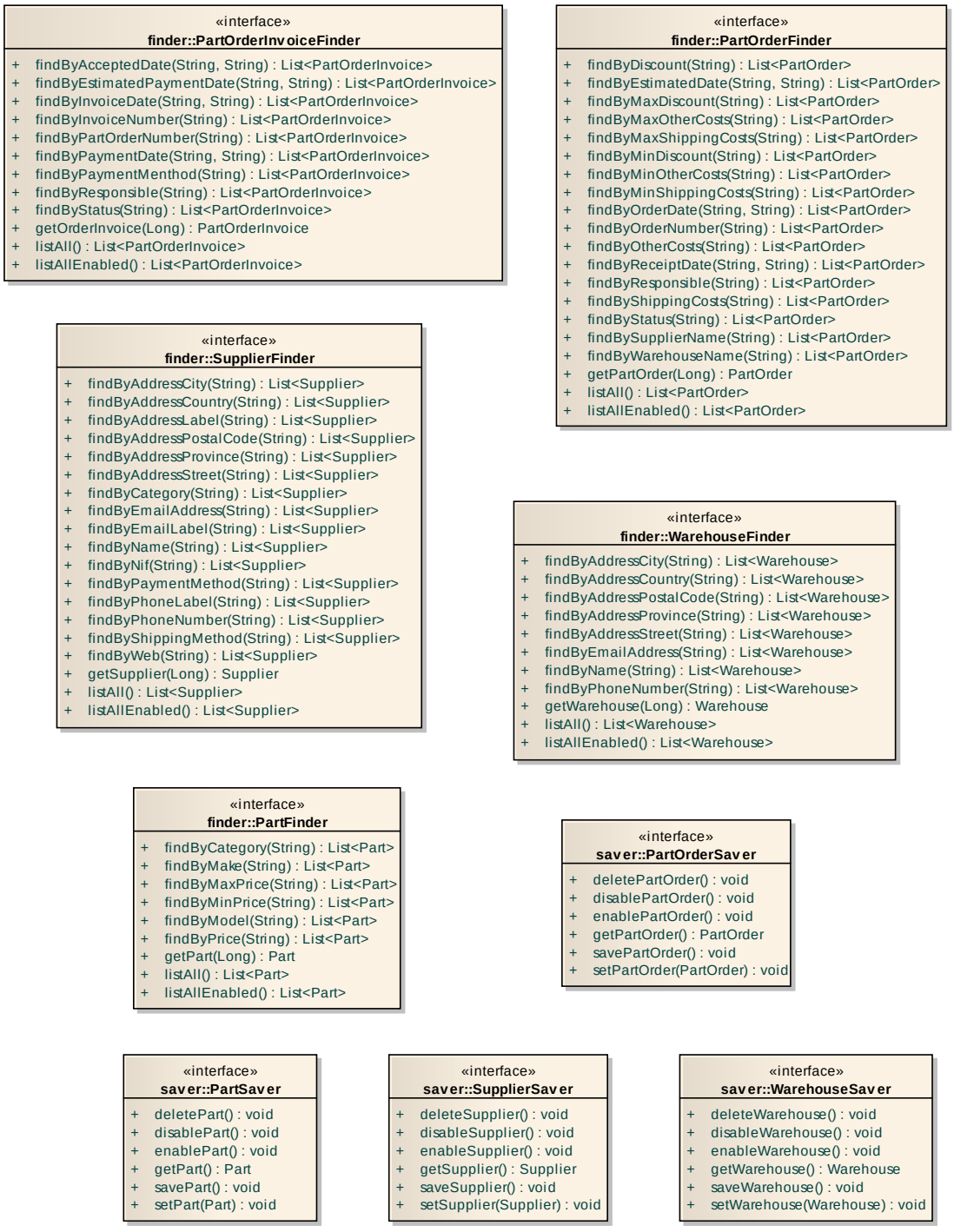

Figura 4.35: Diagrama de clases de los buscadores y almacenadores de Stock API

4.8.3. StockUI

La interfaz de gestión de stocks vuelve a seguir el mismo patrón definido anteriormente, de nuevo con algunos cambios.

Los buscadores, editores y vistas detalladas de los proveedores y almacenes siguen exactamente el mismo diseño anterior. Los buscadores de piezas y pedidos también son iguales, mientras que el buscador de facturas de pedidos tiene la misma peculiaridad que aquel de los vehículos. Al ser necesaria la existencia de un pedido para poder facturarlo, la edición de la factura se realizará a través del editor de pedidos, por lo que el buscador de facturas no mostrará el botón de nueva factura y enlazará al editor de pedidos si pulsamos en el botón correspondiente. De la misma manera, si pulsamos en la vista detallada de facturas, se nos mostrará la vista de los pedidos, con la pestaña de facturas seleccionada.

Por otro lado, en los editores de pedidos y piezas es necesario realizar más cambios. El editor de piezas deberá añadir una pestaña en la que se pueda consultar el stock de dicho objeto en nuestros almacenes. Esta pestaña contendrá una tabla en la que editaremos el stock y una combobox que nos permitirá seleccionar en qué almacenes tenemos guardada cada pieza. Su diseño es el de la figura 4.36.

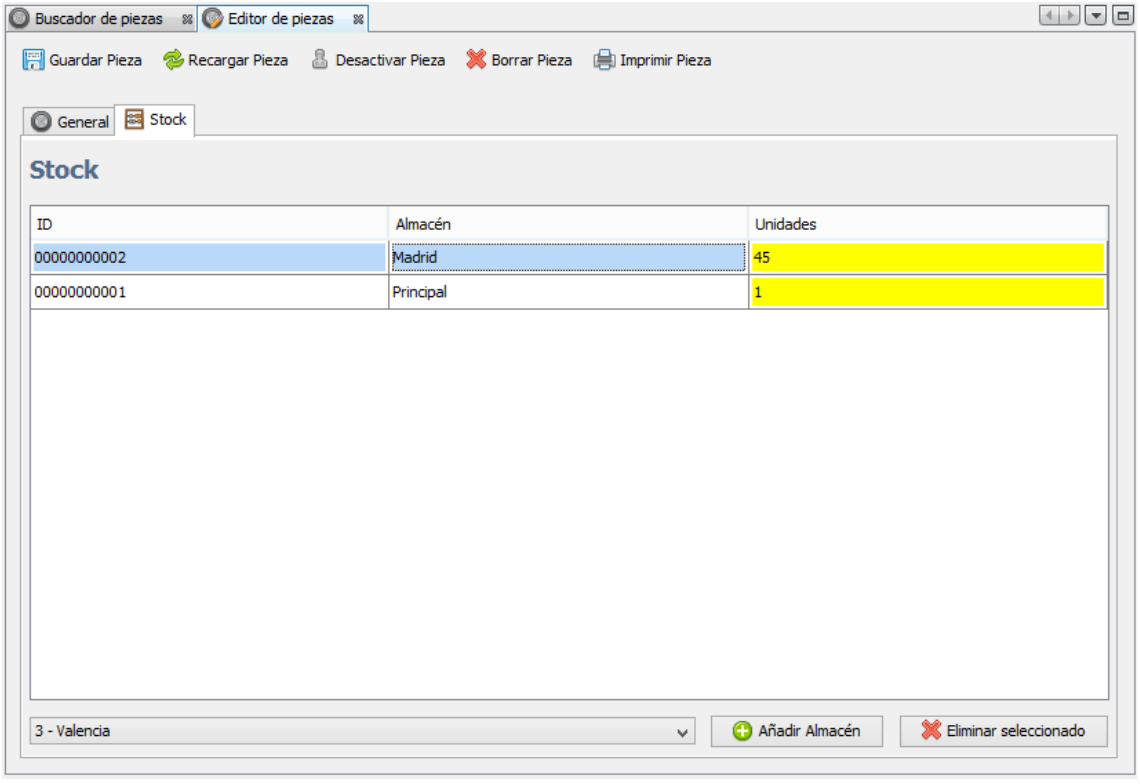

Figura 4.36: Diseño del editor de stock

El editor de pedidos deberá permitir elegir las piezas a pedir y almacenar su factura

en el sistema, por lo que hemos de añadir pestañas para realizar tales acciones. Para la adición de piezas, mostraremos una tabla en la que veremos los datos de la pieza añadida y se nos permitirá editar el número de unidades a pedir. También tendremos un botón que mostrará una ventana flotante con el buscador de piezas para que podamos añadir con comodidad todas las piezas deseadas al pedido. Por último se muestra el total del importe a pagar por las piezas, teniendo en cuenta el precio de cada una, el número de unidades a pedir y el IVA configurado en el programa.

La pestaña de gestión de la factura nos permitirá seleccionar las fechas en la que se emitió, aceptó y pagó así como el responsable de la gestión de dicha factura y otros detalles pertinentes.

En la pestaña de edición del pedido podremos selecionar el proveedor, el estado del pedido, las fechas de realización o entrega o el almacén al que se dirige, entre otros.

El diseño de esta ventana se puede ver en la Figura 4.37.

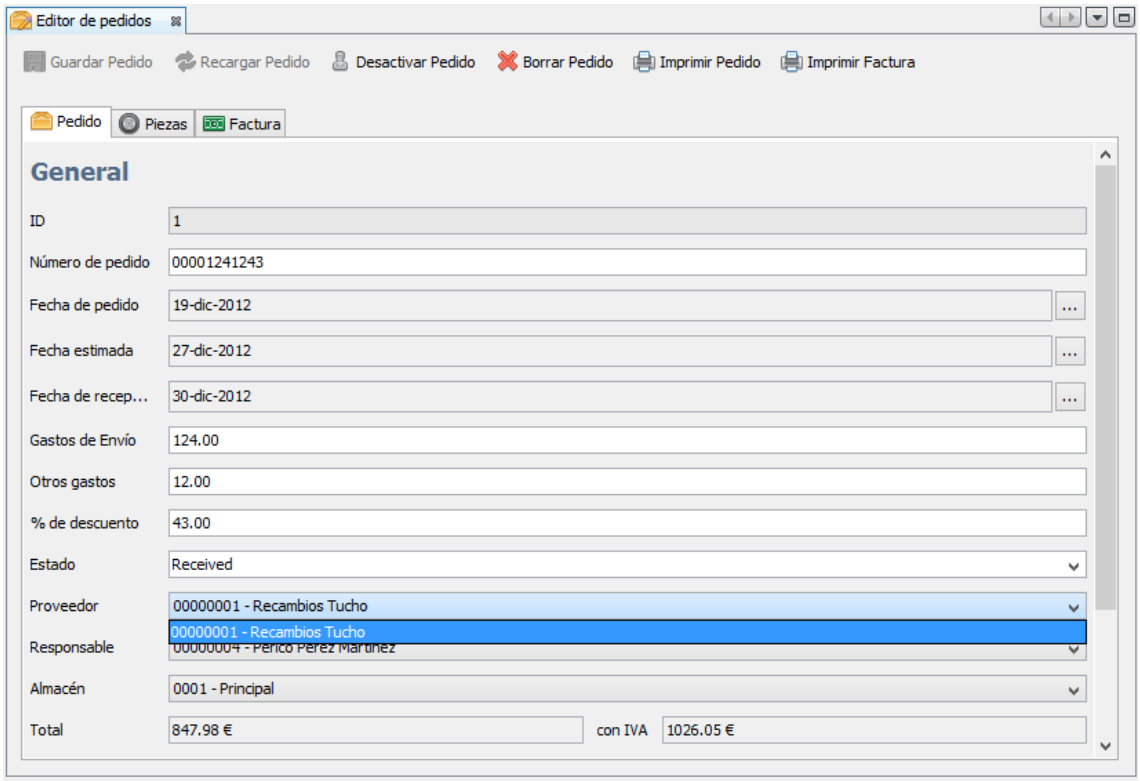

Figura 4.37: Diseño del editor de pedidos

4.9. Séptima etapa: Diseño del módulo de gestión de reparaciones

El último paso en el desarrollo de la aplicación consiste en el diseño e implementación de los módulos dedicados a gestionar reparaciones. Para ello deberemos crear las clases que permitan buscar y almacenar *Órdenes de Reparación*, *Presupuestos*, *Facturas* y la *Mano de Obra* necesaria para realizar la reparación. De nuevo mantenemos la misma división en módulos que en las secciones anteriores.

4.9.1. RepairAPI

Como ya ocurría con las facturas en el módulo de gestión de stock, las facturas de reparaciones están ligadas a una reparación concreta y no pueden existir sin ella, por lo que no nos será necesario diseñar una clase que se ocupe de grabarlas. Lo mismo ocurre con los presupuestos.

El diseño final del módulo se puede consultar en las Figuras 4.38 y 4.39

4.9.2. RepairHibernateDAM

De nuevo realizamos el mismo proceso que en las etapas anteriores: implementamos los métodos definidos en la API usando Hibernate como intermediario con la base de datos.

4.9.3. RepairUI

Nos encontramos con las mismas necesidades que en las ventanas de gestión de stock.

Tendremos que crear un buscador de facturas y otro de presupuestos, que no permitan crear presupuestos o facturas directamente y que al pulsar en el botón de editar nos redireccione al editor de reparaciones. Lo mismo ocurrirá al pulsar en los botones de vista detallada, que nos llevarán a la ventana de vista de reparaciones, con la pestaña correspondiente al registro que hemos seleccionado, a la vista.

El editor de reparaciones ha sido diseñado siguiendo el mismo patrón del editor de pedidos. Tendremos una pestaña en la que podremos introducir los datos de la reparación, otra en la que tendremos los datos del presupuesto, junto con las piezas que se utilizan y la mano de obra necesaria y, por último, una pestaña en la que editar la factura.

Su diseño final se puede ver en la la Figura 4.40.

Figura 4.38: Diagrama de clases de los controladores de Repair API

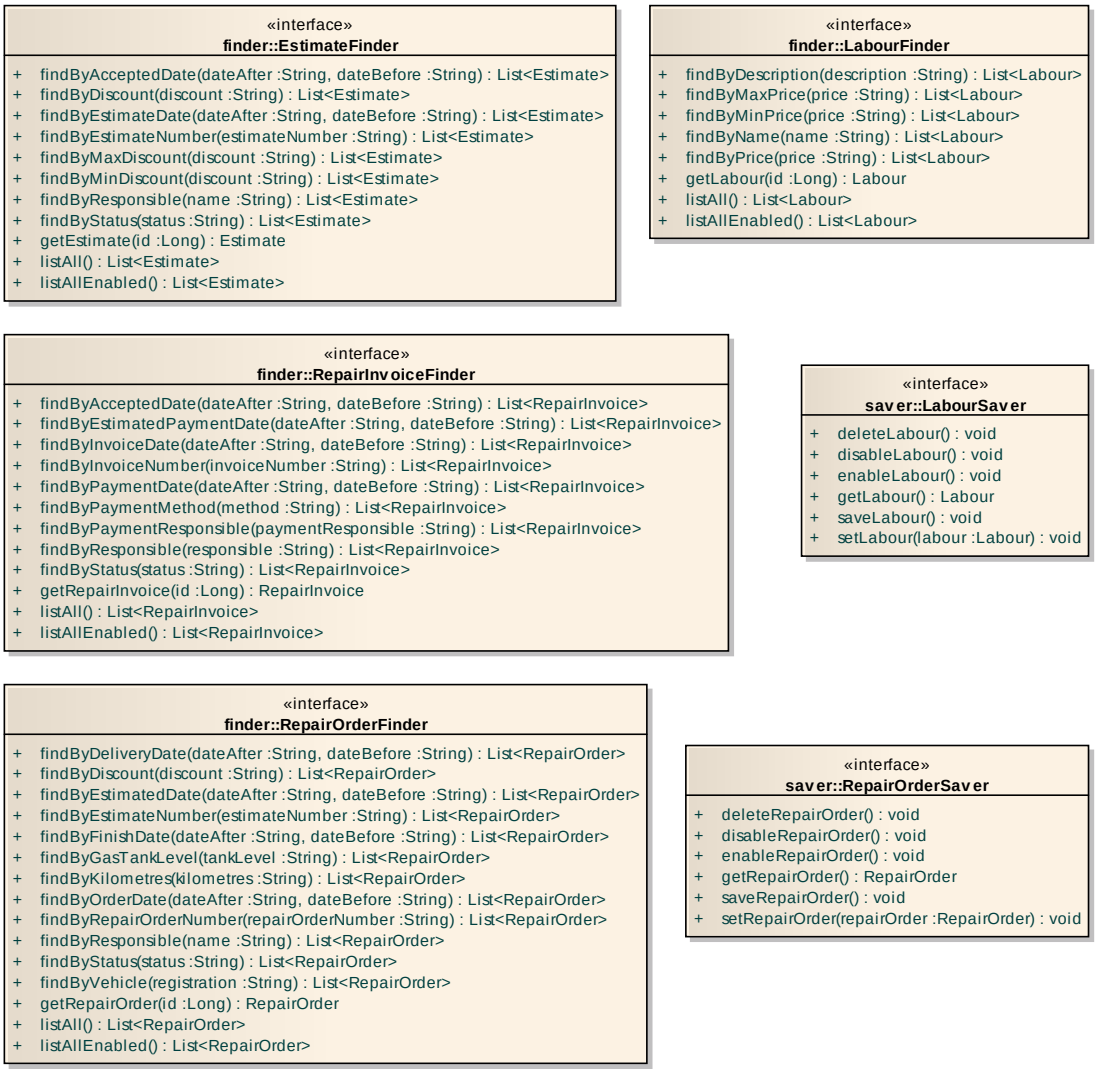

Figura 4.39: Diagrama de clases de los buscadores y almacenadores de Repair API

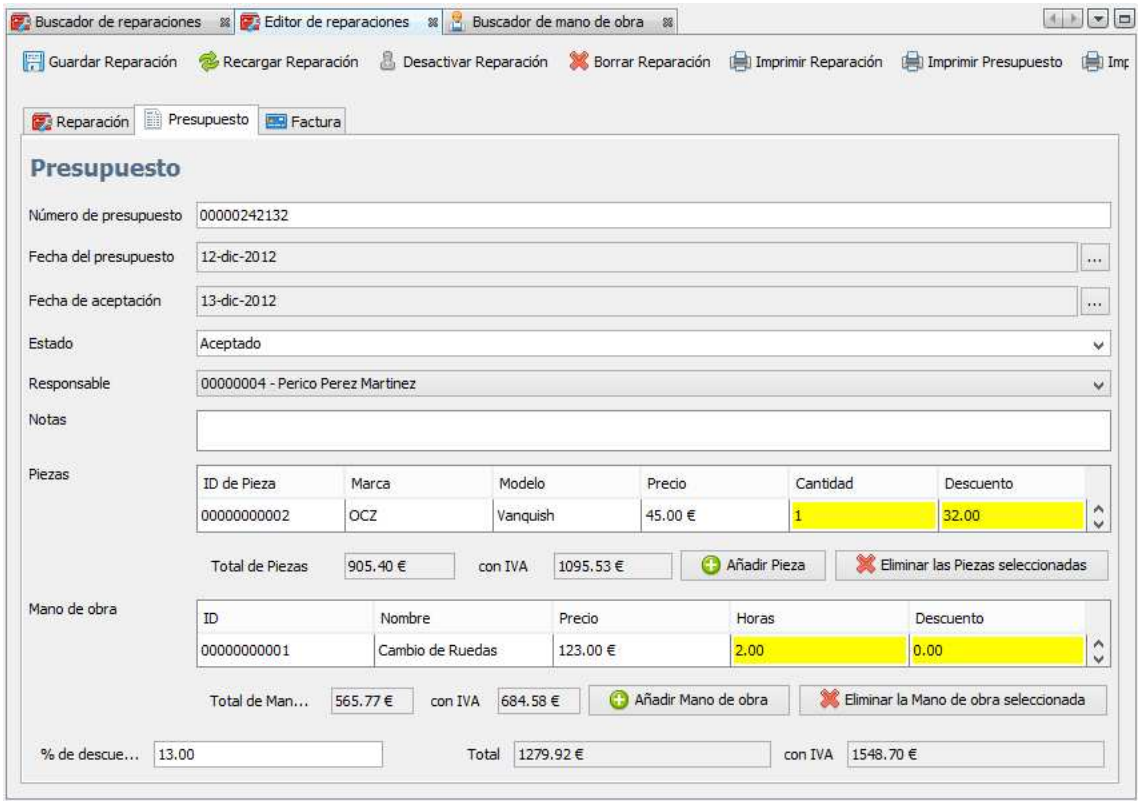

Figura 4.40: Diseño del editor de reparaciones

4.10. Patrones de diseño utilizados

Los patrones de diseño son soluciones bien documentadas que los expertos aplican para solucionar nuevos problemas porque han sido utilizadas con éxito en el pasado. Los expertos identifican partes de un problema que son similares a otros problemas que han encontrado anteriormente. Después, recuerdan la solución aplicada y la generalizan. Finalmente, adaptan la solución general al contexto de su problema actual [17].

La idea principal de los patrones es el concepto de estandarización de la información sobre un problema común y su solución. Representan una evolución importante en la **abstracción** y **reutilización** del software.

4.10.1. Patrones de creación

Singleton. Permite tener una única instancia de una clase en el sistema, a la vez que permite que todas las clases tengan acceso a esa instancia.

Es utilizado para realizar la comunicación de objetos entre las distintas ventanas que conforman la interfaz de usuario. Para cada entidad para la que existe un buscador se crea una clase llamada Lookup que se encarga de almacenar el objeto seleccionado en la tabla de resultados para que las ventanas de edición y vista detallada puedan tener acceso a él. La Plataforma Netbeans también hace uso de este patrón en las clases que hacen un seguimiento de las ventanas abiertas o de los cambios realizados en los campos de texto para permitir el uso de las funciones de deshacer y rehacer.

4.10.2. Patrones de comportamiento

Estrategia. Define un grupo de clases que representan un conjunto de posibles comportamientos. Estos comportamientos pueden ser fácilmente intercambiados en una aplicación, modificando la funcionalidad en cualquier instante.

Para el acceso a los datos se decidió crear una interfaz que defina las operaciones que es posible realizar con los datos y dejar la implementación concreta a cargo de otras clases para que pudieran ser fácilmente intercambiables. En nuestro sistema este acceso se realiza a través de Hibernate y se crearon las implementaciones que permiten conectar a la base de datos y traducir los resultados recibidos al modelo de datos. Si en un futuro quisiéramos acceder a través de otro medio sólo tendríamos que crear una nueva implementación, que estaría disponible junto con la de Hibernate y que podría ser elegida sin interferir con las funcionalidades existentes.

4.10.3. Patrones estructurales

Fachada. Proporciona una intefaz simplificada para un grupo de subsistemas o un sistema complejo.

Se utiliza este patrón en la definición de las interfaces de acceso a los datos. Para cada entidad que pueda ser almacenada en la base de datos existirá una clase que se ocupe de guardarla y otra de buscarla, quedando una clase controladora como fachada a todo el subsistema y que dará acceso a ambas funcionalidades de manera sencilla.

4.10.4. Patrones de sistema

Modelo-vista-controlador. Divide un componente o un subsistema en tres partes lógicas —modelo, vista y controlador— facilitando la modificación o personalización de cada parte.

Esta división se realiza para reducir el acoplamiento entre los métodos de almacenamientos de información y la manera de presentarlos. En nuestro sistema, la interfaz de usuario actúa como vista que se conecta al controlador que consiste en las APIs de acceso a la base de datos por medio de Hibernate, el cual almacena y recupera información del modelo de datos. La capa de vista también hace uso del modelo pero sólo para conocer la clase de los datos con los que trata, sin importarle su implementación interna.

Sesión. Ofrece una forma de que los servidores de sistemas distribuídos sean capaces de distinguir los clientes, permitiendo que las aplicaciones asocien el estado con la comunicación entre el cliente y el servidor.

Para cada cliente y conexión que se realice a la base de datos, se crea una sesión de Hibernate, que permite aislar el trabajo que se va a realizar durante la conexión y el cliente que lo realiza, permitiendo identificar, entre otros, problemas de concurrencia en la modificación de datos.

Transacción. Su propósito es agrupar una colección de métodos de forma que todos ellos finalicen correctamente o fallen de forma colectiva.

Al hacer modificaciones en la base de datos, usamos el patrón transacción para agrupar la lista de cambios a realizar. Si se produjese un error durante la modificación, todos los cambios realizados durante la transacción deben ser deshechos para impedir que la base de datos quede en un estado incorrecto. Si en cambio no se produjese ningún error, la transacción se daría como completa y las modificaciones serían confirmadas por la base de datos.

Capítulo 5

Prestaciones o funcionalidades destacadas

La funcionalidad principal del sistema es la de permitir gestionar las reparaciones a realizar en un taller. Permite almacenar los datos de todos los componentes implicados y definir con precisión en qué consiste la reparación.

Las características detalladas del sistema son las siguientes:

- **Conexión de usuarios:** Los empleados que vayan a hacer uso del sistema tendrán asignado un nombre de usuario y contraseña que les permitirá entrar en la aplicación.
- **Gestión de permisos:** Se puede limitar la cantidad de funcionalidades que se le presentan a determinado usuario, impidiendo que puedan acceder a ciertos datos o realizar modificaciones sobre otros.
- **Búsqueda de registros:** La aplicación ofrece la posibilidad de buscar entre todos los registros existentes en el sistema mediante el uso de sencillos buscadores.
- **Gestión de clientes:** El sistema permite guardar los datos personales de los clientes con los que tengamos relación. También almacenará los datos técnicos de los vehículos que posea junto con la compañía que lo asegura y sus datos.
- **Gestión de empleados:** Es posible almacenar los datos personales de nuestros empleados, así como su puesto de trabajo en la empresa, para luego poderlos identificar como responsables de realizar una reparación o de cobrar una factura.
- **Gestión de piezas:** Se permite guardar información de las piezas con las que trabaja la empresa, como marca, modelo y precio, junto con su stock en nuestros almacenes. También se podrán almacenar pedidos hechos a proveedores que hemos añadido al sistema, y sus facturas.

- **Gestión de reparaciones:** El sistema da la posibilidad de almacenar los datos de una reparación, así como las piezas implicadas y la mano de obra utilizada, junto con el vehículo al que se le ha realizado. Todos estos elementos deben estar ya registrados en el sistema y pueden ser seleccionados con facilidad gracias al uso de los buscadores.
- **Visualización de datos:** Los datos de todos los registros guardados pueden ser consultados con facilidad gracias a ventanas diseñadas únicamente para mostrar estos datos.
- **Internacionalización:** El sistema se ofrece en Castellano e Inglés, idiomas que serán seleccionados automáticamente dependiendo de la configuración del sistema operativo.
- **Configuración:** Se permite configurar el símbolo de moneda a mostrar y los datos del taller en el que se instala el sistema.

Capítulo 6

Pruebas realizadas

Para asegurar el correcto funcionamiento de cualquier sistema es necesario realizar una serie de pruebas que de una manera objetiva prueben que el resultado obtenido tras cierto proceso es el esperado.

La meta de probar es encontrar errores, y una buena prueba es aquella que tienen una alta probabilidad de encontrar un. Por tanto, un sistema informático debe diseñarse e implementarse teniendo en mente la *comprobabilidad*[14].

A continuación se detallan las pruebas realizadas durante el desarrollo de nuestro sistema.

6.1. Pruebas de unidad

En el software orientado a objetos la unidad mínima de prueba es una *Clase*. A diferencia de las pruebas de unidad del software convencional, que tienden a enfocarse en el detalle algorítmico de un módulo y en los datos que fluyen a través de su interfaz, la prueba de clase para el software OO se activa mediante las operaciones encapsuladas por la clase y por el comportamiento de estado de la misma[14].

Las pruebas de unidad deben ser automatizables, repetibles e idempotentes. Debemos obtener el mismo resultado sin importar cuándo se haya ejecutado la prueba y no deberán requerir la participación del usuario para su correcto funcionamiento. También deberemos buscar la máxima completud, intentando que nuestras pruebas abarquen el grueso del código fuente del sistema. Es importante diferenciar entre cubrir la ejecución de cierto código y probar el correcto funcionamiento del mismo. No nos basta con verificar que no se producen errores utilizando parámetros esperados, si no que nos tendremos que centrar principalmente en asegurar que el resultado es correcto cuando se reciben parámetros o

estados incorrectos.

La gran ventaja de las pruebas de unidad es la separación entre la interfaz y la implementación. Una prueba debería dar el mismo resultado con dos implementaciones distintas, siempre que ambas sigan los mismos requisitos.

Para la realización de pruebas de unidad en código Java el sistema más conocido y utilizado es **Junit**, un conjunto de bibliotecas creado por Erich Gamma y Kent Beck.

En el caso de nuestro sistema, se realizaron **pruebas de unidad** en los **módulos de datos** y en los **módulos de acceso a la base de datos**. Estos tests consisten en la inicialización de las clases del modelo de datos y la comprobación del resultado de la asignación de datos a sus atributos, tanto válidos como esperados incorrectos. En las clases con acceso a la base de datos se probó que se guardasen y eliminasen correctamente las entidades, así como que fuera posible recuperarlas posteriormente de la BD. El uso de un sistema de bases de datos en un test de unidad lo convierte en un test de integración.

6.2. Pruebas de integración

Las pruebas de integración permiten comprobar el funcionamiento global de un sistema, en las cuales módulos individuales del sistema son probados como un grupo.

Existen dos estrategias diferentes para las pruebas de integración en los sistemas orientados a objetos. La primera, *prueba basada en hebra*, integra el conjunto de clases requeridas para responder a una entrada o evento del sistema. Cada hebra se integra y prueba de manera individual y una prueba de regresión se aplica para asegurar que no ocurran efectos colaterales. El segundo enfoque de integración, *prueba basada en uso*, comienza la construcción del sistema al probar aquellas clases (llamadas *independientes*) que usan muy pocas clases de servidor. Después de probar las clases independientes, se examina la siguiente capa de clases que usan las clases independientes, llamadas *dependientes*. Esta secuencia de pruebas continúa hasta que se construye todo el sistema y se corresponde con la definida anteriormente como *prueba basada en uso*.

Para la prueba del funcionamiento correcto de los módulos relacionados con la base de datos se decidió evitar la creación de objetos simulados que imitasen el funcionamiento de dicha base de datos pues estos no tendrían en cuenta los posibles errores en la almacenamiento de los datos en un sistema real. Sería posible que tuviésemos un objeto que hayamos comprobado que es válido pero que al intentar almacenarlo no se produzcan los resultados esperados. Esta prueba ayuda a descubrir errores en la definición de las tablas y su traducción a código orientado a objetos.

Las pruebas realizadas consisten en la creación de objetos para las entidades almacena-

bles en el sistema, a los cuales se les asignan diversos datos para posteriormente almacenarlos en la base de datos. Comprobamos también que dichos datos se pueden recuperar correctamente y que el sistema puede detectar conflictos y errores cuando se almacenan objetos en un estado incorrecto.

6.3. Cobertura de las pruebas

Es posible comprobar mediante el software apropiado que nuestras pruebas abarcan buena parte o la totalidad de nuestro código. En el caso de código escrito en lenguaje Java, una buena opción es la herramienta **Covertura**, el cual puede generar informes en PDF o XML en los que podríamos ver la cantidad de código del sistema que se ha ejecutado durante el transcurso de las pruebas.

Hay que tener en cuenta que estos informes sólo nos indican que el código ha sido ejecutado. Queda a nuestra responsabilidad asegurarnos que dicha ejecución es correcta y que no se ha ejecutado solamente como consecuencia de la realización de otra prueba independiente.

6.4. Pruebas de la interfaz de usuario

Para la comprobación del funcionamiento de la interfaz de usuario se decidió realizar las pruebas de manera tradicional, esto es, introduciendo manualmente datos al sistema y comprobando que los resultados son los esperados.

Para ello por cada módulo de interfaces se probaron los editores, creando primero entidades con datos correctos y observando que todo funciona correctamente. Posteriormente se probó que se mostrasen los diálogos de error oportunos cuando se introduce un dato incorrecto y que el sistema avisase correctamente al usuario si se produjese algún conflicto al almacenar los datos.

También se probaron los buscadores, realizando búsquedas por cada atributo y asegurándonos de que obtenemos exactamente los resultados esperados y estos son mostrados de manera correcta.

Para las pruebas de las vistas detalladas, nos aseguramos de que los datos mostrados corresponden al objeto seleccionado, que incluye todos sus atributos y que estos cambian correctamente si se cambia la selección.

Por último se realizó una comprobación del correcto funcionamiento de los permisos de usuario, probando que los buscadores apropiados se ocultan cuando no se tiene permiso para usarlos y que los editores no permiten modificar objetos por usuarios sin los privilegios

correspondientes.

Ya que la interfaz de usuario de nuestro sistema no es más que un visor de datos o un conjunto de cuadros de texto que nos permiten editar información, estas pruebas simplemente aseguran que la traducción entre lo que se muestra en pantalla y lo que se asigna al objeto y viceversa es correcta. El funcionamiento correcto de los objetos y su almacenamiento ya se había probado con anterioridad en las pruebas de unidad e integración. Una vez todos los resultados fueron satisfactorios, la interfaz de usuario es considerada como completa.

6.5. Pruebas de aceptación

Las pruebas de aceptación consisten en una serie de tests conducidos a determinar si los requisitos del software definidos al principio del desarrollo fueron cumplidos. En la ingeniería del software suelen consistir en pruebas de caja negra que examinan las funcionalidades del sistema sin tener en cuenta sus estructuras internas o implementación.

En el caso de nuestro sistema, este fue presentado a un usuario experto, el cual examinó el funcionamiento del programa y aseguró que los requisitos definidos habían sido cumplidos.

6.6. Pruebas de adaptación

Una prueba de adaptación consiste en averiguar la facilidad de uso del sistema y las necesidades de entrenamiento de los usuarios previamente antes de que estos se puedan convertir en usuarios expertos del sistema.

Para la realización de esta prueba se presentaron una serie de actividades a realizar por un grupo de usuarios inexpertos y se observaron sus reacciones y el tiempo tomado por cada ejercicio, realizando cambios en la interfaz en los casos en los que su función no hubiera sido expresada o diseñada correctamente.

Capítulo 7

Planificación y evaluación de costes

7.1. Planificación estimada

La planificación del proyecto fue realizada en base a la división en etapas definidas en la sección 4.2.4. Además de la planificación inicial y las siete etapas de diseño e implementación es necesario asignar tiempo a una etapa final en la que se producirá la redacción de la memoria de trabajo.

La fecha de comienzo del proyecto se fija en el día 9 de Enero de 2012, con fecha de finalización estimada a principios de Julio de 2012. Se decidió dedicar dos semanas a la planificación inicial, que incluiría el análisis de requisitos y la definición de los casos de uso. Una vez completada se pasaría a la definición de la arquitectura del sistema, y al diseño del modelo de datos y la base de datos, hechos que deberían tomar una semana cada uno.

La siguiente etapa consistiría en crear la interfaz de usuario y el primer módulo utilizable del programa, la gestión de empleados, dedicando un mes a ello. El módulo de clientes nos tomaría 3 semanas mientras que la gestión de usuarios llegaría a 4 semanas de trabajo por la necesidad de crear un sistema de gestión de permisos.

Los módulos de gestión de stocks y de reparaciones tomarían 4 semanas cada uno, mientras que la etapa final en la que se redactaría la memoria se esperaría que durara 2 semanas y media.

Con estos datos obtenemos una fecha esperada de finalización en el día 4 de Julio de 2012.

Una vez decida la planificación pasamos al cálculo de los costes estimados de desarrollo.

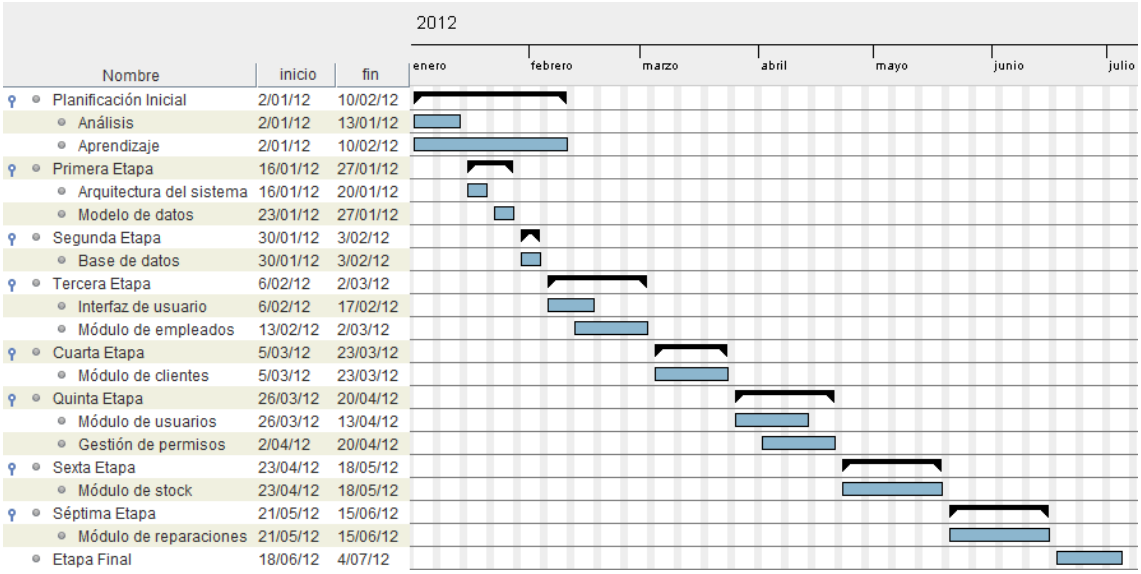

Figura 7.1: Planificación estimada

Estos costes han de incluir tanto los salarios a las personas implicadas en el desarrollo como los gastos en hardware y software necesarios para la creación y prueba del sistema. Gracias al uso de herramientas de software libre, como Netbeans y MySQL, no se ha producido ningún gasto en este departamento.

Siguiendo la planificación anterior se ha estimado que el proyecto tome un total del **118 días laborables**, en los que se trabajaría una media de 6 horas diarias, llegando a calcular un total de **708** horas de trabajo.

Se ha asumido el precio por hora de un programador en 15€ y el de un analista en 25€. Estimamos que las tareas de análisis llevarán un 30 % del tiempo total, quedando el resto para las tareas de programación. También se tienen en cuenta un total de 10 horas empleadas en conversaciones con el director del proyecto.

El desglose total puede ser consultado en el Cuadro 7.1.

Cuadro 7.1: Estimación de costes

Recurso	Porcentaje estimado	Precio/hora	Horas	Total
Analista	30 %	25 €	212,4	5.310 €
Programador	70 %	15 €	495,6	7.434 €
Director	-	25 €	10	250 €
				12.994 €

7.2. Planificación final

Por motivos ajenos al desarrollo del proyecto la fecha de finalización tuvo que ser retrasada hasta Enero de 2013. Este tiempo extra fue aprovechado para mejorar el diseño de todos los componentes desarrollados hasta el momento del cambio y la inclusión de mejoras en el sistema, por lo que la planificación anterior habría de ser modificada para acomodar los nuevos planes de desarrollo.

El desarrollo del sistema fue parado el día 10 de Mayo de 2012 y retomado el 31 de Julio. Durante este tiempo no se realizó ningún trabajo y anteriormente a esa fecha el sistema se encontraba a mitad de desarrollo de la sexta etapa.

Tras retomar el trabajo, se comenzaron a hacer correcciones y mejoras al mismo tiempo que se continuaba con el desarrollo de las etapas restantes. Por este motivo los módulos de gestión de stock y reparaciones tomaron 6 semanas cada uno mientras que la realización de mejoras y correcciones continuó durante 10 semanas más. Por último, la realización de la memoria y retoques finales llevó 5 semanas.

El diagrama final de la planificación puede ser consultado en la Figura 7.2.

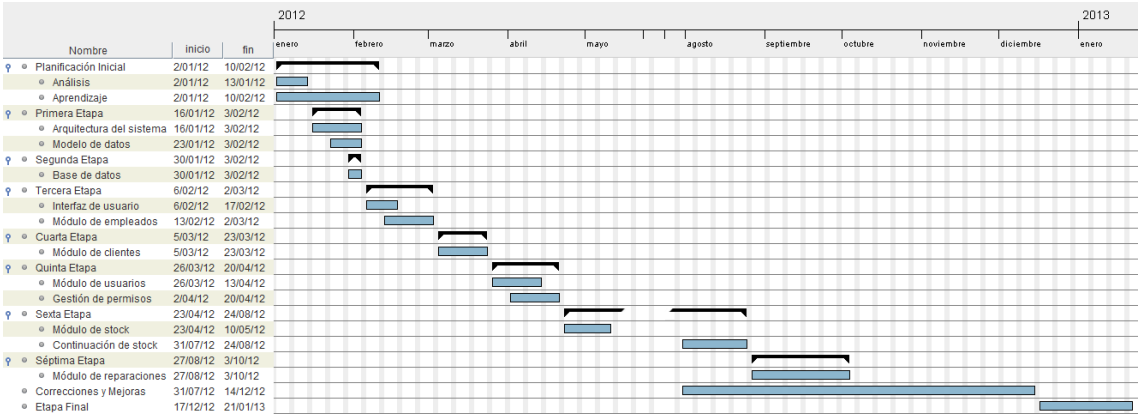

Figura 7.2: Planificación final

Para la estimación de costes final tenemos que tener en cuenta que la carga de trabajo antes y después del paro del desarrollo fue diferente. Hasta Mayo se cumplieron las 6 horas de media diarias, pero tras la vuelta el desarrollo se ralentizó hasta las 3 horas diarias durante los meses de Agosto, Septiembre, Octubre y Noviembre para pasar a una media de 6 horas los meses de Diciembre y Enero.

Consultando el esquema obtenemos que hasta Mayo se trabajaron aproximadamente 19 semanas, obteniendo un total de 570 horas de trabajo. Tras la reanudación se pasaron 19 semanas con una media de 3 horas diarias que nos llevan a un total de 285 horas y en Diciembre y Enero se trabajaron 7 semanas a 6 horas diarias, obteniendo 210 horas.

El total final de horas de desarrollo asciende a **1065** y su coste se desglosa en el Cuadro 7.2.

Cuadro 7.2: Costes finales de desarrollo

Recurso	Porcentaje estimado	Precio/hora	Horas	Importe
Analista	30 %	25 €	319,5	7.988 €
Programador	70 %	15 €	745,5	11.183 €
Director	-	25 €	10	250 €
				19.420 €

Capítulo 8

Conclusiones y contraste de objetivos

Como consecuencia de lo expuesto anteriormente podemos concluir que se han cumplido los requisitos funcionales definidos al principio del desarrollo de nuestro sistema. El objetivo de este proyecto era la creación de una herramienta de gestión sencilla, enfocada a pequeñas y medianas empresas con poca informatización y que necesitan un sistema que pueda cumplir con sus necesidades de igual manera que lo podría hacer algún otro de los caros sistemas de gestión empresarial profesionales y el programa desarrollado cumple perfectamente con esta tarea.

La estructura modular con la que se ha diseñado permite añadir o modificar funcionalidad de manera sencilla, pudiendo ser adaptado a las modificaciones en el modelo de negocio o a las necesidades cambiantes de la empresa. El desarrollo de esta herramienta hace disponible un software moderno, funcional y escalable que espera cubrir un nicho de mercado en el que existen demasiadas aplicaciones desfasadas o con un coste de adopción desproporcionado para una nueva empresa o para aquellas que quieran comenzar o modernizar sus sistemas.

Durante el desarrollo del proyecto se aprendió a crear interfaces de usuario y a integrarlas con sistemas y librerías existentes. Uno de estos sistemas es la *Plataforma Netbeans*, que ha demostrado tener un gran potencial para el desarrollo de aplicaciones de escritorio bajo lenguaje Java, como se puede apreciar por el resultado de este proyecto y la gran cantidad de aplicaciones existentes usando este sistema, tanto comerciales como libres, y mencionadas en su página web oficial [1].

Otro de las utilidades importantes de las que se hace uso es el sistema de mapeado objeto-relacional *Hibernate*, el cual ha facilitado enormemente la ardua tarea de convertir

[1] `http://platform.netbeans.org/screenshots.html`

un grupo de objetos descritos en Java a atributos, tablas y relaciones en una base de datos. Sin embargo, sigue siendo un trabajo con muchas complicaciones, cuya corrección se reduce muchas veces a prueba y error, por la falta de documentación en algunos casos o por los enormes cambios que se han venido produciendo en las últimas versiones, mientras intentan estandarizar todo el sistema para hacerlo compatible con las últimas especificaciones en materia de persistencia hechas por los desarrolladores de Java en Oracle.

Se ha de mencionar también la gran ayuda que proporciona un sistema como *JasperReports* y *DynamicReports*, que permiten generar con sencillez informes imprimibles con los datos almacenados en nuestro sistema, facilidad que no existe actualmente ni en la Plataforma Netbeans ni en las librerías Swing de Java.

El mayor problema que se ha encontrado durante el desarrollo ha sido la falta de manuales y documentación para las librerías mencionadas anteriormente. Tanto para Netbeans como para Hibernate existen en el mercado libros hechos por los mismos autores de las herramientas que pueden actuar perfectamente como tutoriales o manuales de uso pero que evidentemente no son de mucha ayuda en el caso de encontrarte con problemas de compilación o en tiempo de ejecución y teniendo que recurrir a la búsqueda en comunidades de ayuda que en algunos casos sólo pueden ofrecer respuestas anticuadas o parciales.

Otro de los grandes problemas y para el que la única solución es la obtención de experiencia en situaciones de trabajo real es el conocido como *feature creep*, un aumento continuo de las prestaciones que se le quieren otorgar al sistema sin saber apreciar correctamente el tiempo que habremos de destinar a completar cada una de ellas y que, al se agregadas en su totalidad, pueden conllevar un gran aumento en el tiempo de desarrollo o incluso en el alcance y objetivo del sistema. Es importante saber distinguir entre los requisitos definidos al principio del análisis y que deberá cumplir el software de aquellas mejoras o funcionalidades extra que sólo deberíamos implementar después de haber computado correctamente el tiempo que demoraran el trabajo.

Para concluir puedo asegurar que la realización de este sistema me ha enseñado como se ha de gestionar un proyecto de principio a fin y buena parte de los problemas que pueden surgir durante su implementación, así como la disponibilidad de numerosas herramientas cuyo objetivo es facilitar la vida al desarrollador.

Capítulo 9

Líneas futuras

Cualquier tipo de aplicación de gestión empresarial puede ser ampliada con un gran número de mejoras, como el soporte de gestiones avanzadas, automatismos o integración con otros sistemas.

En el caso de nuestro sistema, uno de los cambios que podría reportar grandes beneficios, pero que supondría rehacer gran parte del desarrollo, seria convertir el sistema a una arquitectura 3-tier. Como se explica en la sección 3.4.1, una arquitectura 3-tier o de 3 capas consiste en la división de la aplicación en 3 secciones, cada una de las cuales se especializará en realizar una de estas tres tareas:

- **Capa de Presentación:** Su función es la de mostrar los datos obtenidos de la *Capa de Aplicación* y dar la posibilidad de realizar operaciones sobre ellos. Se correspondería a la interfaz de usuario que hemos desarrollado.
- **Capa de Aplicación:** Permite la comunicación entre la *Capa de Presentación* y la *Capa de Datos*. Se ocupa de acceder y procesar los datos, controlando la funcionalidad de la aplicación.
- **Capa de Datos:** Se compone de servidores de bases de datos cuyo propósito es almacenar los datos del sistema. Se correspondería con el servidor MySQL de nuestro sistema.

Deberíamos pues desarrollar una *Capa de Aplicación* que acceda a nuestro servidor y hacer que nuestra interfaz de usuario pueda comunicarse con ella. Este cambio daría la posibilidad, por ejemplo, de crear diferentes interfaces de usuario en múltiples sistemas, como pueden ser plataformas móviles o sistemas web, que accedan a los mismos datos y de la misma manera que lo haría una aplicación de escritorio, o la posibilidad de crear una lógica de negocio especializada en trabajar con múltiples usuarios simultáneamente de manera más precisa a cómo la hemos implementado en nuestra aplicación.

Una de las posibles ampliaciones podría ser la integración de una gestión avanzada del stock y los almacenes de la empresa, permitiendo hacer una lectura automática de albaranes de entrada de producto o usando lectores de códigos de barra cuando compremos o vendamos una pieza, modificando el stock disponible acordemente.

También sería interesante la posibilidad de conectar nuestro sistema con terminales de venta al público o con sistemas de gestión de contabilidad, pudiendo llevar automáticamente las cuentas de la empresa gracias a conocer tanto los pedidos realizados, como las reparaciones o el salario de los empleados.

Los vehículos actuales llevan instalada de serie una centralita que se ocupa, además de controlar todo el sistema eléctrico del coche, de analizar las lecturas procedentes de los distintos sensores disponibles con el fin de detectar posibles problemas. Existen sistemas informáticos que permiten realizar una lectura de estos análisis, los cuales podríamos querer integrar con nuestro sistema, pudiendo añadir los resultados a la ficha de una reparación, para así saber exactamente qué le ocurre al vehículo antes de comenzar a repararlo.

Por último, se podría adoptar un sistema de adquisición electrónica de los catálogos de piezas de los proveedores o de los manuales de reparación oficiales de los fabricantes de vehículos, pudiendo así añadir directamente todos los datos referentes a una pieza tanto a nuestra gestión de stocks como a los pedidos a realizar, o consultar los procedimientos recomendados para realizar una reparación de manera correcta.

Todas estas mejoras beneficiarían tanto a las pequeñas y medianas empresas a las que va destinado este sistema como a la posibilidad de ser utilizado en grandes talleres o franquicias multinacionales, necesitadas de una gestión más precisa y automatizada debido a la gran cantidad de empleados involucrados y al gran volumen de trabajo registrado.

Apéndice A

Acrónimos

API Application Programming Interface - Interfaz de Programación de Aplicaciones

AWT Advanced Window Toolkit - Cada de Herramientas para Ventanas Avanzada

BD Base de Datos

BOM Bill of Materials - Lista de Materiales

CU Caso de Uso

DAM Data Access Module - Módulo de Acceso a Datos

DMS Dealer Management System - Sistema de Gestión de Concesionarios

DSS Decision Support System - Sistema de Soporte a Decisiones

EIS Executive Information System - Sistema de Información Ejecutiva

ERP Enterprise Resource Planning - Sistema de Planificación de Recursos

GWT Google Web Toolkit - Cada de Herramientas para Web de Google

IBM International Business Machines

ICS Inventory Control System - Sistema de Control de Inventario

IDE Integrated Development Environment - Entorno de Desarrollo Integrado

IRF Fichero de Registro de Inventarios

IVA Impuesto Sobre El Valor Añadido

JDBC Java Database Connectivity - Conectividad a Bases de Datos de Java

JDK Java Development Kit - Kit de Desarrollo de Java

JDO Java Data Object - Objeto de Datos de Java

JPA Java Persistence API - Intefaz de Programación de Persistencia de Java

JPQL Java Persistence Query Language - Lenguaje de Consulta para Persistencia de Java

JRE Java Runtime Environment - Entorno de Ejecución de Java

JVM Java Virtual Machine - Máquina Virtual de Java

ME Mobile Edition - Edición Móvil

MIS Management Information System - Sistema de Información Gerencial

MRP II Manufacturing Requirement Planning - Sistema de Planificación de Requisitos de Fabricación

MRP Materials Requeriments Planning - Sistema de Planificación de Requisitos de Materiales

MS Microsoft

NIF Número de Identificación Fiscal

NSS Número de la Seguridad Social

OAS Office Automation System - Sistema de Automatización de Oficinas

OO Object Oriented - Orientado a Objetos

ORM Object Relational Mapping - Mapeado Objeto-Relacional

P2P Peer to Peer - Punto a Punto

PDF Portable Document Format - Formato de Documentos Portable

PMP Programa Maestro de Producción

RIA Rich Internet Applications - Aplicaciones Ricas de Internet

SGBD Sistema de Gestión de Bases de Datos

SIG Sistema de Información Geográfica

SQL Structured Query Language - Lenguaje de Consulta Estructurado

SWT Standard Widget Toolkit - Caja de Herramietas de Widgets Estándar

TPS Transaction Processing System - Sistema de Procesamiento de Transacciones

UI User Interface - Interfaz de Usuario

UML Unified Modeling Language - Lenguaje de Modelado Unificado

XML Extensible Markup Language - Lenguaje de Marcado Extensible

Manual de usuario

Instalación

Para el correcto funcionamiento del sistema es necesario disponer de un servidor de bases de datos MySQL, de la máquina virtual de Java, de la aplicación de escritorio desarrollada y de un usuario con permiso de acceso al sistema.

El servidor de bases de datos no tiene porque ser un ordenador independiente, si no que puede residir en la misma máquina en la que se instale la aplicación. En el caso de que múltiples máquinas vaya a hacer uso del sistema, el ordenador en el que resida el servidor deberá permitir la conexión del resto de máquinas y estar disponible siempre que se quiera hacer uso de la aplicación.

El proceso de instalación es el siguiente:

- **Instalar MySQL:** Si la empresa no dispone de servidor MySQL, habrá de instalarlo.

  La instalación es altamente dependiente del sistema operativo por lo que se recomienda seguir la guía de instalación proporcionada en el manual de MySQL y disponible en `http://dev.mysql.com/doc/refman/5.0/es/installing.html` para su versión en español o en `http://dev.mysql.com/doc/refman/5.5/en/index.html` para la última versión estable en idioma inglés.

- **Creación de la base de datos:** Es necesaria la creación de las tablas en las que se almacenarán los datos y el primer usuario que tendrá acceso al sistema.

  Para ello se realiza la importación del script *create_tables.sql* incluido en el disco de instalación. Este script se ocupa de crear la base de datos *easyRepair* y todas las tablas y columnas necesarias para guardar los datos, junto con un usuario administrador por defecto para el primer acceso al sistema y un usuario de MySQL con permiso de acceso a la base de datos.

  La base de datos se creará con el nombre *easyRepair*. El usuario de MySQL se llamará *easyUser* y tendrá la contraseña *easyPass*, mientras que el usuario de acceso a la aplicación llevará por nombre *admin* y contraseña *admin*.

Se recomienda personalizar los nombres de usuario y las contraseñas para aumentar la seguridad en el acceso a la base de datos.

- **Instalación del entorno de ejecución de Java:** Para la ejecución de la aplicación es necesaria la instalación de la máquina virtual de Java y del entorno de ejecución en su versión 6 o superior en todas las máquinas cliente. Este entorno es conocido como **JRE** y se recomienda instalar la última versión disponible que incluirá todas las correcciones de seguridad hasta la fecha.

  La descarga de la última revisión del entorno está disponible en `http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html` y su manual de instalación se puede consultar en `http://docs.oracle.com/javase/7/docs/webnotes/install/index.html`.

- **Instalar la aplicación:** El proceso concreto para cada sistema se detalla en las secciones siguientes.

- **Configuración de la base de datos:** En el directorio principal de la aplicación existirá un directorio llamado *netbeans* y dentro de él otro llamado*config*, en el que habrá un archivo de texto con el nombre *hibernate.cfg.xml*. Este archivo se encarga de almacenar la configuración de acceso a la base de datos y será necesaria la edición de las siguientes propiedades:

  - **connection.url:** Correspondiente a la dirección de la máquina en la que se ha instalado el servidor y el nombre de la base de datos. Si el servidor se ha instalado en la máquina local con las opciones por defecto, el valor que habría que introducir sería *jdbc:mysql://127.0.0.1:3306/easyRepair*.
  - **connection.username:** El nombre de usuario con acceso a la base de datos. Por defecto *easyUser*.
  - **connection.password:** La contraseña del usuario antes mencionado. Por defecto *easyPass*.

- **Configuración de la conexión a la BD:** Con la creación de la base de datos se ha creado un usuario por defecto con permiso de administración del sistema. Este usuario tiene el nombre **admin** con la contraseña **admin**. Al ser un usuario por defecto, por seguridad se recomienda que se edite al menos la contraseña antes de hacer disponible la aplicación.

  Cuando se ejecute la aplicación aparecerá una pantalla de login en la que se habrá de introducir el nombre de usuario y contraseña. Una vez accedido al sistema, se habrá de acceder al Navegador de Usuarios. Desde aquí podrá crear nuevos usuarios o editar el ya existente pulsando en el botón *Buscar* o *Listar Todos* para encontrarlo y luego en el botón *Editar Usuario*.

- **Configuración de la aplicación:** El último paso consiste en la configuración de la aplicación para su uso. Accediendo con un usuario que tenga permiso de *Administrador* podremos entrar en la pantalla de opciones de la aplicación. Esta pantalla permite editar la moneda en la que se mostrarán los precios, la tasa de IVA que se aplicará y los datos de la empresa que se mostrarán en los informes imprimibles.

  Para acceder a las opciones pulsar en el menú *Herramientas* y a continuación en la entrada *Opciones*.

Tras estos pasos la aplicación está correctamente instalada y disponible para su uso.

Instalación manual de la aplicación

En el disco se incluye el archivo *easyRepair.zip*, el cual incluye los archivos ejecutables necesarios para el funcionamiento de la aplicación en sistemas Windows, Linux y OS X.

Puede extraer este archivo en el directorio que desee.

Para acceder a la aplicación desde Windows ejecute el archivo *easyRepair.exe* si su sistema es de 32 bits o *easyRepair64.exe* si es de 64, residente en la carpeta *bin*. Para sistemas Mac o Linux, deberá ejecutar el lanzador *easyRepair*, disponible en la misma carpeta.

Se recomienda crear un acceso directo para facilitar el acceso a la aplicación por los usuarios del sistema.

Instalación automática de la aplicación en sistemas Windows y Linux

Si lo desea puede hacer uso del instalador incluido en el disco. Este instalador le guiará en la tarea de escoger el directorio de instalación y la creación de un acceso directo en el Menú de Inicio y en el Escritorio.

Para realizar la instalación con asistente, ejecute el archivo *easyrepair-windows.exe* disponible en el disco de instalación si se encuentra en un sistema Windows o el archivo *easyrepair-linux.sh* si se encuentra en Linux.

Una vez abierto el instalador, este le mostrará la pantalla de bienvenida. Si pulsa en continuar se le mostrará la pantalla de licencia, que deberá aceptar para poder realizar la instalación. A continuación se le pregunta el directorio donde desea instar la aplicación y si quiere crear iconos de acceso directo en el Escritorio y en el Menú de Inicio. Si pulsa en continuar llegará a la pantalla de resumen, que le indicará el directorio de instalación y el tamaño total de esta. Si está de acuerdo pulse en Instalar y se producirá la instalación. Por

último, se le mostrará una ventana de confirmación de la instalación y se le preguntará si quiere ejecutar ahora el programa.

Una vez terminada la instalación podrá acceder a la aplicación haciendo uso de los accesos directos del Escritorio o Menu de Inicio o a través del archivo *easyRepair.exe* si su sistema Windows es de 32 bits o *easyRepair64.exe* si es de 64, residente en la carpeta *bin* dentro del directorio en el que ha realizado la instalación. Para sistemas Linux, deberá ejecutar el lanzador *easyRepair*, disponible en la misma carpeta.

Primeros Pasos

Una vez ejecutado el sistema, la primera ventana que se le muestra al usuario es la pantalla de login.

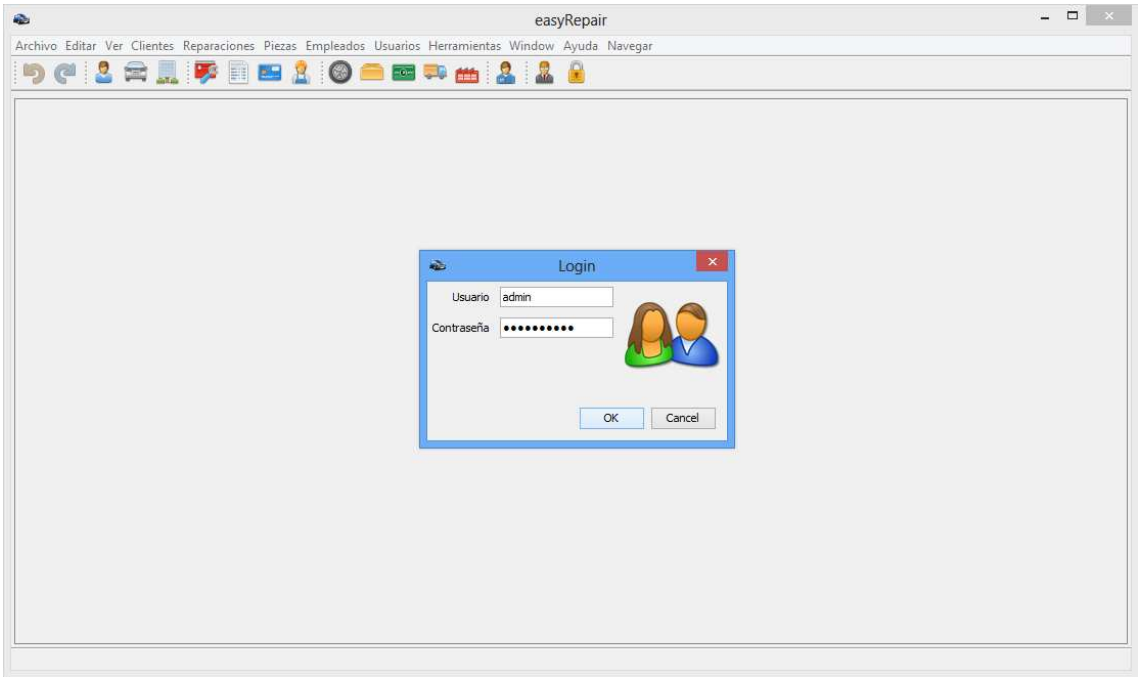

Figura A.1: Ventana de login

En ella deberá proporcionar sus credenciales para que se le permita el acceso al sistema.

Una vez dentro de la aplicación, se ocultará la ventana de conexión y se mostrará la pantalla principal, consistente en una barra de menús y una barra de herramientas que contienen los accesos a los buscadores de cada tipo de dato.

Búsqueda de registros Si quisiese buscar un registro, por ejemplo un cliente, accedería primero al buscador de clientes.

El buscador presenta un cuadro de texto donde introducir la consulta, un selector para elegir a qué atributo del cliente corresponde la consulta y un botón para buscar.

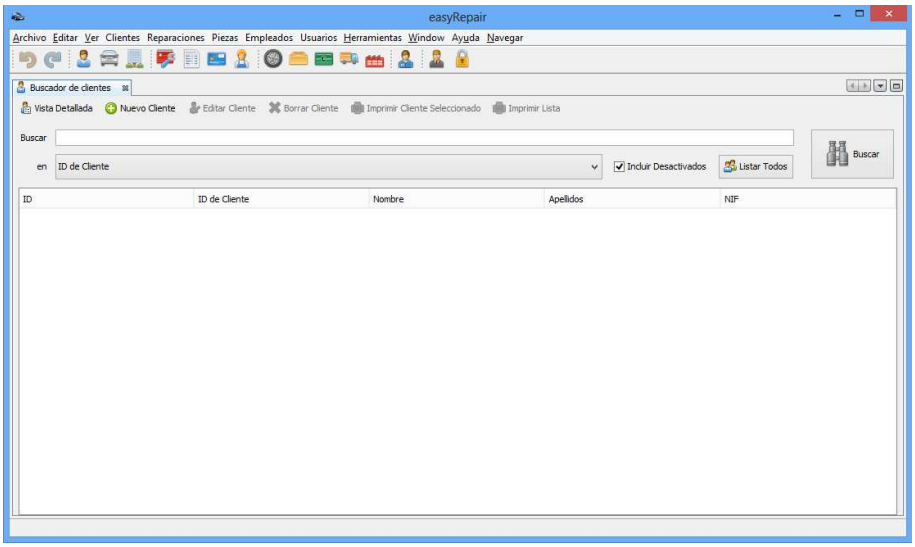

Figura A.2: Ventana del buscador de clientes

Si pulsase en buscar se le mostraría una tabla con los registros coincidentes como se puede apreciar en la Figura A.3.

También podría listar todos o refinar la búsqueda excluyendo a los clientes que se encuentren desactivados.

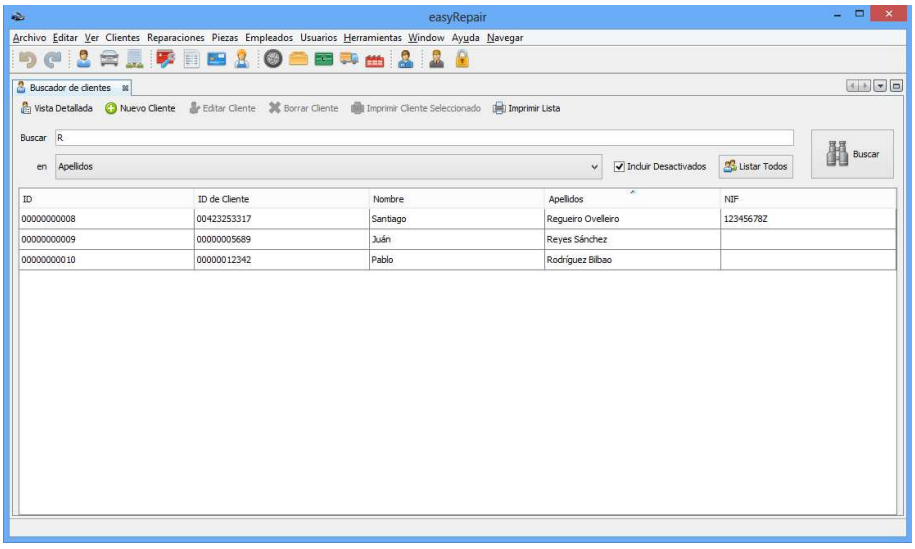

Figura A.3: Ventana del buscador de clientes - Listado

Desde el buscador se accede al resto de operaciones que se pueden realizar con un cliente, que son consultar sus datos, crear uno nuevo, editarlo, borrarlo, imprimir el registro del cliente seleccionado o el listado de todos los mostrados.

Consulta de registros Una vez encontrado un registro a través del buscador, es posible consultar todos sus datos gracias al uso de la ventana de Vista Detallada a la que se accede pulsando en el botón Vista Detallada.

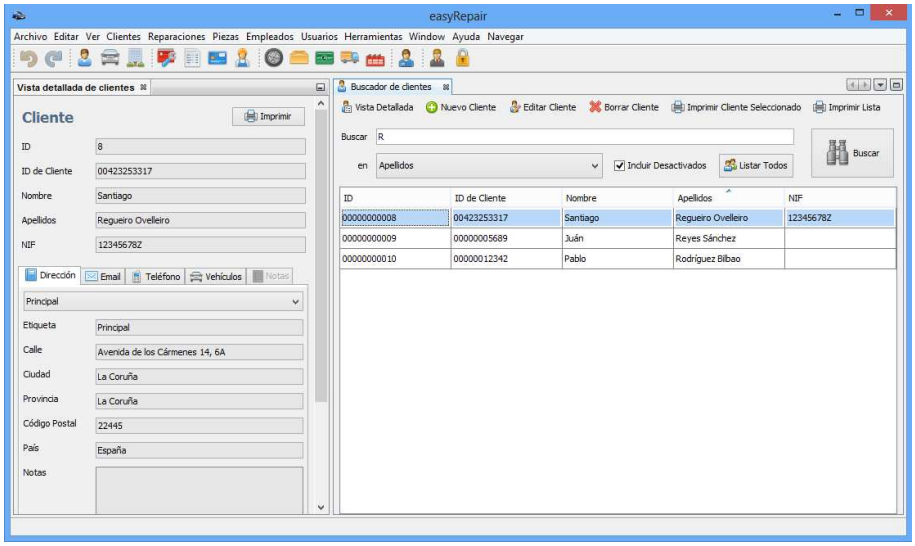

Figura A.4: Ventana de vista detallada

Añadir o editar registros Si pulsase en el botón añadir o editar se abriría la ventana del editor.

En ella podrá modificar los datos del cliente, añadir múltiples teléfonos, direcciones o emails y vehículos.

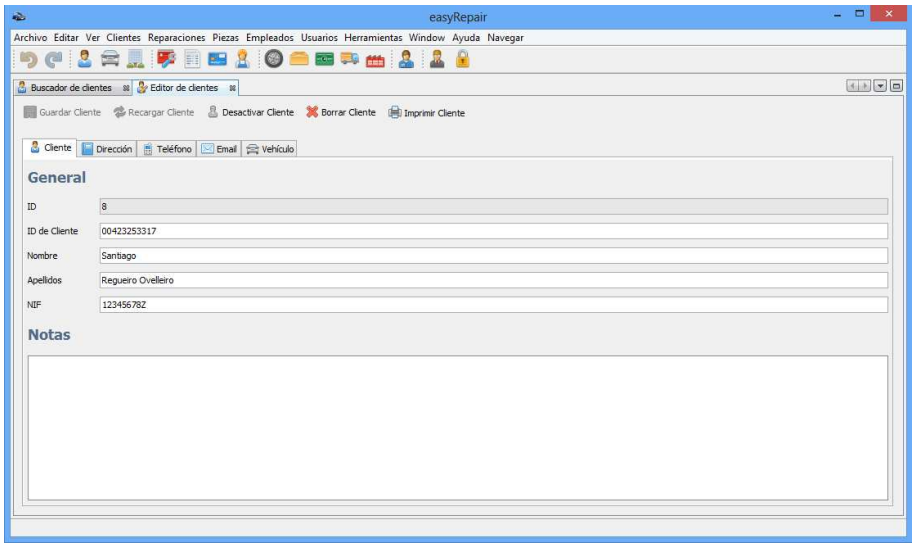

Figura A.5: Ventana del editor

Una vez terminadas las modificaciones, los datos se almacenarán pulsando en el botón Guardar.

Si en algún momento quisiera recuperar la información del registro almacenada en el sistema, podrá hacerlo pulsando en el botón de Recargar. También tiene disponible botones para Activar/Desactivar el registro o para Borrarlo o Imprimirlo.

Impresión Dentro de cada buscador, editor y vista detallada existe un botón que le permitirá realizar la impresión de un documento que contendrá todos los datos del registro seleccionado.

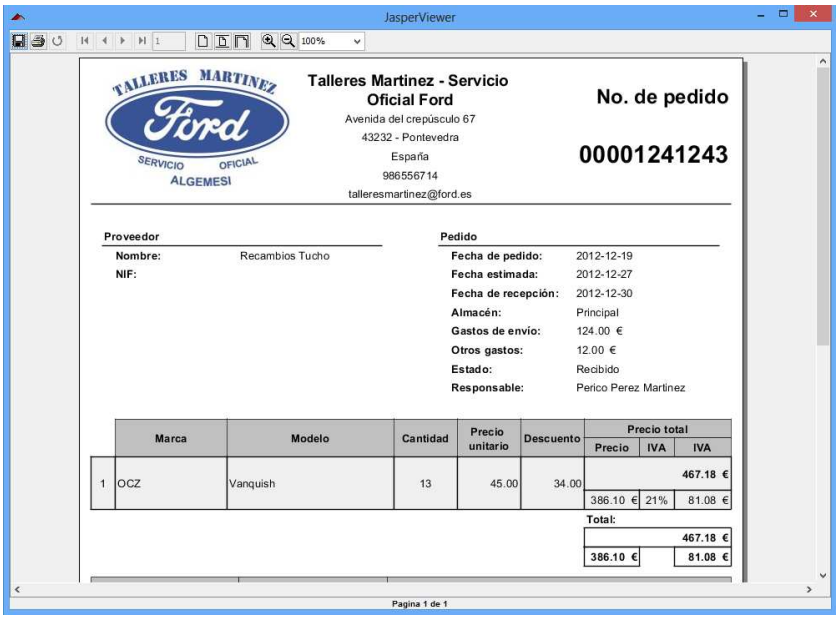

Figura A.6: Ventana de impresión

En la Figura A.6 se muestra la impresión de un pedido de piezas.

Si lo desea también podría imprimir la lista de resultados que se esté mostrando en ese momento en el buscador, como se puede comprobar en la Figura A.7.

Figura A.7: Ventana de impresión de listas

Permisos A cada usuario del sistema le corresponde un rol que define las acciones que puede realizar dentro de la aplicación. Estos permisos se editan dentro de la ventana de edición de roles.

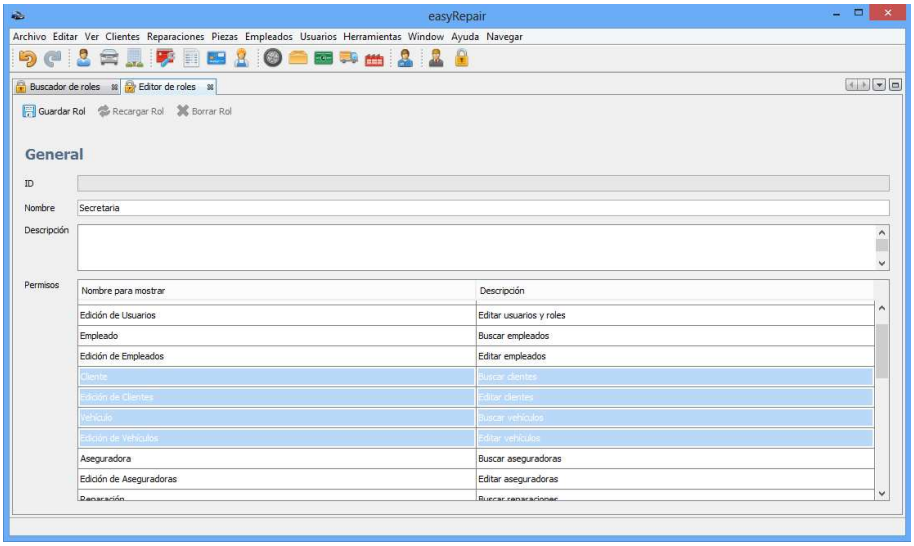

Figura A.8: Ventana del editor de roles

En la Figura A.8 se muestra el rol de la secretaria, que sólo tiene permiso para buscar y editar clientes y vehículos.

Si un usuario no tiene permiso para acceder a un buscador, el acceso disponible en el menú y en la barra de herramientas es eliminado.

Si no tiene permiso para editar, o bien nunca se activarán los botones de edición dispo-

nibles en el buscador, o si se trata de un tipo de dato perteneciente a un editor compuesto, como por ejemplo el editor de reparaciones que está compuesto de la orden de reparación, el presupuesto y la factura, los campos para los que no tiene permiso de edición permanecerán desactivados mientras que se le permitirá editar el resto.

Los permisos distinguen entre habilidad para buscar y habilidad para editar. Para poder editar las opciones de moneda o el nombre del taller, deberá tener permiso de Administrador.

Índice de figuras

Índice de cuadros

Bibliografía

[1] Juan Manuel Ares Casal. El proceso de resolucion de problemas. 2008.

[2] Raúl Oltra Badenes. *Sistemas Integrados de Gestión Empresarial. Evolución histórica y tendencias de futuro.* ISBN 8483638989. Editorial Universitat Politècnica de València, 2012.

[3] Heiko Böck. *The Definitive Guide to Netbeans Platform 7.* ISBN 1430241012. Apress, 2011.

[4] Jose Carneiro. Las ventas de coches marcan un mínimo histórico en septiembre por el nuevo iva, 2012. [Online; accessed 19-December-2012].

[5] Marc Loy ; Robert Eckstein ; Dave Wood ; James Elliott ; Brian Cole. *Java Swing.* ISBN 0596004087. O'Reilly, 2002.

[6] Conepa. La actividad de los talleres de coches se reduce entre un 4 y un 7 por ciento en 2011, 2011. [Online; accessed 19-December-2012].

[7] José Ramón Paramá Gabía. Introducción a los sistemas de bases de datos. 2009.

[8] Jane P. Laudon Kenneth C. Laudon. *Sistemas de información gerencial: organización y tecnología de la empresa conectada en red.* ISBN 970-26-0528-8. Pearson Educación, 2004.

[9] Gavin King. Session (hibernate javadocs), 2010. [Online; accessed 04-January-2013].

[10] Santiago Caballé Llobet. *Aplicaciones distribuidas en Java con tecnología RMI.* ISBN 9788496477957. Delta Publicaciones Universitarias, 2007.

[11] Ramón Muñoz. El plan pive apenas frena la caída de las ventas de coches en octubre, 2012. [Online; accessed 19-December-2012].

[12] Netbeans. Netbeans ide - developing java desktop application based on the netbeans rich-client platform (rcp) framework, 2012. [Online; accessed 04-January-2013].

[13] Josh Juneau ; Mark Beaty ; Carl P. Dea ; Freddy Guime ; John O'Conner. *Java 7 Recipes*. ISBN 1430240563. Apress, 2011.

[14] Roger S. Pressman. *Ingeniería del software: Un enfoque práctico*. ISBN 987-607-15-0314-5. McGraw-Hill, 2010.

[15] Eduardo Mosqueira Rey. Principios de análisis informático. 2010.

[16] Christian Bauer y Gavin King. *Java Persitence with Hibernate*. ISBN 1932394885. Manning, 2007.

[17] Stephen Stelting y Olav Maassen. *Patrones de diseño aplicados a Java*. ISBN 84-205-3839-6. Pearson Educación, 2003.