República de Cuba

Universidad de las Ciencias Informáticas

Facultad 2

Sistema de Gestión Policial (SIGEPOL)

"Módulo de Dependencia Policial"

Trabajo de Diploma

Presentado para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Raúl Martinez Aguirre

Tutor: Ing. Lisbet María González Bravo

Cotutor: Ing. Oigres Alvarez Pérez

"Año del 50 aniversario del triunfo de la Revolución"

Ciudad de la Habana, Cuba. Febrero de 2009.

DECLARACIÓN DE AUTORÍA

Declaro ser autor de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ___ días del mes de _____ del año _____.

_____

Raúl Martinez Aguirre

Autor

_____ _____

Ing. Lisbet María González Bravo

Tutor

Ing. Oigres Alvarez Pérez

Cotutor

Opinión del Tutor del Trabajo de Diploma

Dedicatoria

A Clarisa Aguirre Castillo mi mamá que en paz descanse, que tanto le hubiera gustado participar de este momento.

Agradecimientos

A Juani, Cristina y Vilma por dedicarme tiempo y darme tanto apoyo en este difícil momento de mi vida.

A Mi tío Robe por alentarme, siempre escucharme y ser un ejemplo de graduado universitario.

A mi papá por confiar en mi y tenerme en cuenta en cada momento.

A mi hermano Raudel que nunca dejo de pensar en él.

A mis tutores Lisbet y Oigres, por dedicar algo de su tiempo para que este trabajo se terminara con la mejor calidad posible.

Al equipo de desarrollo de SIGEPOL en general, que compartimos incluso "vacaciones" trabajando duro para lograr que este proyecto tuviera la mejor solución posible y gran aceptación del cliente.

A todos Muchas Gracias.

Resumen

La seguridad ciudadana es uno de los primordiales deberes desde el punto de vista social que todo país debe garantizar, de forma tal, que los ciudadanos queden satisfechos con la actuación de los cuerpos de seguridad ciudadana.

Actualmente la República Bolivariana de Venezuela no cuenta con un sistema de información capaz de integrar a todos los entes de seguridad ciudadana. Por lo tanto, el manejo individual y segmentado de la información almacenada en cada Órgano de Seguridad Ciudadana, trae como inconveniente que no exista un perfil e historial central de cada ciudadano, ni registros delictivos que el mismo pudiera generar. No existe un estricto control del armamento correspondiente a las dependencias policiales ni de los funcionarios que laboran en las mismas. Por esta razón el Ministerio del Poder Popular para Relaciones Interiores y Justicia de la República Bolivariana de Venezuela (MPPRIJ), promueve la realización del Sistema de Gestión Policial (SIGEPOL), que tiene como objetivo la captura de la información sobre la gestión policial a nivel nacional, que sirva de base a la toma de decisiones y ayude a combatir el delito en la República Bolivariana de Venezuela.

Como parte del Sistema de Gestión Policial se define el Módulo de Dependencia Policial, debido a que es necesario gestionar la información de los funcionarios de todas las dependencias policiales así como todo el armamento disponible en las dependencias de la República Bolivariana de Venezuela.

El documento que se presenta a continuación recoge los resultados del trabajo investigativo realizado. Para el desarrollo de la aplicación web que se propone se emplee: RUP como metodología de desarrollo de software, Java (J2EE) como plataforma de desarrollo, Visual Paradigm como herramientas CASE y Eclipse como IDE de desarrollo integrado para la codificación y gestión del código fuente en general.

Índice

Introducción

El Artículo 1^{ero} del Decreto Con Fuerza De Ley De Coordinación De Seguridad Ciudadana de la República Bolivariana de Venezuela establece que: "El presente Decreto Ley tiene por objeto regular la coordinación entre los Órganos de Seguridad Ciudadana, sus competencias concurrentes, cooperación recíproca y el establecimiento de parámetros en el ámbito de su ejercicio". Al lograr la coordinación de las actividades que realizan los órganos policiales, intercambiar información entre los mismos y definir parámetros que estandaricen el accionar diario de estos órganos, se posibilitaría una mayor organización del trabajo policial a nivel nacional y tomar acertadas decisiones para enfrentar las situaciones delictivas.

El MPPRIJ de la República Bolivariana de Venezuela, el 6 de noviembre del 2001, crea la Ley de Coordinación de Seguridad Ciudadana en la que establece las bases para el proyecto Sistema Nacional de Registro Delictivo Emergencia y Desastres (SINARDED), con la finalidad de almacenar y procesar la información de los distintos sucesos policiales que ocurren a nivel nacional diariamente; sin embargo este sistema no fue implementado en su totalidad, por lo que todos sus objetivos no fueron cumplidos y en estos momentos es empleado únicamente en la Dirección de Coordinación Policial. Actualmente las áreas de las dependencias policiales no disponen de un proceso unificado a nivel nacional para la actuación policial. Existen varias diferencias en el proceso de levantamiento de denuncias, reseña del ciudadano y planificación de operativos policiales, así como el modo en que se lleva a cabo la gestión de los funcionarios que pertenecen a una dependencia y el control del armamento con que cuenta la misma varía según la dependencia en que se encuentren cualesquiera de estos procesos anteriormente mencionados.

El MPPRIJ, haciendo alusión a su misión institucional de garantizar la seguridad ciudadana, promueve la formulación y puesta en marcha del Sistema de Gestión Policial (SIGEPOL), este sistema pretende unificar y compartir la información relacionada con denuncias, reseñas, operativos policiales que se registran en las dependencias de las policías estadales y municipales, además de gestionar los funcionarios y el armamento que pertenezcan a las mismas, así como intercambiar información con el CICPC[1] y servir de fuente de información al CTAISC[2]. Todo esto con la finalidad de que los datos en

[1] Cuerpo de Investigaciones Científicas Penales y Criminalística.

[2] Centro de Tratamiento y Análisis de la Información de Seguridad Ciudadana.

línea y compartidos, junto con los análisis que puedan hacerse, ayuden a combatir el delito en la República Bolivariana de Venezuela y alcanzar la tranquilidad ciudadana del pueblo en el marco del estricto cumplimiento de las regulaciones legales.

SIGEPOL estará constituido por un conjunto de módulos, los cuales son:

- Módulo de Administración.
- Módulo de Denuncia.
- Módulo de Reseña.
- Módulo de Operativos Policiales.
- Módulo de Dependencia Policial.

Actualmente el Módulo de Dependencia Policial está en la fase de inicio y desarrollado hasta el flujo de requerimiento.

Cuando a un funcionario se le va a dar ingreso en una dependencia policial, el estado del mismo en la última dependencia que ha laborado debe ser diferente de los estados definidos como de no posible inserción en una dependencia y en estos momentos se dificulta en gran medida obtener esta información de manera confiable. Esto puede traer consigo que un funcionario este activo en dos dependencias simultáneamente o no esté acto para ejercer como funcionario. Además no es posible conocer el historial de las dependencias en las que ha laborado un funcionario, así como los expedientes que se le han imputado por cometer indisciplinas. No se lleva constancia de la hoja de vida de los funcionarios de las dependencias, ya que en todas no se registra de la misma forma los méritos y las sanciones y en muchos casos ni se tiene constancia. No todas las dependencias llevan el registro del armamento que poseen, pues no existe un estricto control de las armas que les son asignadas a los funcionarios de forma permanente o a un corto plazo. Cuando un arma en específico está involucrada en un hecho policial no hay modo de conocer si esta forma parte del parque de arma de una dependencia o es propiedad de un funcionario.

El objetivo del Módulo de Dependencia Policial es solucionar las situaciones antes mencionadas, de ahí que se plantee el siguiente problema científico.

¿Cómo desarrollar una aplicación informática que garantice unificar y gestionar toda la información referente a los funcionarios y el armamento correspondiente a cada una de las dependencias policiales de la República Bolivariana de Venezuela?

El objeto de investigación lo constituye el proceso de gestión de funcionarios y armamento, siendo el campo de acción la informatización de los procesos de gestión de los funcionarios y el armamento en las dependencias policiales de la República Bolivariana de Venezuela.

Entre las principales funcionalidades del Módulo de Dependencia Policial correspondiente al Sistema de Gestión Policial estarán:

- Registrar funcionario en la dependencia policial.
- Dar Ingreso al funcionario.
  - Mostrar historial de dependencias y expedientes.
  - Registrar arma de fuego personal.
- Mostrar Hoja de Vida del Funcionario.
- Registrar Arma de Fuego.
- Asignar y retirar un arma de fuego temporalmente o a largo plazo.
- Reportes sobre los funcionarios y el armamento pertenecientes a la dependencia policial agrupados bajo diversos criterios.

Para darle solución al problema planteado se definió como objetivo general desarrollar una aplicación informática para gestionar los funcionarios y el armamento que pertenecen a las dependencias policiales como parte del Sistema de Gestión Policial.

De acuerdo con este planteamiento se trazaron los siguientes objetivos específicos:

- Realizar un análisis del estado actual de los procesos asociados a los funcionarios y el armamento de las dependencias policiales.
- Realizar el diseño del Módulo de Dependencia Policial a partir de las funcionalidades ya definidas.
- Desarrollar la implementación del Módulo de Dependencia Policial.

Para cumplir con los objetivos y resolver la situación problémica mencionada, se plantea un grupo de tareas de investigación:

1. Analizar los procesos de funcionarios y armamento que se llevan a cabo en las dependencias policiales.
2. Analizar las leyes que rigen el registro de funcionarios y el control del armamento en los Órganos Policiales y de Seguridad Ciudadana en Venezuela para conformar el Marco Legal.
3. Valorar la metodología de desarrollo de software, plataforma, lenguaje y el conjunto de herramientas de desarrollo definidas para la elaboración de la aplicación.

4. Identificar y seleccionar los patrones de arquitectura y diseño más apropiados para la elaboración del producto de software.
5. Realizar el diseño del software a partir del análisis del negocio.
6. Implementar los elementos de diseño en términos de componentes de implementación

Para cumplir con las tareas antes mencionadas se desarrollará un producto de software flexible, seguro, confiable y orientado a las necesidades del cliente acorde con las tecnologías de punta en el ámbito informático, las leyes venezolanas vigentes y los requerimientos necesarios.

El presente documento está compuesto por 3 capítulos:

En el Capítulo 1 **"Fundamentación del Tema"** se exponen los fundamentos generales que sirven de soporte teórico en la solución del problema. Se analizan las herramientas y lenguajes de programación que se han utilizado en los módulos ya desarrollados, para comprobar que son las idóneas para el desarrollo del Módulo de Dependencia Policial. Además se plantea la metodología a emplear en el desarrollo del mismo.

En el Capítulo 2 **"Diseño del Sistema"** se construye la solución propuesta, se modelan diagramas de clases que representan las funcionalidades del Módulo de Dependencia Policial aplicando los patrones de arquitectura y diseño seleccionados.

En el Capítulo 3 **"Implementación de la Solución"** se representa el diagrama de componentes que detalla la forma en que está estructurado el sistema, reflejando la transformación de los elementos del modelo del diseño en términos de componentes, ficheros que contienen código fuente, ejecutables, así como las dependencias entre ellos.

Capítulo 1 Fundamentación del Tema

1.1. Introducción

El objetivo fundamental de este capítulo es realizar la investigación sobre las actividades policiales; particularmente los procesos de control de los funcionarios y el armamento perteneciente a una dependencia policial y exponer los fundamentos generales que sirven de soporte teórico en la solución y concepción del problema. Se abordan aspectos de los distintos tipos de software, especificando en el tema de los sistemas de gestión, así como elementos que se deben tener en cuenta para su funcionamiento. Se justifica la selección de la metodología, tecnologías, técnicas y herramientas que se emplean en el desarrollo del módulo.

1.2. Seguridad Ciudadana

Se identifica como seguridad ciudadana a la acción integrada que desarrolla un país, con la colaboración de la ciudadanía, destinada a asegurar su convivencia pacífica y la erradicación de la violencia. Del mismo modo, contribuir a la prevención de la comisión de delitos y faltas. (1)

La seguridad ciudadana está orientada a prevenir males sociales como la violencia, las infracciones y el crimen; acciones dañinas que generan peligros para el apropiado desarrollo de las actividades económico-socio-cultural de toda sociedad humana actual.

La inseguridad ciudadana se ha convertido en uno de los grandes desafíos de las sociedades contemporáneas. El impacto del fenómeno sobre la calidad de la vida de los ciudadanos obliga a los gobiernos nacionales y locales y a los sectores organizados de la sociedad, a diseñar esquemas alternativos a los existentes que, siendo en su cometido de disminuir los niveles de inseguridad, no sacrifiquen el avance de la Democracia y el respeto por los Derechos Humanos y las Garantías Ciudadanas. (2)

Entre las acciones que suelen realizarse para tributar a la seguridad ciudadana se encuentran el dictado de leyes de seguridad ciudadana, que regulan los métodos y parámetros de actuación de los Órganos de Seguridad Ciudadana ante una situación; la creación de Órganos de Seguridad para velar el cumplimiento de las regulaciones establecidas; entre otras.

1.2.1. Seguridad Ciudadana en Venezuela

La seguridad de los ciudadanos debe ser atendida de acuerdo al mandato establecido en el artículo 55 de la Constitución de la República Bolivariana de Venezuela, el cual establece "Toda persona tiene derecho a la protección del Estado, a través de los órganos de seguridad ciudadana regulados por ley, frente a situaciones que constituyan amenazas, vulnerabilidad o riesgos para la integridad física de las personas, sus propiedades, el disfrute de sus derechos y el cumplimiento de sus deberes".

La cantidad de delitos que se cometen en esta nación de América del Sur ha originado gran incertidumbre, teniendo en cuenta la situación política vivida en estos últimos años, generada por una oposición que busca crear caos para tratar de llegar al poder, lo cual ha influido para que el gobierno tome cartas en este asunto con la decisión y firmeza que se requiere para solventar el problema de la inseguridad. (3)

Venezuela encara hoy el desafío de adecuar sus estructuras administrativas al nuevo proceso socio-político que se teje en el país, marcado por los principios de justicia social y respeto a los derechos y garantías constitucionales, con clara orientación hacia la inclusión de todos los ciudadanos en el goce y ejercicio pleno de los mismos.

En la población de la República Bolivariana de Venezuela están presentes problemas como el desempleo y la vivienda; pero uno de los más preocupantes es la inseguridad ciudadana. En ese sentido el MPPRIJ conjunto con los Órganos de Seguridad Ciudadana ha tomado medidas para el combate y prevención del delito, así como asegurar una convivencia pacífica en el país. En este marco es que surge la idea y puesta en marcha del Sistema de Gestión Policial con la finalidad de ayudar a coordinar las acciones de los órganos policiales.

1.2.2. Marco Legal o Seguridad Ciudadana según la Ley

Para establecer la normativa jurídica que sirve de base para la creación del Sistema de Gestión Policial del MPPRIJ de la República Bolivariana de Venezuela se tomará como referencia:

- Constitución de la República Bolivariana de Venezuela.

La Constitución de la República Bolivariana de Venezuela, publicada en Gaceta Oficial Extraordinaria No. 5 453 de la República Bolivariana de Venezuela en Caracas, viernes 24 de Marzo de 2000, expresa:

Artículo 9: Cuando resulte inminente el desbordamiento de la capacidad de respuesta del órgano actuante para controlar la situación, debido a su magnitud o complejidad de la misma, asumirá la

responsabilidad de la coordinación y el manejo de ésta, el Órgano de Seguridad Ciudadana que disponga de los medios y la capacidad de respuesta para ello.

Artículo 18. El Consejo de Seguridad Ciudadana tendrá por objeto el estudio, formulación y evaluación de las políticas nacionales en materia de seguridad ciudadana.

Artículo 36: La coordinación nacional de seguridad ciudadana y las coordinaciones regionales de seguridad ciudadana, como administradores de las bases de datos de sus respectivas localidades, procesarán la data e información suministrada por los integrantes del sistema y generarán los reportes de información relacionados con el comportamiento de la acción delictiva, emergencias y situaciones de desastres.

Artículo 322: Establece la organización de los Órganos de Seguridad Ciudadana, como medio para garantizar la protección de los ciudadanos y sus hogares en el disfrute de los derechos fundamentales, incorpora la creación de instituciones e instrumentos legales que permitan abordar integral y eficazmente la problemática de la inseguridad ciudadana.

Los Órganos de Seguridad Ciudadana son:

- La Policía Nacional.
- Las Policías de cada Estado.
- Las Policías de cada Municipio.
- El cuerpo de investigaciones científicas, penales y criminalísticas.
- El cuerpo de bomberos y administración de emergencias de carácter civil.
- La organización de protección civil y administración de desastre.

Corresponde a la Policía Nacional atender las situaciones con implicaciones internacionales, incluyendo delitos con proceso ejecutivo fraccionado entre varios países y con implicaciones que trascienden a más de un estado, entre otras.

Las policías estadales y municipales comparten las mismas funciones, según el ámbito territorial y nivel de complejidad, intensidad de intervención y especialidad de la situación a ser controlada. Deberán actuar de inmediato en la atención temprana del conflicto o situación de que se trate, independientemente de su complejidad, extensión o repercusión, al tiempo que deberán informar y requerir la participación de los cuerpos policiales más próximos en orden ascendente cuando la situación rebase sus posibilidades.

Estos Órganos de Seguridad Ciudadana tienen entre sus deberes comunes organizar y desarrollar sistemas informáticos, comunicacionales, administrativos y de cualquier otra naturaleza que permitan optimizar la coordinación entre los distintos órganos de seguridad ciudadana.

1.3. Gestión Policial en Venezuela

La actividad policial, parte de la administración del Estado, puede concebirse como “la disposición de una fuerza organizada para el mantenimiento del orden público mediante la vigilancia (aspecto preventivo) y la aprehensión de los infractores a los fines de imposición de una sanción (aspecto represivo), sanción a cargo de la propia instancia policial, de otras dependencias administrativas o de una instancia jurisdiccional”. (4)

Uno de los objetivos principales para garantizar la seguridad ciudadana en la República Bolivariana de Venezuela es fortalecer las políticas, estrategias, lineamientos y directrices que conlleven a la automatización del control de gestión de las acciones policiales, a fin de proveer a los diferentes entes y/o unidades del MPPRIJ de tecnología de información y comunicación que permitan lograr el cumplimiento de sus tareas sobre una plataforma tecnológica de avanzada. Además de promover y ejecutar las políticas del Estado en materia de investigación del fenómeno delictivo, mantener actualizado el registro de la población penal, llevar las estadísticas del ramo de prisiones, patronatos o presos libertados. (5)

Actualmente en la República Bolivariana de Venezuela cada dependencia policial controla y almacena por separado la información referente al armamento, los funcionarios y los expedientes de estos últimos, que incluyen sus méritos y sanciones.

Por lo tanto, el manejo individual y segmentado de la información asociada a cada dependencia policial, trae como consecuencia directa el inconveniente de que no existe un control centralizado de la información de los funcionarios y el armamento existente en las dependencias, dificultando de esta manera la obtención de una visión general de la documentación que se gestiona en los Órganos de Seguridad Ciudadana dificultando en gran medida la manera en que se llevan a cabo estos procesos de gestión.

1.3.1. Propuesta del Proceso Automatizado de Dependencia Policial

La propuesta para el proceso de registrar un funcionario transcurre como sigue:

El administrador del sistema de la dependencia policial busca el funcionario en la Base de Datos de la dependencia para comprobar que no se encuentre registrado en la misma. Si el funcionario no está registrado, se procede a registrar por primera vez en la dependencia introduciendo todos los datos básicos y comprobando su historial de dependencias y expedientes anteriores, en caso de que el último estado del funcionario seleccionado en la última dependencia en la que ha laborado sea

diferente de los estados definidos como no posible inserción en una dependencia no será posible ingresarlo en la dependencia, de lo contrario se le da ingreso en la dependencia. Si ya se encuentra registrado solo se pasa a darle ingreso y se registra los datos del arma de fuego personal en caso de poseerla.
Los datos de los funcionarios pueden ser modificados, así como su estado, jerarquía, cargo, tipo.
Si el funcionario comete alguna falta se le abre un expediente y se registra una sanción, la que le puede modificar el estado al funcionario. Al igual que se registran los méritos otorgados.
La propuesta para el proceso de gestionar un arma de fuego transcurre como sigue:
El parquero, que es la persona encargada de llevar el control de todas las armas con que cuenta la dependencia, registra el arma con todos los datos necesarios en la dependencia policial. Cuando el jefe del departamento al que pertenece el funcionario dentro de la dependencia le asigna un arma de fuego, el parquero retira dicha arma del parque, y se le cambia el estado a esta arma.
El parquero registra la entrega y devolución de las armas de fuego que dispone el parque de armas en la dependencia policial.
Los datos del arma de fuego pueden ser modificados, al igual que su estado.

Este módulo proporciona reportes sobre los funcionarios y armamento registrados en la dependencia policial agrupadas bajo diversos criterios. Como los méritos que le han otorgado a un funcionario, los expedientes que le han sido abiertos además de las sanciones que le han aplicado y en el caso de las armas según los criterios de asignadas o entregadas.

1.4. Conceptos asociados al dominio del problema

1.4.1. Software

El software es un conjunto de programas de cómputo y procedimientos necesarios para hacer posible la realización de una tarea específica, en contraposición a los componentes físicos del sistema, es decir, es el soporte lógico a todos los componentes intangibles de un ordenador o computadora. Está formado por una serie de instrucciones y datos, que permiten aprovechar todos los recursos que el ordenador tiene, de manera que pueda resolver gran cantidad de problemas. Una computadora en sí, es sólo un conglomerado de componentes electrónicos; el software le da vida al computador, haciendo que sus componentes funcionen de forma ordenada.

El software es un conjunto de instrucciones detalladas que controlan la operación de un sistema computacional. Adopta varias formas en distintos momentos de su ciclo de vida:

- Código fuente: escrito por programadores, contiene el conjunto de instrucciones destinadas a la computadora.
- Código objeto: resultado del uso de un compilador sobre el código fuente. Consiste en una traducción de este último.
- Código ejecutable: resultado de enlazar uno o varios fragmentos de código objeto. El código ejecutable es directamente inteligible por la computadora. (6)

Entre las principales funciones de un software se encuentran:

- Administrar los recursos del ordenador.
- Proporcionar las herramientas para optimizar estos recursos.
- Actuar como intermediario entre el usuario y la información almacenada.

Entre los tipos de software más utilizados se encuentran los softwares de gestión

1.4.2. Software de Gestión

La gestión, en el amplio mundo de la informática se entiende por el subsistema encargado de la gestión y control de los repositorios de información, de los grupos de usuarios, y de los procesos de soporte para otros subsistemas.

El software de gestión se encarga de definir y controlar los flujos de trabajo que son utilizados por los otros subsistemas, y de la definición de parámetros para el funcionamiento del sistema.

Las empresas, independientemente de su tamaño, enfrentan demandas respecto a rentabilidad, calidad, tecnología y desarrollo sostenible. Un sistema de gestión eficiente, diseñado a la medida de sus procesos comerciales, puede ayudar a enfrentar los desafíos del cambiante mercado global de hoy. (7)

La gestión de la información puede definirse como el conjunto de actividades realizadas con el fin de controlar, almacenar, y posteriormente, recuperar adecuadamente la información producida, recibida o retenida por cualquier organización en el desarrollo de sus actividades. Un sistema de gestión puede ayudar a centrar, organizar y sistematizar los procesos para la gestión y mejora. (8)

Entre los tipos de softwares de Gestion más usados están los del tipo ERP (Enterprise Resource Planning) que son los usados para la planificación de recursos empresariales, los del tipo BI (Business Intelligence) que se especializan en el análisis de negocio, además PLM (Product Lyfecycling Management) que se basan en la gestión de ciclo de vida de un producto.(9)

1.4.3. Aplicaciones

Aplicación es el término que se utiliza para designar un programa que se ejecuta en la computadora. Para evaluar si una aplicación está realmente bien construida no solo basta con que realice su tarea correctamente, sino también que sea fácil de utilizar por el usuario. Es decir, que el usuario se pueda relacionar con ella de forma rápida y comprensible. Para esto la aplicación dispone de un diseño el cual se llama Interfaz de usuario o Conexión de usuario, actualmente casi todo el esfuerzo de quienes diseñan aplicaciones está orientado a lograr una interfaz lo más amistosa e intuitiva posible. Sin embargo, es difícil establecer categorías genéricas significativas para las aplicaciones del software.

Una aplicación web es un sistema informático que los usuarios utilizan accediendo a un servidor web a través de Internet o de una intranet. Las aplicaciones web son populares debido a la practicidad del navegador web como cliente ligero. Otra razón de su popularidad es la facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software en miles de potenciales clientes. Es importante mencionar que una página Web puede contener elementos que permiten una comunicación activa entre el usuario y la información lo cual permite que el usuario acceda a ella de modo interactivo, gracias a que la página responderá a cada una de sus acciones. (10)

1.5. Lenguaje Unificado de Modelado (UML)

El Lenguaje Unificado de Modelado (UML) es un lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema que involucra una gran cantidad de software.

Está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos, permite la modelación de sistemas con tecnología orientada a objetos. Los diagramas son entes importantes de UML, cuya finalidad es presentar diversas perspectivas de un sistema, a las cuales se les conoce como modelo. Un modelo UML describe lo que supuestamente hará un sistema, pero no dice cómo implementarlo. El modelo gráfico de UML tiene un vocabulario en el que se identifican: elementos, relaciones y diagramas.

1.6. Metodologías de Desarrollo de Software

Las metodologías de desarrollo de software son un conjunto de procedimientos, técnicas, herramientas y un soporte documental que ayuda a los desarrolladores a realizar un nuevo software. Las

metodologías de desarrollo de software contribuyen a mejorar la calidad y a realizar un software en el tiempo esperado y con el coste estimado.

Van indicando paso a paso todas las actividades a realizar para lograr el producto informático deseado, proponen qué personas deben participar en el desarrollo de las actividades y qué papel deben de tener. Además detallan la información que se debe producir como resultado de una actividad y la información necesaria para comenzarla. (11)

Una metodología puede seguir uno o varios modelos de ciclo de vida, es decir, el ciclo de vida indica qué es lo que hay que obtener a lo largo del desarrollo del proyecto pero no cómo hacerlo. La metodología indica cómo hay que obtener los distintos productos parciales y finales. Algunos de los aspectos positivos de las metodologías son:

- Son interactivas e incrementales.
- Fácil de dividir el sistema en varios subsistemas independientes.
- Se fomenta la reutilización de componentes.

1.6.1. Metodología RUP

Para controlar, planificar, organizar y guiar el desarrollo del Módulo de Dependencia se decidió utilizar como metodología de desarrollo del software RUP (Proceso Unificado de Desarrollo de Software). RUP es el resultado de varios años de desarrollo y uso práctico en el que se han unificado técnicas de producción, es un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas de software, para diversas áreas de aplicación, distintos tipos de organizaciones, disímiles niveles de aptitud y diferentes tamaños de proyectos.

RUP divide el proceso de desarrollo en ciclos, teniendo un producto funcional al final de cada ciclo, cada ciclo se divide en fases que finalizan con un hito donde se debe tomar una decisión importante, las cuatro fases que incluye RUP son:

- **Inicio**, el objetivo en esta fase es determinar la visión del proyecto.
- **Elaboración,** en esta fase el objetivo es determinar la arquitectura óptima.
- **Construcción,** en esta fase el objetivo es obtener la capacidad operacional inicial.
- **Transición,** el objetivo es llegar a obtener el release del proyecto.

Vale mencionar que el ciclo de vida que se desarrolla por cada iteración, es llevado bajo dos disciplinas:

Disciplina de Desarrollo

- Modelamiento del Negocio: Describe los procesos de negocio, identificando quiénes participan y las actividades que requieren automatización.
- Requerimientos: Define qué es lo que el sistema debe hacer, para lo cual se identifican las funcionalidades requeridas y las restricciones que se imponen.
- Análisis y Diseño: Describe cómo el sistema será realizado a partir de la funcionalidad prevista y las restricciones impuestas (requerimientos), por lo que indica con precisión lo que se debe programar.
- Implementación: Define cómo se organizan las clases y objetos en componentes, cuáles nodos se utilizarán y la ubicación en ellos de los componentes y la estructura de las capas de la aplicación.
- Pruebas: Busca los defectos a lo largo del ciclo de vida.

Disciplina de Soporte

- Administración de Configuración y Cambios: Describe cómo controlar los elementos producidos por todos los integrantes del equipo de proyecto en cuanto a utilización/actualización concurrente de elementos, control de versiones, etc.
- Administración del proyecto: Involucra actividades con las que se busca producir un producto que satisfaga las necesidades de los clientes.
- Ambiente: Contiene actividades que describen los procesos y herramientas que soportarán el equipo de trabajo del proyecto; así como el procedimiento para implementar el proceso en una organización.
- Instalación: Produce release(versión terminada y estable de una aplicación informática) del producto y realiza actividades para entregar el software a los usuarios finales.

RUP está basado principalmente en tres características fundamentales:

Dirigido por casos de uso: Los casos de uso reflejan lo que los usuarios futuros necesitan y desean lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. A partir de aquí los casos de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso (como se llevan a cabo).

Centrado en la arquitectura: La arquitectura de un sistema es la organización o estructura de sus partes más relevantes. Muestra una visión del sistema completo, describe los elementos del modelo

que son más importantes para su construcción. El modelo de arquitectura se representa a través de vistas en las que se incluyen los diagramas de UML.

Iterativo e Incremental: Para facilitar el trabajo, el proyecto se realiza mediante iteraciones que terminan con un incremento. Cada iteración comprende diferentes disciplinas y de esta resulta una versión de un producto que irá creciendo en cada iteración. (12)

Como RUP es un proceso, en su modelación define como sus principales elementos:

- **Actividades** (“cómo”), Es una tarea que tiene un propósito claro, es realizada por un trabajador y manipula elementos.
- **Trabajadore**s (“quién”), Definen el comportamiento y responsabilidades (rol) de un individuo, grupo de individuos, que trabajan en conjunto como un equipo. Ellos realizan las actividades y son propietarios de elementos. Vienen a ser las personas o entes involucrados en cada proceso.
- **Artefactos** (”qué”), Un artefacto puede ser un documento, un modelo, o un elemento de modelo, o sea productos tangibles que son producidos, modificados y usados por las actividades.
- **Flujo de actividades** (“cuándo”), Secuencia de actividades realizadas por trabajadores y que produce un resultado de valor observable.

RUP es un ejemplo de las llamadas Metodologías Tradicionales, conocidas también como metodologías pesadas, las cuales ponen especial énfasis en la planificación y control del proyecto, así como en la especificación precisa de requisitos y el modelado. Existen además las denominadas Metodologías Ágiles*,* dirigidas sobre todo a equipos de desarrollo pequeños, con mayor fuerza en aspectos humanos asociados al trabajo en equipo e involucran al cliente en el proceso como parte activa del propio equipo de desarrollo y están orientadas sobre todo a la generación de código con ciclos cortos de desarrollo, pero no resulta factible utilizar este tipo de metodologías teniendo en cuenta que el cliente proceden de otro país y no puede formar parte del equipo de desarrollo, además el equipo de desarrollo es grande e inestable, pues en su mayoría son estudiantes y profesores que pueden pasar a cumplir otras funciones en cualquier momento, por lo que se necesitará una buena organización y abundante documentación para que el avance del proyecto no se vea afectado y se asegure la continuidad del mismo.

Asimismo la diferencia más notable entre estos dos grupos es que mientras los métodos pesados intentan obtener los resultados apoyándose principalmente en la documentación ordenada, los

métodos ligeros o ágiles tienen como base de sus resultados la comunicación e interacción directa con todos los usuarios involucrados en el proceso.

1.7. Plataforma de Desarrollo de Software

Una plataforma de desarrollo es el entorno común en el cual se desenvuelve la programación de un grupo definido de aplicaciones.

1.7.1. Plataforma Java

La plataforma Java es un entorno o plataforma de computación creada por la Sun Microsystems[3], capaz de ejecutar aplicaciones desarrolladas con el uso del lenguaje de programación Java y un conjunto de herramientas de desarrollo. En este caso, la plataforma no es un hardware específico o un sistema operativo, sino una máquina virtual encargada de la ejecución, y un conjunto de librerías estándares que ofrecen funcionalidades comunes. (13)

Para el desarrollo de la aplicación se seleccionó la Edición Empresarial: J2EE o Java EE (Java Platform, Enterprise Edition).

Una de las principales razones por la que se seleccionó esta plataforma es porque se cuenta con limitaciones legales por parte del cliente: el decreto Ley No. 3.390 de la Gaceta Oficial de la República Bolivariana de Venezuela establece que la Administración Pública Nacional empleará prioritariamente Software Libre Desarrollado con Estándares Abiertos, en sus Sistemas, Proyectos y Servicios Informáticos, permitiendo el uso de otro tipo de software solo en casos de no ser posible la adquisición de Software Libre Bajo Estándares Abiertos, en tal caso, los órganos y entes de la Administración Pública Nacional deberán solicitar ante el Ministerio de Ciencia y Tecnología, la autorización para adoptar otro tipo de soluciones bajo normas y criterios establecidos por ese Ministerio. (14)

J2EE es independiente del sistema operativo, es una plataforma madura y con una amplia gama de proveedores y documentación. Además el usuario no necesitará de requerimientos de hardware ni de software para acceder al sistema, solo se debe equipar con buen hardware el servidor de aplicaciones.

[3] Es una empresa informática de Silicon Valley, fabricante de semiconductores y software. Las siglas SUN se derivan de «Stanford University Network», proyecto que se había creado para interconectar en red las bibliotecas de la Universidad de Stanford.

Java es uno de los lenguajes de programación más elaborados y más utilizados para la creación de software de empresa. La evolución de Java, que ha pasado de ser un medio para desarrollar *applets*[4] para ser ejecutados en navegadores, a un modelo de programación capaz de manejar las aplicaciones de una empresa de hoy en día. (15)

Los componentes principales de la plataforma son la Máquina Virtual de Java (JVM) y el lenguaje de programación Java.

[4] Un *applet* es un componente de *aplicación* que se ejecuta en el contexto de otro programa Los Java applet son pequeños programas hechos en Java, que se transfieren con las páginas web y que el navegador ejecuta en el espacio de la página.

Máquina Virtual de Java (*JVM*)

El término JVM se refiere a la especificación abstracta de una máquina de software para ejecutar programas Java. Es el núcleo de ese lenguaje de programación, por lo que resulta imposible ejecutar cualquier programa Java sin la ejecución de alguna implantación de la JVM, o sea, el código no se ejecuta directamente sobre un procesador físico, sino sobre un procesador virtual Java. Es la encargada de traducir los bytecode (código resultante de la compilación del código fuente) en las instrucciones nativas, permitiendo la portabilidad de las aplicaciones.

Lenguaje de Programación Java

Este lenguaje nace bajo la filosofía de la Programación Orientada a Objetos (*OOP*), lo que permite lograr una mayor reutilización del código y una forma más eficiente de producir software, es una técnica centrada principalmente en los datos y la manera de llegar a ellos. Fue creado a partir de C++ y diseñado para integrarse sin mucha complejidad a entornos de red, ideal para ambientes bajo filosofía Cliente-Servidor. Es portable, sus programas pueden ser ejecutados en multiplataformas, esto posibilita hacer más liviana la carga de trabajo de un sistema, debido a que puede distribuirse en diferentes equipos de cómputo sin los problemas de incompatibilidad de estructuras de datos, o sea, se pueden utilizar recursos situados en diferentes ubicaciones geográficas, característica que favorece el desarrollo de SIGEPOL dado la modularidad y distribución del mismo como sistema.

Por todas las características anteriormente mencionadas, es que se propone el empleo de esta plataforma que integra todo lo necesario para que el desarrollo de la aplicación en cuestión sea favorable, sin costes elevados y ajustado a cronogramas.

1.8. Herramienta Case

Las herramientas CASE (Ingeniería de Software Asistida por Ordenador) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software, reduciendo el coste de las mismas en términos de tiempo y de dinero. Las herramientas CASE permiten organizar y manejar la información de un proyecto informático. Permite que los sistemas (especialmente los complejos, en el caso de SIGEPOL), se tornen más flexibles, más comprensibles y además mejorar la comunicación entre los participantes.

Estas herramientas pueden servir de apoyo en todos los aspectos del ciclo de vida de desarrollo del software, en tareas como el proceso de realizar un diseño del proyecto, cálculo de costes, compilación automática, documentación o detección de errores entre otras. (16)

1.8.1. Visual Paradigm para UML

Visual Paradigm para UML es una herramienta CASE profesional, fácil de utilizar y que soporta el ciclo de vida completo del desarrollo de software; usa al UML como lenguaje de modelado, característica que ayuda a la construcción de aplicaciones con calidad, de manera intuitiva, a menor coste; además de que facilita la comunicación entre los miembros del equipo de desarrollo, al garantizar el uso de un lenguaje estándar común. (17)

Se decidió utilizar esta herramienta CASE para el modelado del diseño de la aplicación, ya que entre sus principales características y facilidades se encuentran:

- Brinda la posibilidad de crear un conjunto bastante amplio de artefactos utilizados con mucha frecuencia durante la confección de algún diagrama en el desarrollo del software. Todos estos, cumpliendo con el Estándar UML 2.0.
- Disponibilidad en múltiples plataformas (multiplataforma): Microsoft Windows (98, 2000, XP, o Vista), Linux, Mac OS X, Solaris o Java. (18)
- Puede ser utilizado en varios idiomas, sus componentes se encuentran relacionados, por lo que se hace muy fácil la creación de cualquier tipo de diagrama, no se requieren grandes conocimientos para su manejo.
- Brinda un número considerable de estereotipos, lo que permite un mayor entendimiento de los diagramas.
- Generación de código e ingeniería inversa: brinda la posibilidad de generar código a partir de los diagramas, para plataformas como Java, así cómo obtener diagramas a partir del código, lo que facilita en gran medida el entendimiento entre analistas e implementadores, a la vez que es posible obtener el código, partiendo de un diagrama elaborado por el analista, también es posible visualizar diagramas con un alto nivel de detalle, a partir del código implementado por un programador.
- Integración con distintos Ambientes de Desarrollo Integrados (IDE): se integra fácilmente con varios IDEs de desarrollo, entre los que se encuentra Eclipse, que es el que se utilizará para el desarrollo de la aplicación.

- Generación de documentación: brinda la posibilidad de documentar todo el trabajo sin necesidad de utilizar herramientas externas.

1.9. Patrones arquitectónicos de diseño.

En ingeniería del software, un patrón es una solución ya probada y aplicable a un problema que se presenta una y otra vez en el desarrollo de distintas aplicaciones y en distintos contextos. Es importante destacar que un patrón no es en general una solución en forma de código directamente, sino una descripción de cómo resolver el problema y ante qué circunstancias es aplicable.
Entre los principales patrones que se emplearán en el desarrollo del módulo se encuentran:

1.9.1. De arquitectura

Modelo-Vista-Controlador (*MVC*)

El patrón Modelo-Vista-Controlador separa el modelamiento del dominio, la presentación, y las acciones basadas en las entradas hechas por el usuario en tres clases fundamentales:
Modelo: Administra y maneja el comportamiento y los datos del dominio de aplicación, da respuestas a peticiones de información sobre el estado de la aplicación (normalmente desde la Vista), y responde con instrucciones de cambio de estado (usualmente desde el controlador) a la vista.
Vista: Gestiona lo relacionado con mostrar la información al usuario.
Controlador: El controlador interpreta los eventos que son lanzados por la entrada estándar del usuario (normalmente mouse y teclado), informando de los mismos al modelo y/o la vista para que se ejecuten los cambios apropiadamente. (19)

Tres Capas (*Layers*)

El modelo tres capas permite desglosar la aplicación en partes lógicas que favorecen a una mejor organización en el desarrollo de la misma. Este patrón es fácilmente acoplable con el MVC. La arquitectura quedaría separada en lo relacionado con la gestión de interfaz (presentación), negocio y acceso a datos, además de las clases del dominio.

1.9.2. De diseño

Básicos: GRASP (General Responsibility Assignment Software Patterns)

Creador

El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos, tarea muy frecuente en los sistemas orientados a objetos. El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento.

Beneficios:

- Se brinda soporte a un bajo acoplamiento, lo cual supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización. Es probable que el acoplamiento no aumente, pues la clase creada tiende a ser visible a la clase creador, debido a las asociaciones actuales que llevaron a elegirla como el parámetro adecuado.

Controlador:

Este patrón ofrece una guía para tomar decisiones apropiadas que generalmente se aceptan. Un defecto frecuente al diseñar controladores consiste en asignarles demasiada responsabilidad. Normalmente un controlador debería delegar a otros objetos el trabajo que ha de realizarse mientras coordina la actividad.

Beneficios:

- Mayor potencial de los componentes reutilizables. Garantiza que la empresa o los procesos de dominio sean manejados por la capa de los objetos del dominio y no por la de la interfaz. Desde el punto de vista técnico, las responsabilidades del controlador podrían cumplirse en un objeto de interfaz, pero esto supone que el código del programa y la lógica relacionada con la realización de los procesos del dominio puro quedarían incrustados en los objetos interfaz o ventana.
- Reflexionar sobre el estado del caso de uso. A veces es necesario asegurarse de que las operaciones del sistema sigan una secuencia legal o poder razonar sobre el estado actual de la actividad y las operaciones en el caso de uso subyacente.

Experto:

Experto es un patrón que se usa más que cualquier otro al asignar responsabilidades; es un principio básico que suele utilizarse en el diseño orientado a objetos. Expresa simplemente la "intuición" de que los objetos hacen cosas relacionadas con la información que poseen. Nótese que el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos. Ello significa que hay muchos expertos "parciales" que colaboraron en la tarea.

El patrón Experto ofrece una analogía con el mundo real.

Beneficios:

- Se conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un bajo acoplamiento, lo que favorece al hecho de tener sistemas más robustos y de fácil mantenimiento.

- El comportamiento se distribuye entre las clases que cuentan con la información requerida, alentando con ello definiciones de clases "sencillas" y más cohesivas que son más fáciles de comprender y de mantener.

Bajo Acoplamiento:

El Bajo Acoplamiento es un principio que se debe recordar durante las decisiones de diseño, es la meta principal que es preciso tener presente siempre. Es un patrón evaluativo que el diseñador aplica al juzgar sus decisiones de diseño. El Bajo Acoplamiento estimula asignar una responsabilidad de modo que su colocación no incremente el acoplamiento tanto que produzca los resultados negativos propios de un alto acoplamiento.

El Bajo Acoplamiento soporta el diseño de clases más independientes, que reducen el impacto de los cambios, y también más reutilizables, que acrecientan la oportunidad de una mayor productividad. No puede considerarse en forma independiente de otros patrones como Experto o Alta Cohesión, sino que más bien ha de incluirse como uno de los principios del diseño que influyen en la decisión de asignar responsabilidades. (20)

Beneficios:

- No se afectan por cambios de otros componentes
- Fáciles de entender por separado
- Fáciles de reutilizar

Alta Cohesión:

Como el patrón Bajo Acoplamiento, también Alta Cohesión es un principio que se debe tener presente en todas las decisiones de diseño, es la meta principal que ha de buscarse en todo momento. Grady Booch señala que se da una alta cohesión funcional cuando los elementos de un componente "colaboran para producir algún comportamiento bien definido".

El patrón Alta Cohesión presenta semejanzas con el mundo real, ya que si alguien asume demasiadas responsabilidades -sobre todo las que debería delegar-, no será eficiente.

Beneficios:

- Mejoran la claridad y la facilidad con que se entiende el diseño.
- Se simplifican el mantenimiento y las mejoras en funcionalidad.
- A menudo se genera un bajo acoplamiento.
- La ventaja de una gran funcionalidad soporta una mayor capacidad de reutilización, porque una clase muy cohesiva puede destinarse a un propósito muy específico.

Polimorfismo:

Como el patrón Experto, el uso del patrón Polimorfismo está acorde al espíritu del patrón "Lo hago yo mismo". Es un principio fundamental en que se fundan las estrategias globales, o planes de ataque, al diseñar cómo organizar un sistema que se encargue del trabajo. Un diseño basado en la asignación de responsabilidades mediante el polimorfismo puede ser extendido fácilmente para que realicen nuevas variantes.

Beneficios:

- Es fácil agregar las futuras extensiones que requieren las variaciones imprevistas.

Avanzados: GOF (Gang of Four)

Fábrica Abstracta (Abstract Factory):

Es un patrón creacional en la clasificación de los patrones GOF. Proporciona una interfaz para crear familias de objetos sin especificar su clase de forma concreta.

Consecuencias:

- Se potencia el encapsulamiento, puesto que se aísla a los clientes de las implementaciones.
- Se incrementa la flexibilidad del diseño, resultando fácil cambiar de familia de productos.
- Se refuerza la consistencia (alta cohesión y bajo acoplamiento), puesto que se restringe el uso a productos de una sola familia cada vez.

Fachada (Facade)

Simplifica el acceso a un conjunto de clases o interfaces. Proporciona una interfaz unificada de un subsistema sin ocultar sus interfaces. Permite acceder a elementos del sistema y realizar operaciones más complejas con ellos de forma transparente.

Consecuencias:

- Oculta a los clientes parte de la complejidad de los subsistemas.
- Disminuye el acoplamiento entre subsistemas y cliente.
- Disminuye el acoplamiento (las interfaces de por sí reducen el acoplamiento).
- Ocultación de componentes del subsistema.
- Que el cliente pueda utilizar toda la funcionalidad del sistema.

Instancia única o Solitario (Singleton)

El patrón Singleton garantiza que una clase sólo tenga una instancia y proporciona un punto de acceso global a esta instancia.

Consecuencias:

- Acceso controlado a la única instancia. Puede tener un control estricto sobre cómo y cuándo acceden los clientes a la instancia.
- Espacio de nombres reducido. El patrón Singleton es una mejora sobre las variables globales.
- Permite el refinamiento de operaciones y la representación. Se puede crear una subclase de Singleton.
- Permite un número variable de instancias. El patrón hace que sea fácil cambiar de opinión y permitir más de una instancia de la clase Singleton.
- Más flexible que las operaciones de clase (static en C#, Java).

1.10. Frameworks

En el desarrollo de software, un marco de trabajo (framework en inglés) es una estructura de soporte definida sobre la cual un proyecto puede ser organizado y desarrollado; puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre aplicaciones para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Para desarrollar una aplicación Web es necesario tener en cuenta que esta filosofía está basada en el estilo arquitectónico Cliente-Servidor, por lo que es necesario definir ambos comportamientos. Por ello se propone, para el lado del cliente, la utilización del siguiente marco de trabajo para el trabajo con javascript.

Dojo

Dojo es un marco de trabajo que contiene APIs[5] y controles (widgets) para facilitar el desarrollo de aplicaciones Web que utilicen el modelo AJAX. Con la utilización de Dojo, se gana en facilidad y agilización del proceso de desarrollo de este tipo de aplicaciones; permite reutilizar código y promueve buenas prácticas de desarrollo; resuelve problemas comunes de compatibilidad entre navegadores. Dojo contiene un sistema de empaquetado inteligente, efectos de Interfaz de Usuario (UI), APIs de arrastrar y soltar (*drag and drop APIs*), APIs de controles (*widget APIs*), abstracción de eventos, almacenamiento de APIs en el cliente, e interacción de APIs con AJAX, y la habilidad de degradar cuando AJAX/JavaScript no es completamente soportado en el cliente. Proporciona una gama más

[5] Interfaz de Programación de Aplicaciones

amplia de opciones en una sola biblioteca, que otros frameworks del mismo tipo. Por estas potencialidades ya mencionadas Dojo se hace una opción viable. (21)

Para darle soporte a toda la gestión de información y manejo de páginas dinámicas que llegan a los clientes, se plantea para el lado del servidor la utilización del siguiente marco de trabajo, dadas las potencialidades del mismo que se exponen a continuación.

Spring Framework

Marco de Trabajo de código abierto que facilita la creación y desarrollo de aplicaciones para la plataforma Java. Ofrece muchas libertades a los desarrolladores, además de que cuenta con una abundante documentación. Se basa y promueve el empleo de las mejores prácticas de programación aceptadas en la actualidad.

Está compuesto por siete módulos principales que colaboran y brindan una amplia gama de funcionalidades fáciles de emplear, que garantizan una arquitectura robusta para cualquier proyecto que tenga su basamento sobre Spring. Entre los principales módulos se encuentran:

- Módulo núcleo (Core Container and Supperting Utilities): Contiene la Fábrica de Clases (BeanFactory) que es el corazón de toda aplicación sobre Spring. *BeanFactory* aplica la Inversión de Controles (Invertion of Control o IoC), específicamente la Inyección de Dependencias (Dependency Injection), para separar la configuración de la aplicación y sus especificaciones de dependencias entre objetos de la lógica de negocio de la aplicación.
- Módulo AOP (Aspect-Orient Programming): Brinda soporte a la Programación Orientada a Aspectos, con la que es posible manejar y solucionar problemáticas como las auditorías y las transacciones de datos con la Base de Dato o entre capas de abstracción dentro de la aplicación, manteniendo la legibilidad y limpieza en el código que define la lógica de negocio de la aplicación en cuestión.
- Módulos ORM y JDBC abstraction and DAO: Brindan soporte a la gama de funcionalidades y problemáticas comunes en el trabajo con Bases de Datos y abstraen al programador de las acciones básicas de intercambio con la Base de Dato permitiendo que sea posible mantener el código de acceso a datos limpio y simple. Además, ORM implementa mecanismos de enganches para lograr compatibilidad e integración con otros frameworks que soportan el mapeo de objetos relacionales como Hibernate e iBatis.
- Módulo MVC (Model – View – Controller): Brinda soporte al desarrollo de aplicaciones web basando el intercambio de los clientes (dígase navegadores web) con el servidor, en la filosofía que sigue el patrón arquitectónico del mismo nombre. Esta facilidad también emplea la *IoC* para

lograr separar limpiamente la lógica de controlador de los objetos de negocio, también permite unir declarativamente, parámetros de una petición a objetos de negocio. Además incorpora un mecanismo de validación de formularios muy útil en la comprobación de los datos provenientes del cliente. (22)

Por todas las características ya mencionadas, es posible afirmar que Spring es un framework muy simple, conveniente y flexible, pero al mismo tiempo muy poderoso; facilita la manipulación de objetos, elimina la necesidad de usar distintos y variados tipos de ficheros de configuración, mejora la práctica de programación y suaviza la curva de aprendizaje favorablemente para el desarrollador; es por ello que se propone como una alternativa viable para el desarrollo de la aplicación en cuestión.

iBatis:

Es un marco de trabajo especializado en dar soporte a las operaciones y problemáticas más comunes referentes a las interacciones entre el acceso a datos de una aplicación y la base de datos correspondiente. Resulta un componente de software encargado de traducir entre objetos y registros, basado en capas, situado entre la lógica de negocio y la capa de origen de datos. Compuesto de dos paquetes complementarios pero independientes: IBatis Data Access Objects (DAO), que implementa la capa de abstracción (ibatis-dao.jar) e iBatis SQL MAPS, que implementa la capa de persistencia (ibatis-sqlmap.jar). IBatis Data Access Objects (DAO) oculta los detalles de la capa de persistencia y proporciona un API común para todas las aplicaciones iBatis Data Mapper proporciona un modo simple y flexible de mover los datos entre los objetos Java y la base de datos relacional, se tiene toda la potencia de SQL sin una línea de código *JDBC (Java DataBase Connectivity)*. (23)

Para su uso no requiere de grandes esfuerzos de aprendizaje, consta de buena documentación y es un proyecto *Open Source*. Ofrece un sistema de mapeo directo en ficheros XML. Proporciona mecanismos para el acceso a procedimientos almacenados, con la debida consideración por parte del programador de no implementar lógica de negocio en los mismos.

Dada las condiciones en la que se desarrolla la aplicación, así como las características y facilidades antes mencionadas, se propones este framework de persistencia como una opción factible.

1.11. Herramientas de desarrollo

Entre las principales herramientas de desarrollo que se utilizarán en el proyecto se encuentran:

Eclipse.

Eclipse es un ambiente integrado de desarrollo de código abierto basada en Java. Es un desarrollo de IBM cuyo código fuente fue puesto a disposición de los usuarios. En sí mismo Eclipse es un marco y

un conjunto de servicios para construir un entorno de desarrollo a partir de componentes conectados (plug-in). Eclipse contiene una serie de perspectivas, cada perspectiva proporciona una serie de funcionalidades para el desarrollo de un tipo específico de tarea. La perspectiva Java combina un conjunto de vistas que permiten ver información útil cuando se está escribiendo código fuente, mientras que la perspectiva de depuración contiene vistas que muestran información útil para la depuración de los programas Java. (24)

La característica clave de Eclipse es la extensibilidad. Eclipse es una gran estructura formada por un núcleo y muchos plug-ins que van conformando la funcionalidad final. La forma en que los plug-ins interactúan es mediante interfaces o puntos de extensión; así, las nuevas aportaciones se integran sin dificultad ni conflictos. Está compuesto de tres subproyectos: la Plataforma Eclipse, la Java Development Tool y el Plug-in Development Environment. El éxito de la Plataforma Eclipse depende de cómo sea capaz de admitir una amplia gama de herramientas de desarrollo para reproducir lo mejor posible las herramientas existentes en la actualidad. (25)

En el desarrollo de la aplicación se incluyen un conjunto de plug-ins que amplían sus funcionalidades como plataforma de desarrollo de aplicaciones sobre la web. Los plug-ins más importantes son los siguientes.

- Web Tool Platform (WTP): Para soportar todas las funciones necesarias para desarrollar aplicaciones web sobre J2EE.
- Spring IDE: Para facilitar el uso sobre los beans y xml definidos por Spring Framework.
- Subeclipse: Para permitir de una forma mucho más ágil y cómoda el desarrollo colaborativo de software en un equipo de desarrolladores.

1.12. Conclusiones

En este capítulo se realizó un análisis de los principales conceptos que deben conocerse para comprender la solución que se propone. Se describió como transcurren los procesos de una dependencia policial. Se estudiaron las metodologías y patrones de arquitectura y diseño a utilizar a lo largo del desarrollo del sistema propuesto, y se fundamentó la elección de las herramientas y tecnologías a utilizar en la elaboración de la aplicación.

Capítulo 2 Diseño del Sistema

2.1. Introducción

En este capítulo se describe la propuesta de diseño del Módulo de Dependencia Policial. Se muestra la arquitectura del Sistema de Gestión Policial, se presenta el diagrama de paquetes que se propone para estructurar los elementos del diseño analizados desde dos enfoques, horizontal desde el punto de vista modular, y vertical, entrando en la estructura de capas en las que estará dividido cada módulo. Se describen además los diagramas de clases de cada uno de estos paquetes divididos por capas, y la descripción de las principales clases del diseño.

2.2. Descripción de la Arquitectura

Establecer las bases sólidas que soporten la construcción, escalabilidad, correcto funcionamiento y mantenimiento de una aplicación informática, requiere un estudio detallado de los patrones seguidos a lo largo del desarrollo del software, las buenas prácticas de programación, el entorno de desarrollo de la misma, las prestaciones que ha de satisfacer y los requerimientos necesarios para su puesta en marcha. La descripción de la arquitectura define los principales aspectos tomados en cuenta en el diseño de un sistema. La arquitectura de SIGEPOL está organizada desde dos enfoques, uno vertical y otro horizontal.

Enfoque horizontal

El sistema está dividido en varios módulos los cuales responden a un conjunto de funcionalidades y casos de uso específicos del cliente. Todos estos módulos interactúan y comparten datos de interés en dependencia de la funcionalidad de cada uno y siguiendo un estricto régimen de seguridad de la información.

Enfoque Vertical

El desarrollo de cada módulo responde a un modelo multicapas. Este modelo está integrado fundamentalmente por Objetos del Dominio (Domain).y las siguientes capas:

- Capa presentación.
- Capa de negocio.
- Capa acceso a datos.
- Capa de persistencia.

2.2.1. Modelo Basado en Capas

Una capa es una agrupación lógica de un conjunto de clases acopladas entre sí. Desde el punto de vista de la Ingeniería del Software, la división de un sistema en capas facilita el diseño modular (cada capa encapsula un aspecto concreto del sistema) y permite la construcción de sistemas débilmente acoplados (si se minimiza las dependencias entre capas, resultará más fácil sustituir la implementación de una capa sin afectar al resto del sistema). Además, el uso de capas también posibilita la reutilización. El Módulo de Dependencia Policial posee un conjunto de clases agrupadas jerárquicamente en capas, de acuerdo a la funcionalidad que brinda cada una de ellas, ya sea presentación, lógica de negocio, acceso a datos y persistencia. Cada capa para completar su funcionamiento delega tareas a la capa que se encuentra por debajo en la jerarquía, brindándole los datos necesarios para las mismas o solicitando de ella la información requerida.

A continuación se muestra como está estructurada la arquitectura del módulo y posteriormente se explican cada una de sus partes.

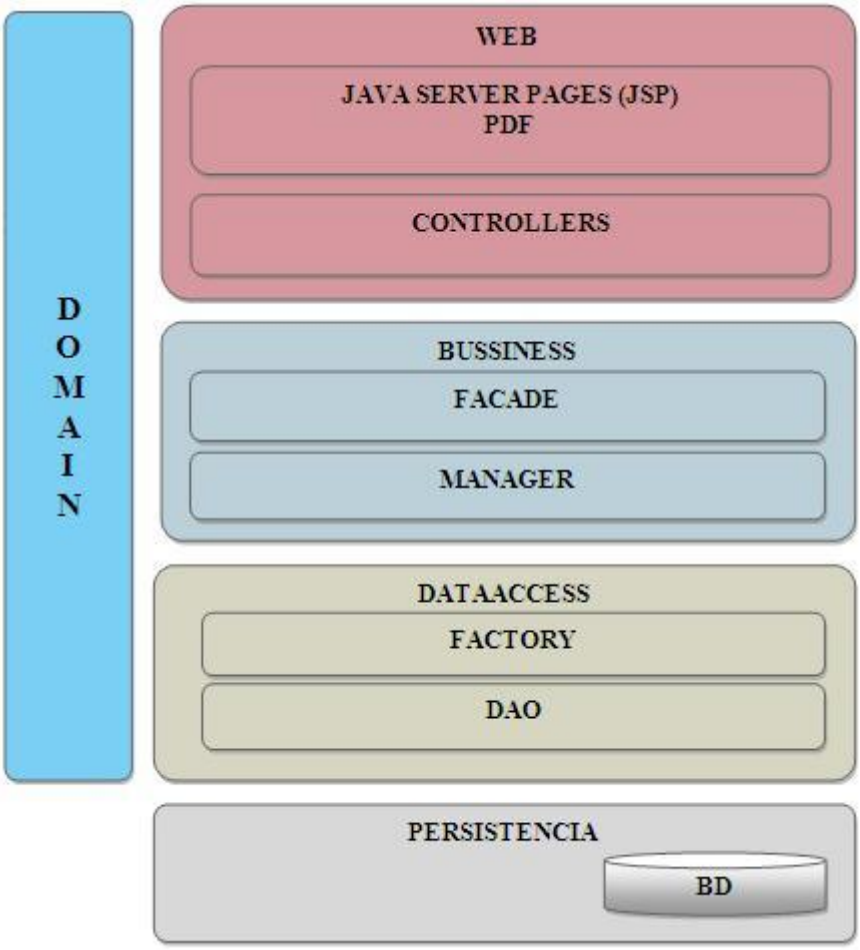

Fig. 2.1. Modelo de la Arquitectura.

Para que exista una mejor comprensión y teniendo en cuenta, que las clases de cada capa para realizar sus responsabilidades se auxilian de las clases de la capa inferior, se comenzará la explicación por la capa más baja en la jerarquía, pero para ello se hace necesario conocer la razón por la cual las clases del dominio (DOMAIN), se representa verticalmente en la figura 2.1 que representa el modelo de la arquitectura.

Dominio(DOMAIN)

El dominio delimita el área del problema que se estudiará, por ejemplo, los procesos que ocurren en una dependencia policial en la República Bolivariana de Venezuela.

Está formado por las clases que representan de manera abstracta los conceptos encontrados en el dominio. Las mismas contienen la información que el sistema debe manipular a través de las capas

que posee, dándole en cada una de ellas el uso necesario, ya sea, visualizar, verificar o persistir, garantizando sus propiedades y restricciones.
Dado que las demás capas se van a encargar de hacer cumplir las restricciones del negocio, el dominio constituye el soporte para la transferencia de datos desde el acceso a datos hasta la capa de presentación y viceversa.

Capa de Persistencia

Esta capa corresponde a los almacenes de datos, contiene todas las estructuras necesarias para poder almacenar la información del dominio que debe persistir y está contenida físicamente en el servidor de bases de datos. Es única y común a todos los módulos de SIGEPOL y debe garantizar que el modelo de datos esté al menos, en tercera forma normal.

Capa de acceso a datos (DATA ACCESS)

El término de Objeto de Acceso a Datos o en inglés, Data Access Object (DAO), es ampliamente usado en el desarrollo de software. Los DAOs son responsables de hacer persistir los objetos del dominio, constituyen una abstracción entre la capa de persistencia y la capa de negocio, y encapsulan la forma en que ambas se relacionan. Oculta completamente los detalles de implementación de la fuente de datos a sus clientes. Las implementaciones de los DAOs estarán disponibles para los objetos (típicamente para los objetos de negocio) haciendo uso de la inyección de dependencias con los objetos de negocio y las instancias de los DAOs, configurada en el contenedor de inversión de control de Spring Framework.
Manteniendo la filosofía de que es mejor programar orientado a interfaces que a clases, en la capa de Acceso a Datos se definen varias interfaces, que son expuestas a la capa de negocio a través de una fábrica, que construye para cada una de sus implementaciones, una instancia única. La fábrica se implementó utilizando el patrón *AbstractFactory.*
Los métodos fundamentales declarados en estas interfaces, se encaminan a guardar, eliminar, buscar y actualizar objetos del dominio en la base de datos. Las implementaciones de estas interfaces heredan de la clase SqlMapClientDaoSupport, implementada en Spring, haciendo más sencillas las mismas.

Capa de Negocio

Las clases del negocio implementan la lógica de negocio de las aplicaciones. En esta capa se separa la lógica de la aplicación de los datos mediante la utilización de entidades del dominio, que carecen de lógica, y son utilizados como objetos que transitan por las diversas capas arquitectónicas.
El Manager agrupa la lógica de negocio de cada una de las entidades que son representativas en el modelo. Dentro del Manager se encuentran las interfaces definidas para dar soporte a las funcionalidades que contiene la aplicación. Sus implementaciones se encargan de manejar la lógica del negocio, controlar el flujo de información con la capa de acceso a datos y el chequeo adecuado de las transacciones del negocio.
La solución de diseño para lograr un bajo acoplamiento está en el empleo del patrón estructural Facade (Fachada). En Facade se encuentran las interfaces que brindan a la capa de presentación los métodos necesarios para resolver las funcionalidades especificadas en los casos de uso. Sus implementaciones delegan su labor a los Managers especializados en gestionar la información que poseen o necesitan.

Capa de Presentación

La capa de presentación descansa sobre una capa de servicios de negocio. Esto significa que esta capa será fina y no contendrá lógica de negocio, sino simplemente lo concerniente a aspectos de presentación, por ejemplo, el código para manipular las interacciones web.
Spring Web MVC Framework, es una implementación del patrón arquitectónico llamado MVC (Modelo Vista Controlador), el cual se utilizará en esta capa de presentación.
En el módulo MVC que implementa Spring, define al *DispatcherServlet* como el controlador frontal de la aplicación, este es el encargado de atender todas las peticiones que se le realicen al servidor. El componente responsable de manipular la petición en el MVC de Spring es un *Controller.*

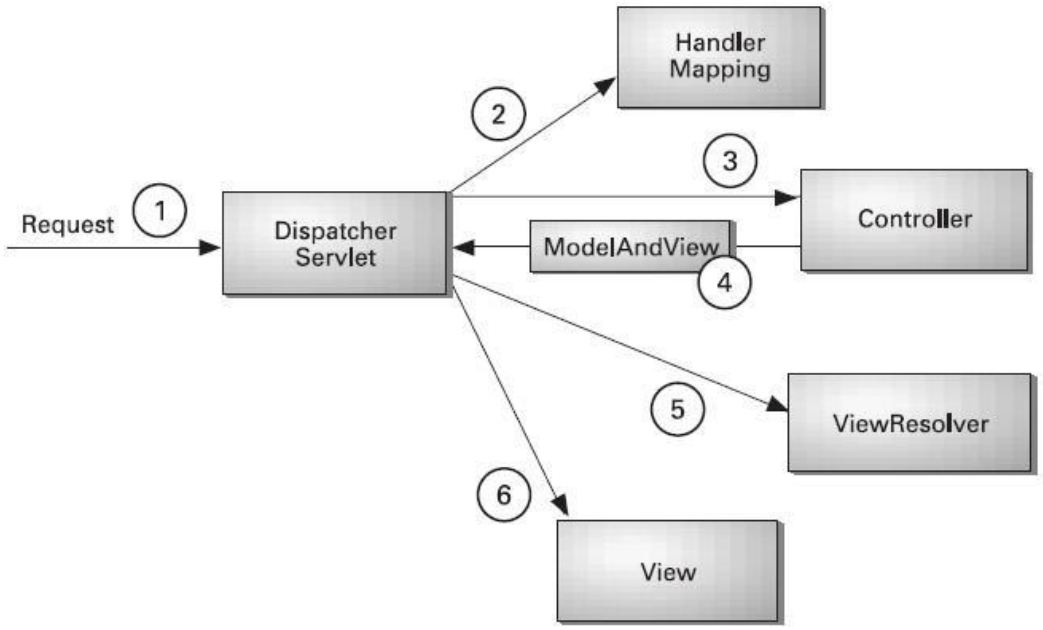

Fig. 2.2. Spring MVC.

Para que el *DispatcherServlet* logre identificar qué *Controller* manipulará la petición actual, este se auxilia del o los *HandlerMappings*, Un *HandlerMapping,* usualmente es el encargado de mapear los patrones de URL con el objeto Controller asociado. Una vez que el *DispatcherServlet* conoce el objeto *Controller*, le delega la responsabilidad de atender la petición para que realice la lógica de negocio para la que fue diseñado y devuelva un objeto *ModelAndView* al *DispatcherServlet* con el objeto *View* o el nombre lógico del mismo. Si el objeto *ModelAndView* contiene el nombre lógico del *View*, entonces el *DispatcherServlet* consulta al *ViewResolver* para localizar al objeto *View* que visualizará la respuesta (response) al cliente.

En su módulo MVC, Spring implementa una serie de controladores personificados que facilitan la construcción de nuevos controladores mediante la herencia.

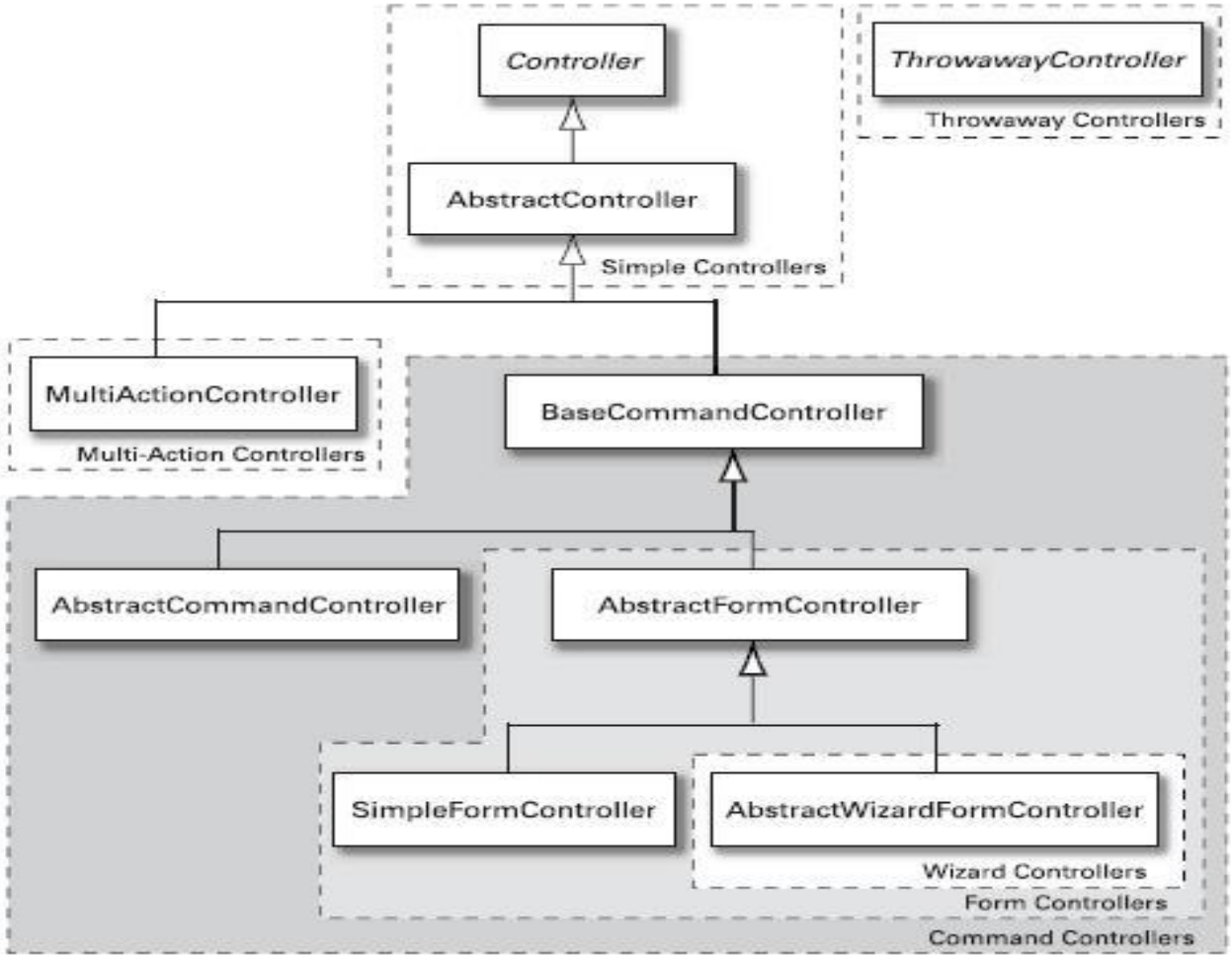

Fig. 2.3. Jerarquía de Controladores de Spring MVC.

Dentro de *Controllers*, se encuentran las clases controladoras de la capa, que son las responsables de procesar las peticiones HTTP, y a partir de las funcionalidades expuestas por las fachadas de la capa de negocio, construir el modelo correspondiente a la nueva vista que se ha de mostrar finalmente al usuario. Los mismos heredan de alguno de los controladores especificados en la jerarquía de Spring MVC y en algunos casos se van a auxiliar de clases validadoras que chequearán que los objetos del dominio posean la información correcta y necesaria para poder realizar el proceso del negocio en que están inmersos.

Modelo: Estos objetos contienen los datos resultantes de la ejecución de la lógica de negocio, los cuales deberían ser mostrados en la respuesta.

Vistas: Estos objetos son responsables de mostrar el modelo en la respuesta de la petición. La forma de mostrar el modelo podrá ser de diferentes tipos de vistas, por ejemplo, archivos JSP, HTML, PDF,

entre otras. Las vistas no son responsables de modificar los datos o incluso de obtener los datos; estas simplemente sirven para mostrar los datos del modelo que han sido suministrados por un controlador.

2.3. Diagrama General de Clases del Diseño

El diagrama general de clases del diseño representa la interacción entre las clases de las diferentes capas por las que está integrado el Módulo de Dependencia Policial. Se ha utilizado como elemento característico el color definido en la figura 2.1 para identificar las clases pertenecientes a cada capa. Las clases representadas con el color verde indican las clases que proporcionan los frameworks utilizados y se representan para mayor comprensión del diagrama.

SimpleFormController: Clase implementada en spring que se especializa en atender las peticiones de un formulario web. Brinda la posibilidad de configurar la vista del formulario, que se va a utilizar y la vista que debe mostrarse luego de ser enviado (submit). Tiene en cuenta la validación de los datos enviados, para lo cual se le ha de especificar un objeto que implemente la interface Validator del framework. Posee métodos que al ser reimplementados en las clases hijas, facilita enviar al formulario los datos específicos que necesite.

Validator: Es una interface utilizada por spring para la validación. Puede ser implementada por clases que se especialicen en validar los datos de un tipo de objeto específico.

AbstractController: Clase implementada en el spring que sirve de base a otros controladores.

SqlMapClientDaoSupport: Clase implementada por spring, que sirve de base para el trabajo con el framework iBatis, que se utiliza en la capa de acceso a datos.

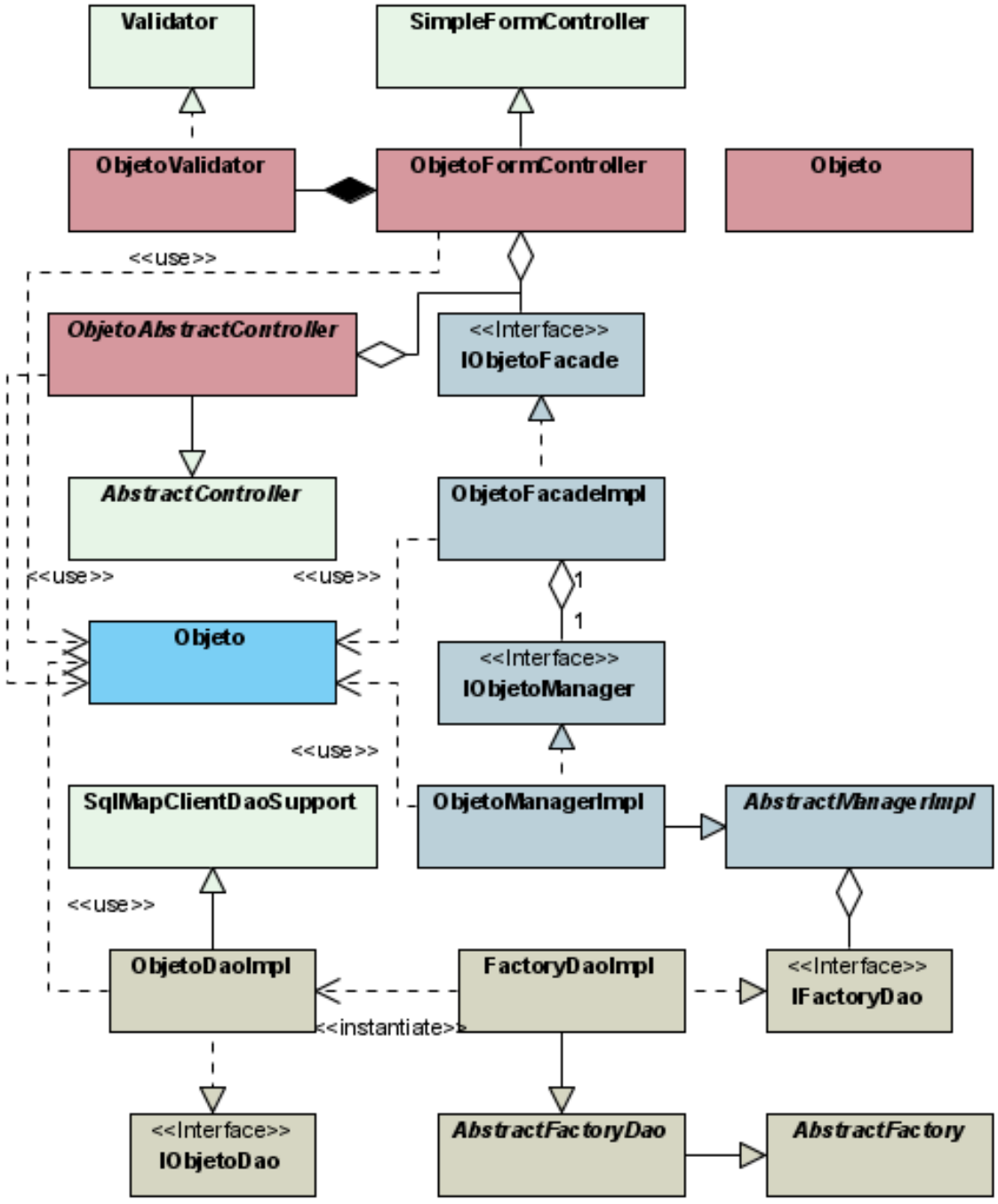

Fig. 2.4. Diagrama General de Clases del Diseño.

2.4. Diagrama de Paquetes

Los diagramas de paquetes se usan para reflejar la organización de paquetes y sus elementos, muestran como un sistema está dividido en agrupaciones lógicas mostrando las dependencias entre esas agrupaciones. Un paquete de diseño es una colección de clases, relaciones, realizaciones de

casos de usos, diagramas y otros paquetes. Es usado para estructurar el modelo de diseño mediante su división en partes más pequeñas y agrupar elementos relacionados de dicho modelo con propósitos organizacionales.

Para el diseño de este módulo las clases se agruparon por paquetes según su funcionalidad y las entidades que gestionan. Los paquetes definidos son: paquete Gestionar Funcionario, paquete Gestionar Arma, paquete Gestionar Dependencia, y paquete Común. Cada paquete contiene cuatro diagramas de clases correspondientes a la Capa de Presentación, Capa de Negocio, Capa de Acceso a Datos y el Dominio. Esta división se hace necesaria para comprender con más precisión cada uno de los elementos que contiene, debido a la complejidad de los diagramas y el gran número de clases contenidos en los mismos.

Se realizará una breve descripción de las principales clases contenidas en el paquete. En la descripción de las clases solo se especificarán los atributos y métodos principales. En cada clase por cada atributo definido existirán los métodos *get* y *set* que aunque no se describan estarán presentes.

Los paquetes especificados no comprenden las funcionalidades definidas para la seguridad del sistema debido a la integración existente con el CAS (Central Autentication Service) de SIGEPOL, que es una aplicación que se encarga de filtrar todas las peticiones antes de ejecutar alguna otra acción. Además controla la apertura y cierre de la sesión para los usuarios que permanezcan inactivos, y garantiza que quien se autentica es un humano.

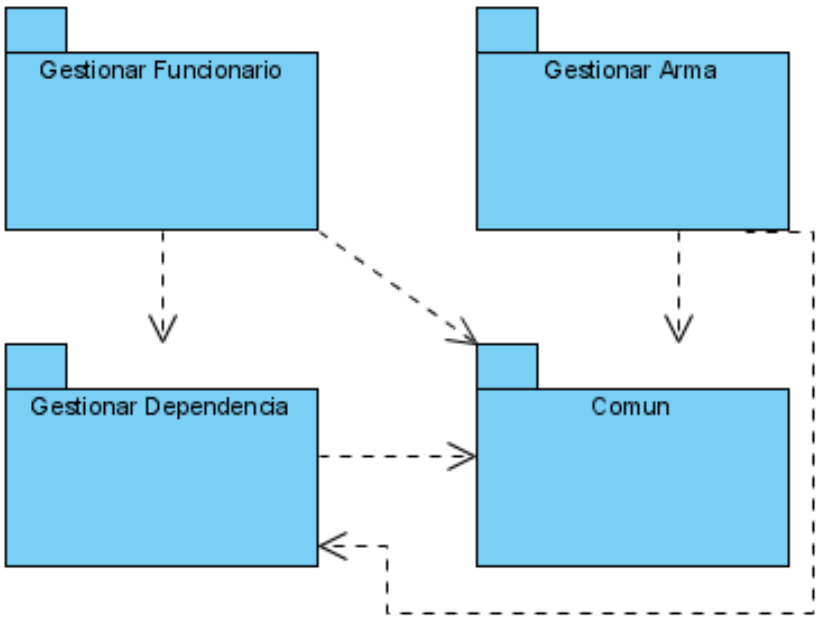

Fig. 2.5. Diagrama de Paquetes del Diseño.

2.5. Paquete Gestionar Funcionario

El paquete Gestionar Funcionario permitirá gestionar toda la información acerca de los funcionarios de las dependencias policiales.

La aplicación permitirá registrar un nuevo funcionario, mostrar el listado de funcionarios que cumplan con los criterios de búsqueda introducidos por el usuario, registrar un arma de fuego personal del mismo, mostrar el historial de las dependencias donde ha laborado, así como mostrar los detalles del funcionario que se seleccione. Una vez seleccionado un funcionario se podrán modificar algunos de sus datos personales, así como la dirección, número de teléfono y el estado en que se encuentra el mismo. Se registrará las sanciones y los méritos, se podrá abrir un expediente al funcionario por alguna indisciplina cometida.

2.5.1. Diagrama de Clases de la Capa Presentación

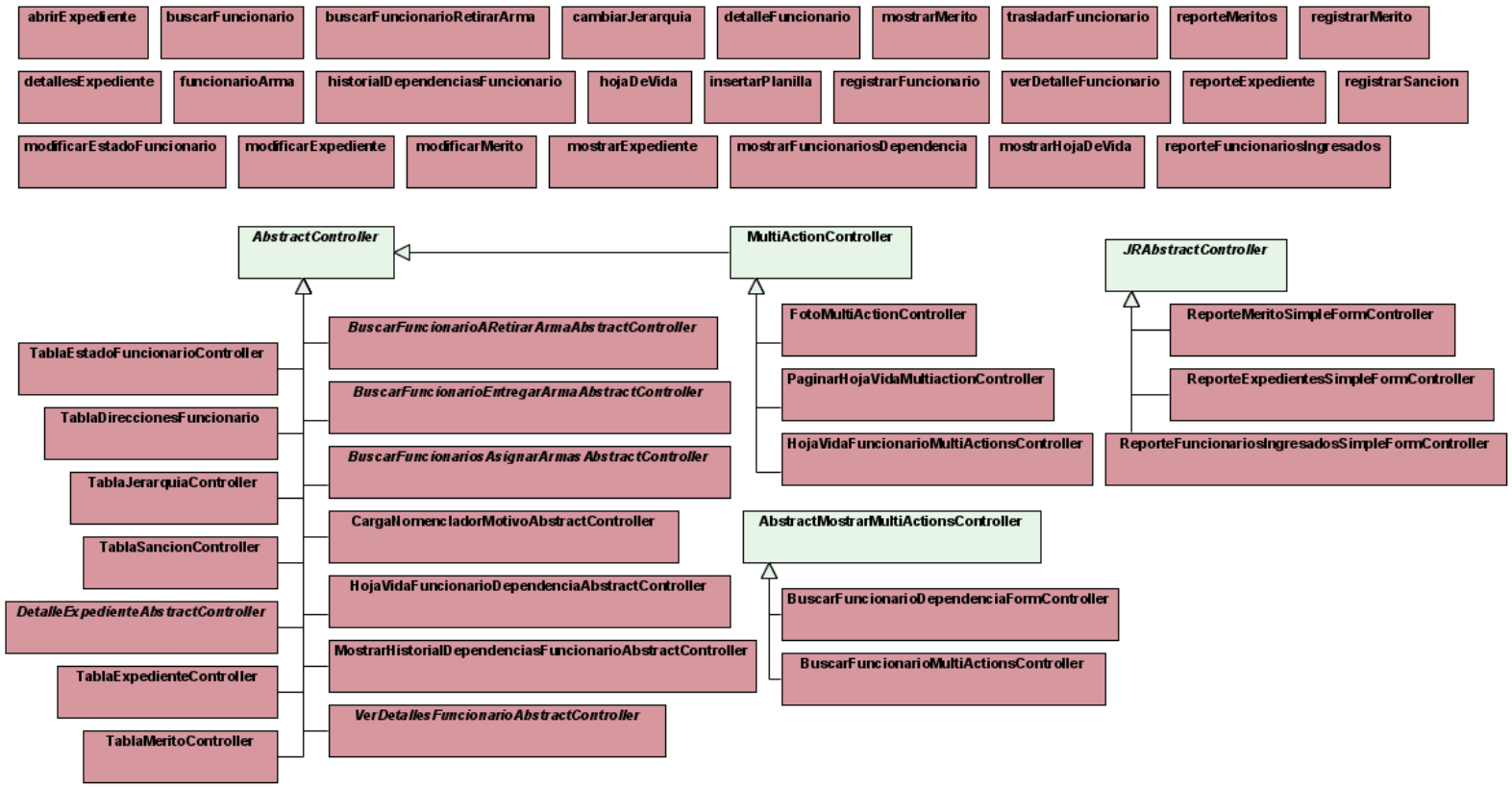

Fig. 2.6. Diagrama de Clases de la Capa Presentación del Paquete Gestionar Funcionario. (a)

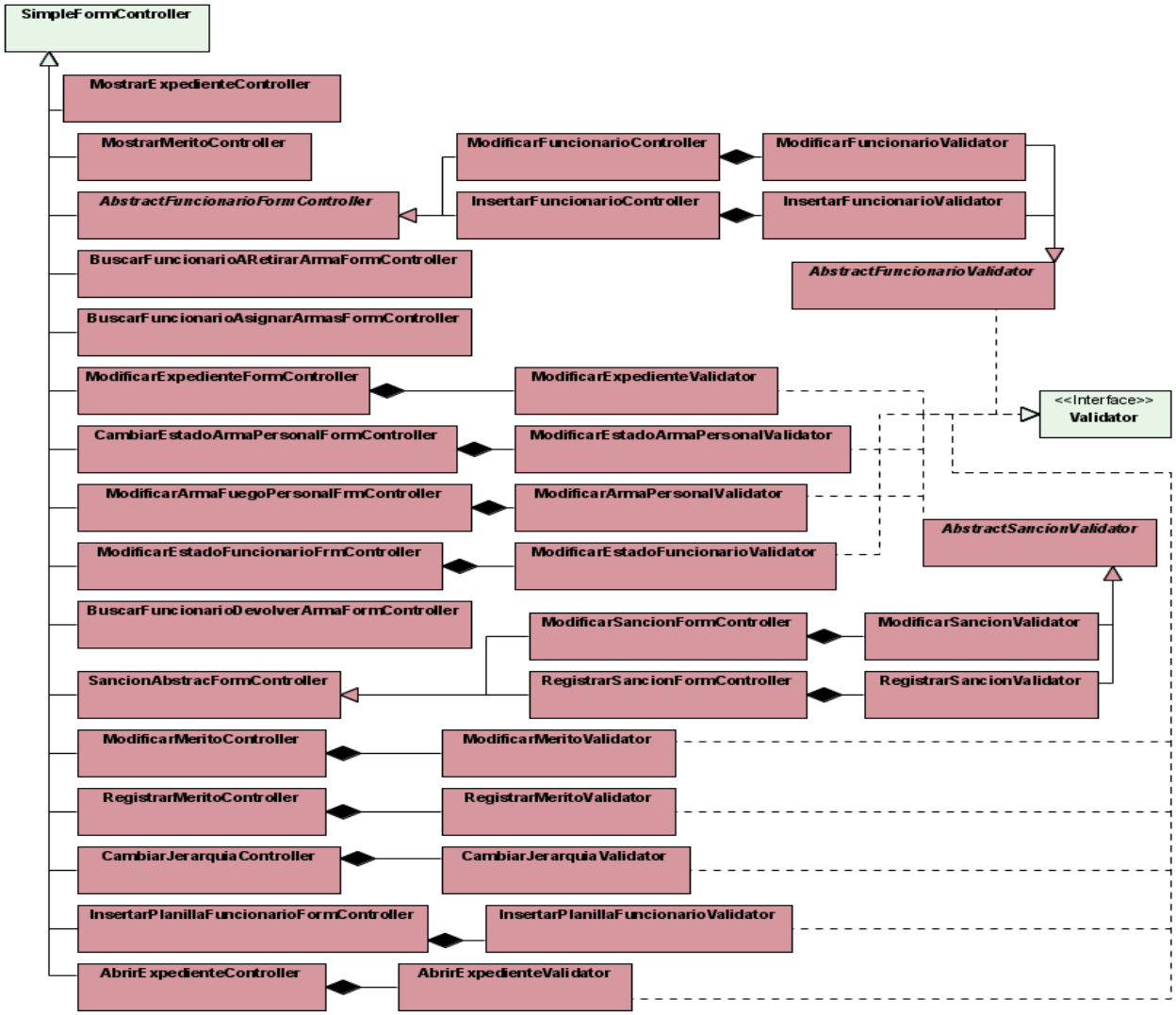

Fig. 2.6. Diagrama de Clases de la Capa Presentación del Paquete Gestionar Funcionario. (b)

Clase BuscarFuncionarioDependenciaFormController

Propósito:

Controlador que atiende la petición buscar funcionario de una dependencia policial.

Atributos:

- IFuncionarioFacade funcionarioFacade: Permite acceder a las funcionalidades proporcionadas por la interfaz “IFuncionarioFacade”.
- INomencladorFacade nomencladorFacade: Permite acceder a las funcionalidades proporcionadas por la interfaz “INomencladorFacade”.

Métodos:

- Map<String, Object> bindParameters(HttpServletRequest request): Método auxiliar para capturar los parámetros que son enviados desde la interfaz (cliente) como parámetros de búsqueda.
- ModelAndView mostrarPagina (HttpServletRequest request, HttpServletResponse response): Confecciona todos los elementos del modelo que van a ser mostrados en la vista formulario que constituyen los criterios de búsqueda de un funcionario y retorna el ModelAndView correspondiente.
- ModelAndView listar (HttpServletRequest request, HttpServletResponse response): Confecciona el ModelAndView compuesto por los funcionarios encontrados que serán listados en la vista.

Observaciones:

- Hereda de la clase abstracta AbstractMostrarMultiActionsController.

Clase InsertarFuncionarioController

Propósito:

Controlador que atiende la petición de registrar funcionario.

Atributos:

No aplica

Métodos:

- Map<String,Object> referenceData(HttpServletRequest arg0, Object arg1, Errors arg2): Permite suministrar datos que se necesitan en la vista del formulario.
- void onBind(HttpServletRequest request, Object command, BindException errors): Obtiene del request el acta y la foto del funcionario y se los adiciona para que contengan toda la información necesaria y pueda ser validado.

- ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors): Si no existen errores (almacenados en errors) en la validación del formulario, procesa el mismo y retorna el modelo y/o la vista que se va a mostrar.
- Object formBackingObject(HttpServletRequest request): Retorna por defecto una instancia de la clase comando que tiene configurada, pero se puede sobrescribir para obtener ese objeto con valores de la base de datos.
- ModelAndView showForm(HttpServletRequest request, HttpServletResponse response, BindException errors): Si no existen errores (almacenados en errors) actualiza las listas de armas y direcciones del funcionario a nivel de sesión.

Observaciones:

- Hereda de la clase abstracta AbstractFuncionarioFormController

Clase HojaVidaFuncionarioDependenciaAbstractController

Propósito:

Controlador que atiende la petición mostrar las hojas de vida de los funcionarios.

Atributos:

- IFuncionarioFacade funcionarioFacade: Permite acceder a las funcionalidades proporcionadas por la interfaz “IFuncionarioFacade”.

Métodos:

- ModelAndView handleRequestInternal(HttpServletRequest arg0, HttpServletResponse arg1): Método que atiende la petición sin un objeto command asociado; ejecuta la búsqueda auxiliándose de las funcionalidades que brinda la fachada funcionarioFacade. Crea un objeto ModelAndView con los datos de la búsqueda y lo devuelve.

Observaciones:

Hereda de la clase abstracta AbstractController

2.5.2. Diagrama de Clases de la Capa Negocio

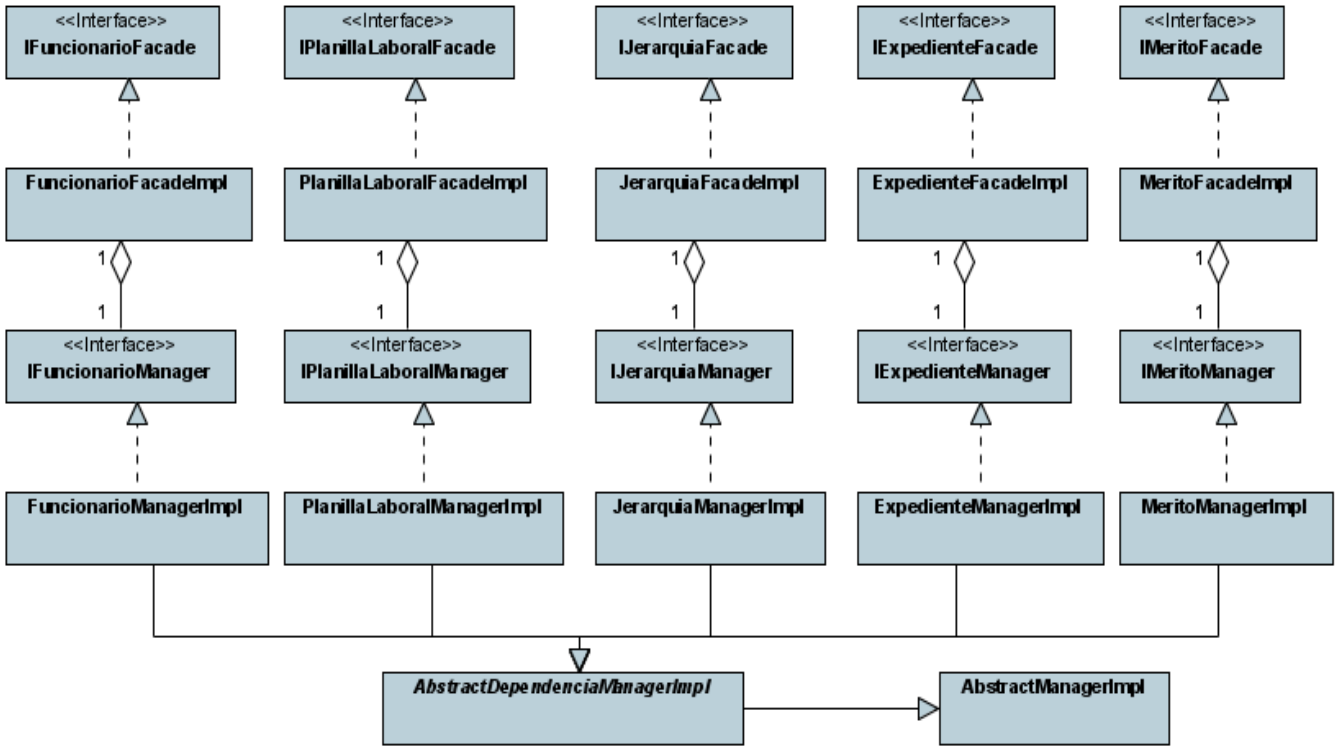

Fig. 2.7. Diagrama de Clases de la Capa Negocio del Paquete Gestionar Funcionario.

Clase FuncionarioManagerImpl

Propósito:

Implementa la interfaz IFuncionarioManager que define las posibles acciones sobre el funcionario a nivel de lógica de negocio.

Atributos:

- List<ArmaFuegoPersonal> listaDeArmasPersonales: Almacena la lista de armas de fuego personales del funcionario.
- List<HistorialEstadoFuncionario> listaHistorialEstadoFuncionario: Almacena el historial de los estados del funcionario.
- IDireccionManager direccionManager: Almacena el identificador de la dirección del funcionario.
- int idActualEstado: Almacena el identificador del estado actual del funcionario.
- int idActualJerarquia: Almacena el identificador actual de la jerarquía del funcionario.
- String idDependencia: Almacena el identificador de la dependencia del funcionario.

Métodos:

- void vaciarlistaHistorialEstadoFuncionario(): Elimina los elementos de la lista de historiales de los estados del funcionario.
- void adicionarArmaPersonal(ArmaFuegoPersonal arma): Adiciona el arma de fuego que se pasa por parámetros en la lista de armas que se tiene como parámetros.
- Map<String, Object> obtenerListaArmasPersonales(int posicionInicial, int cantidad): Devuelve el listado paginado de armas personales del funcionario.
- AltaFuncionario buscarDatosFuncionario(String idAltaFuncionario): Retorna los datos de un funcionario en forma de acta dado el identificador del acta del funcionario.
- AltaFuncionario buscarFuncionarioDependenciaId(String idFuncionario): Devuelve los datos de un funcionario en forma de acta dado el identificador del funcionario.
- void eliminarFuncionario(String idFuncionario): Elimina el funcionario que se corresponde con el identificador del funcionario que se pasa por parámetro.
- void modificarFuncionario(AltaFuncionario alta): Actualiza los datos de un funcionario dado el alta de funcionario.
- String registrarFuncionario(AltaFuncionario alta): Adiciona un funcionario dado el alta del mismo.
- FuncionarioSIGEPOL obtenerFuncionarioArma(String serialArma, String tipoSerial): Retorna un funcionario dado el serial y el tipo de serial del arma personal.
- FuncionarioResult obtenerFuncionarioSIGEPOL(String cedula): Devuelve un funcionario dado su cédula.
- List<UbicacionPolicial> obtenerUbicacionesPoliciales(String cedula): Retorna la lista de ubicaciones policiales de un funcionario dado su cédula.
- Map<String, Object> listarFuncionarioDependencia(Map<String, Object> criterios): Retorna el listado de funcionarios según los criterios de búsqueda y la dependencia.
- Map<String, Object> listarFuncionario(Map<String, Object> criterios): Retorna el listado de funcionarios según los criterios de búsqueda.
- AltaFuncionario ultimaAltaFuncionarioDependencia(String idFuncionario, String idDependencia): Devuelve la última acta de un funcionario dado el identificador del mismo y el de la dependencia en la que laboraba.
- PlanillaLaboral ultimaPlanillaLaboralAltaFuncionario(String idAltaFuncionario): Devuelve la última planilla laboral de un funcionario dado el identificador del alta del funcionario.

- Map<String, Object> listarFuncionarioAsignarArmas(Map<String, Object> criterios): Devuelve el listado de funcionarios a los cuales se les debe asignar armas que cumplen además con los criterios de búsqueda.
- List<AltaFuncionario> reporteFuncionariosIngresados(Map<String, Object> map): Devuelve el listado de las altas de funcionarios ingresados.
- List<Map<String, Object>> historialDependencias(String idFuncionario): Devuelve el historial de las dependencias en las que ha laborado un funcionario dado el identificador del mismo.
- Map<String, Object> obtenerListaHistorialEstadoFuncionario(Map<String, Object> criterios): Devuelve el historial de los estados del funcionario que cumple con los criterios de búsqueda.
- Map<String, Object> obtenerListaDireciones(Map<String, Object> criterios): Devuelve el listado de direcciones dado los criterios de búsqueda.
- AltaFuncionario buscarDatosAlta(String idAltaFuncionario): Devuelve el alta del funcionario dado su identificador.
- void registrarPlanillaLaboral(PlanillaLaboral planillaLaboral): Adiciona la planilla laboral que se pasa por parámetros.
- Map<String, Object> obtenerListaArmasPersonalesIdFuncionario(Map<String, Object> criterios): Devuelve el listado de armas personales que cumplen con los criterios de búsqueda.
- AltaFuncionario seleccionarPorIdPrestamo(String idPrestamo): Devuelve el alta de funcionario dado el identificador del préstamo de arma.
- AltaFuncionario datosBasicosFuncionario(String idFuncionario): Devuelve los datos de un funcionario en forma de alta dado el identificador del funcionario.
- byte[] obtenerFotoFuncionario(String idFuncionario): Devuelve la foto del funcionario en forma de arreglos de byte dado el identificador del funcionario.
- Map<String, Object> listarFuncionarioEntregarArma(Map<String, Object> criterios): Devuelve el listado de funcionarios que están por entregar armas y cumplen además con los criterios de búsqueda.
- Map<String, Object> listarFuncionarioRetirarArmas(Map<String, Object> criterios): Devuelve el listado de funcionarios que están por retirar armas y cumplen además con los criterios de búsqueda.
- void darBajaFuncionarioEnDependencia(String idAltaFuncionario, Date fechaEgreso, String idFuncionario): Elimina un funcionario de una dependencia dado el identificador del funcionario, el identificador del alta de funcionario y la fecha de egreso del mismo.

- void cambiarEstadoArmaEnLista(String idArma, HistorialEstadoArma estado): Actualiza el estado de un arma dado su identificador y el nuevo estado que se le pasa por parámetro.
- UsuarioSistema usuarioPorIdFuncionario(String idFuncionario): Devuelve el usuario del sistema dado el identificador del funcionario.

Observaciones:

- Hereda de la clase abstracta AbstractDependenciaManagerImpl e implementa la Interfaz IFuncionarioManager.

2.5.3. Diagrama de Clases de la Capa Acceso a Datos

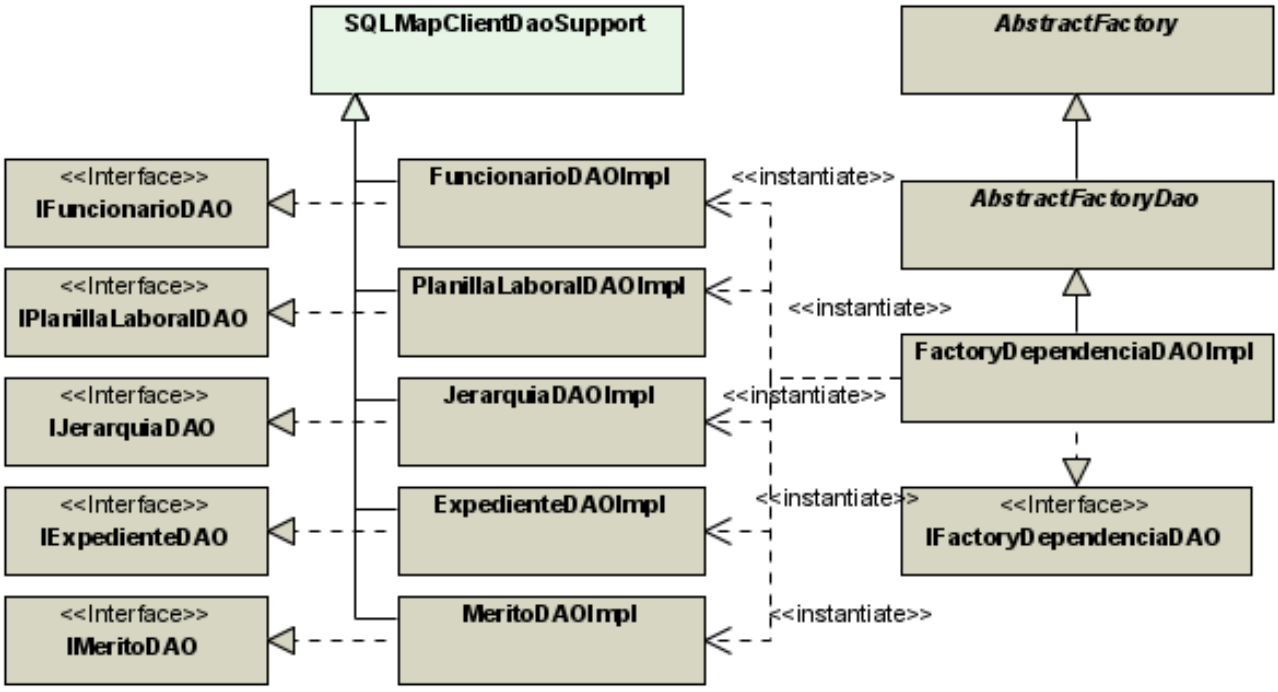

Fig. 2.8. Diagrama de Clases de la Capa Acceso a Datos del Paquete Gestionar Funcionario.

Clase ExpedienteDAOImpl

Propósito:

Implementa la interfaz IExpedienteDao para el manejo de posibles acciones sobre un Expediente a nivel de acceso a datos.

Atributos:

No aplica

Métodos:

- Map<String, Object> obtenerListaExpedientes(Map<String, Object> criterios): Invoca al procedimiento almacenado que devuelve el listado de expediente dado criterios de búsqueda.
- Map<String, Object> buscarExpedientes(String idAltaFuncionario, Integer inicio, Integer cantidad): Invoca al procedimiento almacenado que devuelve el listado de expediente dado el identificador del funcionario y los criterios de paginación.
- void abrirExpediente(Expediente expediente): Inserta el expediente confeccionado del funcionario.
- Expediente detalleExpediente(String idExpediente): Devuelve todos los detalles de un expediente dado su identificador.
- boolean expedienteCerrado(String idExpediente): Devuelve verdadero en caso de que el estado de un expediente sea cerrado, de lo contrario devuelve falso dado el identificador del expediente.
- void modificarExpediente(Expediente expediente): Se modifica el expediente del funcionario.
- Sancion obtenerSancionExpediente(String idExpediente): Devuelve la última sanción dado el identificador del expediente del funcionario.
- Sancion obtenerSancionExpediente(String idFuncionario,String idExpediente): Devuelve la última sanción dado el identificador del expediente del funcionario y el identificador del funcionario.
- List<Expediente> listarExpedientes(String idFuncionario, String idDependencia) Obtiene el listado de expediente que tiene asociado un funcionario, dado su identificador y el identificador de la dependencia policial.
- String existeExpediente(Expediente expediente): En caso de que se encuentre registrado el expediente en la base de datos se retorna su identificador.
- List<AltaFuncionario> reporteExpedientes(Map<String, Object> map): Invoca al procedimiento almacenado que devuelve el listado de altas del funcionario dado criterios de búsqueda.
- String idAltaExpediente(String idExpediente): Obtiene el identificador del alta de expediente de funcionario dado el identificador del expediente.
- void registrarSancion(Sancion sancion): Inserta una sanción en el expediente del funcionario.
- void modificarSancion(Sancion sancion): Modifica la sanción en el expediente del funcionario.
- void actualizarCerrado(String idExpediente, boolean estado): Modifica el estado de un expediente dado el identificador del expediente.
- boolean mostrarCambioEstado(int sancionN): Retorna el estado de un expediente dado el identificador de la sanción.

- boolean mostrarCerrado(String idExpediente): Retorna el estado del expediente dado su identificador

Observaciones:

- Hereda de la clase SqlMapClientDaoSupport e implementa la interfaz IExpedienteDAO.

Clase PlanillaLaboralDAOImpl

Propósito:

Implementa la interfaz IPlanillaLaboralDAO que define las posibles acciones sobre una planilla laboral a nivel de acceso a datos.

Atributos

- No aplica

Métodos:

- String insertarPlanillaLaboral (PlanillaLaboral planilla): Inserta la planilla laboral del funcionario retornando el identificador de la misma.
- PlanillaLaboral ultimaPlanillaLaboralFuncionario (String idAlta): Devuelve la última planilla laboral del funcionario dado el identificador del acta de la planilla.
- AltaFuncionario ultimaAltaFuncionario (String idFuncionario, String idDependencia): Devuelve la última acta del funcionario dado el identificador del funcionario y el identificador de la dependencia.
- List<Dependencia> dependenciasFuncionario(String idFuncionario): Obtiene el listado de las dependencias a las cuales ha pertenecido el funcionario dado el identificador del funcionario.
- List<Nomenclador> cargarTiposFuncionario(): Obtiene el listado de los tipos de funcionarios.
- List<Nomenclador> cargarCargosFuncionario(): Obtiene el listado de los cargos de funcionarios.
- Map<String, Object> obtenerListaPlanillas (Map<String, Object> parametros): Invoca al procedimiento almacenado que devuelve el listado de planillas.
- List<PlanillaLaboral> obtenerListaPlanillas(String idFuncionario, String idDependencia): Obtiene el listado de planillas laborales de un funcionario dado el identificador del funcionario y el identificador de la dependencia.

Observaciones:

- Hereda de la clase SqlMapClientDaoSupport e implementa la interfaz IPlanillaLaboralDAO.

2.5.4. Diagrama de Clases del Dominio

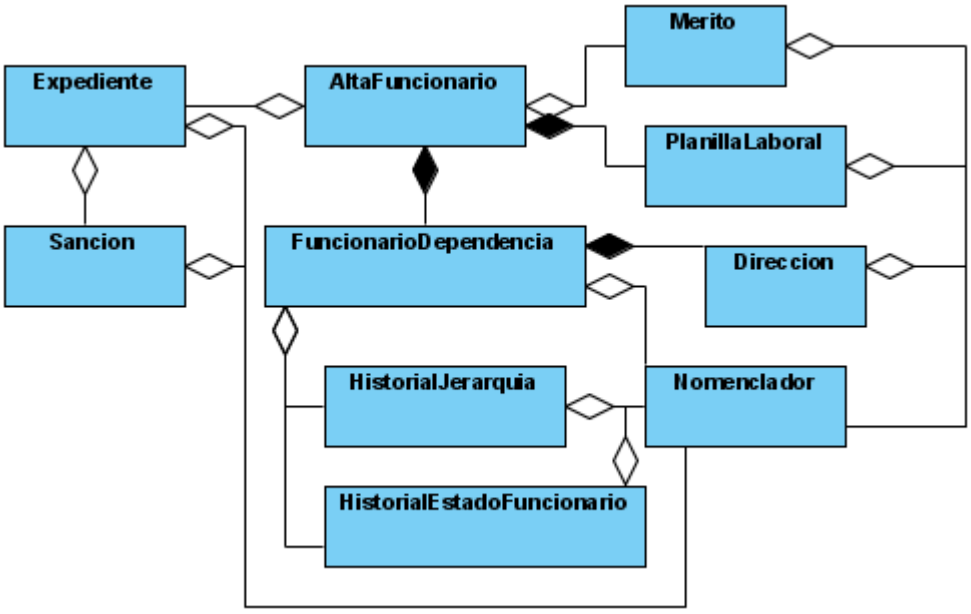

Fig. 2.9. Diagrama de Clases del Dominio del Paquete Gestionar Funcionario.

Clase FuncionarioDependencia

Propósito:

Clase de dominio propuesta para almacenar y propagar verticalmente los datos básicos del funcionario.

.Atributos*:*

- String lugarNacimiento: Almacena el lugar de nacimiento del funcionario.
- int activado: Almacena el estado del funcionario.
- Nomenclador formacionAcademica: Almacena la formación académica del funcionario.
- Nomenclador gradoInstruccion: Almacena el grado de instrucción del funcionario.
- HistorialJerarquia historialJerarquia: Almacena el historial de jerarquía del funcionario.
- HistorialEstadoFuncionario estado: Almacena el historial de los estados del funcionario.
- ArmaFuegoPersonal armasPersonales: Almacena las armas de fuego personal del funcionario.
- Direccion direcciones: Almacena las direcciones del funcionario.

Métodos:

- JSONObject createJSONObject(): Crea la representación en JSON de un objeto de tipo FuncionarioDependencia.

Observaciones:

- Hereda de la clase del dominio Funcionario.

Clase AltaFuncionario

Propósito:

Clase de dominio propuesta para almacenar y propagar verticalmente los datos del alta del funcionario.

.Atributos:

- String idAltaFuncionario: Almacena el lugar de nacimiento del funcionario.
- FuncionarioDependencia funcionario: Almacena el estado del funcionario.
- String idAltaFuncionario: Almacena el identificador del alta del funcionario.
- FuncionarioDependencia funcionario: Almacena la información de la relación Funcionario Dependencia.
- String credencial: Almacena la credencial.
- Date fechaIngreso: Almacena la fecha de ingreso.
- Date fechaEgreso: Almacena la fecha de egreso.
- Expediente expediente: Almacena la información del expediente.
- PlanillaLaboral planilla: Almacena la información de la planilla laboral.
- Merito merito: Almacena la información del mérito del funcionario.
- String idDependencia: Almacena el identificador de la dependencia.
- String nombreDependencia: Almacena el nombre de la dependencia.

Métodos:

- JSONObject createJSONObject(): Crea la representación en JSON de un objeto de tipo Altafuncionario.
- JSONObject createJSONObjectFuncionario(): Crea la representación en JSON de un objeto de tipo Funcionario.

Observaciones:

No aplica.

2.6. Paquete Gestionar Arma

El paquete Gestionar Arma permite llevar el control de toda la información acerca de las armas de fuego de las dependencias policiales. Sus funcionalidades principales son registrar, asignar y retirar un arma, actualizar los datos y modificar el estado de la misma, mostrar la información del armamento de

las dependencias policiales. Además de registrar la entrega y devolución del arma de fuego temporalmente.

2.6.1. Diagrama de Clases de la Capa Presentación

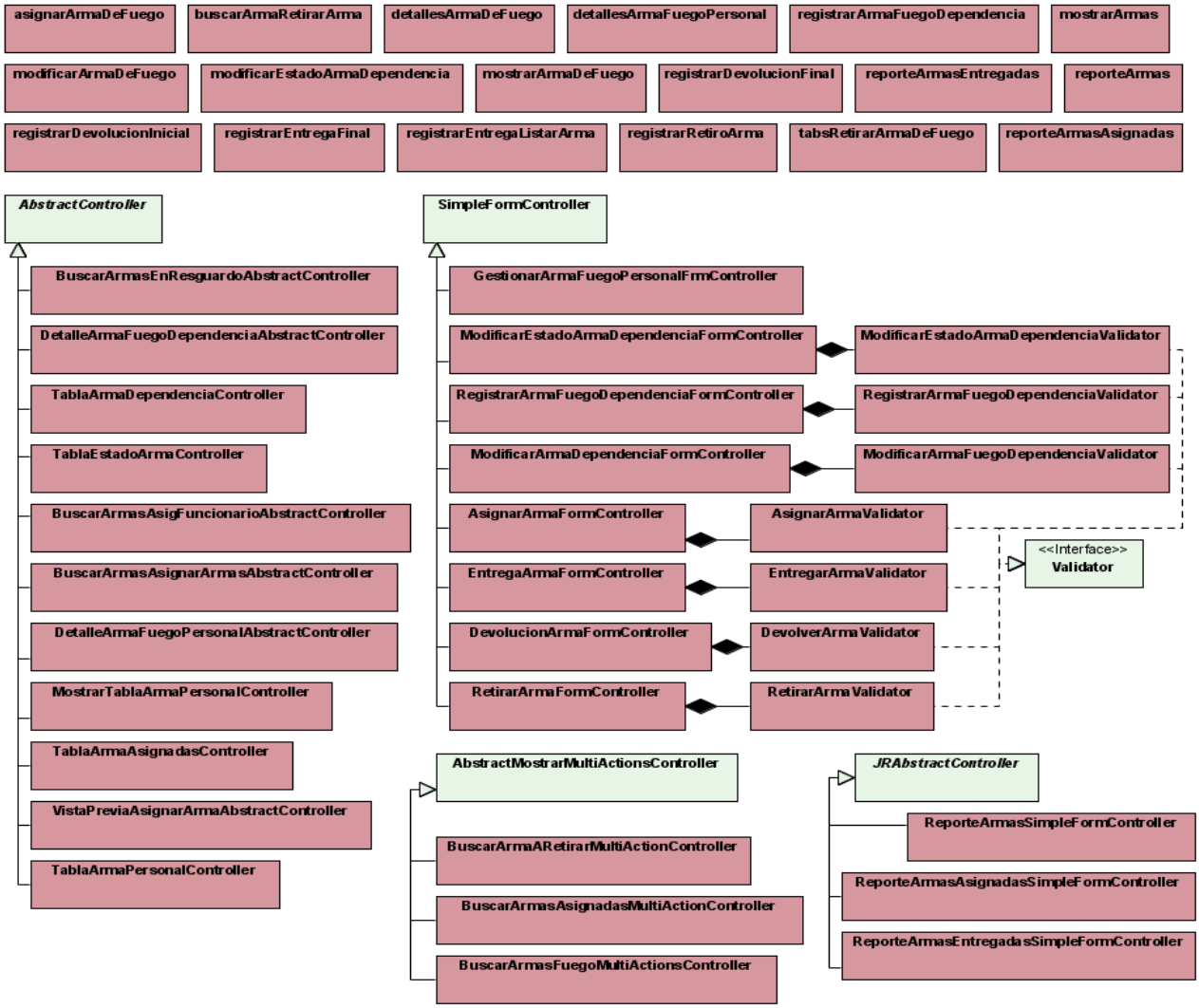

Fig. 2.10. Diagrama de Clases de la Capa Presentación del Paquete Gestionar Arma.

Clase RegistrarArmaFuegoDependenciaFormController

Propósito:

Controlador que atiende la petición de registro del arma de fuego.

Atributos:

- IArmaDeFuegoFacade armaFacade: Referencia a la fachada (interfaz) de arma de fuego que define las posibles acciones sobre el arma.

- FuncionarioFacade funcionarioFacade: Referencia a la fachada (interfaz) de funcionario que define las posibles acciones sobre el funcionario.

Métodos:

- ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors): Si no existen errores (almacenados en errors) en la validación del formulario, procesa el mismo y retorna el modelo y/o la vista que se va a mostrar.
- Map<String,Object> referenceData(HttpServletRequest request): Permite suministrar datos que se necesitan en la vista del formulario
- void initBinder(HttpServletRequest request, ServletRequestDataBinder binder): Define las estrategias de conversiones de tipos de datos. Permite registrar los editores para aquellas propiedades del objeto comando (command) que no sean String.

Observaciones:

- Hereda de la clase SimpleFormController.

Clase BuscarArmaARetirarMultiActionController

Propósito:

Controlador que atiende las peticiones para eliminar, buscar y mostrar la página que contiene los formularios necesarios para insertar el arma a retirar.

Atributos:

- IArmaDeFuegoFacade armaFacade: Referencia a la fachada (interfaz) del arma de fuego que define las posibles acciones sobre el arma.
- FuncionarioFacade funcionarioFacade: Referencia a la fachada (interfaz) del funcionario que define las posibles acciones sobre el funcionario.

Métodos:

- Map<String, Object> bindParameters(HttpServletRequest request): Método auxiliar para capturar los parámetros que son enviados desde la interfaz (cliente) como parámetros de búsqueda.
- ModelAndView mostrarPagina (HttpServletRequest request, HttpServletResponse response): Confecciona todos los elementos del modelo que van a ser mostrados en la vista del formulario que constituyen los criterios de búsqueda de un arma de fuego y retorna el ModelAndView correspondiente.
- ModelAndView listar(HttpServletRequest request, HttpServletResponse response): Confecciona el ModelAndView compuesto por las armas a retirar encontradas que serán listadas en la vista.

Observaciones

- Hereda de la clase AbstractMostrarMultiActionsController.

2.6.2. Diagrama de Clases de la Capa Negocio

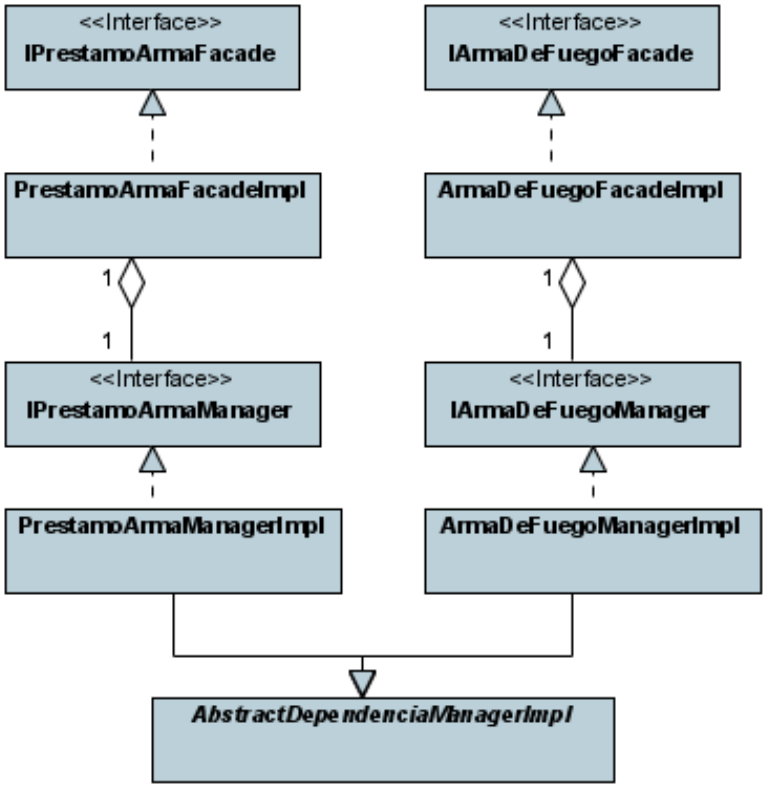

Fig. 2.11. Diagrama de Clases de la Capa Negocio del Paquete Gestionar Arma.

Clase ArmaDeFuegoManagerImpl

Propósito:

Implementa la interfaz IArmaDeFuegoManager que define las posibles acciones sobre el arma de fuego a nivel de lógica de negocio.

Atributos:

- List<ArmaFuegoDependencia> listaDeArmasDependencia: Almacena el listado de armas de la dependencia.
- List<HistorialEstadoArma> listaHistorialEstado: Almacena el listado de los historiales de los estados de las armas de fuego.

- List<PrestamoArma> listaPrestamoArma: Almacena el listado de los préstamos de armas de fuego de la dependencia.

Métodos:

- void llenarListaHistorialEstadoIdArma(String idArma): Adiciona los estados de las armas de fuego a la lista de historiales dado el identificador del arma.
- void llenarListaPrestamoArmaIdFuncionario(String idFuncionario): Adiciona los préstamos del arma de fuego a la lista de préstamos dado el identificador del funcionario al que se le presta.
- void vaciarListaDeArmasDependencia(): Elimina todos las armas del listado de armas de la dependencia.
- void vaciarListaHistorialEstado(): Elimina todos los historiales de estados del listado de historiales de estado de la dependencia.
- void vaciarListaPrestamoArma(): Elimina todos los préstamos del listado de préstamos de la dependencia.
- Map<String, Object> obtenerListaArmasDependencia(int posicionInicial, int cantidad): Devuelve el listado de armas de la dependencia.
- Map<String, Object> detalleArmaFuegoDependencia (String idArma): Devuelve los detalles de un arma de fuego dado el identificador del arma de fuego.
- void insertarArmaDependencia(ArmaFuegoDependencia arma): Adiciona una nueva arma de fuego con los datos pasados en el parámetro “arma”.
- void modificarArmaDependencia(ArmaFuegoDependencia armaFuego): Modifica los datos del arma de fuego existente con los nuevos datos pasados en el parámetro “armaFuego”.
- Map<String, Object> buscarArmasPorDependenciaPaginado(ArmaFuegoDependencia arma,
- integer posicionInicial, Integer cantidad, String usuario): Devuelve el listado de armas por dependencia paginado dado el usuario.
- Map<String, Object> detalleArmaFuegoPersonal(String idArmaFuego): Devuelve los detalles de un arma de fuego personal dado el identificador del arma de fuego.
- Map<String, Object> obtenerListaHistorialEstadoArma(int posicionInicial, int cantidad): Devuelve la lista de historiales de estados de las armas de la dependencia paginada.
- Map<String, Object> buscarArmasEnResguardo(Map<String, Object> criterios): Devuelve el listado de las armas en resguardo dado los criterios de búsqueda.

Observaciones:

- Hereda de la clase AbstractDependenciaManagerImpl e implementa la interfaz IArmaDeFuegoManager.

2.6.3. Diagrama de Clases de la Capa Acceso a Datos

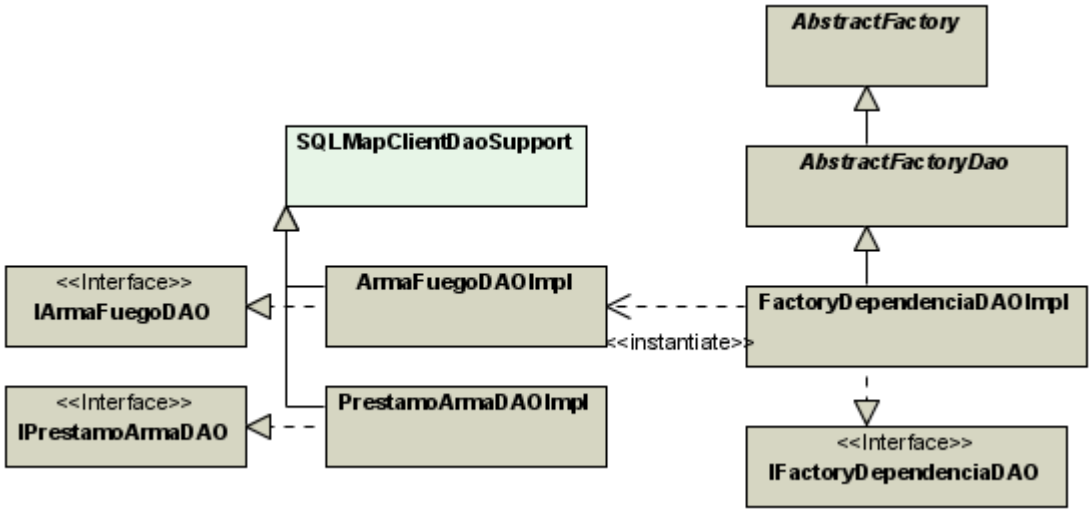

Fig. 2.12. Diagrama de Clases de la Capa Acceso a Datos del Paquete Gestionar Arma.

Clase PrestamoArmaDAOImpl

Propósito:

Implementa la interfaz IPrestamoArmaDao que define las posibles acciones sobre un préstamo de arma de fuego a nivel de acceso a datos.

Atributos:

- No aplica

Métodos:

- AsignarArmaFuncionario(PrestamoArma prestamoArma): Asigna un arma al funcionario en forma de préstamo.
- void registrarDevolucionDeArma(PrestamoArma prestamoArma): Registra la devolución de un préstamo de arma.
- Date fechaHoraPosibleDevolucion(String idPrestamoArma): Devuelve la fecha de la posible devolución del arma prestada dado el identificador del préstamo de arma.

Observaciones:

- Hereda de la clase SqlMapClientDaoSupport e implementa la interfaz IPrestamoArmaDAO.

2.6.4. Diagrama de Clases del Dominio

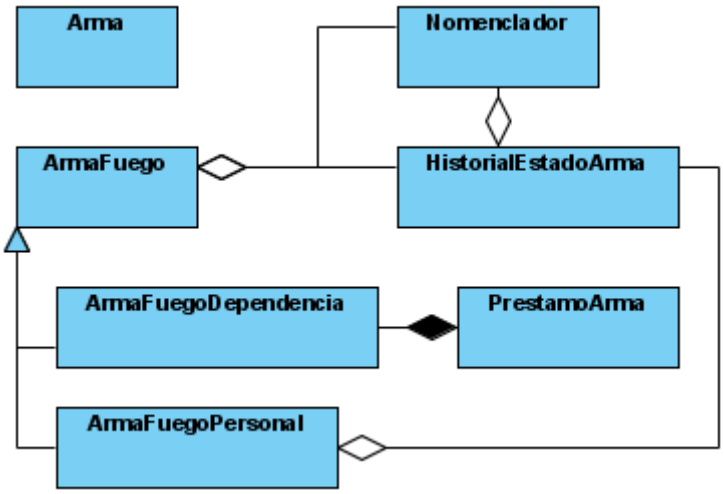

Fig. 2.13. Diagrama de Clases del Dominio del Paquete Gestionar Arma.

Clase ArmaFuego

Propósito:

Clase de dominio propuesta para almacenar y propagar verticalmente los datos del arma de fuego en cuestión.

Atributos:

- String idArma: Almacena el identificador del arma.
- Nomenclador modelo: Almacena el modelo del arma.
- String calibre: Almacena el calibre del arma.
- Nomenclador tipo: Almacena el tipo de arma.
- String serialPrimario: Almacena el serial primario.
- String serialSecundario: Almacena el serial secundario.
- String serialTerciario: Almacena el serial terciario.
- Nomenclador marca: Almacena la marca del arma.
- HistorialEstadoArma historialEstados: Contiene el historial de los estados del arma.

Métodos:

- JSONObject createJSONObject(): Crea la representación en JSON de un objeto de tipo ArmaFuego.

Observaciones:

No aplica.

Clase PrestamoArma

Propósito:

Clase de dominio propuesta para almacenar y propagar verticalmente los datos del préstamo de arma de fuego.

Atributos:

- String idPrestamoArma: Almacena el identificador del préstamo de arma.
- AltaFuncionario funcionario: Almacena la información del alta del funcionario
- ArmaFuegoDependencia arma: Almacena la información del arma de fuego.
- Date fechaEntrega: Fecha del préstamo
- Date fechaDevolucion: Fecha de devolución.
- String descripcionEntrega: Almacena la información referente a la descripción de la entrega.
- String descripcionDevolucion: Almacena la información referente a la descripción de la devolución.
- String fechaPosibleDevolucion: Fecha de la posible devolución.
- String horaPosibleDevolucion: Hora de la posible devolución.
- String motivoIncumplimiento: Almacena la información del motivo del incumplimiento.

Métodos:

- JSONObject createJSONObject(): Crea la representación en JSON de un objeto de tipo Arma.
- JSONObject createJSONObjectPrestamoArmas(): Crea la representación en JSON de un objeto de tipo PrestamoArmas.

Observaciones:

No aplica.

2.7. Paquete Dependencia

El paquete Dependencia agrupa las clases responsables de gestionar la información de las dependencias policiales. Se centra principalmente en adicionar, eliminar, actualizar, y mostrar la información de las dependencias policiales.

2.7.1. Diagrama de Clases de la Capa Presentación

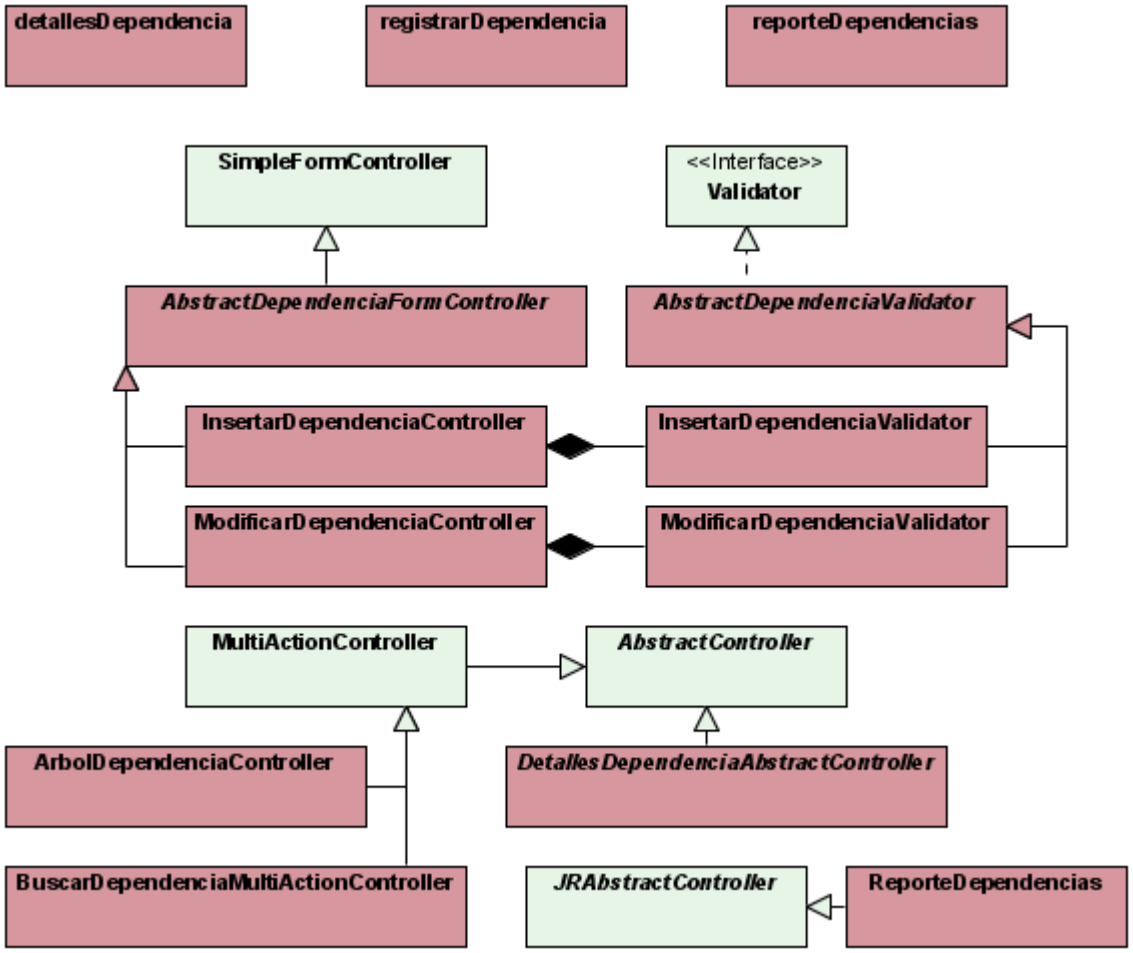

Fig. 2.14. Diagrama de Clases de la Capa Presentación del Paquete Gestionar Dependencia

Clase InsertarDependenciaController

Propósito:

Controlador que atiende la petición de registrar dependencia.

Atributos:

No aplica

Métodos:

- ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors): Si no existen errores (almacenados en errors) en la validación del formulario, procesa el mismo y retorna el modelo y/o la vista que se va a mostrar.
- Map<String,Object> referenceData(HttpServletRequest request): Permite suministrar datos que se necesitan en la vista del formulario void initBinder (HttpServletRequest arg0, ServletRequestDataBinder arg1).

Observaciones:

- Hereda de la clase abstracta AbstractDependenciaFormController.

2.7.2. Diagrama de Clases de la Capa Negocio

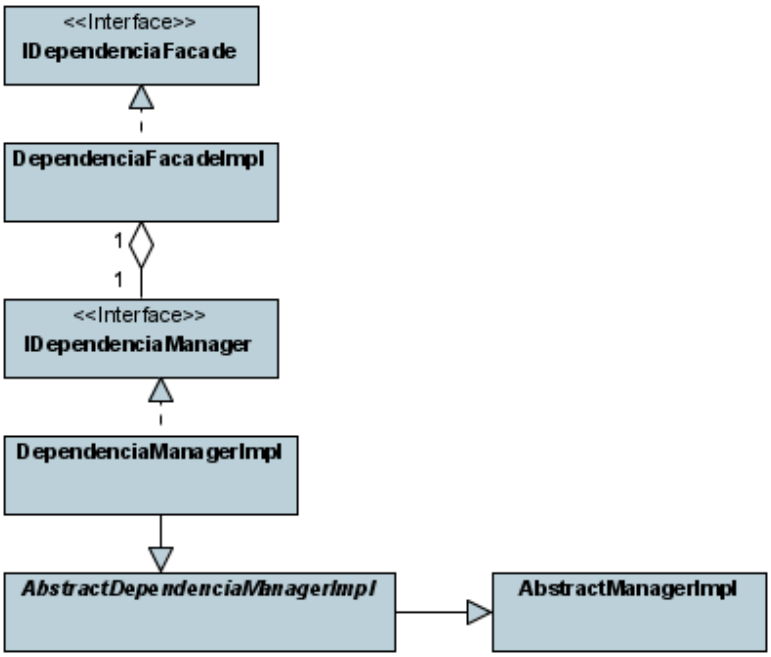

Fig. 2.15. Diagrama de Clases de la Capa Negocio del Paquete Gestionar Dependencia.

Clase DependenciaManagerImpl

Propósito:

Implementa la interfaz IDependenciaManager que define las posibles acciones sobre la dependencia a nivel de lógica de negocio.

Atributos:

No aplica

Métodos:

- String registrarDependencia(final Dependencia dependencia): Crea un objeto dependencia y lo adiciona devolviendo el identificador del mismo.
- void modificarDependencia(final Dependencia dependencia): Modifica los datos de la dependencia dado los nuevos datos que se pasan por parámetro.
- void eliminarDependencia(String idDependencia): Elimina una dependencia dado el identificador de la misma.
- Dependencia buscarDependenciaPorId(String idDependencia): Devuelve una dependencia dado el identificador de la dependencia que se pasa por parámetros.
- Map<String, Object> buscarDependenciaPaginado(Dependencia dependencia, int posicionInicial, int cantidad): Devuelve un listado paginado de dependencias.
- boolean existeDependencia(Dependencia dependencia): Devuelve “true” en caso de que exista la dependencia que se pasa por parámetros de lo contrario retorna “false”.
- List<Dependencia> reporteDependencias(String tipoDependencia, String estado, String municipio): Devuelve el listado de las dependencias que cumplen con los criterios de búsqueda pasados por parámetros.

Observaciones:

- Hereda de la clase abstracta AbstractDependenciaManagerImpl e implementa la interfaz IDependenciaManager

2.7.3. Diagrama de Clases de la Capa Acceso a Datos

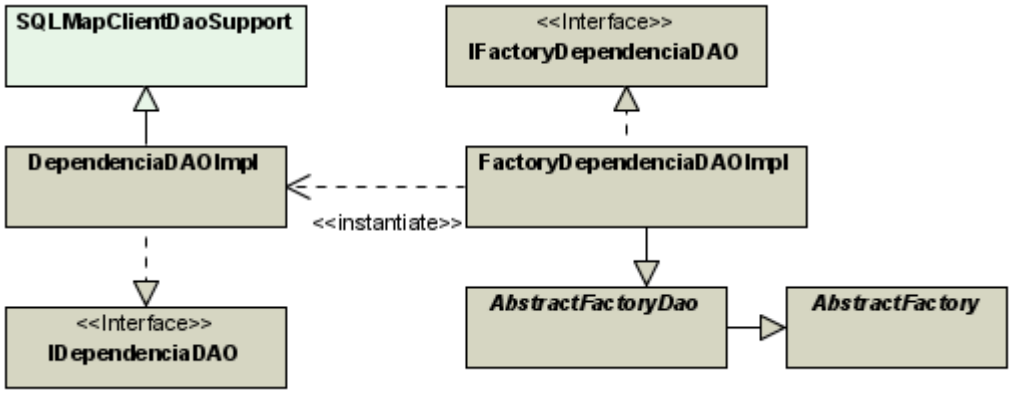

Fig. 2.16. Diagrama de Clases de la Capa Acceso a Datos del Paquete Gestionar Dependencia.

Clase DependenciaDAOImpl

Propósito:

Implementa la interfaz IDependenciaDao que define las posibles acciones sobre una dependencia policial a nivel de acceso a datos.

Atributos:

- No aplica

Métodos:

- Direccion direccionDependencia(String idDireccion): Devuelve la dirección de una dependencia dado el identificador de dirección.
- void actualizar(Dependencia dependencia): Modifica los campos de una dependencia tomando como base los de la dependencia que se pasa por parámetros.
- void eliminar(String idDependencia): Elimina una dependencia dado el identificador de la dependencia.
- String insertar(Dependencia dependencia): Adiciona una dependencia.
- Dependencia seleccionarPorId(String idDependencia): Devuelve una dependencia dado el identificador de la dependencia.
- List<Dependencia> buscarDependencias(Dependencia dependencia): Devuelve el listado de dependencias dado el código y el nombre de la dependencia que se pasa por parámetros.

- Map<String, Object> buscarDependenciaPaginado(Dependencia dependencia, int posicionInicial, int cantidad): Devuelve el listado de dependencias paginado dado el código y el nombre de la dependencia que se pasa por parámetros.
- void eliminarDireccionDependencia(String idDependencia): Elimina la dirección de una dependencia dado el identificador de la dependencia.
- void insertarTelefonos(Dependencia dependencia): Adiciona la lista de teléfonos de una dependencia.
- void elminarTelefonos(String idDependencia): Elimina los teléfonos de una dependencia dado el identificador de la dependencia.
- List<String> listaTelefonos(String idDependencia): Devuelve el listado de teléfonos de una dependencia dado el identificador de la dependencia.
- boolean existeDependencia(Dependencia dependencia): Devuelve “true” en caso de que exista la dependencia que se pasa por parámetros, de lo contrario retorna “false”.
- List<Dependencia> reporteDependencias(String tipoDependencia, String estado, String municipio): Devuelve el listado de las dependencias que cumplen con los criterios de búsqueda pasados por parámetros.

Observaciones:

- Hereda de la clase SqlMapClientDaoSupport e implementa la interfaz IDependenciaDAO.

2.7.4. Diagrama de Clases del Dominio

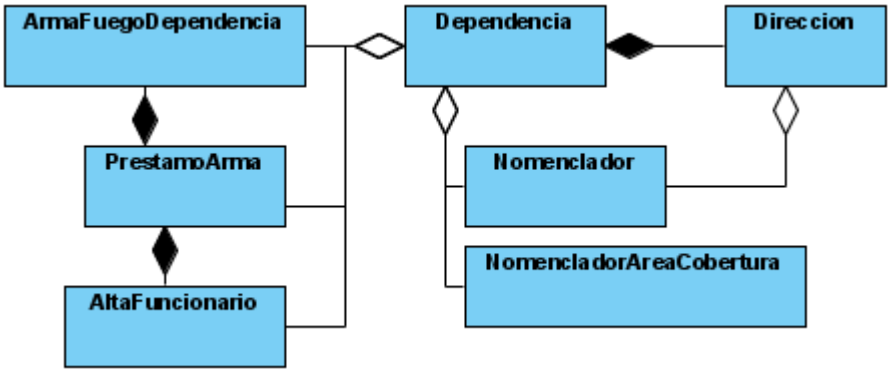

Fig. 2.17. Diagrama de Clases del Dominio del Paquete Gestionar Dependencia.

Clase Dependencia

Propósito:

Clase de dominio propuesta para almacenar y propagar verticalmente los datos asociados a una dependencia policial en la aplicación.

Atributos:

- String idDependencia: Almacena el identificador de la dependencia.
- String codigoDependencia: Almacena el código de la dependencia.
- String nombreDependencia: Almacena el nombre de la dependencia.
- Dependencia dependenciaSuperior: Almacena la información de la dependencia superior.
- String descripción: Almacena la descripción de la dependencia.
- String email: Almacena la dirección de correo electrónico de la dependencia.
- String fax: Almacena el fax de la dependencia.
- List<String> telefonos: Almacena el listado de teléfonos de la dependencia.
- Nomenclador tipoDependencia: Almacena el tipo dependencia.
- Nomenclador organismoSeguridad: Almacena el tipo de organismo de seguridad.
- Direccion dirección: Almacena la dirección de la dependencia.
- int activado: Almacena el estado de la dependencia.
- List<NomencladorAreaCobertura> acEstados: Almacena los estados que forman parte del área de cobertura de la dependencia.
- List<NomencladorAreaCobertura>acMunicipios: Almacena los municipios que forman parte del área de cobertura de la dependencia.
- List<Nomenclador> acParroquias: Almacena las parroquias que forman parte del área de cobertura de la dependencia.
- List<Nomenclador> acLocalidades: Almacena las localidades de la dependencia.
- String idAreaCobertura: Almacena el identificador del área de cobertura de la dependencia.
- ArmaFuegoDependencia arma: Almacena la información del arma de fuego de la dependencia.
- PrestamoArma préstamo: Almacena la información del préstamo del arma de fuego de la dependencia.
- AltaFuncionario alta: Almacena la información del alta del funcionario de la dependencia.

Métodos:

- JSONObject createJSONObject(): Crea la representación en JSON de un objeto de tipo Dependencia.

Observaciones:

No Aplica.

2.8. Paquete Común

Este paquete recoge el conjunto de clases que son utilizadas por todos los módulos que conforman SIGEPOL. Entre estas clases tenemos las que gestionan las direcciones, los teléfonos y los nomencladores.

2.8.1 Diagrama de Clases de la Capa Presentación

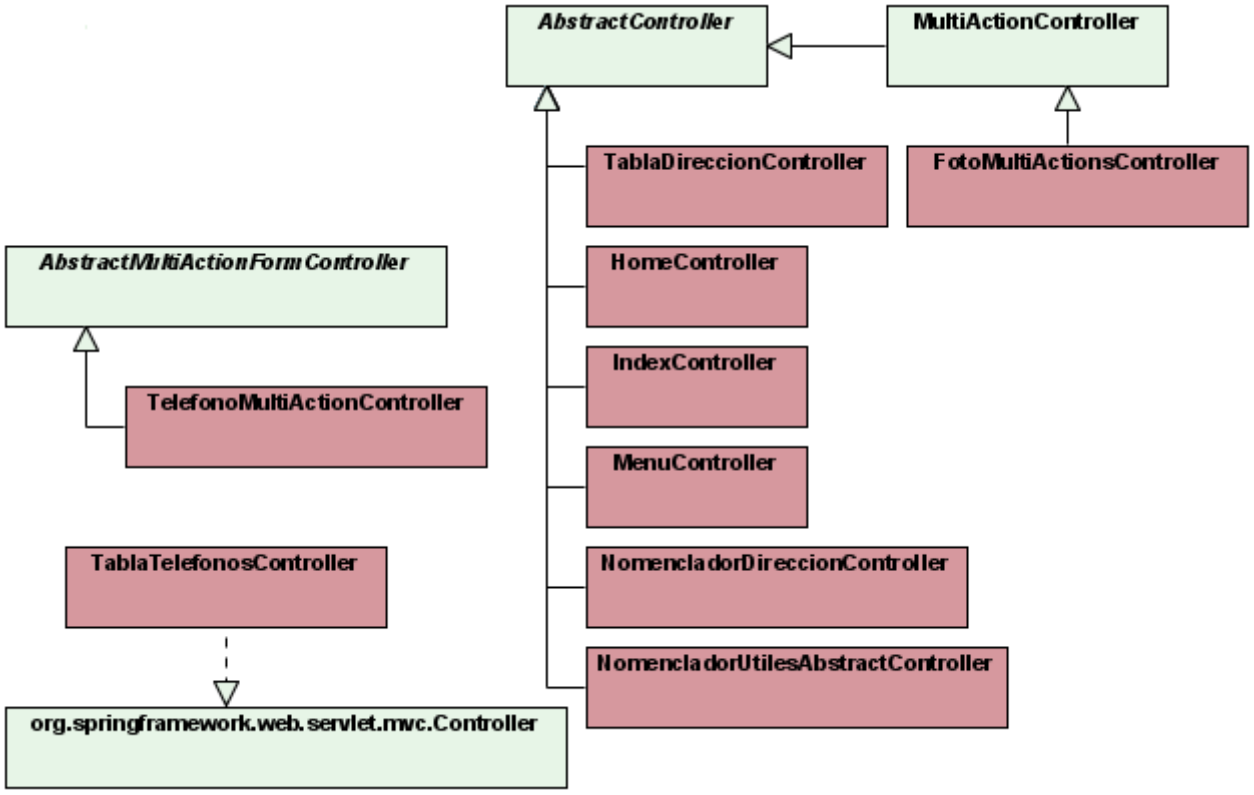

Fig. 2.18. Diagrama de Clases de la Capa Presentación del Paquete Común.

2.8.2. Diagrama de Clases de la Capa Negocio

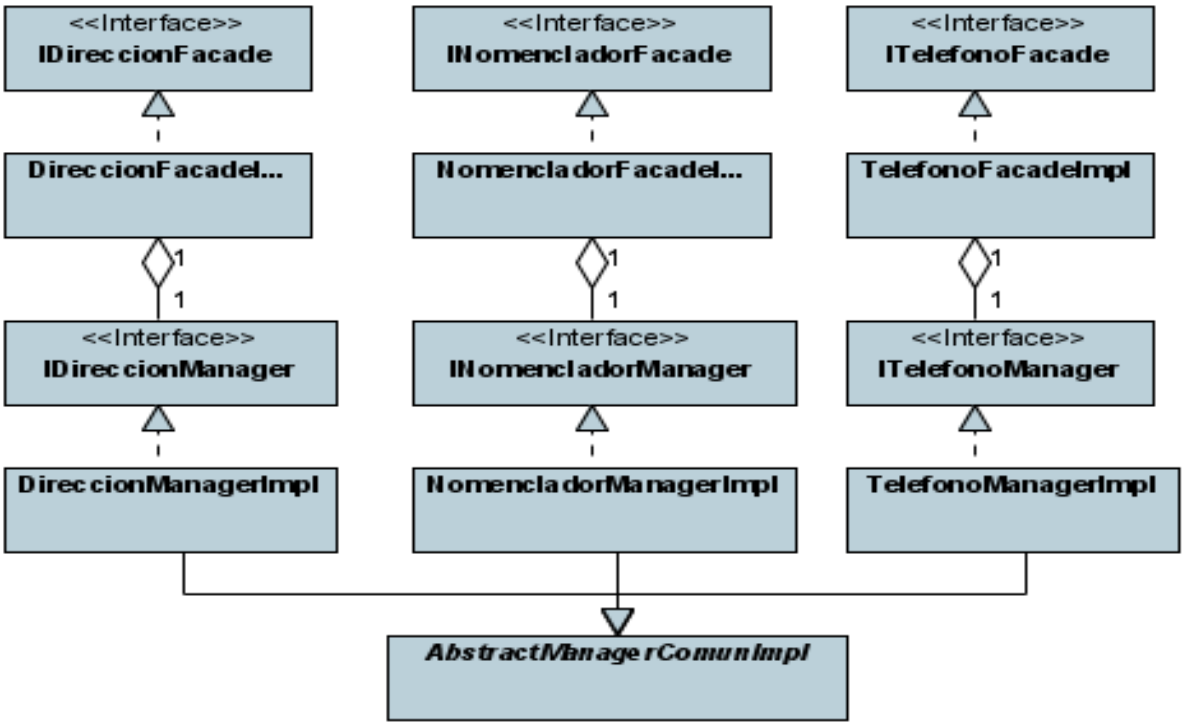

Fig. 2.19. Diagrama de Clases de la Capa Negocio del Paquete Común.

Clase DireccionManagerImpl

Propósito

Implementa la interfaz IDireccionManager que define las posibles acciones sobre la dirección a nivel de lógica de negocio.

Atributos

- private List<Direccion> listaDirecciones**:** Almacena el listado de direcciones.

Métodos

- String insertarDireccion(Direccion direccion): Invoca al procedimiento almacenado que inserta la dirección que se pasa por parámetros en la base de datos.
- void modificarDireccion(Direccion direccion): Modifica la dirección que se pasa por parámetro.
- void eliminarDireccionMemoria(String idDireccion): Elimina del listado de direcciones que está en memoria la dirección que se corresponde con el identificador de la dirección que se pasa por parámetros.

- Direccion obtenerDireccionMemoria(String idDireccion): Retorna del listado de direcciones que está en memoria la dirección que se corresponde con el identificador que se pasa por parámetros.
- List<Direccion> obtenerListaDireccionesMemoria(): Retorna el listado de direcciones que se tiene cargado en memoria.
- void insertarDireccionMemoria(Direccion direccion): Inserta la dirección que se pasa por parámetros.
- void asignarListaDireccionesMemoria(List<Direccion> direcciones): Asigna al listado de direcciones que se tiene por atributo, el que se pasa por parámetros.
- void limpiarListaDireccionesMemoria(): Elimina todos los elementos de la lista de direcciones.

Observaciones:

- Hereda de la clase abstracta AbstractManagerComunImpl e implementa la interfaz IDireccionManager.

2.8.3 Diagrama de Clases de la Capa Acceso a Datos

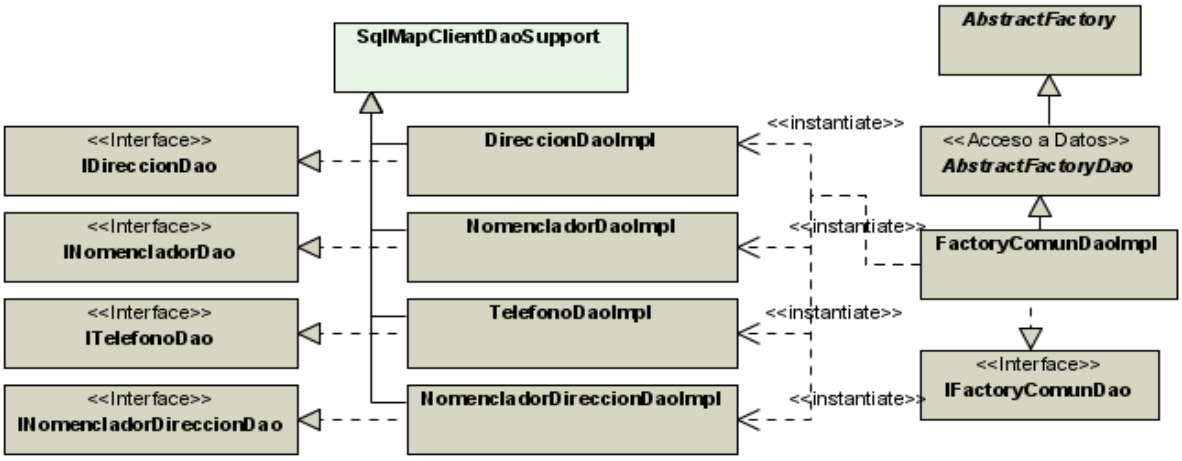

Fig. 2.20. Diagrama de Clases de la Capa Acceso a Datos del Paquete Común.

2.8.4. Diagrama de Clases del Dominio

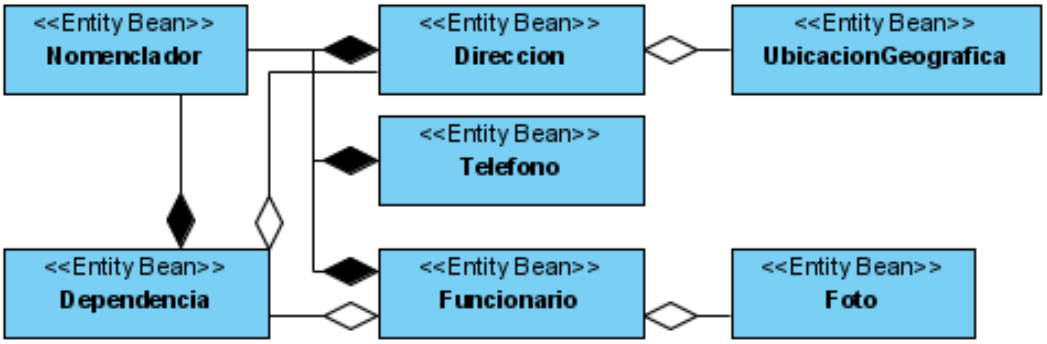

Fig. 2.21. Diagrama de Clases del Dominio del Paquete Común.

Clase Funcionario

Propósito:

Clase de dominio propuesta para almacenar y propagar verticalmente los datos asociados a un agente de seguridad ciudadana que pertenece a una dependencia.

Atributos:

- String idFuncionario: Identificador del funcionario.
- Date fechaNacimiento: Fecha de nacimiento del funcionario.
- String sexo: Sexo del funcionario.
- String primerNombre: Primer nombre del funcionario.
- String segundoNombre: Segundo nombre del funcionario. Puede quedar sin especificar.
- String primerApellido: Primer apellido del funcionario.
- String segundoApellido: Segundo apellido del funcionario. Puede quedar sin especificar.
- String credencial: Número de la credencial del funcionario.
- String cedula: Número de cédula del funcionario.
- Nomenclador jerarquia: Jerarquía del funcionario.
- Date fechaIngreso: Fecha de ingreso a la dependencia policial.
- Dependencia dependencia: Dependencia a la que pertenece el funcionario.
- Nomenclador cargoFuncionario: Cargo que desempeña el funcionario dentro de la dependencia policial.
- Foto foto: Imagen del funcionario.

Métodos:

- JSONObject createJSONObject(): Crea la representación en JSON de un objeto de tipo Funcionario.

Observaciones:

No Aplica**.**

2.9. Conclusiones

En este capítulo se realizó una breve descripción de la arquitectura del sistema SIGEPOL, se presentó el diagrama de paquetes del diseño propuesto, así como los diagramas de clases de cada una de las capas correspondientes a estos. Además se realizó una descripción de las principales clases del diseño.

Capítulo 3 Implementación de la solución

3.1. Introducción

En el presente capítulo se representan en términos de componentes los elementos del modelo del diseño definidos en el capítulo anterior. A continuación se modela un diagrama de componentes, con los componentes principales que constituyen la aplicación del Módulo de Dependencia Policial. Se realiza una breve descripción de cada componente, especificando su propósito y contenido.

3.2. Modelo de Implementación

Los componentes son la parte física y reemplazable de un sistema que está compuesto por un conjunto de interfaces y proporciona la realización de dicho conjunto. Se usan para modelar los elementos físicos que pueden hallarse en un nodo por lo que empaquetan elementos como clases, colaboraciones e interfaces. Son independientes entre ellos y tienen su propia estructura e implementación. Tienen relaciones de traza con los elementos del modelo que implementan.

3.2.1. Diagrama de Componentes.

Los diagramas de componentes se utilizan para modelar la vista estática de un sistema. Muestran la organización y las dependencias lógicas entre un conjunto de componentes software, sean éstos componentes de código fuente, librerías, binarios o ejecutables. El uso más importante de estos diagramas es mostrar la estructura de alto nivel del modelo de implementación.

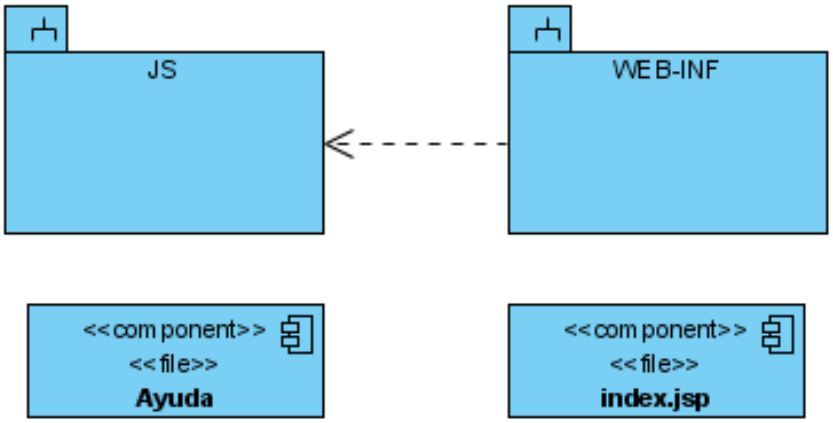

Fig. 3.1 Diagrama de Componentes.

3.2.2. Descripción de los Componentes.

a) <u>Subsistema JS</u>

Propósito

Contenedor físico de los ficheros que contienen código JavaScript.

Contenido

- VerDetallesFuncionario.js
- TrasladarFuncionario.js
- TabsRetirarArmaDeFuego.js
- TablaFuncionario.js
- Resultado.js
- ReporteMeritos.js
- ReporteFuncionariosIngresados.js
- ReporteExpedientes.js
- ReporteDependencias.js
- ReporteArmasEntregadas.js
- ReporteArmasAsignadas.js
- ReporteArmas.js
- RegistrarSancion.js
- RegistrarRetiroArma.js
- RegistrarMerito.js
- RegistrarFuncionario.js
- RegistrarEntregaListarArmas.js
- RegistrarEntregaFinal.js
- RegistrarDevolucionInicial.js
- RegistrarDevolucionFinal.js
- RegistrarArmaFuegoDependencia.js
- MostrarMerito.js
- MostrarHojaDeVida.js
- MostrarHistorialDependencia.js
- MostrarFuncionariosDependencia.js
- MostrarExpediente.js

- MostrarArmas.js
- MostrarArmaDeFuego.js
- ModificarMerito.js
- ModificarExpediente.js
- ModificarEstadoFuncionario.js
- ModificarEstadoArmaDependencia.js
- ModificarArmaDeFuego.js
- InsertarPlanilla.js
- VistaPreviaAsignacionArmamento.js
- HojaDeVida.js
- GestionarDependencia.js
- DetallesExpediente.js
- DetallesArmaFuegoPersonal.js
- DetallesArmaDeFuego.js
- DetalleFuncionario.js
- DialogoEstadoArmas.js
- DialogoArmas.js
- CambiarJerarquia.js
- BuscarFuncionarioRetirarArma.js
- BuscarFuncionario.js
- BuscarDependencia.js
- BuscarArmaRetirarArma.js
- AsignarArmaDeFuego.js
- ArbolDependencias.js
- AbrirExpediente.js
- Core.js
- Check.js
- Index.js
- Page.js
- Validator.js
- Directions.js

- Undo.js
- Útil.js
- ObtenerFuncionario.js
- ObtenerFuncionarioArma.js
- DialogoFuncionarios.js
- DialogoDirecciones.js
- DialogoTelefonos.js

b) Componente Ayuda

Propósito

Contenedor físico de los ficheros que conforman la ayuda de la aplicación.

c) Componente index.jsp

Propósito

Constituye el punto de entrada a la aplicación.

d) Subsistema WEB-INF

Propósito

Contenedor físico de los ficheros que conforman el núcleo de la aplicación.

Contenido

- Subsistema JSP.
- Subsistema Classes.
- sigepol-common.jar
- seguridad.jar
- web.xml

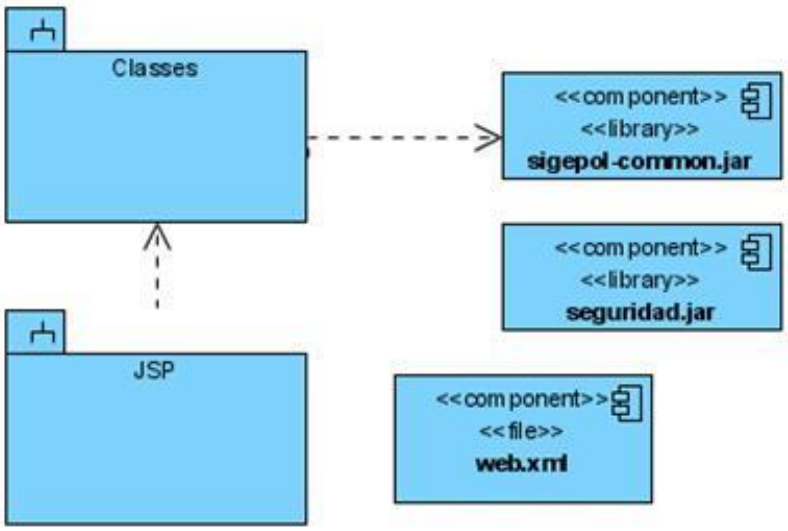

Fig. 3.2. Diagrama de Componentes del Subsistema WEB-INF**.**

d.1) <u>Subsistema JSP</u>

Propósito

Contenedor físico de las interfaces de usuarios, archivos con extensión JSP.

Contenido

- verDetalleFuncionario.jsp
- trasladarFuncionario.jsp
- tabsRetirarArmaDeFuego.jsp
- reporteMeritos.jsp
- reporteFuncionariosIngresados.jsp
- reporteExpediente.jsp
- reporteDependencias.jsp
- reporteArmasEntregadas.jsp
- reporteArmasAsignadas.jsp
- reporteArmas.jsp
- registrarSancion.jsp
- registrarRetiroArma.jsp
- registrarMerito.jsp
- registrarFuncionario.jsp
- registrarEntregaListarArmas.jsp

- registrarEntregaFinal.jsp
- registrarDevolucionInicial.jsp
- registrarDevolucionFinal.jsp
- registrarDependencia.jsp
- registrarArmaFuegoDependencia.jsp
- mostrarMerito.jsp
- mostrarHojaDeVida.jsp
- mostrarFuncionariosDependencia.jsp
- mostrarExpediente.jsp
- mostrarDependencia.jsp
- mostrarArmas.jsp
- mostrarArmaDeFuego.jsp
- modificarMerito.jsp
- modificarExpediente.jsp
- modificarEstadoFuncionario.jsp
- modificarEstadoArmaDependencia.jsp
- modificarArmaDeFuego.jsp
- insertarPlanilla.jsp
- vistaPreviaAsignacionArmamento.jsp
- hojaDeVida.jsp
- historialDependenciasFuncionario.jsp
- funcionarioArma.jsp
- detallesExpediente.jsp
- detallesDependencia.jsp
- detallesArmaFuegoPersonal.jsp
- detallesArmaDeFuego.jsp
- detalleFuncionario.jsp
- tablaFuncionario.jsp
- tablaArmaFuego.jsp
- resultado.jsp
- dialogoEstadoArmas.jsp

- dialogoArmas.jsp
- datosTabla.jsp
- arbolDependencias.jsp
- cambiarJerarquia.jsp
- buscarFuncionarioRetirarArma.jsp
- buscarFuncionario.jsp
- buscarArmaRetirarArma.jsp
- asignarArmaDeFuego.jsp
- abrirExpediente.jsp
- include.jsp
- taglibs.jsp
- dialogoDirecciones.jsp
- dialogoFuncionarios.jsp
- dialogoTelefonos.jsp
- tabla.jsp
- tablaDirecciones2.jsp
- tablaFuncionarios.jsp
- tablaDirecciones1.jsp
- tablaTelefonos.jsp
- nomencladorDireccion.jsp
- userDisable.jsp
- logoffMessage.jsp
- logoff.jsp
- loginErrors.jsp
- locationError.jsp
- index.jsp
- home.jsp
- expiredSession.jsp
- dataMenu.jsp
- changePassword.jsp
- accessDenied.jsp

d.2) <u>Subsistema Classes</u>

Propósito

Contenedor físico del compilado de cada una de las clases de la aplicación, archivos con extensión CLASS, JASPER y XML.

Contenido

- BuscarArmaARetirarMultiActionController.class
- BuscarArmasAsigFuncionarioAbstractController.class
- BuscarArmasAsignadasMultiActionController.class
- BuscarArmasAsignarArmasAbstractController.class
  BuscarArmasEnResguardoAbstractController.class
- BuscarArmasFuegoMultiActionsController.class
- DetalleArmaFuegoDependenciaAbstractController.class
- DetalleArmaFuegoPersonalAbstractController.class
- GestionarArmaFuegoPersonalFrmController.class
- ModificarArmaDependenciaFormController.class
- ModificarEstadoArmaDependenciaFormController.class
- MostrarTablaArmaPersonalController.class
- RegistrarArmaFuegoDependenciaFormController.class
- TablaArmaAsignadasController.class
- TablaArmaDependenciaController.class
- TablaArmaPersonalController.class
- TablaEstadoArmaController.class
- AbstractDependenciaFormController.class
- ArbolDependenciaController.class
- BuscarDependenciaMultiActionController.class
- DetallesDependenciaAbstractController.class
- InsertarDependenciaController.class
- ModificarDependenciaController.class
- AbrirExpedienteController.class
- DetalleExpedienteAbstractController.class

- ModificarExpedienteFormController.class
- ModificarSancionFormController.class
- MostrarExpedienteController.class
- RegistrarSancionFormController.class
- SancionAbstracFormController.class
- TablaExpedienteController.class
- TablaSancionController.class
- HojaVidaFuncionarioMultiActionsController.class
- AbstractFuncionarioFormController.class
- BuscarFuncionarioARetirarArmaAbstractController.class
- BuscarFuncionarioARetirarArmaFormController.class
- BuscarFuncionarioAsignarArmasFormController class
- BuscarFuncionarioDependenciaFormController.class
- BuscarFuncionarioDevolverArmaFormController.class
- BuscarFuncionarioEntregarArmaAbstractController.class
- BuscarFuncionarioMultiActionsController. class
- BuscarFuncionariosAsignarArmasAbstractController.class
- CambiarEstadoArmaPersonalFrmController.class
- CargaNomencladorMotivoAbstractController.class
- FotoMultiActionController.class
- HojaVidaFuncionarioDependenciaAbstractController.class
- InsertarFuncionarioController.class
- ModificarArmaFuegoPersonalFrmController.class
- ModificarEstadoFuncionarioFrmController.class
- ModificarFuncionarioController.class
- MostrarHistorialDependenciasFuncionarioAbstractController.class
- PaginarHojaVidaMultiactionController.class
- TablaDireccionesFuncionario.class
- TablaEstadoFuncionarioController.class
- VerDetallesFuncionarioAbstractController.class
- CambiarJerarquiaController.class

- TablaJerarquiaController.class
- ModificarMeritoController.class
- MostrarMeritoController.class
- RegistrarMeritoController.class
- TablaMeritoController.class
- InsertarPlanillaFuncionarioFormController.class
- AsignarArmaFormController.class
- DevolucionArmaFormController.class
- EntregaArmaFormController.class
- RetirarArmaFormController.class
- VistaPreviaAsignarArmaAbstractController.class
- ReporteArmasAsignadasSimpleFormController.class
- ReporteArmasEntregadasSimpleFormController.class
- ReporteArmasSimpleFormController.class
- ReporteDependencias.class
- ReporteExpedientesSimpleFormController.class
- ReporteFuncionariosIngresadosSimpleFormController.class
- ReporteMeritoSimpleFormController.class
- ModificarArmaFuegoDependenciaValidator.class
- ModificarEstadoArmaDependenciaValidator.class
- RegistrarArmaFuegoDependenciaValidator.class
- AbstractDependenciaValidator.class
- InsertarDependenciaValidator.class
- ModificarDependenciaValidator.class
- AbrirExpedienteValidator.class
- AbstractSancionValidator.class
- ModificarExpedienteValidator.class
- ModificarSancionValidator.class
- RegistrarSancionValidator.class
- AbstractFuncionarioValidator.class
- InsertarFuncionarioValidator.class

- ModificarArmaPersonalValidator.class
- ModificarEstadoArmaPersonalValidator.class
- ModificarEstadoFuncionarioValidator.class
- ModificarFuncionarioValidator.class
- CambiarJerarquiaValidator.class
- ModificarMeritoValidator.class
- RegistrarMeritoValidator.class
- InsertarPlanillaFuncionarioValidator.class
- AsignarArmaValidator.class
- DevolverArmaValidator.class
- EntregarArmaValidator.class
- RetirarArmaValidator.class
- FuncionarioPlanillaLaboral.class
- ArmaDeFuegoFacadeImpl.class
- DependenciaFacadeImpl.class
- ExpedienteFacadeImpl.class
- FuncionarioFacadeImpl.class
- FuncionarioServiceFacadeImpl.class
- JerarquiaFacadeImpl.class
- MeritoFacadeImpl.class
- PlanillaLaboralFacadeImpl.class
- PrestamoArmaFacadeImpl.class
- IArmaDeFuegoFacade.class
- DependenciaFacade.class
- IExpedienteFacade.class
- IFuncionarioFacade. class
- IFuncionarioServiceFacade.class
- IJerarquiaFacade.class
- IMeritoFacade.class
- IPlanillaLaboralFacade.class
- IPrestamoArmaFacade.class

- AbstractDependenciaManagerImpl.class
- ArmaDeFuegoManagerImpl.class
- DependenciaManagerImpl.class
- ExpedienteManagerImpl.class
- FuncionarioManagerImpl.class
- JerarquiaManagerImpl.class
- MeritoManagerImpl.class
- PlanillaLaboralManagerImpl.class
- PrestamoArmaManagerImpl.class
- IArmaDeFuegoManager.class
- IDependenciaManager.class
- IExpedienteManager.class
- IFuncionarioManager.class
- IJerarquiaManager.class
- IMeritoManager.class
- IPlanillaLaboralManager.class
- IPrestamoArmaManager.class
- ArmaFuegoDAOImpl.class
- DependenciaDAOImpl.class
- ExpedienteDAOImpl.class
- FuncionarioDAOImpl.class
- JerarquiaDAOImpl.class
- MeritoDAOImpl.class
- PlanillaLaboralDAOImpl.class
- PrestamoArmaDAOImpl.class
- IArmaFuegoDAO.class
- IDependenciaDAO.class
- IExpedienteDAO.class
- IFuncionarioDAO.class
- IJerarquiaDAO.class
- IMeritoDAO.class

- IPlanillaLaboralDAO.class
- IPrestamoArmaDAO.class
- FactoryDependenciaDAOImpl.class
- IFactoryDependenciaDAO.class
- AreaCobertura.class
- Arma.class
- FuncionarioResult.class
- FuncionarioSIGEPOL.class
- UbicacionPolicial.class
- UbicacionPolicialDaoResult.class
- AccionDenuncia.class
- AccionesDelFuncionario.class
- AccionOperativo.class
- AccionResenna.class
- AltaFuncionario.class
- ArmaFuego.class
- ArmaFuegoDependencia.class
- ArmaFuegoPersonal.class
- Dependencia.class
- Expediente.class
- FuncionarioDependencia.class
- HistorialEstadoArma.class
- HistorialEstadoFuncionario.class
- HistorialJerarquia.class
- Merito. class
- PlanillaLaboral.class
- PrestamoArma.class
- Sancion.class
- FuncionarioServiceEndPoint.class
- Funcionario.class
- Direccion.class

- DireccionManagerImpl.class
- NomencladorFacadeImpl.class
- DireccionMultiActionsController.class
- FotoMultiActionsController.class
- TablaDireccionController.class
- TelefonoMultiactionController.class
- TablaTelefonosCelular.class
- NomencladorUtilesAbstraController.class
- NomencladorDireccionController.class
- MenuController.class
- IndexController.class
- HomeCOntroller.class
- FotoMultiactionsController.class
- IDireccionFacade.class
- DireccionFacadeImpl.class
- DireccionManager.class
- INomencladorFacade.class
- NomencladorManager.class
- NomencladorManagerImpl.class
- ITelefonoFacade.class
- ITelefonoManager.class
- TelefonoManagerImpl.class
- IDireccionDao.class
- IFuncionarioDao.class
- ITelefonoDao.class
- INomencladorDao.class
- INomencladorDireccionDao.class
- DireccionDaoImpl.class
- FuncionarioDaoImpl.class
- NomencladorDaoImpl.class
- NomencladorDireccionDaoImpl.class

- Nomenclador.class
- Dependencia.class
- Telefono.class
- Foto.class
- UbicacionGeografica.class
- TelefonoDaoImpl.class
- ReporteArmas.jasper.
- ReporteArmasAsignadas.jasper
- ReporteArmasEntregadas.jasper
- ReporteDependencias.jasper
- ReporteExpediente.jasper
- ReporteFuncionariosIngresados.jasper
- ReporteMeritos.jasper
- sqlMapConfig.xml
- sigepol-dependencia-ws-context.xml
- sigepol-dependencia-servlet.xml
- sigepol-dependencia-security-context.xml
- sigepol-dependencia-report-context.xml
- sigepol-dependencia-menu-context.xml
- sigepol-dependencia-dataaccess-context.xml
- sigepol-dependencia-context.xml
- WebServiceMap.xml
- PrestamoArma.xml
- PlanillaLaboral.xml
- Nomenclador-Funcionario.xml
- Nomencladores.xml
- Nomenclador-Dependencia.xml
- Merito.xml
- Jerarquia.xml
- Funcionario.xml
- Expediente.xml

- Dependencia.xml
- ArmaFuego.xml

d.3) sigepol-common.jar

Propósito

Contenedor físico de clases que implementan las funcionalidades comunes a todos los módulos que conforman SIGEPOL.

d.4) seguridad.jar

Propósito

Contenedor físico de las nuevas clases que modifican la seguridad original del framework Acegi, además de poseer nuevas funcionalidades que garantizan la comunicación entre aplicaciones.

d.5) web.xml

Propósito

Descriptor de despliegue para la aplicación Web.

3.3. Conclusiones

Los diferentes diagramas de componentes confeccionados en este capítulo, representan en términos de componentes y subsistemas de implementación, los elementos del modelo de diseño obtenidos en el capítulo anterior y establecen las dependencias entre uno y otro.

Conclusiones

El presente trabajo finaliza dando cumplimiento al objetivo general trazado de desarrollar una aplicación informática para los procesos de gestionar de funcionarios y control del armamento perteneciente a las dependencias policiales como parte del Sistema de Gestión Policial para la República Bolivariana de Venezuela.

Se efectuó un estudio de las herramientas y tecnologías de desarrollo que se utilizaron en la confección de la solución, además de los procesos de gestión de los funcionarios y el armamento de las dependencias. Se definió un diseño que modela el sistema y brinda soporte a todos los requisitos funcionales y no funcionales. Un aspecto de gran utilidad en la confección y concepción del diseño fue la comprensión y utilización de patrones, que permitieron aplicar buenas prácticas y brindar soluciones probadas a problemas comunes.

Se estructuró un diagrama de componentes que representa la implementación del sistema en términos de componentes, quedando construida una aplicación capaz de garantizar rapidez y efectividad en los procesos de gestión de funcionarios y registro de armamento en las dependencias policiales.

Recomendaciones

Realizar las pruebas a la versión operacional obtenida del Módulo de Dependencia Policial, que permitan validar y evaluar la calidad del producto desarrollado.

Incorporar nuevos reportes relacionados con el proceso de la gestión de funcionarios y el control del armamento en las dependencias policiales de manera que el análisis de los mismos contribuya a perfeccionar la toma de decisiones para ayudar a combatir y prevenir el delito en la hermana República Bolivariana de Venezuela.

Referencias Bibliográficas

1. **Seguridad Ciudadana, Venezuela.** [Online] [Cited: Diciembre 12, 2007.] Disponible en: http://www.seguridadidl.org.pe/sistema/leysinasec.pdf.

2. **Monografías.com.** [Online] Monografias.com S.A. . [Cited: Febrero 4, 2007.] Disponible en: http://www.monografias.com/trabajos28/seguridad-ciudadana/seguridad-ciudadana.shtml.

3. **MIJ.** *Reportajes.htm.* 2006. nº.

4. **GABALDÓN, LUIS GERARDO**. *Cfr.* p. 116. 1.

5. **ESCAMILLO, J. C.** Ministerio del Poder Popular para Relaciones de Interiores y Justicia. [Online] [Cited: Enero 7, 2008.] Disponible en: http://www.mpprij.gob.ve/spip.php?article=2071.

6. **Wikipedia.** [Online] [Cited: Diciembre 5, 2007.]. Disponible en: http://es.wikipedia.org/wiki/Software.

7. **DNV.** [Online] [Cited: Diciembre 8, 2007.]. Disponible en: http://www.dnv.es/certificacion/sistemasdegestion/index.asp.

8. **Bustelo C, Amarilla R.** *Gestión del Conocimiento y Gestión de la Información.* 2001. VIII(34): 226-230 .

9**.** Habituales TIPOS de SOFTWARE DE GESTIÓN. Disponible en **:**http://www.alimarket.es/html2/rev/a017/c02.sof.pdf

10. **Wikipedia.** ***Aplicaciones Web.*** [Online] [Cited: Diciembre 8, 2007.] Disponible en: http://es.wikipedia.org/wiki/Aplicaci%C3%B3n_web.

11. **Sanchez, María A. Mendoza**. Informatizate. [Online] Junio 7, 2004. [Cited: Enero 9, 2008.] Disponible en: http://www.informatizate.net/articulos/metodologias_de_desarrollo_de_software_07062004.html.

12. **JACOBSON, I. B., GRADY Y RUMBAUGH, JAMES**. *El Proceso Unificado de Desarrollo de Software.* La Habana : FÉLIX VARELA, 2004.

13. **Wikipedia.** ***Plataforma Java.*** [Online] [Cited: Febrero 5, 2008.] Disponible en: http://es.wikipedia.org/wiki/Plataforma_Java.

14. **FSF.** ***Fundación Software Libre.*** [Online] [Cited: Enero 8, 2008.] Disponible en: http://www.fsfla.org/svnwiki/legis/venezuela/3390.

15. **Allamaraju, Subrahmanyam, et al.** *Programación Java Server con J2EE.* 1.3. 1206.

16. **Wikipedia.** ***Herramienta CASE.*** [Online] [Cited: Febrero 4, 2008.] Disponible en: http://es.wikipedia.org/wiki/Herramienta_CASE.

17. **Vizcaíno, Aurora, García, Felix Oscar and Caballero, Ismael.** *Una Herramienta CASE para ADOO: Visual Paradigm.* [Documento] s.l. : UNIVERSIDAD DE CASTILLA-LA MANCHA.

18. **Visual Paradigm.** [Online] [Cited: Enero 23, 2008.] Disponible en: http://www.visual-paradigm.com/product/vpuml.

19. **Burbeck, Steve.** *Application Programming in Smalltalk-80: How to use Model-View-Controller.*

20. **Larman, Craig.** Introducción al análisis y diseño orientado a objetos. *UML y Patrones. Tomo I.*

21. **WM.** ***elwebmaster.com.*** [Online] [Cited: Enero 25, 2008.] Disponible en: http://www.elwebmaster.com/articulos/top-5-javascript-framework.

22. **C. Walls, R. Breidenbach**. *Spring in Action.* s.l. : Manning Publications, 2005.

23. **Clinton Begin, Brandon Goodin, Larry Meadors.** *iBatis in Action.* 2007. Manning Publications.

24. **Gutierrez, Juan.** Universidad de Valencia. *Eclipse (2.1) y Java.* [Online] 2004. [Cited: Febrero 10, 2008.] Disponible en: http://www.uv.es/~jgutierr/MySQL_Java/TutorialEclipse.pdf.

25. **García Puebla, Ivón.** Gestor de Contenidos. *Eclipse: una herramienta profesional al alcance de todos.* [Online] Agosto 2, 2005. [Cited: Febrero 11, 2008.] Disponible en: http://www.gui.uva.es/~laertes/nuke/index.php?option=com_content&task=view&id=56&Itemid=41.

Bibliografía

1. **Almaer, Dion**. *"The Dojo Toolkit in Practice".* 2006

2. **Becerril C., Francisco.** *Java a su alcance.* s.l. : McGRAW-HILL INTERAMERICANA EDITORES.

3. **C. Walls, R. Breidenbach**. *"Spring in Action", Manning Publications, 2005.*

4. **Catalogo de Antipatrones.** [En línea] [Citado el: 18 de 02 de 2008.] Disponible en: http://c2.com/cgi/wiki?AntiPatternsCatalog.

5. **Clinton Begin, Brandon Goodin, Larry Meadors.** *"iBATIS in Action", Manning Publications*.

6. **Constitución de la República Bolivariana de Venezuela.** [En línea] [Citado el: 16 de Diciembre de 2007.] Disponible en: http://www.venezuela-oas.org/Constitucion%20de%20Venezuela.htm.

7. **Craig Larman**. *UML y Patrones*, Introducción al análisis y diseño Orientado a Objetos.

8. **Darren Davison, Steven Devijver, Colin Yates**. *Expert Spring MVC and Web Flow*.

9. **DeMichiel**, L. *"Enterprise JavaBeans Specification, Version 2.1", Sun Microsystems, November 12, 2003*. Disponible en: http://java.sun.com/products/ejb/docs.html

10. **DEVX.** ***An Introduction to Antipatterns in Java Applications.*** [En línea] [Citado el: 02 de 03 de 2008.] Disponible en: http://www.devx.com/Java/Article/29162/1954.

11. **Gamma, Erich, y otros.** *Design Patterns*. Elements of Reusable Object Oriented Software, Adison-Wesley Longman,1995.

12. **Heilmann, Christian**. *"Beginning JavaScript with DOM Scripting and Ajax".* 2006

13. **Jacobson, I. and Booch, G. y Rumbaugh, J.** *"El Proceso Unificado de Desarrollo de software". 2000.* 2000.

14. **Java Anti-Patterns.** [En línea] [Citado el: 03 de 03 de 2008.] Disponible en: http://www.odi.ch/prog/design/newbies.php.

15. **Larman, Craig.** *Uml y Patrones.Introducción al análisis y diseño orientado a objetos*. [En línea] México, 1999. [Citado el: 24 de Enero de 2008.] Disponible en: http://bibliodoc.uci.cu/pdf/reg00061.pdf ISBN 970-17-0261-1

16. **VENEZUELA, G. D.** LEY DE COORDINACION DE SEGURIDAD CIUDADANA. 2001, nº Disponible en: http://www.mij.gov.ve/ley_coord_sc.htm.

17. **Visual Paradigm**. *Visual Paradigm for UML Model –Code-Deploy Platform*. [En línea][Citado el: 9 de Enero de 2008.] Disponible en: http://www.visual-paradigm.com/product/vpuml/ .

18. **Wikipedia.** ***La enciclopedia libre.*** [En línea] [Citado el: 03 de 03 de 2008.] Disponible en: http://es.wikipedia.org/wiki/Antipatr%C3%B3n_de_dise%C3%B1o .