Universidad Nacional Autónoma de México

Facultad de Ingeniería

Arquitectura base de un Sistema Integral de Gestión de Información

I N F O R M E

Que para obtener el título de
Ingeniero en Computación
p r e s e n t a :
MARCO ANTONIO CAMACHO ESTRADA

Aval: Ing. Laura Sandoval Montaño

Ciudad Universitaria. 2010

Índice

Antecedentes..1
Estructura orgánica general de la CDHDF..2
Estructura orgánica de la Dirección General de Administración..3
Estructura orgánica de la Jefatura de Unidad de Tecnologías de Información..4

Definición del problema..5

Marco teórico..6
Análisis y diseño orientado a objetos..6
UML..11
Métodos ágiles..16
Desarrollo Iterativo e incremental..17
Pruebas..22

Participación profesional..23
Inicio..24
Elaboración..28

Resultados y aportaciones..59

Conclusiones..61

Bibliografía..62

Anexo 1. Diagramas de casos de uso..63

Anexo 2. Casos de uso..65

Anexo 3. Especificaciones suplementarias..75

Antecedentes

Desde su creación en septiembre de 1993, la Comisión de Derechos Humanos del Distrito Federal se ha encargado de recibir las quejas y denuncias sobre presuntas violaciones a los derechos humanos cuando éstas fueran imputadas a cualquier autoridad o servidor público que desempeñe un empleo o cargo en la administración pública del Distrito Federal o en los órganos de impartición de justicia. Si se acredita una violación y no se logra una conciliación con la autoridad responsable, la CDHDF puede emitir una Recomendación [CDHDF06-08].

Para llevar a cabo la gestión del proceso desde que se realiza un servicio de orientación verbal o escrito a alguna persona que presenta una queja, y las gestiones a las que se puede derivar (un expediente de queja o una recomendación), se pusieron en marcha varios sistemas de captura, independientes para cada modelo de gestión, desarrollados en Clipper[1]; uno por cada área para poder satisfacer las demandas de cómputo. Recabar más detalles sobre este producto no fue posible, pues no existe ningún tipo de documentación o artefacto disponible.

Con la finalidad de facilitar el trabajo a los empleados y optimizar la calidad del servicio, en septiembre de 2005 se puso en marcha el Sistema Integral de Atención a Peticionarios (SIAP) que ya contaba con la integración de las distintas áreas del Programa de Defensa; era una aplicación web programada en Java pero con carencias como producto[2]. Específicamente, la manera en como estaba diseñado lo hacía muy complicado para modificar y extender, no utilizaba patrones de diseño básicos y en la vista (JSP's) se realizaban tareas de control y lógica del programa y del negocio.

Bajo estas complicaciones técnicas, y con el desarrollo y la madurez de nuevas tecnologías y metodologías disponibles, se hizo necesaria la aparición de un nuevo sistema de cómputo que satisfaciera las demandas actuales y que, a la vez, previera cambios probables y el crecimiento en el futuro.

---

1 Clipper es un lenguaje de programación procedural e imperativo, orientado a manejo de datos. Fue creado en 1985 y la última versión fue lanzada en 1997 [Wiki09a].

2 Otro de sus objetivos era mantener la integridad de la información, sin embargo, se llegaron a detectar algunas inconsistencias en los datos que hacían muy complicada la extracción de datos para los informes de la CDHDF, las cuales, nunca fueron documentadas y por ello no se tiene registro oficial de ellas.

Estructura orgánica general de la CDHDF

La CDHDF está integrada por el Presidente, el Consejo, los visitadores generales y el personal profesional, técnico y administrativo [CDHDF06-08] organizados en distintos órganos y áreas de apoyo, cuya estructura se muestra a continuación:

CONSEJO
PRESIDENCIA
CONTRALORÍA INTERNA
SECRETARÍA EJECUTIVA
SECRETARÍA PARTICULAR
DIRECCIÓN GENERAL DE QUEJAS Y ORIENTACIÓN
DIRECCIÓN EJECUTIVA DE SEGUIMIENTO
DIRECCIÓN EJECUTIVA DE INVESTIGACIÓN Y DESARROLLO INSTITUCIONAL
PRIMERA VISITADORÍA GENERAL
SEGUNDA VISITADORÍA GENERAL
TERCERA VISITADORÍA GENERAL
CUARTA VISITADORÍA GENERAL
SECRETARÍA TÉCNICA
DIRECCIÓN GENERAL DE EDUCACIÓN Y PROMOCIÓN DE LOS DD.HH.
PROGRAMA DE DEFENSA
PROGRAMA DE PROMOCIÓN
PROGRAMA DE FORTALECIMIENTO INSTITUCIONAL
DIRECCIÓN GENERAL DE ADMINISTRACIÓN
DIRECCIÓN GENERAL DE COMUNICACIÓN SOCIAL

Figura 1: Estructura orgánica de la CDHDF.

En el Programa de Defensa están involucradas aquellas áreas que se encargan de la atención a peticionarios, la formulación de la queja y su investigación y seguimiento; en el Programa de Promoción están involucradas las áreas que se encargan de la educación, capacitación y difusión de los derechos humanos, así como de la vinculación de la CDHDF con otras instituciones; la Dirección Ejecutiva de Investigación y Desarrollo Institucional, es el área encargada del Programa de Fortalecimiento Institucional, y se encarga de la investigación, formación profesional y planeación estratégica de los programas en derechos humanos.

Estructura orgánica de la Dirección General de Administración

La Dirección General de Administración (DGA) es el área encargada de atender las necesidades administrativas de los diferentes órganos y áreas de apoyo de la Comisión de conformidad a los lineamientos generales, normas, políticas, manuales y procedimientos administrativos aprobados por el Consejo [CDHDF06-08].

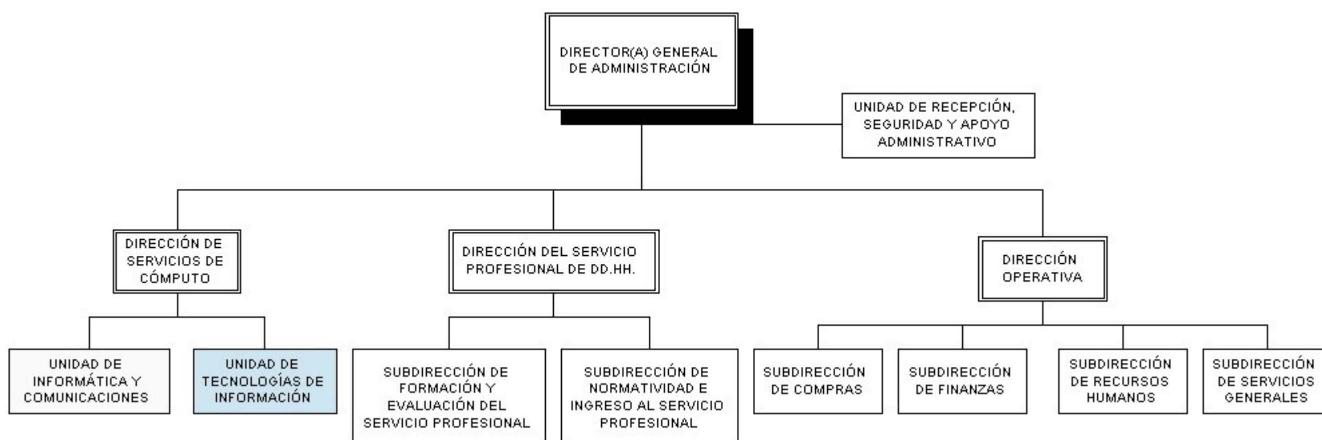


Figura 2: Estructura de la Dirección General de Administración.

Para su funcionamiento, la DGA cuenta con tres direcciones, una de ellas es la Dirección de Servicios de Cómputo, cuya vacante está desocupada por políticas presupuestales. De dicha Dirección, se desprende la Unidad de Tecnologías de Información, que es la Unidad para la cual laboro.

Estructura orgánica de la Jefatura de Unidad de Tecnologías de Información

La estructura de la Jefatura de Unidad de Tecnologías de Información se muestra en la siguiente figura:

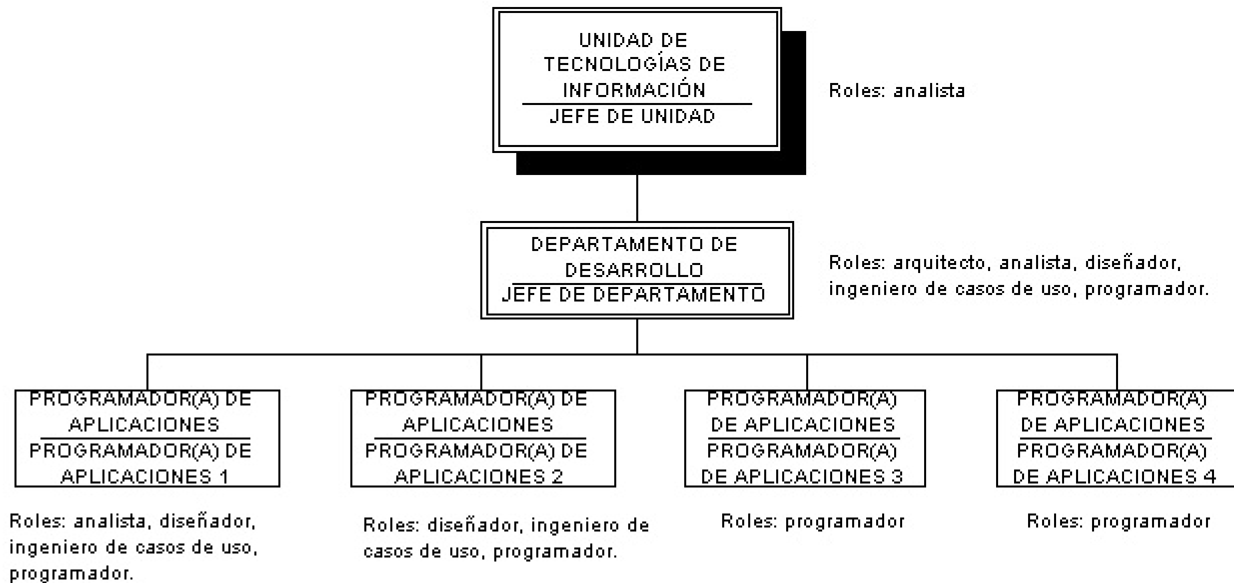


Figura 3: Estructura de la Unidad de Tecnologías de Información.

Según la estructura de la CDHDF, la Unidad está compuesta por el Jefe de Unidad, el Jefe de Departamento de Desarrollo, y cuatro Programadores de Aplicaciones.

Desde marzo de 2009, poseo oficialmente el cargo de Jefe de Departamento de Desarrollo, pues anteriormente, la estructura de la Dirección de servicios de cómputo estaba organizada de manera distinta, en la cual, oficialmente el área de Desarrollo contaba con un sólo trabajador bajo el nombramiento de Programador de aplicaciones, y cuatro trabajadores contratados bajo la figura fiscal de asimilados al salario como Prestadores de servicios profesionales, situación en la que permanecí dos años diez meses. Sin embargo, la estructura real del área era la misma a la de ahora en cuanto a responsabilidades se refiere.

El organigrama de la figura 3 muestra también los roles que cada empleado tomó durante el desarrollo del proyecto. Los roles que tomé fueron los de arquitecto, analista, diseñador, ingeniero de casos de uso y programador.

Definición del problema

El Sistema Integral de Atencion a Peticionarios (SIAP), resolvió en su momento el principal reto para la Comisión en cuanto a tener una organización de la información en gran parte de su Programa de Defensa de los derechos humanos, que no resolvían los antiguos sistemas programados en Clipper. Sin embargo, después de la puesta en marcha del SIAP y con las nuevas necesidades de la Comisión, surgieron nuevos problemas por resolver: la extensión y modificación de algunos módulos del SIAP; la integración al Sistema de cómputo de las demás áreas y la dificultad del SIAP para integrarlas, pues su diseño no permitía una integración sencilla o la adición de módulos; y la carencia de interfaces bien definidas.

Es así como se planteó la necesidad de un nuevo sistema que resolviera eficientemente las demandas de los principales servicios que proporciona la CDHDF, que a saber, se centran en tres programas institucionales, los cuales son: el Programa de Defensa; el Programa de Promoción, Educación y Difusión; y el Programa de Fortalecimiento Institucional; siendo las necesidades del primer programa las que el SIIGESI, en su primera etapa, debió cubrir.

La primera etapa del desarrollo del Sistema Integral de Gestión de Información estuvo enfocada en la Ingeniería de Software para el Programa de Defensa, para lo cual, el problema a resolver se centró en el establecimiento de una arquitectura estable del sistema en las fases de inicio y elaboración (del Proceso Unificado), considerando la posterior integración al sistema de los demás programas de la Institución y tomando en cuenta los resultados del análisis de requerimientos, tanto funcionales como no funcionales, que dieron paso al diseño e implementación.

Marco teórico

Análisis y diseño orientado a objetos

Las habilidades en diseño y análisis orientados a objetos son esenciales para la creación de software bien diseñado, robusto y mantenible usando cualquier lenguaje orientado a objetos como Java. Sin embargo, conocer un lenguaje orientado a objetos es un primer paso necesario pero insuficiente para crear sistemas de objetos. De manera similar, la notación UML es útil, pero no es lo más importante para aprender en análisis OO.

El **análisis** se sefiere a una *investigación* de los requerimientos del problema más que una solución. En tanto, el **análisis orientado a objetos** enfatiza la búsqueda y descripción de los objetos en el dominio del problema.

El **diseño** hace referencia a una solución conceptual (en software y hardware) para satisfacer los requerimientos en vez de una implementación. Las ideas de diseño excluyen con frecuencia detalles obvios o de bajo nivel para los clientes. Durante el **diseño orientado a objetos** se hace énfasis en definir los objetos del software y cómo ellos colaboran para satisfacer los requerimientos.

El modelado es diseñar software antes de codificar, es una parte esencial en los grandes proyectos de software y muy útil para proyectos medianos e incluso para pequeños proyectos. Un modelo es análogo a los planos de un edificio. Al utilizar un modelo, los responsables del éxito de un desarrollo de software pueden asegurarse que la funcionalidad requerida es completa y correcta, y que el diseño del programa es robusto y puede escalarse fácilmente. El modelado es entonces, la única forma de visualizar un diseño y verificarlo con los requerimientos antes de codificar.

Patrones de diseño

Para dominar el diseño orientado a objetos, se requiere del conocimiento de una cantidad considerable de principios flexibles.

Una manera de pensar en el diseño de objetos es en términos de responsabilidades, roles y colaboraciones, los cuales forman parte de un enfonque llamado diseño orientado a las responsabilidades (RDD por sus siglas en inglés).

Las responsabilidades están relacionadas con las obligaciones o el comportamiento de un objeto en términos de su rol. Básicamente, tales responsabilidades son de dos tipos: *hacer* y *conocer*. Las primeras incluyen: crear un objeto o hacer un cálculo, iniciar una acción en otros objetos, controlar y coordinar actividades en otros objetos; mientras que las segundas incluyen: conocer sobre los datos encapsulados privados y los objetos relacionados.

Las colaboraciones se refieren al hecho de que algún método de un objeto necesite colaborar con los métodos de otros.

Una forma útil de aprender a entender el diseño y de aplicar el razonamiento de una manera metódica, racional y explícita, son los patrones GRASP (General Responsibility Assignment Software Patterns). Es un enfoque para entender y usar los principios básicos de diseño basados en patrones de asignación de responsabilidades. Hay nueve patrones GRASP, los cuales listo y explico brevemente a continuación:

Creador

El problema principal del creador es saber quién tiene la responsabilidad de crear un objeto A. La solución es la siguiente:

Asignar la responsabilidad a una clase B para que cree una instancia de la clase A, si alguno de los siguientes enunciados es verdadero:

- B “contiene” A.
- B registra A.
- B usa A.
- B tiene los datos iniciales de A.

En ocasiones, cuando se requiere de cierta complejidad para crear un objeto, tal como reciclar instancias para obtener un mejor rendimiento, creación condicional (basada un valor de una propiedad externa) de instancias de una familia de clases similares, etc., es mejor delegar la creación de los objetos a una clase auxiliar, proveniente de otros patrones de diseño como el *ConcreteFactory* o el *AbstractFactory* en vez de usar el creador.

Experto

El experto es uno de los principios más básicos para asignar responsabilidades en diseño orientado a objetos. La solución al problema de saber cuál es el principio para asignar responsabilidades es asignar una responsabilidad a la clase que tiene la información suficiente para satisfacerla.

Hay ocasiones en la que usar el experto no es una buena elección de diseño, por ejemplo cuando hay responsabilidades que deberían satisfacerse usando capas adicionales al dominio del problema, esto aumenta la cohesión y mantiene un bajo acoplamiento.

Bajo acoplamiento

El acoplamiento es una medida para saber la relación de un elemento con otro. Esto incluye saber: si están conectados, si uno tiene conocimiento del otro o alguno depende del otro. Si hay dependencia, cuando un elemento cambia, puede afectar al otro.

Este patrón consiste entonces en conocer la forma de reducir el impacto del cambio, y la solución es asignar responsabilidades tal que el acoplamiento permanezca bajo. Una clase que tiene alto acoplamiento puede forzar a cambios en las clases relacionadas cuando ésta cambia, es difícil de entender y difícil de reutilizar porque su uso requiere de clases adicionales de los cuales depende.

Las formas comunes de acoplamiento de un TipoX a un TipoY en un lenguaje orientado a objetos incluyen:

- El TipoX tiene un atributo que hace referencia a una instancia TipoY.
- Un TipoX usa un servicio del objeto TipoY.
- TipoX tiene un método que hace referencia a una instancia de TipoY, a menudo, mediante un parámetro o una variable local TipoY, o el objeto TipoY regresado de un mensaje.
- El TipoX es directa o indirectamente una subclase de TipoY
- TipoY es una interfaz, y TipoX la implementa.

El bajo acoplamiento se usa a menudo para evaluar alternativas de diseño para escoger la que tenga menor acoplamiento. Cuando una librería es estable no hay mucho problema cuando se tiene un alto acoplamiento.

Controlador

El problema consiste en conocer cuál es el primer objeto después de la capa de interfaz de usuario que controle las operaciones del sistema.

Hay al menos dos alternativas, asignar la responsabilidad a un objeto que represente alguna de las siguientes opciones:

- que represente el total del "Sistema", un "objecto raíz", un dispositivo en el que el software esté ejecutándose, o un subsistema mayor
- que represente un escenario del caso de uso en el cual ocurra la operación del Sistema.

Alta cohesión

La alta cohesión se refiere a la forma de mantener los objetos para que sean entendibles, manejables y enfocados en las operaciones que les corresponden. Un elemento con varias responsabilidades relacionadas, las cuales no realizan mucho trabajo, tiene una alta cohesión.

La alta cohesión, al igual que el bajo acoplamiento, se usa a menudo para evaluar alternativas en el diseño. Una consecuencia natural de tener una alta cohesión es que el acoplamiento se mantiene bajo.

Una clase que tiene baja cohesión es difícil de comprender, de reutilizar, de mantener y es afectada por el cambio. Nótese que éstos son también problemas derivados de un alto acoplamiento, por lo que la cohesión y el acoplamiento estan fuertemente relacionados.

Polimorfismo

En el diseño de software a menudo se tratan las siguientes interrogantes: ¿cómo tratar una serie de alternativas basadas en el tipo de dato? ¿Cómo crear componentes de software que se puedan conectar con otros?

Si un programa es diseñado usando sentencias *if-then-else*, entonces, si surge una nueva variante, se requiere modificación de la lógica en varias partes del código. Esta solución hace que sea difícil de ampliar un programa con nuevos cambios por que ellos tienden a afectar en varias líneas de código.

La solución a las dos preguntas es la siguiente: cuando hay alternativas o comportamientos relacionados que varían de acuerdo al tipo, asignar una responsabilidad para el comportamiento usando

operaciones polimórficas para los tipos de datos en donde el comportamiento varíe. Por lo tanto, no se debe utilizar lógica condicional para elegir alternativas basadas en el tipo de dato.

Fabricación pura

¿Qué objeto debe tener la responsabilidad cuando no se quiere violar el principio de *bajo acoplamiento* ni de *alta cohesión* y la solución que ofrece el *experto* no es la apropiada?

Los diseños orientados a objetos se caracterizan por implementar clases que representan conceptos en el dominio del problema del mundo real. Sin embargo, hay muchas soluciones en las cuales, asignar responsabilidades conduce a problemas de pobre acoplamiento o pobre cohesión, y, ulteriormente, a un bajo potencial de reuso.

Para subsanar el problema de diseño al que puede llegar a conducir la utilización de clases basadas en objetos puros del dominio del problema, se puede asignar una cantidad de responsabilidades con alta cohesión a una clase *artificial* de tal forma que la cohesión permanezca alta y la cohesión baja. Una clase de este tipo es una clase de *fabricación pura*.

Indirección

La indirección ofrece una solución al problema de cómo desacoplar objetos de tal forma que haya bajo acoplamiento y el potencial de reuso permanezca alto, o a cuál clase asignar una responsabilidad para evitar un acoplamiento directo entre dos objetos.

La solución que aporta la *indirección* es asignar la responsabilidad a un objeto intermedio, de tal forma que los otros objetos no estén directamente acoplados.

Variaciones protegidas

¿Cómo se pueden diseñar objetos para que las variaciones en ellos no tengan un impacto indeseable en otros objetos?

Este patrón establece que se deben identificar puntos de inestabilidad o de variaciones predecibles en los objetos y crear una interfaz que satisfaga las responsabilidades para, con ella, envolver los objetos implicados y protegerse de los cambios en sus implementaciones.

UML

El Lenguaje de Modelado Unificado (UML) es un lenguaje visual para especificar, construir y documentar artefactos de sistemas [OMG03a]. De esta cita, el punto clave es “lenguaje visual”. Por lo tanto lo más importante para un desarrollador no es utilizar toda la notación UML, más aún, se puede usar UML al dibujar diagramas incompletos e informales para profundizar en las partes difíciles de un problema o el espacio de la solución. Estos diagramas pueden ser dibujados en una pizarra como un bosquejo, esta práctica es una manera común de aplicar UML y es favorecida por los métodos ágiles.

UML es útil para especificar, visualizar y documentar modelos para sistemas de software, incluyendo su estructura y diseño, de tal forma que se cumpla con todos sus requerimientos. UML también se puede usar para modelar sistemas que no sean software. Con la ayuda de herramientas basadas en UML disponibles en el mercado, se puede analizar los requerimientos de una aplicación y el diseño mediante cualquiera de los trece tipos de diagramas estándar.

Se puede modelar en UML casi cualquier tipo de aplicación, ya sea en un ambiente combinado con hardware, sistemas operativos, lenguajes de programación o redes.

UML tiene tres clases principales de diagramas. Los diagramas estáticos describen la estructura lógica invariable de los elementos de software representando clases, objetos, estructuras de datos y las relaciones entre ellas. Los diagramas dinámicos muestran cómo cambian las entidades de software durante la ejecución, representando el flujo de ejecución o la forma en que cambian de estado las entidades. Los diagramas físicos muestran la estructura física invariable de las entidades de software representando entidades físicas como archivos fuente, bibliotecas, archivos binarios y archivos de datos, y las relaciones que existen entre ellas.

UML cuenta con diagramas de clases, de componentes, de objetos, de despliegue, de paquetes, de actividades, de casos de uso, de secuencia, entre otros. A continuación se presenta un breve compendio de algunos diagramas relevantes.

Diagramas estáticos

Diagramas de clases

Los diagramas UML de clases son usados para representar clases, interfaces y sus asociaciones bajo un modelado de objetos estático.

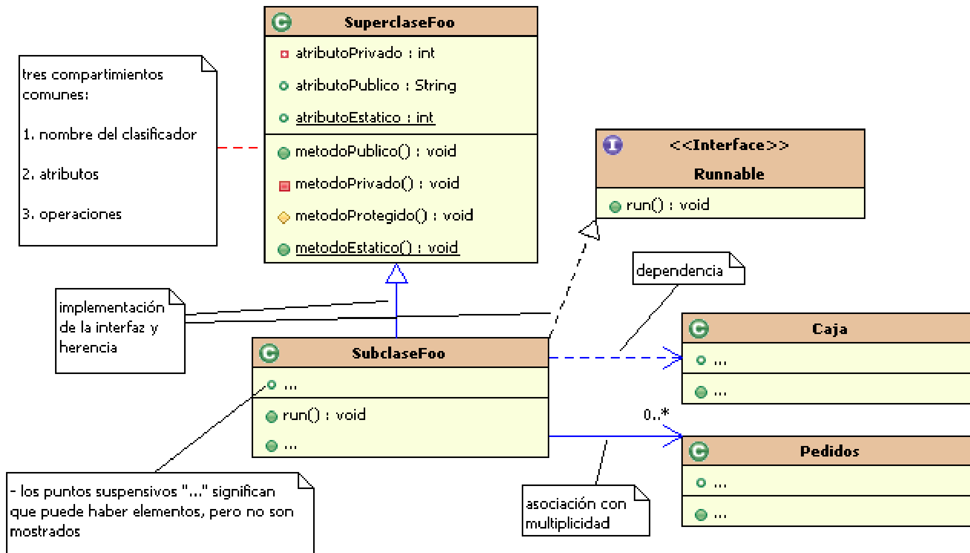


Figura 4: Notación de un diagrama de clases

La figura 4 muestra la notación general que se utiliza al momento de escribir diagramas de clases. Existen tres compartimientos para una clase: una para el nombre de la clase (o clasificador), otra para los atributos de clase y una tercera para las operaciones. Los atributos pueden mostrarse de las siguientes maneras:

●**texto del atributo** en el compartimiento de la clase, por ejemplo: *atributo1 : Atributo*

●**líneas de asociación**

●**ambas**

El formato completo para la notación de texto del atributo es:

visibilidad nombre: tipo multiplicidad = default {cadena-propiedad}

Se prefiere la notación de texto del atributo cuando se trata de tipos de datos comunes, y la notación de líneas de asociación para los casos restantes.

El último compartimiento de una clase muestra las firmas de las operaciones. El formato oficial para escribirlas es:

visibilidad nombre (lista-de-parámetros) : tipo-de-retorno {cadena-propiedad}

Las generalizaciones son mostradas con una línea sólida y una flecha triangular desde la subclase a la superclase. Las clases abstractas pueden expresarse ya sea mediante la etiqueta *{abstract}* o escribiendo el nombre de la clase con letras cursivas.

Las líneas de dependencia, las cuales son más comunes en los diagramas de paquetes, expresan una relación de conocimiento entre un elemento cliente y un elemento proveedor. La dependencia puede darse por los siguientes motivos:

- el elemento cliente tiene un atributo del tipo del proveedor
- el cliente envía un mensaje al proveedor (por medio de un atributo, un parámetro, una variable local, global o de visibilidad de clase)
- el cliente recibe un parámetro del tipo del elemento proveedor
- el proveedor es una superclase o una interfaz

Diagramas de casos de uso

UML provee una notación para dibujar los diagramas de caso de uso e ilustrar nombres, casos de uso, actores y relaciones entre ellos. Los diagramas de caso de uso tienen una relevancia secundaria, pues los casos de uso son documentos de texto antes que diagramas.

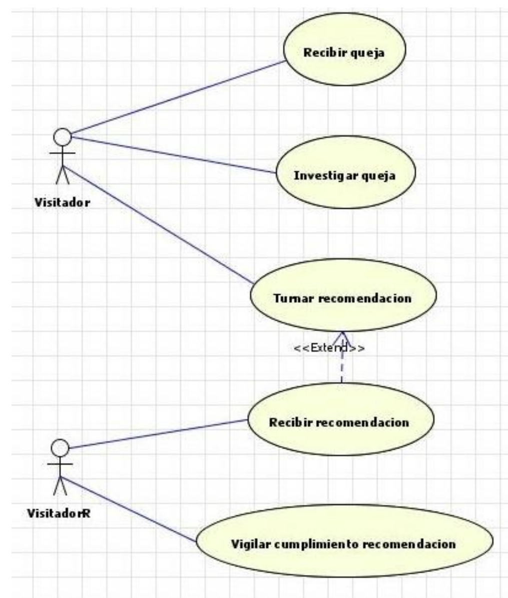


Figura 5: Diagrama parcial de casos de uso

Un diagrama de caso de uso es excelente para ilustrar el contexto de un sistema, y funciona como una forma de comunicación que sintetiza el comportamiento de un sistema y sus actores. Una ejemplo parcial de un caso de uso se muestra en la figura 5.

Diagramas dinámicos

Diagramas de secuencia

Los diagramas de interacción UML ilustran cómo los objetos interactúan a través de mensajes. Existen dos tipos: diagramas de interacción y de secuencia. La ventaja de los diagramas de interacción sobre los de secuencia es que ahorran más espacio y se comprende un poco mejor el flujo de la secuencia. Sin embargo, los diagramas de secuencia son más convenientes para documentar, realizar ingenieria inversa y generar código mediante una herramienta que lo permita.

Cada *mensaje* entre objetos es representado con una flecha sólida entre las líneas de vida verticales (figura 6). En UML, un mensaje inicial es llamado *found message*, y significa que el remitente del mensaje no es especificado, no es conocido o proviene de una fuente aleatoria. El foco de control se especifica usando una barra de *especificación de la ejecución*, la cual, es opcional. Hay dos formas de mostrar el resultado de retorno de un mensaje: usando la sintaxis *variableRetorno = mensaje(parámetro)* o, usando una línea de mensaje de respuesta hacia la barra de ejecución.

En UML, para expresar condiciones o bucles, se utilizan *marcos.* Los marcos son regiones o fragmentos de diagramas; tienen una *etiqueta* u *operador* y una *marca*. Existen los siguientes tipos de marcos: *alt*, sirve para exresar alternativas mediante fragmentos mutuamente excluyentes; *loop*, sirve para expresar la ejecución de un fragmento mientras se establezca una condición; *opt*, se utiliza para ejecutar un fragmento si una marca es verdadera; *par*, expresa fragmentos que se ejecutan en paralelo; y *region*, que expresa una región crítica en la cual sólo un hilo puede ejecutarse.

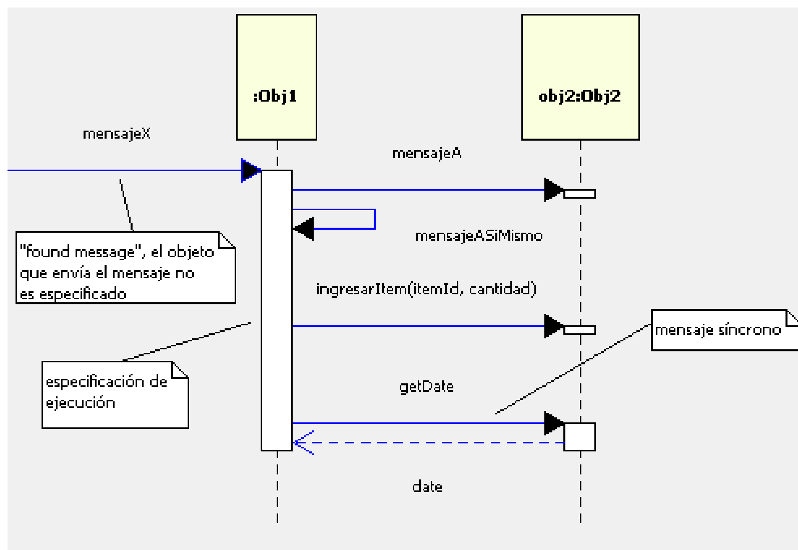


Figura 6: Mensajes y control de ejecución en un diagrama de secuencia.

Diagramas de actividades

Un diagrama de actividades muestra actividades secuenciales y paralelas en un proceso. Estos diagramas son útiles para modelar procesos de negocio, flujos de trabajo, flujos de datos y algorítmos complejos. Un diagrama de actividades ofrece una notación variada para mostrar una secuencia de actividades y puede ser aplicado para cualquier propósito, sin embargo, es ampliamente usado para visualizar flujos de trabajo de negocio, procesos y casos de uso.

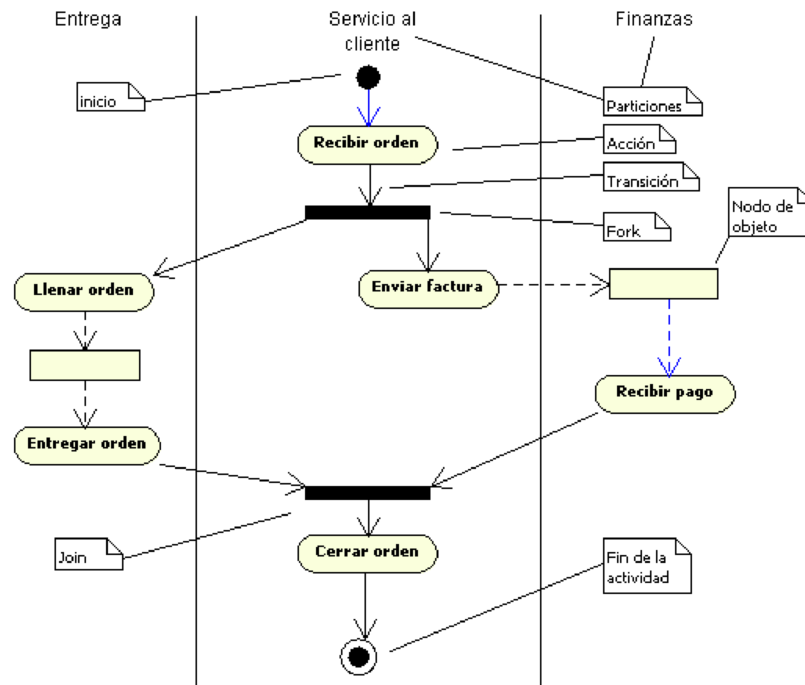


Figura 7: Notación para un diagrama de actividades básico

La figura 7 muestra la notación básica para dibujar un diagrama de actividades, compuesta por acciones, particiones, *forks*, *joins*, y nodos de objeto. Con este tipo de diagramas se puede mostrar tanto el flujo de control como el flujo de datos.

Diagramas físicos

Diagramas de despliegue

Un diagrama de despliegue muestra en qué nodos computacionales se alojan concretamente los artefactos de software (tales como archivos ejecutables), muestra el despliegue de elementos de software en la arquitectura física y la comunicación entre los elementos físicos. Los diagramas de despliegue son útiles para comprender la arquitectura física o de despliegue.

Métodos ágiles

Los métodos para el **desarrollo ágil** se aplican comúnmente para el desarrollo iterativo y evolutivo, emplean una planeación adaptativa, promueven entregas que se incrementan, e incluyen otras prácticas que alientan la *agilidad* de respuesta al cambio.

Aunque es complicado definir con exactitud los métodos ágiles, se puede decir que éstos promueven prácticas y principios que reflejan una sensibilidad hacia la simplicidad, comunicación, equipos autogestivos, pasos ligeros y más. Estos principios se pueden resumir en el manifiesto ágil, elaborado por diecisiete críticos de los modelos del desarrollo de software en 2001 [AManifiesto01]:

- valorar más a los individuos y su interacción que a los procesos y herramientas
- valorar más el software que funciona que la documentación exhaustiva
- valorar más la colaboración con el cliente que la negociación contractual
- valorar más la respuesta al cambio que el seguimiento de un plan.

El modelado ágil supone que el propósito del modelado es, primeramente, entender, no documentar, e implica ciertas prácticas y valores, que incluyen:

- adoptar un método ágil no significa evitar cualquier tipo de modelado
- el propósito del modelado y de los modelos es comprender el problema y exhortar la comunicación, no documentar
- no aplicar UML del todo o en la mayor parte del diseño del software. Modelar y aplicar UML para un pequeño porcentaje, que sea difícil o complejo
- usar la herramienta más simple que sea posible
- no modelar sólo. Modelar en parejas o tríos en una pizarra con el objetivo de descubrir, entender y compartir la comprensión
- crear modelos en paralelo
- comprender que todos los modelos serán inexactos y el código final o el diseño serán distintos que el modelo
- los desarrolladores deben diseñar modelado OO para ellos mismos, no para que lo implementen otros programadores.

Desarrollo Iterativo e incremental

Un **proceso de desarrollo de software** describe un enfoque para la construcción, despliege, y posiblemente, el mantenimiento del software, o, de manera más general, es un conjunto de actividades **necesarias** para transformar los requerimientos del usuario en sistemas de software [RJB99].

Una práctica clave tanto en el Proceso Unificado como en algún otro método moderno es el **desarrollo iterativo**. En este enfoque de ciclo de vida, el desarrollo es organizado en una serie de pequeños projectos cortos y de duración fija llamados **iteraciones**. El resultado de cada uno es un sistema parcial probado, integrado y ejecutable. Cada iteración incluye su propio análisis de requerimientos, desarrollo, implementación y actividades de prueba.

El ciclo de vida iterativo está basado en un refinamiento y robustecimiento de un sistema a través de múltiples iteraciones, con la retroalimentación y adaptación como núcleos para converger en un sistema adecuado. El sistema crece con el tiempo, iteración por iteración, de tal forma que este enfoque es llamado también **desarrollo iterativo e incremental**. Puesto que la realimentación y la adaptación rigen las especificaciones y el diseño, también es conocido como **desarrollo iterativo y evolutivo**.

El resultado de cada iteración es un sistema ejecutable pero incompleto; el sistema no será puesto en ambiente de producción hasta después de varias iteraciones. El resultado de una iteración no es un prototipo experimental sino un subconjunto del sistema final.

El desarrollo iterativo y evolutivo está basado en la actitud de arropar el cambio y la adaptación como algo inevitable y esencial, en vez de luchar en contra del cambio al intentar insatisfactoriamente de completar el conjunto de requerimientos, típico de un modelo en cascada.

En cada iteración se escoje un subconjunto de requerimientos, se diseña, implementa y prueba. En las primeras iteraciones el resultado de la elección de los requerimientos podría no ser como se deseaba, pero eso ayuda a que exista retroalimentación de los usuarios y desarrolladores.

Los beneficios del desarrollo iterativo incluyen:

- menos fallas en el proyecto, mejor productividad, tasas bajas de defectos.
- mitigación temprana de los altos riesgos

- progreso visible desde el principio
- realimentación desde el comienzo, compromiso del usuario, adaptación. Un camino para refinar un sistema que se aproxime a las necesidades reales de los usuarios
- el equipo de desarrollo no se satura con ánalisis o con pasos largos y complejos
- el aprendizaje en cada iteración puede ser usado metódicamente para mejorar el proceso de desarrollo en sí.

En sistemas que cambian frecuentemente o en sistemas complejos, la realimentación y la adaptación son claves:

- realimentación desde el comienzo del proyecto cuando los programadores obtienen las especificaciones y cuando el cliente ve los demos para refinar los requerimientos
- realimentación en las pruebas para refinar el diseño o los modelos.
- realimentación del progreso del equipo de desarrollo para adaptar y mejorar la agenda y los tiempos estimados.

El Proceso Unificado

El Proceso Unificado es un proceso genérico, es decir, un *framework* que puede ser especializado por una amplia gama de sistemas de software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de competencias, y diferentes tamaños de proyectos [RJB99].

El Proceso Unificado (PU) ha emergido como un proceso de desarrollo de software iterativo popular para construir sistemas orientados a objetos [Larman05]. El PU es muy flexible y abierto, está dirigido por los casos de uso, exhorta la adopción de buenas prácticas tales como el ciclo de vida iterativo y el desarrollo dirigido por el riesgo, así como la adopción de buenas prácticas de otros métodos iterativos tales como la programación extrema (XP), Scrum y otros. Es orientado al riesgo (centrado en la arquitectura) cuando se intenta identificar y definir estrategias para enfrentar los riesgos críticos del proyecto, resolviendo los puntos más difíciles sobre todo en las primeras fases.

Plantea un tipo de desarrollo iterativo e incremental en el cual el proyecto se organiza en una serie de mini-proyectos de duración fija (de 2 a 6 semanas) llamados iteraciones. Puesto que al término de cada iteración el sistema es probado, integrado y ejecutado, la salida es un sub-conjunto del sistema con calidad de producción final, así, el usuario tiene en producción algunas funcionalidades mientras se

van desarrollando otras. En las subsecuentes iteraciones los cambios son resultado de la retroalimentación y son bienvenidos.

El término desarrollo iterativo e incremental se refiere a una combinación al generar versiones comenzando con un subsistema funcional pequeño al cual se le va agregando alguna funcionalidad en cada versión (desarrollo incremental) y al entregar un sistema completo desde el principio y luego cambiar la funcionalidad de algún subsistema en cada nueva versión.

Un proyecto con el PU organiza el trabajo y las iteraciones a través de cuatro grandes fases:

1. **Inicio**. Se define la viabilidad del proyecto, se definen los casos de uso, el alcance y se obtienen estimaciones iniciales.
   El inicio es el primer paso para establecer una visión común del problema y del alcance del proyecto. Incluye tal vez el 10% del análisis de los requerimientos que no sean críticos, y los pasos necesarios para crear un ambiente de desarrollo que permita avanzar a la fase de elaboración.

2. **Elaboración**. Visión refinada, implementación iterativa de la arquitectura del núcleo, resolución de los riesgos críticos, identificación de los requerimientos y alcance y estimaciones reales.
   En esta fase se tiene una visión más refinada del proyecto, se implementa iterativamente el núcleo de la arquitectura, se resuelven los riesgos críticos y se identifican la mayoría de los requerimientos y su alcance.

3. **Construcción.** Implementación iterativa del riesgo no crítico remanente y preparación para el despliegue.
   En la fase de construcción se implementan los riesgos no críticos, remanentes, elementos sencillos y se prepara para el despliegue. Suele ser la fase más larga del proyecto.

4. **Transición.** Pruebas beta, despliegue.
   En la transición se obtienen pruebas beta y se despliega el producto en el entorno de los usuarios, lo que implica la capacitación de éstos.

El resultado de cada fase es un conjunto de artefactos, llamado *milestone*. Un milestone es muy útil para manejar el proyecto, tomar decisiones y evaluar el avance obtenido.

El Proceso Unificado describe actividades mediante **disciplinas** que constan de un conjunto de actividades en un área tales como el análisis de requerimientos:

- Modelado de negocio
- Requerimientos
- Diseño
- Implementación
- Pruebas
- Despliegue
- Configuración y manejo de cambios
- Administración del proyecto
- Entorno

El siguiente cuadro [Larman05] resume las disciplinas del PU y sus artefactos asociados, expresando también, para las siguientes fases, el grado aproximado de desarrollo de cada uno de estos artefactos.

c: comienzo de la construcción del artefacto.

r: refinamiento del artefacto.

Componentes de UP		**Fases del UP**			
Disciplina	**Artefacto**	**Inicio**	**Elaboración**	**Construcción**	**Transición**
Modelado de negocio	Modelo del dominio		c		
Requerimientos	Modelo de Casos de Uso	c	r		
	Visión y Análisis del negocio	c	r		
	Especificación Complementaria	c	r		
	Glosario	c	r		
Diseño	Modelo de diseño		c	r	
	Documentación de Arquitectura		c		
	Modelo de datos		c	r	
Implementación	Modelo de implementación		c	r	r
Gestión de proyecto	Plan de desarrollo	c	r	r	r
Pruebas	Modelo de pruebas		c	r	
Entorno	Marco de desarrollo	c	r		

Tabla 1: Artefactos creados a lo largo del Proceso Unificado.

En el PU, la elección de los artefactos es libre ya que no todos los proyectos requieren todos los

artefactos, ni con igual grado de profundidad o detalle. Se recomienda usar pocos artefactos, eligiendo los de mayor valor práctico para cada proyecto.

La figura 8 muestra la relación entre las fases del PU y las disciplinas. Se observa que la fase de Inicio se enfoca en el modelado de negocio y el levantamiento de algunos requerimientos para poder definir el alcance del proyecto. Por su parte, el análisis y el diseño son más exhaustivos en la fase de Elaboración, pues se pretende identificar los riesgos críticos y la realización de los casos de uso de más relevancia para la formación de la arquitectura del sistema. Las pruebas son realizadas premanentemente en las fases de Elaboración y Construcción; escribir pruebas unitarias por cada método que se vaya a implementar antes de codificar, es una práctica alentada por los métodos ágiles. Se debe notar que la relación entre las fases y disciplinas establece la diferencia fundamental del Proceso Unificado con el módelo de ciclo de vida en cascada, otorgándole de esa manera, el carácter de modelo iterativo e incremental.

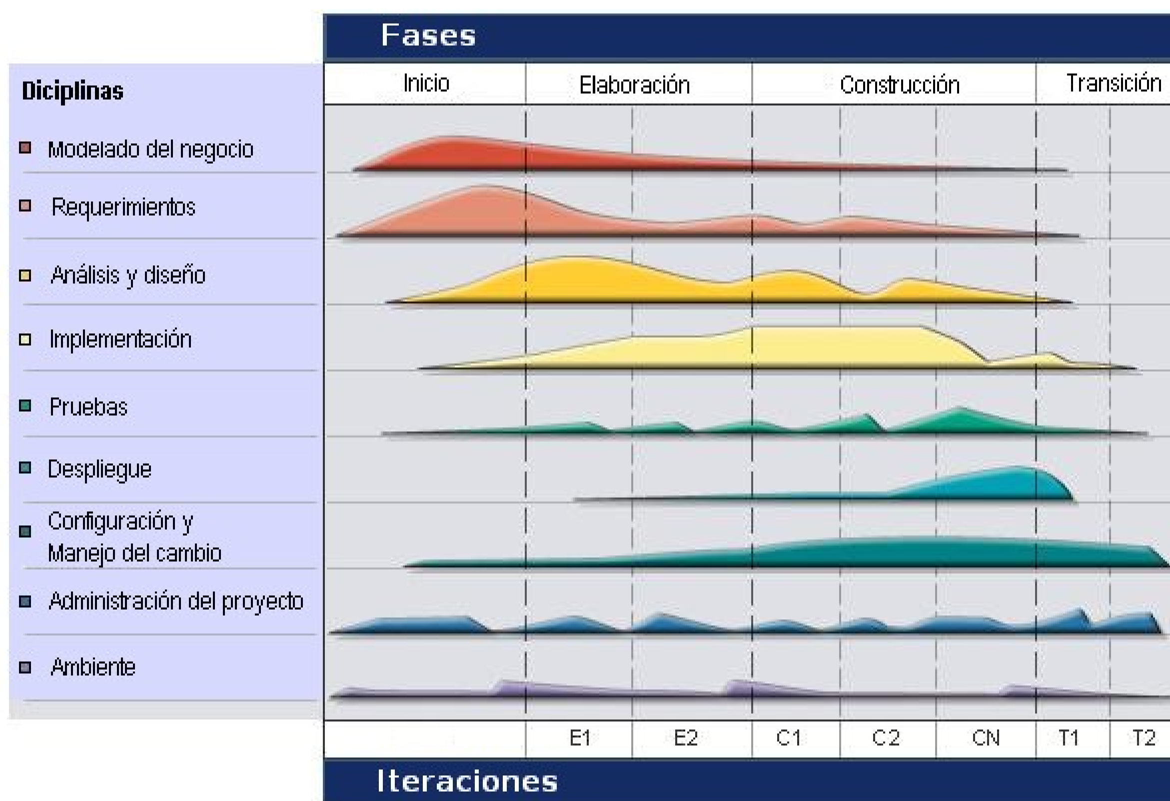


Figura 8: Disciplinas y fases en el Proceso Unificado

Pruebas

La programación extrema (XP) promueve como práctica importante de la disciplina de pruebas escribir las pruebas antes de codificar. Esto, además de varias otras prácticas de pruebas, es conocido como **desarrollo dirigido a pruebas** (test-driven development TDD).

Escribir las unidades de prueba antes, ayuda a clarificar los detalles de las interfaces y el comportamiento que deberían tener los métodos, las pruebas se tornan automáticas conforme se van escribiendo más y más pruebas, lo que da confianza para hacer cambios y poder detectar variaciones inesperadas en el comportamiento.

Unidades de prueba con JUnit

Una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código. Una fase posterior a las pruebas unitarias es la de las pruebas de integración, lo que sirve para asegurar el funcionamiento correcto de un subsistema o sistema.

Las pruebas unitarias tienen como objetivo aislar cada parte del programa y mostrar que las partes individuales son correctas. Estas pruebas aisladas proporcionan cinco ventajas básicas:

1. **Fomentan el cambio.** Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
2. **Simplifican la integración.** Las pruebas permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
3. **Documentan el código**: Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
4. **Separación de la interfaz y la implementación**: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
5. **Los errores están más acotados y son más fáciles de localizar**.

JUnit es un framework que se usa para hacer pruebas unitarias en Java. Utilizándolo, se puede evaluar el comportamiento de cada uno de los métodos de clase y JUnit devolverá el éxito o el fallo si el método cumple o no con lo que se espera que haga.

Participación profesional

Las aplicaciones empresariales deben ser más que un conjunto de módulos de código, deben estar estructuradas de tal modo que permitan su escalabilidad, una ejecución segura bajo condiciones extremas y su estructura debe estar definida claramente de tal forma que los programadores encargados del mantenimiento puedan eficazmente encontrar y corregir errores una vez que los autores del código se hayan movido a otro proyecto. Eso implica que los programas deben estar bien diseñados para mantener una buena estructura que trate la complejidad. Otro beneficio de una buena estructura es que se puede reusar código, y con ello el diseño se orienta a una colección de aplicaciónes o módulos autocontenidos. Con el tiempo, se pueden contruir librerías de modelos de componentes por cada módulo. Cuando alguna otra aplicación necesita la misma funcionalidad, el diseñador puede importar el módulo de la librería. Al tratarse el módulo en la codificación, el desarrollador puede rápidamente importar el código del módulo en la aplicación.

Bajo la premisas anteriores, detallo las actividades que realicé en la ingeniería de software del SIIGESI en las etapas de inicio y elaboración del Proceso Unificado, principalmente, en las disciplinas de análisis y diseño, pero también, de manera menos prioritaria, en las disciplinas de modelado de negocios y requerimientos, pruebas e implementación para obtener los cimientos de una arquitectura estable y bien definida. Muestro y explico el razonamiento realizado en el flujo de trabajo de diseño, para obtener algunos diagramas de clases y secuencia necesarios para disminuir la brecha de representación entre los requerimientos y la implementación, inspirados en el modelo de dominio y el modelo de casos de uso, y me baso en los patrones de diseño generales *GRASP* y algunos patrones de diseño *GoF*[3].

3 Los patrones de diseño GoF (*Gang-of-four*) fueron detallados por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides en el libro *Design Patterns*.

Inicio

Debido a la naturaleza de la organización (la CDHDF) para la cual trabajo, el software lo produce una unidad para otras unidades de la misma organización (in-house development [RJB99]). Generalmente, cuando los sistemas son creados por la necesidad específica de un área, la fase de inicio es muy útil para comenzar a entender los requerimientos, sin embargo, el sistema en cuestión sustituye un sistema heredado (legacy system) cuya comprensión de los requerimientos no parte totalmente de las ideas que surgan al momento de conocer de una necesidad, por lo que definir la viabilidad del proyecto en esta fase podría parecer trivial. Por lo tanto, las mayores consideraciones en el inicio, tomando en cuenta la naturaleza del sistema a construir, debió centrarse, por un lado, en los riesgos que puede involucrar el reemplazo o la transición del antiguo sistema al nuevo y ,por otro, en el análisis inicial del proyecto.

La mayor parte de los riesgos son mitigados en el Proceso Unificado desde un punto de vista de la arquitectura [RJB99]. Es útil comenzar a vislumbrar una arquitectura candidata desde esta fase. Otro riesgo a considerar, es cómo manejar el despliegue para sustituir el sistema heredado. Ese es un tema que se aborda a fondo en el proyecto en la fase de Transición, sin embargo, conviene visualizar desde el inicio algunas posibilidades. Kent Beck, creador de la programación extrema (XP), recomienda sustituir el antiguo sistema de manera incremental, sustituyendo nuevas unidades de funcionamiento que se consideren estables y manejables, por las anteriores [Beck04]. De esta manera, se abate el riesgo de poner el sistema en ambiente de producción después del plazo señalado. Esa consideración influencia la arquitectura, sin embargo, ése un tema que no se abarca en el presente informe. Lo que interesa es, entonces, el establecimiento de una arquitectura candidata y la captura de requerimientos, que sirvieron como base para continuar en la siguiente fase.

Captura de los requerimientos

La fase de inicio enfatiza la disciplina del levantamiento de requerimientos y modelado del negocio. Esta disciplina incluye identificar y detallar los casos de uso, y conocer o crear un modelo de dominio que sirva posteriormente para influenciar las clases de diseño. Los requerimientos se dividen, de acuerdo a su impacto directo o indirecto en el funcionamiento del Sistema, en dos: requerimientos funcionales y requerimientos no funcionales [Larman05].

Captura de los requerimientos funcionales. Definir el alcance del proyecto es una tarea que formó parte de la primera sesión de trabajo con la gente del negocio. En esta sesión, como en las

subsecuentes, participé como arquitecto y analista, en donde levanté, junto con mi jefe y un compañero analista, algunos requerimientos básicos para tomarlos como punto de partida. Se estableció que la primera parte del proyecto abarcaría el Programa de Defensa, que involucra a la Dirección General de Quejas y Orientación, las cuatro Visitadurías Generales, y la Dirección Ejecutiva de Seguimiento.

Para los requerimientos, se identificaron algunos actores:

- Peticionario
- Orientador
- Admisor
- Visitador
- VisitadorRecomendaciones

También se identificaron siete casos cuyo diagrama se puede consultar en el Anexo 1:

- Procesar orientación
- Recibir queja
- Turnar queja
- Investigar queja
- Turnar recomendación
- Recibir recomendación
- Vigilar cumpimiento recomendación

de los cuales seleccioné los siguientes casos de uso señalándolos como prioritarios:

- Procesar orientación
- Procesar gestión.

Dichos casos fueron detallados por los ingenieros de casos de uso (anexo 2).

Captura de los requerimientos no funcionales. En el anexo 3 se muestran las especificaciones suplementarias capturadas por mí (bajo mi rol de arquitecto) correspondientes con la visión de los requerimientos en esta fase que no pueden ser capturados en los casos de uso, y que, generalmente, guían en buena medida la arquitectura del Sistema. En el documento se muestra los componentes que se requirieron para la construcción del Sistema, estos son de dos tipos:

- componentes comprados
- componentes de software libre y código abierto

El componente comprado que se consideró para esta fase es un API para la generación de documentos de texto desde una plantilla. Si bien OpenOfice pone gratuitamente un framework a disposición de los desarrolladores, hubo que considerar algunos demos para MS Word, de los cuales eventualmente se eligió uno (MS Word) para utilizarse. Mientras que en la categoría de los componentes de sofware libre y código abierto se consideró el Framework Struts (MVC Web) y el Framework Hibernate (manejo de persistencia).

Apache Struts es el framework que elegí para crear la aplicación web, principalmente, por dos razones: la más importante, el conocomiento de su uso por parte del equipo de desarrollo; y por ser un framework flexible de alto rendimiento que ayuda a la separación de intereses (*separation of concerns principle*) al utilizar una arquitectura Modelo-Vista-Controlador (MVC). Por su parte, Hibernate ha emergido entre la comunidad Java como un framework de persistencia robusto y flexible.

Arquitectura candidata

En este punto, propuse la siguiente arquitectura candidata de la figura 9. En el diagrama se nota una arquitectura en tres capas. En la capa superior, se encuentran los manejadores de Apache Struts, que son el punto de contacto entre los mapeos xml de struts (por medio del archivo struts-config.xml), que controlan el flujo de las peticiones web, y la lógica de la aplicación dada por las clases del dominio. La capa intermedia es precisamente la capa de dominio, cuyas interacciones con la tercera capa (de servicios técnicos) son necesarias para el almacenamiento de datos (ya sea en una base de datos o archivos de texto) y para el servicio de almacenamiento y creación de documentos de texto. Debe notarse que no se contempló la interacción con un componente externo o una tercera parte (ej. *web services*), que de haberlos, formarían parte de la capa de servicios técnicos.

Modelo del dominio

Aunque la elaboración de un modelo de dominio en la fase de inicio no es muy común, realicé, con ayuda de los analistas, un modelo tomando como base el modelo de dominio del sistema anterior (figura 10). Esto ayudó a mejorar la comprensión del sistema en esta fase tanto para la gente de negocio como para el equipo de desarrollo.

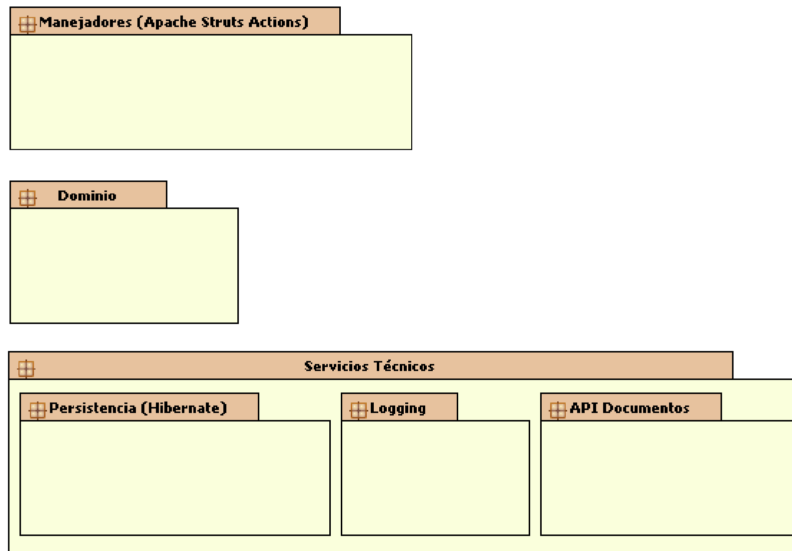


Figura 9: Diagrama de paquetes de la Arquitectura candidata.

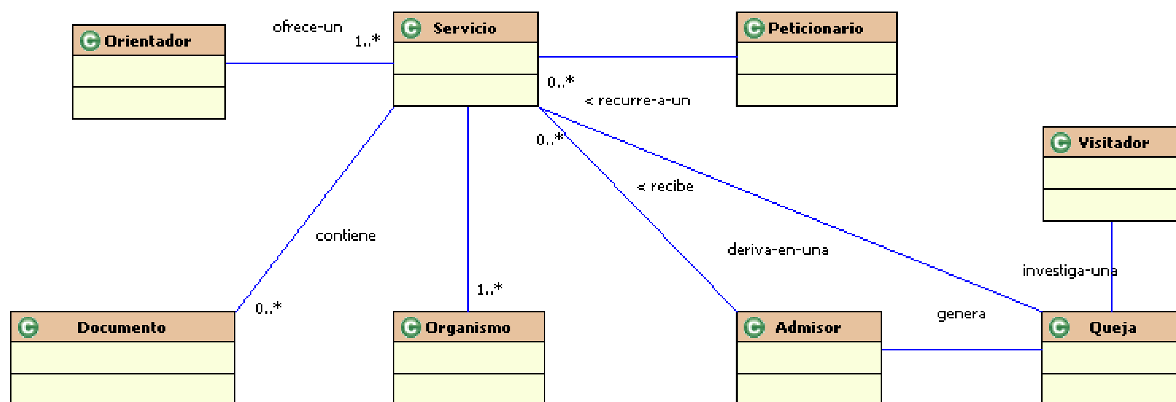


Figura 10: Modelo del dominio en la fase de inicio

Resumen de las actividades realizadas en la fase de Inicio

La fase de inicio comenzó con algunos planteamientos sobre la viabilidad del proyecto y los riesgos implicados, la identificación de actores y casos de uso, de los cuales elegí dos, cuya relevancia para la arquitectura me es mayor, y que los ingenieros de caso de uso capturaron conjuntamente con la gente de negocio, dando como resultado el comienzo de un modelo de casos de uso; capturé algunos requerimientos no funcionales a través del documento de especificaciones suplementarias; sugerí una arquitectura candidata que constó de tres capas (manejadores, dominio y servicios técnicos); finalmente, elaboré, juto con los analistas, un modelo de dominio inicial en base al sistema heredado, que sirvió como artefacto útil en los comienzos de la fase de elaboración.

Elaboración

En esta fase capturé, junto con los ingenieros de caso de uso, los requerimientos remanentes, formulando los requerimientos funcionales en el texto de los casos de uso.

Analicé y diseñé el flujo principal de algunos de los casos de uso más representativos. Realicé completamente el caso de uso *Generar documentos:* analicé, diseñé, escribí algunas pruebas unitarias e implementé el diseño. Para las directrices de seguridad, detalladas en las especificaciones suplementarias, hice el análisis y diseño correspondiente.

Finalmente, establecí la cimentación de la arquitectura.

Modelo del dominio

Revisando los casos de uso, identifiqué, junto con los analistas, los siguientes conceptos que ayudaron en la elaboración de un modelo del dominio:

Peticionario	Visitador	Agraviados	Involucrados
Orientador	Admisor	Organismos	Grupo Agraviado
Gestión Servicio	Gestión	Expediente de queja/Queja	Administrador
Caso	Remisión	Expedientillo	Información histórica de la conclusión
Documento	CDHDF	Catálogo de violaciones a los DDHH.	Información histórica de los Visitadores removidos
Dirección de Orientación	Dirección de Admisibilidad y Registro de Quejas	Visitaduría General	Dirección Ejecutiva de Seguimiento

Tabla 2: Conceptos del dominio del problema.

Estos conceptos pertenecen a la categoría de objetos de transacción (Caso); *items* de la transacción (Gestion); registro (Información histórica); roles, gente y organizaciones (Peticionario, CDHDF, Administrador, etc.); objetos físicos (Documentos); catálogos (Catálogo de violaciones a los DDHH) [Larman05].

El modelo de dominio resultante se muestra en la figura 11.

Puede observarse que hay conceptos que no fueron utilizados en el modelo del dominio. Por ejemplo, para la creación de los documentos se requiere conocer el área ("Dirección ejecutiva de seguimiento", "Dirección Admisibilidad y Registro de Quejas", etc.), sin embargo, para clarificar el entendimiento del problema, éste no es muy relevante para ser incluido como concepto en el modelo. No obstante, el hecho de que cada empleado pertenece a un área, hace de él un concepto importante a considerar, si bien no aparece en el modelo.

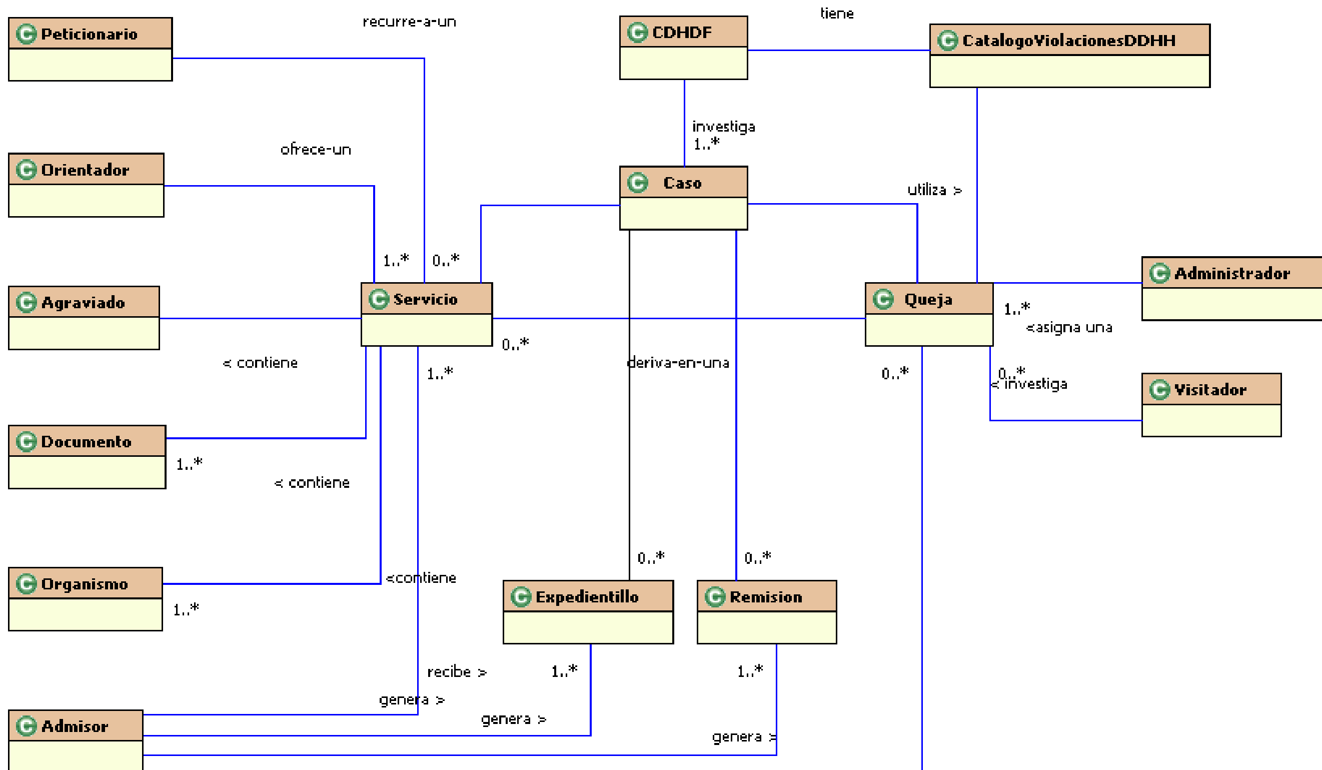


Figura 11: Modelo del dominio para la fase de elaboración.

Captura y refinación de la mayoría de los requerimientos

Derivado de la captura de requerimientos en las sesiones semanales, identifiqué, junto con los analistas, los siguientes casos de uso:

- Procesar orientación
- Procesar gestión
- Capturar involucrados
- Capturar perfil socio-económico
- Buscar datos

- Capturar organismos
- Generar documentos
- Recibir documentos
- Recibir queja
- Investigar queja
- Turnar recomendación
- Recibir recomendación
- Vigilar cumplimiento recomendación
- Manejar usuarios
- Manejar seguridad
- Manejar directorio institucional
- Iniciar Sistema

Se debe notar que aparecieron nuevos casos de uso con respecto a los existentes en la fase de inicio: Capturar involucrados, Capturar perfil socio-económico, Buscar datos, Capturar organismos, Generar documentos, Manejar usuarios, Manejar seguridad, Manejar directorio institucional e Iniciar sistema.
Los actores que identificamos al detallar los casos de uso son:

- Peticionario
- Orientador
- Admisor
- Visitador
- VisitadorRecomendaciones
- Administrador del sistema

Priorización de los casos de uso

Del total de los casos de uso, seleccioné, a lo largo de las iteraciones que compusieron esta fase, los siguientes casos de uso prioritarios desde el punto de vista del negocio y de relevancia para la arquitectura:

- Procesar orientación
- Procesar gestión
- Generar documentos
- Capturar organismos
- Capturar involucrados

- Recibir queja
- Investigar queja

Estos siete casos de uso representan la mayor cantidad de tareas que se realizan diariamente en la CDHDF. Tan sólo en 2008, la CDHDF generó y envió 4,861 documentos de medidas precautorias (*Generar documentos*), brindó 39,365 servicios (*Procesar orientación*), de los cuales se admitieron y registraron 7,814 quejas (*Procesar gestión*, *Recibir queja* e *Investigar queja*) [CDHDF09a]. Su detalle y estudio fue útil para establecer las bases de la arquitectura, y para mitigar los riesgos en etapas tempranas del proyecto.

Detalle de los casos de uso

Los ingenieros de casos de uso se enfocaron a capturar los casos de uso que seleccioné o, en su caso, a completar los que desde la fase de inicio ya se habían comenzado a detallar. Por iniciativa de mi jefe, por acuerdo mutuo y debido a la relevancia que tiene para la Comisión generar cientos de documentos diariamente[4], me dediqué a detallar el caso de uso *Generar documentos*. El resultado de esta actividad puede ser revisada en el Anexo 2 en donde se encuentran detallados los siete casos de uso escogidos por mí. Los casos de uso “Buscar datos”, “Turnar recomendación” y “Vigilar cumplimiento recomendación” también se detallaron. Los casos de uso restantes no se detallaron completamente en esta fase; de esta manera, el 100% de los casos de uso fueron identificados (para el Programa de Defensa), y cerca del 70% de los casos de uso fueron descritos en detalle.

Estructura del Modelo de casos de uso

En la mayoría de los casos de uso (“Procesar orientación” y “Procesar gestión” por ejemplo), se advierte, al leer lo especificado, la frecuencia con que estos requieren generar documentos. Es por ello que en el modelo de casos de uso (Anexo 1), identifiqué a “Generar documentos” como un nuevo caso de uso relacionado a los que lo requieren, con lo cual, se eliminan redundancias y se obtiene un Modelo de casos de uso mejor estructurado. Lo mismo se aplica para “Capturar organismos”, “Capturar involucrados” y “Buscar datos”. Para representar la relación de estos casos de uso en el modelo, utilicé un mecanismo de inclusión, que en los diagramas de caso de uso está dado por el estereotipo *include*, y en el texto de los casos de uso sobresale por estar subrayados (por ejemplo: Generar documentos).

4 Dato estimado proporcionado por la Unidad de Tecnologías de Información.

Análisis y diseño

Procesar orientación

Por cada caso de uso fue conveniente dibujar un diagrama de secuencia de sistema para el flujo principal, que se convirtió en un artefacto más para guiar el diseño. Para el caso de uso *Procesar orientación* elaboré un diagrama de secuencia de sistema para el escenario principal:

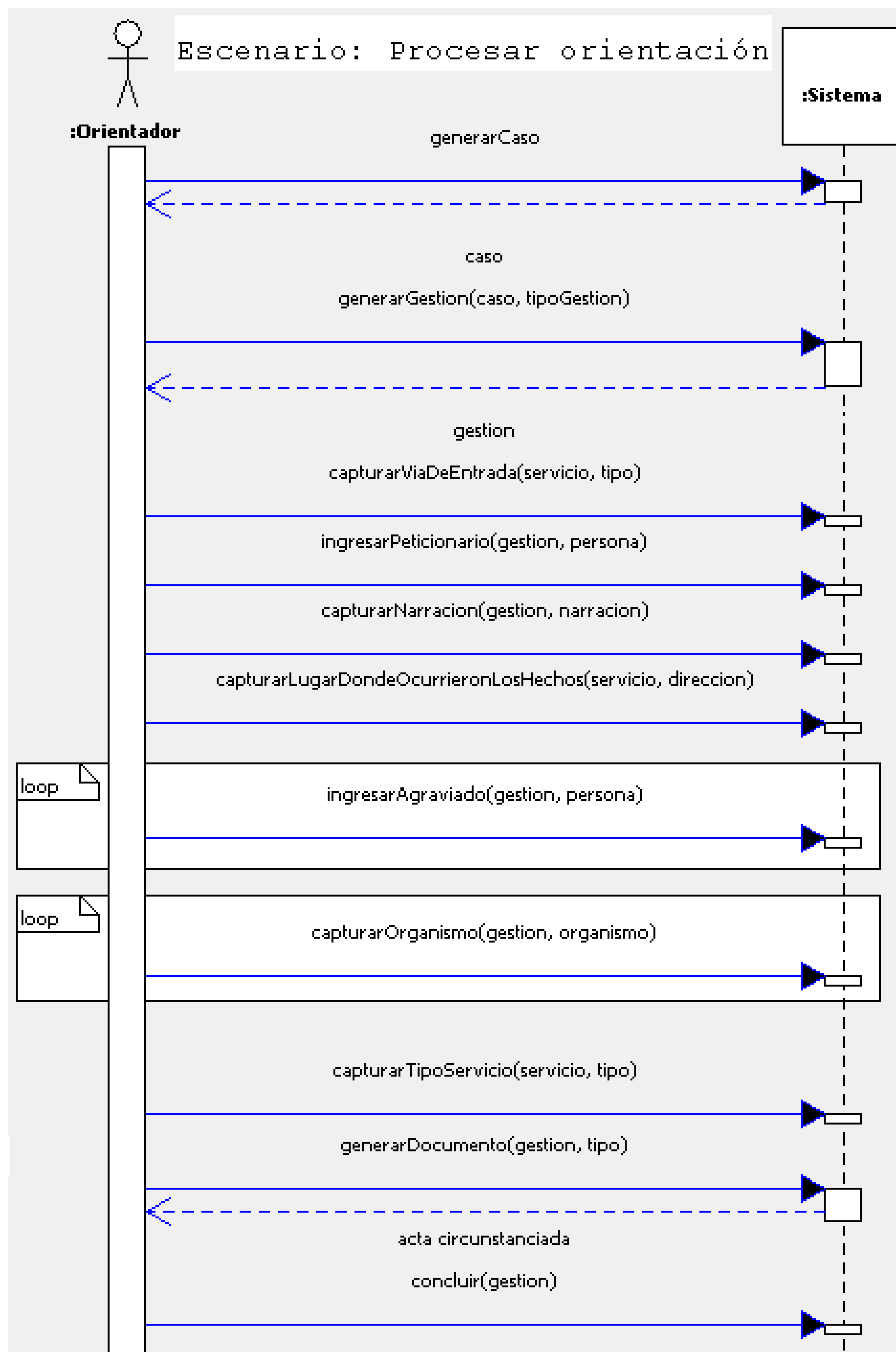


Figura 12: Diagrama de secuencia de sistema para Procesar orientación.

Diseño para generarCaso

Inspirado en el modelo del dominio, de CDHDF deriva el nombre de *OrientacionHandler*, por ser la clase que funge como controladora para el servicio de orientación, que a su vez, por el patrón de diseño del creador, es una candidata para crear Caso. Al crearse, *Caso* debe contener una colección vacía de Gestiones. El título del caso contiene una nomenclatura y un número consecutivo. Del caso de uso *Procesar Orientación* (Anexo 2) se sabe que se requiere reiniciar el número consecutivo (con el cual se registra el caso) automáticamente al inicio de cada año. Para lograr esto, se hace la llamada al método *verificarContadores* de la clase de utilería *ContadorService* que es creada por medio de la clase *UtilServiceFactory* que utiliza el patrón de diseño *singleton* para crearse. *verificarContadores* verifica el contador no sólo para el caso, sino también para cualquier otro contador que requiera reiniciarse cada año automáticamente (por ejemplo, los contadores de las gestiones, documentos, etc), por lo que éste servicio deberá ser requerido por cada uno de sus clientes. *Caso* delega la responsabilidad de generar la nomenclatura del título del caso a *CasoDAO* por medio de la llamada a *obtenerNomenclatura,* y a *NomenclaturaService* mediante el mensaje *generarTituloCaso.* A continuación, se persiste el objeto *Caso* por medio de *CasoDAO*, que es la clase que Hibernate utiliza para persistir el objeto. Se debe notar que agregué *tipo* como parámetro de *mostrarCaso*, esta observación surgió durante la sesión de diseño, que no se había contemplado en la sesión de análisis (ver figura 12).

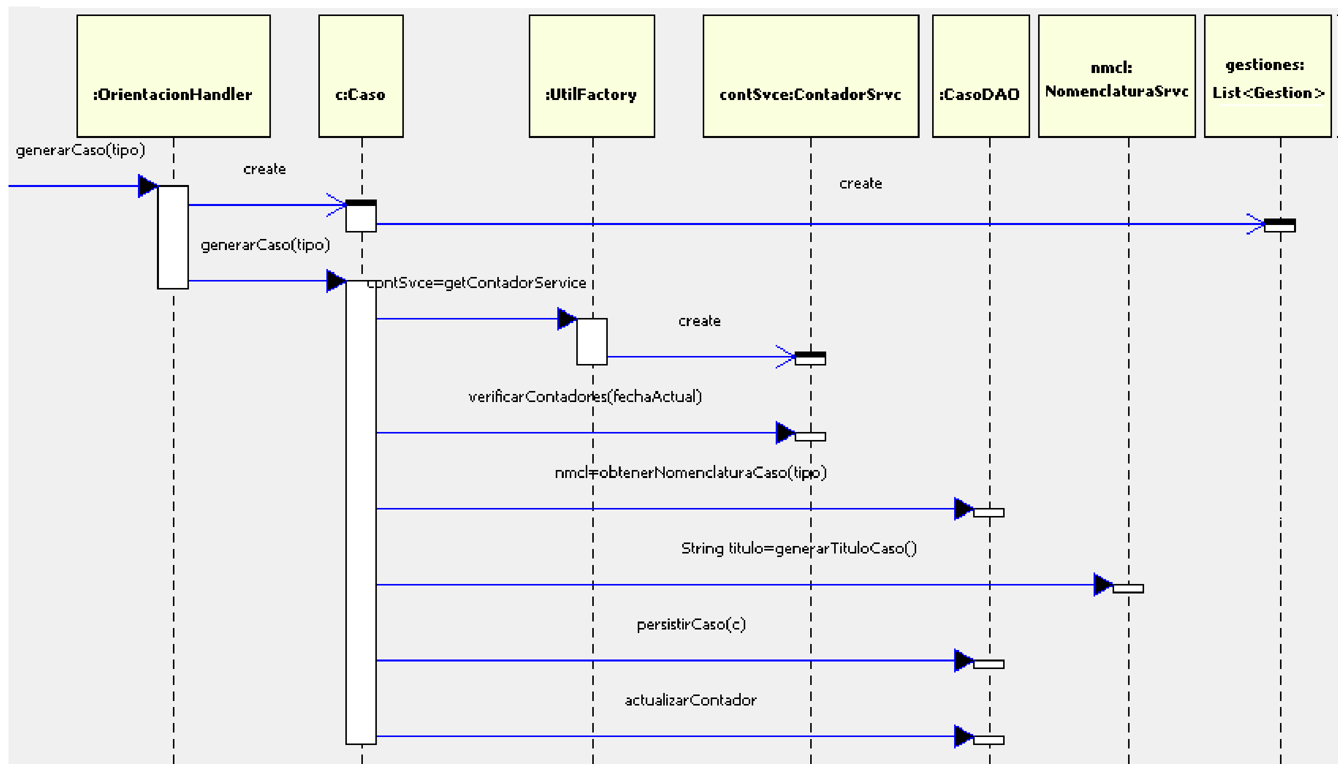


Figura 13: Diagrama de secuencia para generarCaso.

generarServicio

OrientacionHandler, la clase controladora, recibe el mensaje para generar el servicio en un caso. ServicioDAO, se encarga de crear y almacenar el servicio en la BD, el cual, es agregado al conjunto de gestiones del Caso (figura 14).

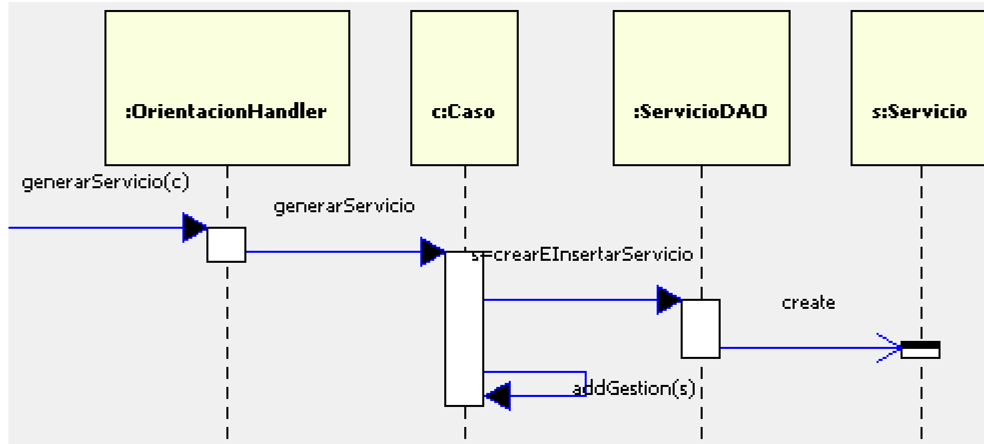


Figura 14: Diagrama de secuencia para generarServicio.

capturarViaDeEntrada, capturarNarracion, capturarLugarDondeOcurrieronLosHechos y capturarTipoServicio

Estos mensajes son muy similares en su estructura, y básicamente, realizan operaciones de actualización. En la figura 15 se muestra el diagrama de secuencia para *capturarViaDeEntrada*. *capturarNarracion*, *capturarLugarDondeOcurrieronLosHechos* y *capturarTipoServicio* son análogos. *OrientacionHandler* recibe el mensaje, enseguida, envía un mensaje a la clase de persistencia correspondiente para actualizar el valor en servicio.

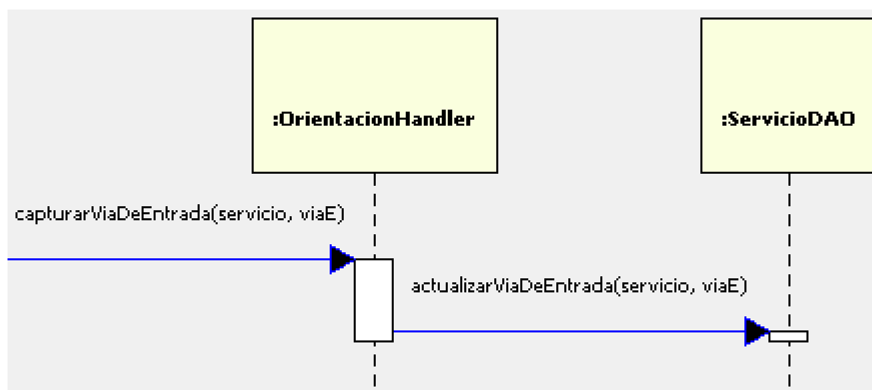


Figura 15: Diagrama de secuencia para capturarViaDeEntrada.

ingresarPeticionario, ingresarAgraviado, capturarOrganismo y generarDocumentos
Explicaré estos mensajes al realizar sus casos de uso correspondientes, a saber, Capturar involucrados, Capturar organismos y Generar documentos.

concluirGestion
Revisando los casos de uso *Procesar orientación* e *Investigar Queja*, se observa que las respectivas gestiones requieren ser concluidas; se sabe que las recomendaciones también se concluyen, es decir, todas las gestiones del Programa de Defensa se concluyen. El diagrama de secuencia de la figura 16 muestra la manera general de cómo se concluyen las gestiones. *OrientacionHandler* recibe el mensaje concluir y como parámetro se le pasa la gestión que se va a concluir. *Gestion*, que es una clase abstracta (figura 17), define el método *concluir*, el cual implementa la lógica para persistir la información de la conclusión, y también hace una llamada al método *concluirLogic*, que es un método abstracto de *Gestion*, cuya lógica es implementada en cada una de las clases concretas (*Servicio*, *Expediente* y *Recomendacion*). El hecho de haber definido un método que implemente un cuerpo general (*template method*), cuyas instrucciones deben ser ejecutadas de la misma manera por cada clase concreta, y que además, requieran implementar lógica específica por medio de un método protegido (*hook method* o *primitive operation* [GHJV95]), se le conoce como *Template method pattern*, que ofrece una solución adecuada para este mensaje.

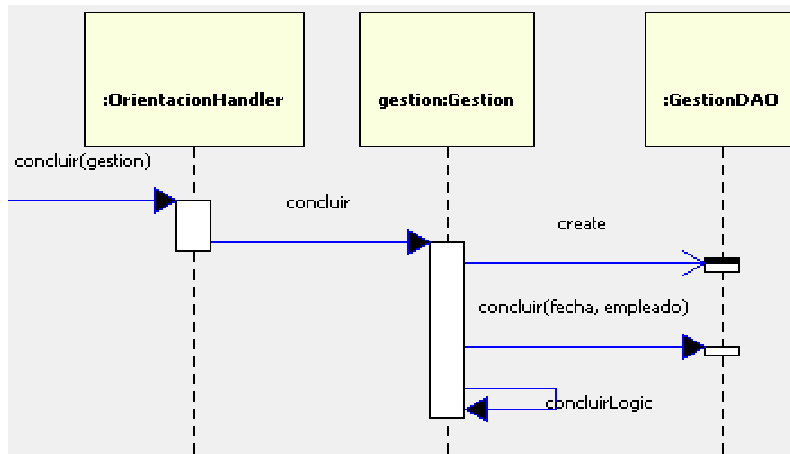


Figura 16: Diagrama de secuencia para concluir una gestión.

En el diagrama de la figura 17 se observa la estructura de clases de la gestión *Servicio* en el momento en que diseñé el caso de uso Procesar orientación para su flujo principal.

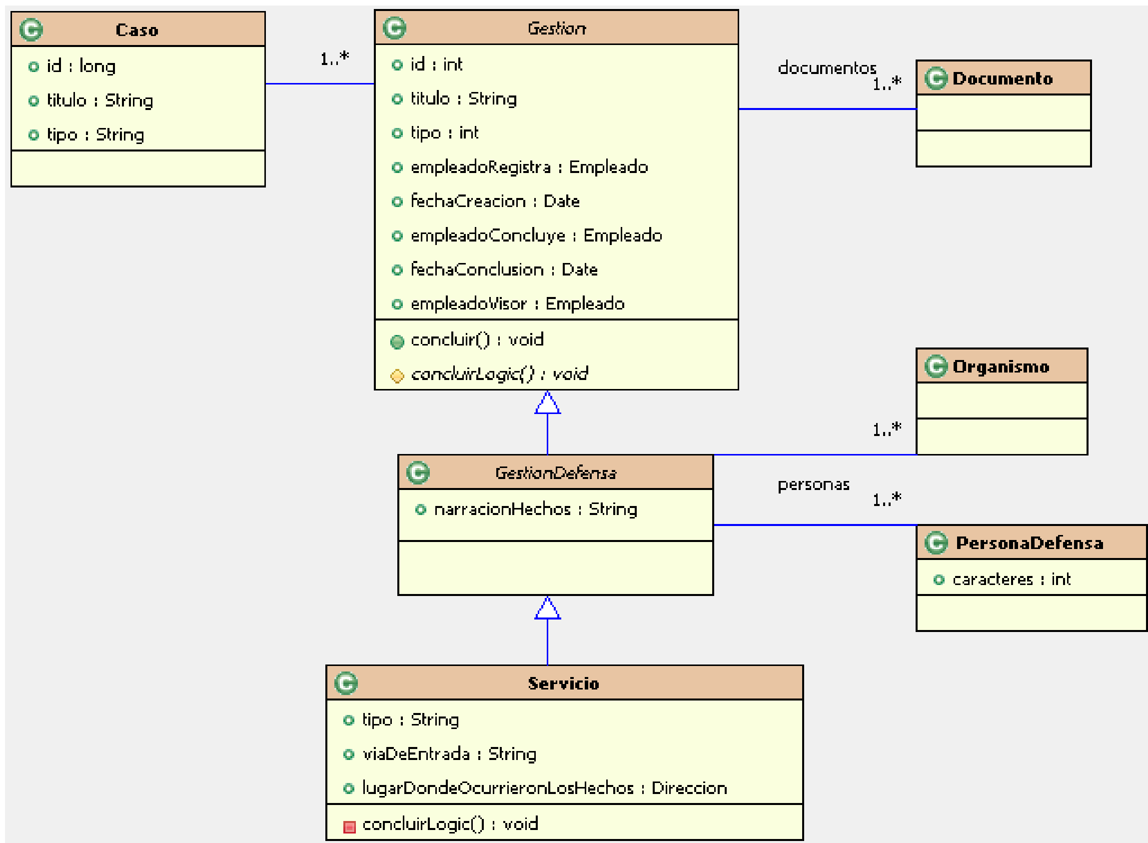


Figura 17: Diagrama de clases para Gestion-Servicio

Capturar involucrados

Nuevamente, para el escenario principal este caso de uso, elaboré un diagrama de secuencia de sistema (figura 18), para propocionar una mejor idea de los mensajes que debe recibir el Sistema, además de servir como punto de partida para comprender el escenario a desarrollar y como puente entre las especificaciones (caso de uso) y el diseño[5].

En la figura 18 se observan cuatro mensajes que conforman el escenario principal del caso de uso, cuya realización por cada mensaje explico a continuación:

5 El puente entre las especificaciones y el diseño para mejorar la comprensión entre los objetos de software y el *mundo real* es llamado *brecha de baja representación* o *Low represtational gap* [Larman05].

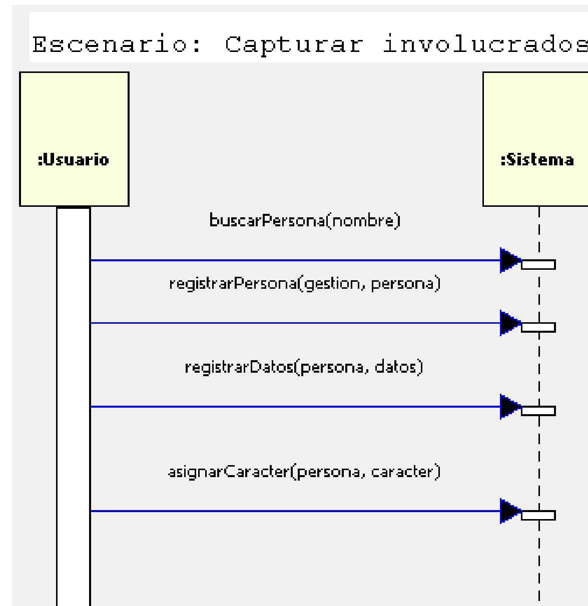


Figura 18: Diagrama de secuencia de sistema para Capturar involucrados.

buscarPersona

OrientacionHandler recibe el mensaje *buscarPersona*. Como parámetro recibe una persona. La clase *Buscador* es la encargada de realizar las tareas de búsqueda y tiene visibilidad con la clase *BusdacorDAO* para realizar las consultas necesarias. En el diagrama de secuencia se observa que *Buscador* manda el mensaje *listarPersonas* a *BuscadorDAO*, el cual, regresa un conjunto de personas de acuerdo al criterio de búsqueda. De acuerdo con los requerimientos especiales del caso de uso (anexo 2), el Sistema debe buscar a la persona considerando las posibles combinaciones de sus nombres. Para satisfacer esto, *Buscador* manda el mensaje *tieneVariosNombres* a *Persona*. En caso de que la persona tenga varios nombres, se buscan las coincidencias de las personas con el nombre combinado y el resultado se agrega a la lista de personas encontradas (ver figura 19).

Figura 19: Diagrama de secuencia para buscarPersona.

registrarPersona

OrientacionHandler recibe el mensaje *registrarPersona*, y recibe como parámetros la gestión y la persona. *Gestion* es la clase responsable de registar la persona. La persona se persiste sólo si no ha sido ya persistida (su id es mayor a cero), es decir, si fue encontrada en la búsqueda. Enseguida se agrega la persona a la colección de personas de la gestión, y *GestionDAO* es responsable de persistir la adición de la persona por medio del mensaje *actualizarGestion* (ver figura 20).

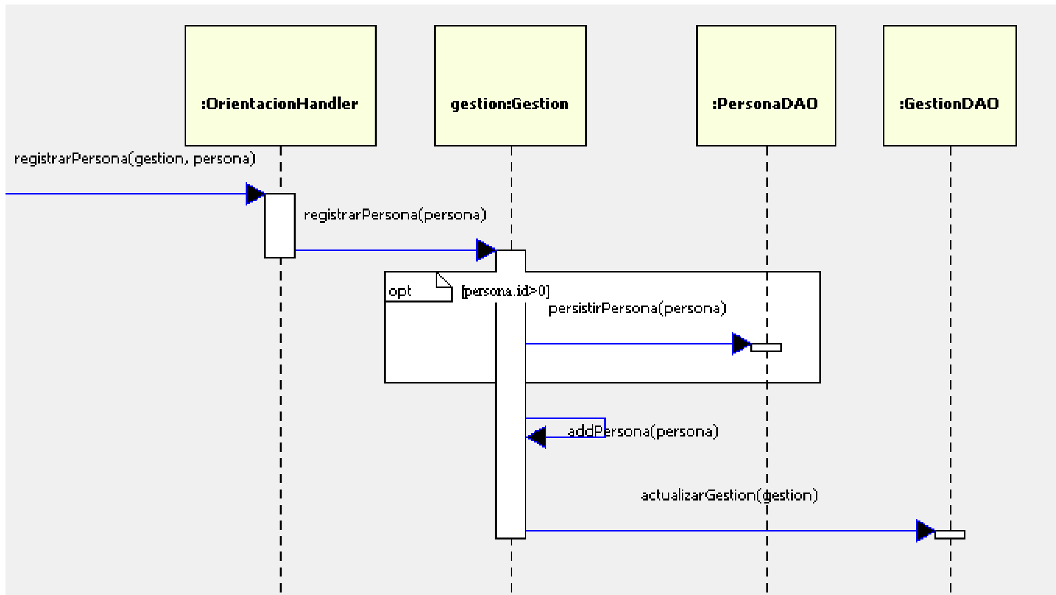


Figura 20: Diagrama de secuencia para registrarPersona

registrarDomicilios

En el Diagrama de Secuencia de Sistema para *Capturar involucrados* (figura 20) se observa el mensaje *registrarDatos*. Estos datos son: los domicilios, teléfonos y correos electrónicos. A continuación muestro el diagrama de secuencia para *registarDomicilio*:

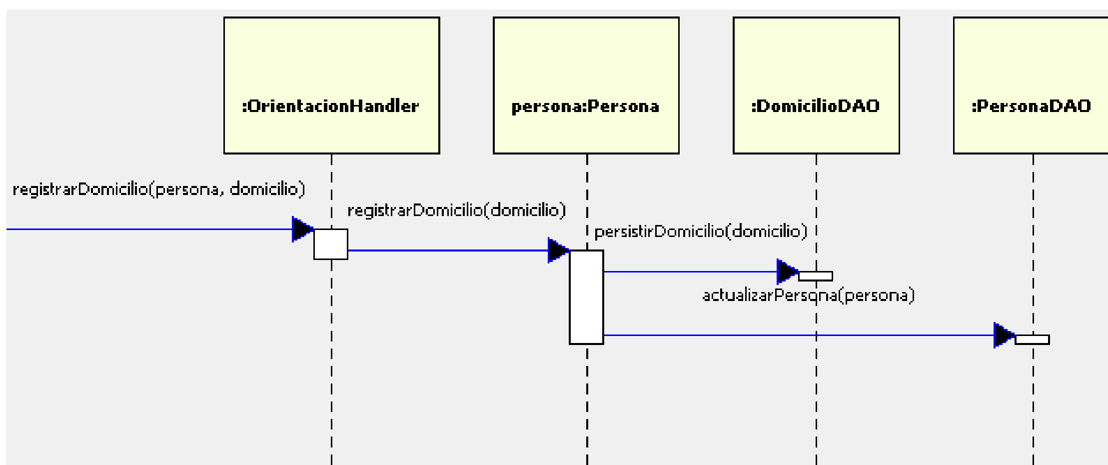


Figura 21: Diagrama de secuencia para registrarDomicilio.

OrientacionHandler delega la tarea de registrar el domicilio a *Persona*, que a su vez es la responsable

de persistir el domicilio a través de las clases de persistencia *DomicilioDAO* y *PersonaDAO*. Operaciones similares se realizan para registrar los teléfonos y correos electrónicos.

asignarCaracter

asignarCaracter es simple de diseñar: a *Persona* se le asigna el carácter y se persiste. Lo interesante es que una persona tiene un carácter (peticionario, agraviado o ambos) para gestiones del Programa de Defensa. La disciplina de modelado de negocios no se realizó para el Programa de Promoción[6], por lo que se desconocen cuestiones de diseño para éste. Lo más conveniente es hacer una especialización de *Persona*: *PersonaDefensa*, cuyos atributos y métodos se muestren sólo a sus clientes[7] (ver figura 22).

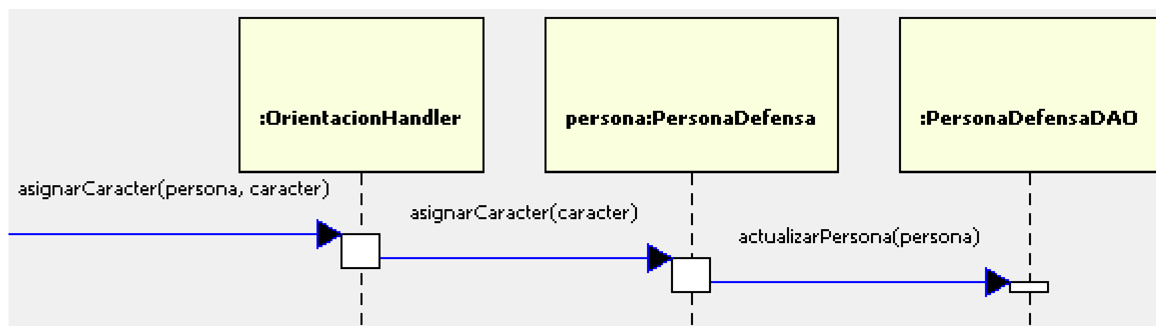


Figura 22: Diagrama de secuencia para asignarCaracter.

Generar documentos

Este caso de uso es de mucha relevancia para la arquitectura del sistema. Los casos de uso más importantes como *Procesar orientación*, *Procesar gestión*, *Recibir queja* e *Investigar queja*, entre otros, incluyen la generación de documentos (ver casos de uso en el anexo 2). Además, los documentos son la base para la comunicación tanto interna como externa de la Comisión. Se utilizan documentos para acordar la recepción de una Queja por parte de las Visitadurías, para dar fe de eventos en los sucitados, para notificar a los peticionarios y a los organismos sobre la conclusión de una Queja, para canalizar servicios a algún otro organismo, etc. [CDHDF10-08].

Para el Programa de Defensa, la generación de documentos es continua e importante. Muy probablemente también lo es para el Programa de Promoción. Por su relevancia y porque se contempló la necesidad de crear nuevos tipos de documentos para ambos programas, la arquitectura debió ser los suficientemente robusta para

6 La integración del Programa de Promoción al sistema será posterior a la del Programa de Defensa.
7 Principio de la segregación de interfaces [Martin04].

soportar la adición de nuevas clases, que representen los nuevos tipos, sin mayor esfuerzo de diseño.

La idea fue desarrollar un subsistema con un conjunto cohesivo de interfaces y clases que colaboren para proveer el servicio de crear documentos sin variar la parte de la lógica del sistema; que cuente con clases abstractas que contenga métodos abstractos y concretos; y que permita definir subclases para crear nuevos tipos de documentos.

El flujo principal del caso de uso *Generar documentos* es muy sencillo, y su diagrama de secuencia de sistema es el siguiente:

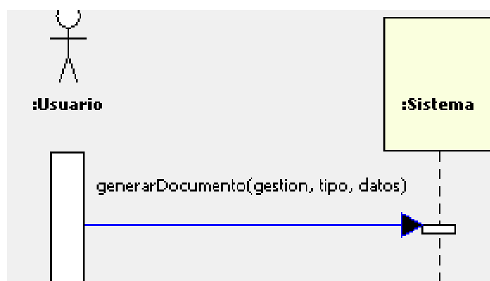


Figura 23: Diagrama de secuencia de sistema para generarDocumento

Se observa en la figura 23 un solo evento, g*enerarDocumento*, que tiene como parámetros la gestión donde se creará el documento, el tipo, y datos adicionales como el plazo, observaciones, contacto, etc.

Realizar este caso de uso requirió mayores esfuerzos de diseño. Primero hubo que plantearse qué objeto crearía el documento, lo persistiría y generaría su archivo físico correspondiente. Para realizar la primera tarea, concebí algunas clases de fabricación pura: *IGeneradorDocumentos*, *GeneradorDocumentosAbstract*, y sus tipos derivados.

Al leer el caso de uso, identifiqué que básicamente existen tres tipos de documentos: aquellos que se envían a organismos, aquellos que se envían a peticionarios, y los internos; además, por cada tipo básico existen *tipos nominales* o simplemente *tipos* (Acuerdo de conclusión, Medidas precautorias, Acta circunstanciada, etc). Los documentos a organismos persisten el documento con información sobre el plazo de respuesta, organismo al que va dirigido, contacto, etc., mientras que los documentos a peticionarios requieren el peticionario a quien va dirigido. Los documentos internos no requieren ese tipo de datos. Por estas razones fue conveniente usar una estructura polimorfa y crear tres clases abstractas que generen cada tipo de básico de documento: *GeneradorDocumentosOrg*,

GeneradorDocumentosPeticionario y *GeneradorDocumentosInternos*.

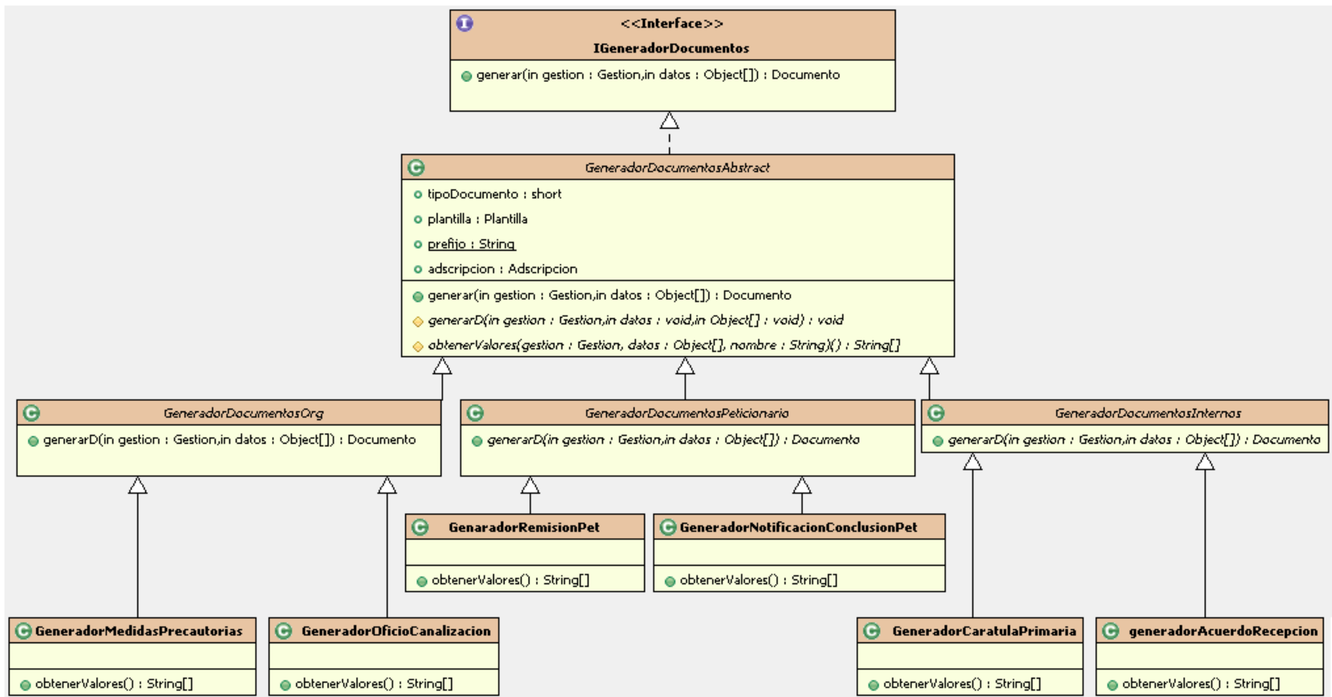


Figura 24: Clases para la generación de documentos.

Nuevamente utilicé el patrón de diseño *Template method*. *IGeneradorDocumentos* define el método *generar* (*template method*), el cual es implementado por la clase abstracta *GeneradorDocumentosAbstract*, en cuyo cuerpo se definen operaciones que se realizan para toda clase concreta y, adicionalmente, manda el mensaje *generadorD* (*hook method*) a sí misma. *generadorD* está definido como abstracta en *GeneradorDocumentosAbstract*, por lo que *GeneradorDocumentosOrg*, *GeneradorDocumentosPeticionario* y *GeneradorDocumentosInternos* lo implementan de acuerdo al comportamiento requerido.

Cada tipo de documento requiere cierta información que debe contener su archivo físico. Las clases concretas que deriven de *GeneradorDocumentosAbstract* deben generar respectivamente los tipos de documentos. Cada generador debe ser capaz de obtener los valores de texto que se requieren para generar el archivo físico de texto. Por esta razón, *GeneradorDocumentosAbstract* define el método abstracto *obtenerValores*, cuyo cuerpo es implementado por cada clase concreta. En el diagrama de la figura 24 sólo se muestran las clases concretas *GeneradorMedidasPrecautorias*, *GeneradorOficioCanalizacion*, *GeneradorRemisionPeticionario*, *GeneradorNotificacionConclusionPeticionario*, *GeneradorCaratulaPrimaria* y *GenerardorAcuerdoRecepcion* por razones de espacio, pero debe entenderse que los generadores para

los tipos de documentos restantes tienen la misma estructura. Más aún, de esta forma se observa que si se requiere crear un nuevo tipo de documento, sólo se debe crear su clase generadora, asociarle una estrategia de conteo, saber de qué tipo básico es, e implementar *obtenerValores*. Esto satisface la necesidad de generar nuevos tipos rápida y sencillamente.

La nomenclatura para nombrar estas clases es *GeneradorNombreDelTipoDeDocumento*. Esta convención debe considerarse para nuevas clases generadoras.

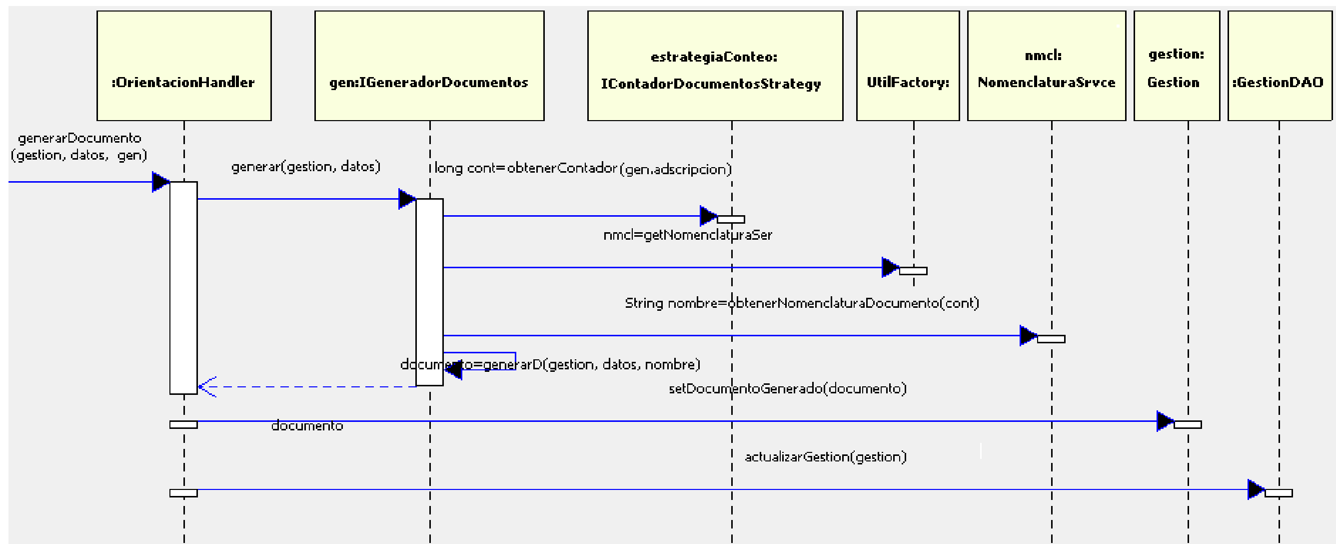


Figura 25: Colaboraciones al generar documentos.

El diagrama de secuecia de la figura 25 muestra las colaboraciones entre los objetos. *OrientacionHandler* recibe el mensaje *generarDocumento* y recibe como parámetros la gestión, los datos que se requieren, y el generador. Se observa que en este diagrama no se muestra la creación de alguna clase concreta de *IGeneradorDocumentos*. En una capa superior a la del dominio, debe existir una estrategia para crear el generador de acuerdo al tipo de documento solicitado (posiblemente con algún *Factory* en un ojbeto que extienda de *Action* de Struts). Mientras tanto, para estas colaboraciones, el polimorfismo hace que no importe de qué tipo concreto se trate.

El generador de documentos tiene como atributo a *estrategiaConteo* de tipo *IContadorDocumentosStrategy*, que es el responsable de obtener los contadores para la generación de los números de documentos. Cada generador contiene un tipo real de estrategia de conteo, pudiendo ser ésta *ContadorDocumentoPorAdscripcionStrategy* o *ContadorDocumentoPorTipoDocumento*,

dependiendo de lo que se requiera según las especificaciones del caso de uso (figura 26).

Una vez que se obtiene el contador, se genera la nomenclatura del documento. Nuevamente la clase *NomenclaturaService* que crea *UtilFactory*, es un auxiliar para generar la nomenclatura requerida. Se tiene entonces la información necesaria para crear el documento: se ejecuta el *hook method generarD* que regresa el objeto *documento* de tipo *Documento*.

Finalmente, *documento* se agrega a la lista de documentos de *gestion* y se persiste.

Las colaboraciones para *generarD* se muestran en el diagrama de secuencia de la figura 27, y corresponden a la implementación de *GeneradorDocumentosOrg*.

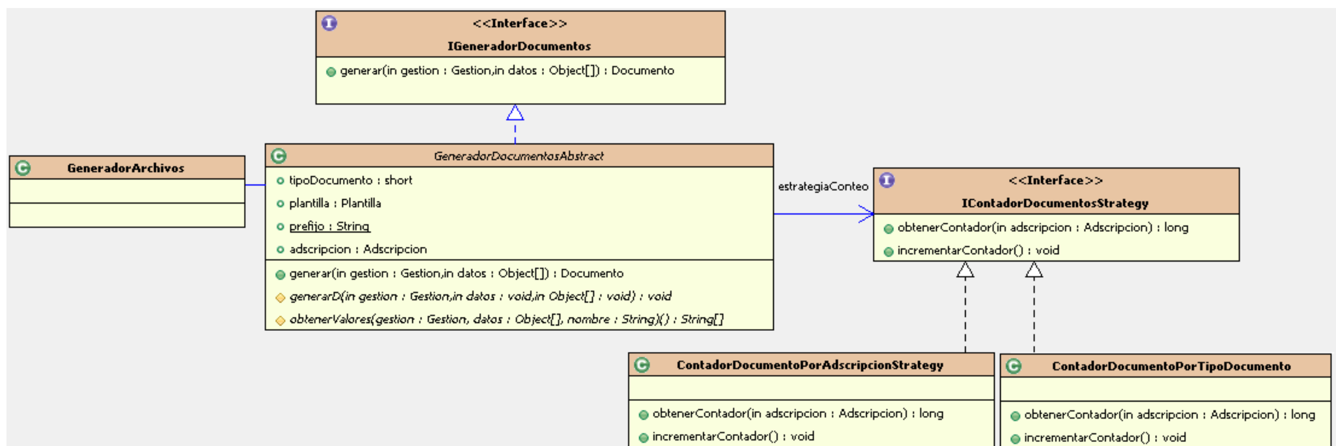


Figura 26: Estructura de la generación de documentos.

gen recibe el mensaje *GenerarD*, recibe como parámetros *gestion*, *datos* y *nombre;* envía el mensaje *obtenerValores* a sí misma, delega en *generadorArchivos* la responsabilidad de genear el archivo, y persiste el documento con la información requerida. Finalmente, *gen* regresa el documento generado.

Otro aspecto interesante en este caso de uso, es que se requiere probar las API's para generar físicamente el archivo de texto tanto para MSOfice Word y OpenOfice Text. Para este problema encaja muy bien como solución el patrón de diseño *Variaciones protegidas,* y, en un sentido más amplio, puede concebirse como un *Adaptador* [GHJV95] para proteger la generación de archivos del cambio. Puesto que la implementación es suceptible de modificaciones, se envuelven las clases generadoras en una interfaz, utilizando polimorfismo, y se crean las dos implementaciones (o posibilitar implementaciones futuras). De esa manera, el diseño queda poco acoplado (ver figura 28).

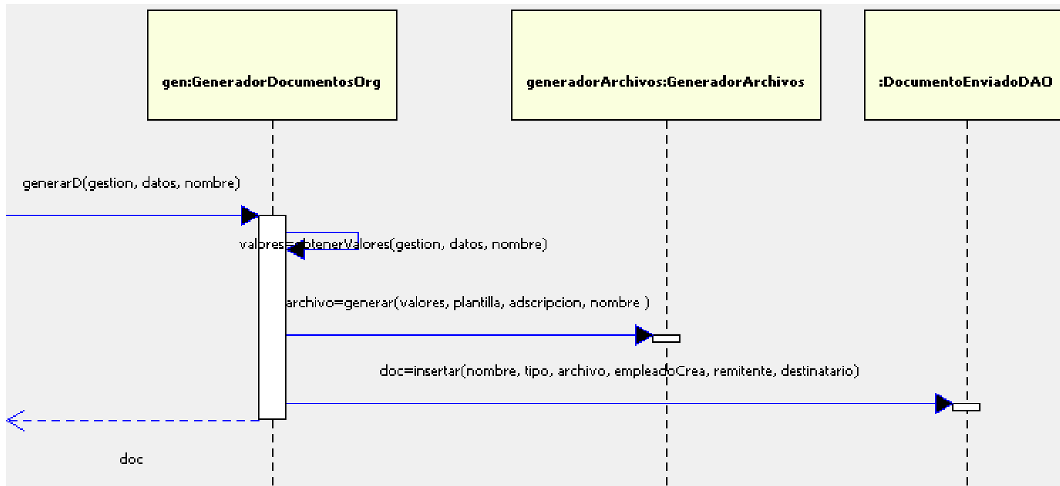


Figura 27: Diagrama de secuencia para el método ancla generarD.

En el diagrama de clases de la figura 28 se observa que *GeneradorArchivos* tiene asociada la interfaz *IGeneradorArchivosTextAdapter*, y define el método *generar.*

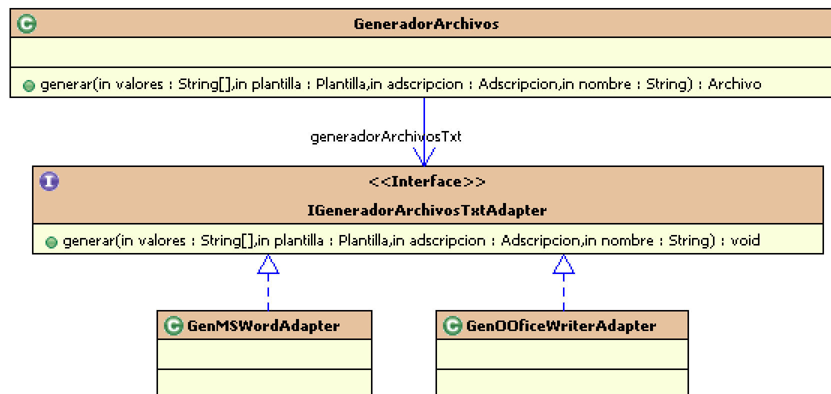


Figura 28: Diagrama de clases para generar archivos.

El diagrama de la figura 29 se observan las colaboraciones para generar un archivo. Primero, se obtiene la ruta de dónde se guardará el archivo, se crea e inserta el archivo y, finalmente, se genera el archivo de texto.

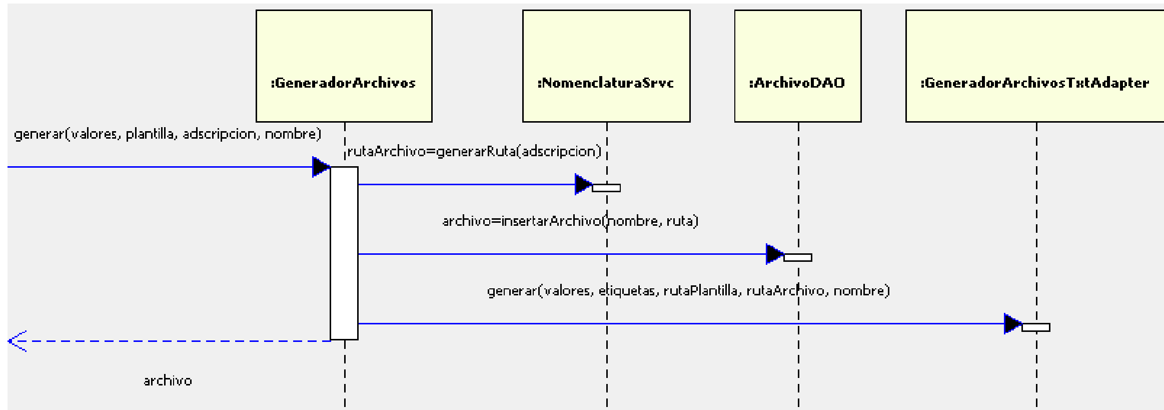


Figura 29: Diagrama de secuencia para la generación de archivos.

Los diseños y colaboraciones anteriores ayudaron a conocer mejor la estructura de los documentos: las asociaciones entre las clases y atributos En el diagrama de la figura 30 se observa que uno o más documentos están asociados a una gestión y uno o más archivos están asociados a un documento. Por su parte, *Documento* es un tipo abstracto, cuyas especializaciones están dadas por *DocumentoEnviado* y *DocumentoInterno*.

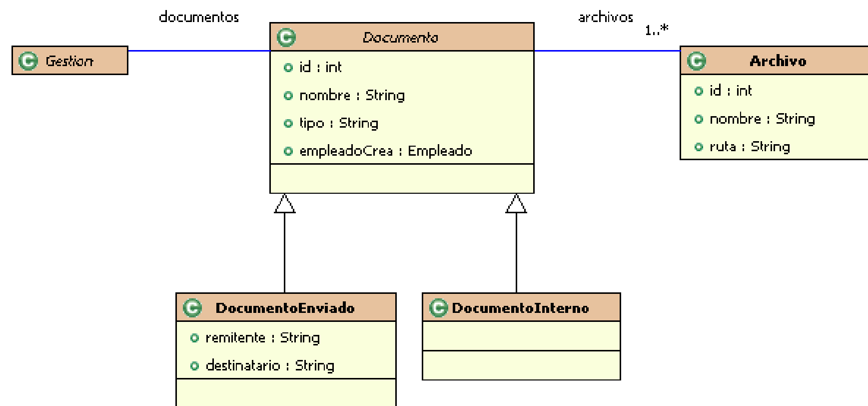


Figura 30: Estructura de los documentos.

Seguridad

En el apartado *seguridad* de las especificaciones suplementarias (anexo 3), se definieron algunas reglas generales de seguridad con que debe contar el sistema. Este es un tema de mucha relevancia para la arquitectura puesto que el servicio de seguridad debió ser lo suficientemente robusto para satisfacer las especificaciones, ademas, debe ser estable, creciente, y aplicable para todas las gestiones del sistema.

El usuario requiere autorización para:

- autenticarse en el sistema
- realizar una operación específica
- modificar sólo gestiones que se generan de acuerdo a su adscripción
- realizar operaciones si un expediente le pertenece (sólo para el caso de Visitadurías)
- realizar operaciones de administración.

El modelado de la figura 31 surgió al analizar el problema. El empleado (usuario) debe pertenecer a un grupo de usuarios, cada grupo debe tener permisos para realizar operaciones en el sistema. Así, Empleado pertenece a un Grupo y cuenta con diversos Permisos. Adicionalmente, las operaciones *tienePermiso* y *perteneceAGrupo* son necesarias para verificar si el Empleado cuenta con un permiso específico y si pertenece a un grupo respectivamente.

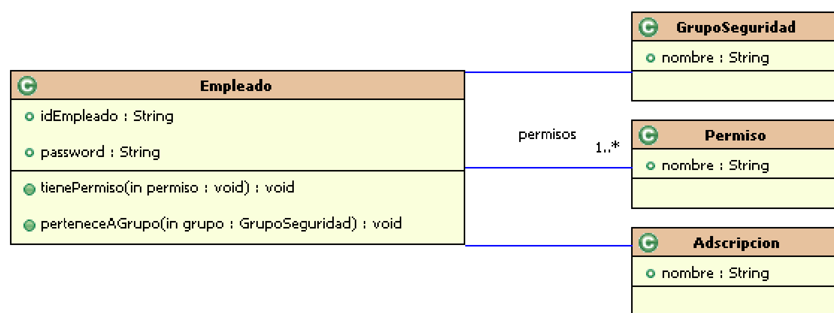


Figura 31: La clase Empleado y sus asociaciones.

Las bases para el servicio de seguridad estuvieron dadas en el modelado de la figura 31, sin embargo, si se hubiese comenzado a codificar tan sólo con las operaciones definidas en él, el resultado habría sido tener un alto acoplamiento en el código del cliente con respecto a clases como *Empleado*, *Adscripcion* y *GrupoSeguridad*, baja cohesión y un esquema frágil. Es por ello que requerí de un esfuerzo de análisis mayor, de no basarme únicamente en los objetos del dominio y de utilizar algunos patrones de diseño que ofrecieran una solución adecuada al problema.

De acuerdo a las especificaciones, se requirieron al menos tres estrategias de seguridad: aquella que verifica el permiso, la que verifica la adscripción y la que verifica la pertenencia del expediente.

El diagrama de la figura 32 muestra la aplicación de los patrones de diseño *Strategy* y *Composite* para este problema.

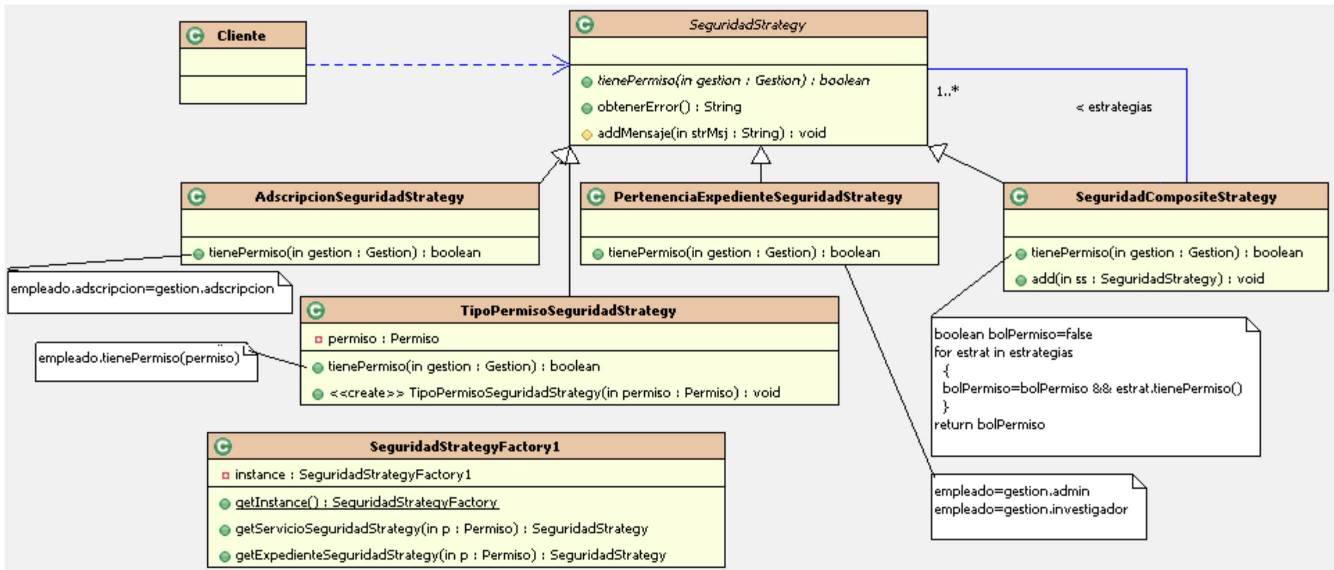


Figura 32: Estrategias de seguridad.

SeguridadStrategy es la interfaz (por conveniencia, es en realidad una clase abstracta) que define la operación necesaria para conocer si un Empleado tiene permiso bajo una situación dada. Esa situación la define concretamente la gestión a la cual se le realiza alguna operación, entonces, una composición de estrategias tendrá que aplicar a dicha gestión (más adelante explicaré cómo se aplica para el caso de expedientes de queja).

Lo que se requirió verificar son algunos datos como la **adscripción**, el **permiso** para realizar una operación, o si el Expediente le **pertenece** al empleado que requiere realizar alguna operación sobre él. *gestion* es la clase que posee esa información (figura 16), por ello, es parámetro de *tienePermiso*, que es el método responsable de saber si alguna estrategia de seguridad ha pasado o no la verificación.

SeguridadCompositeStrategy implementa la interfaz *SeguridadStrategy*, y tiene como atributo una colección de estrategias de seguridad (*estrategias*) e implementa el método *tienePermiso*, el cual prueba por cada estrategia en la colección, su respectivo método *tienePermiso*, y regresa un valor booleano, resultado final que autoriza o deniega un acceso.

AdscripcionSeguridadStrategy, también implementa la interfaz *SeguridadStrategy*, y es la estrategia de seguridad que sirve para probar si la adscripción a la cual pertenece una gestión es la misma que la

adscripción a la cual pertenece el empleado.

TipoPermisoSeguridadStrategy es otra estrategia de seguridad, y con la implementación del método *tienePermiso* prueba si el empleado tiene un permiso en especial.

PertenenciaExpedienteSeguridadStrategy es la estrategia de seguridad aplicable sólo a gestiones de tipo Expediente de queja (o Queja). En el método *tienePermiso* se prueba si el empleado es Administrador o Investigador del Expediente.

Para la estrategia *TipoPermisoSeguridadStrategy*, la semántica del nombre del método *tienePermiso* corresponde exactamente a lo que realiza la operación: evaluar si el empleado tiene un permiso. En cambio, para las demás estrategias, la semántica se debe entender como: "tiene permiso para". Por ejemplo, para *AdscripcionSeguridadStrategy*, debe entenderse "si un empleado tiene permiso por adscripción para acceder a una gestión".

Con este diseño (figura 32), se puede crear tantas nuevas estrategias de seguridad como lo requiera una gestión. De esta manera, si Servicio requiriera una nueva estrategia de seguridad para que un Empleado pueda acceder a él, tan solo se crea la nueva estrategia y se agrega a la composición de estrategias del método *getServicioSeguridadStrategy* de la clase *SeguridadStrategyFactory*, que es la clase creadora (patrones *Factory* y *Singleton*). Más aún, si se crean nuevas gestiones (para el Programa de Defensa, por ejemplo), únicamente hay que escribir un método por cada gestión en *SeguridadStrategyFactory* que regrese la composición de estrategias necesarias, aplicables para la respectiva gestión.

El diagrama de la figura 33 muestra las creaciones y colaboraciones para aplicar una estrategia de seguridad compuesta, aplicada a un Expediente.

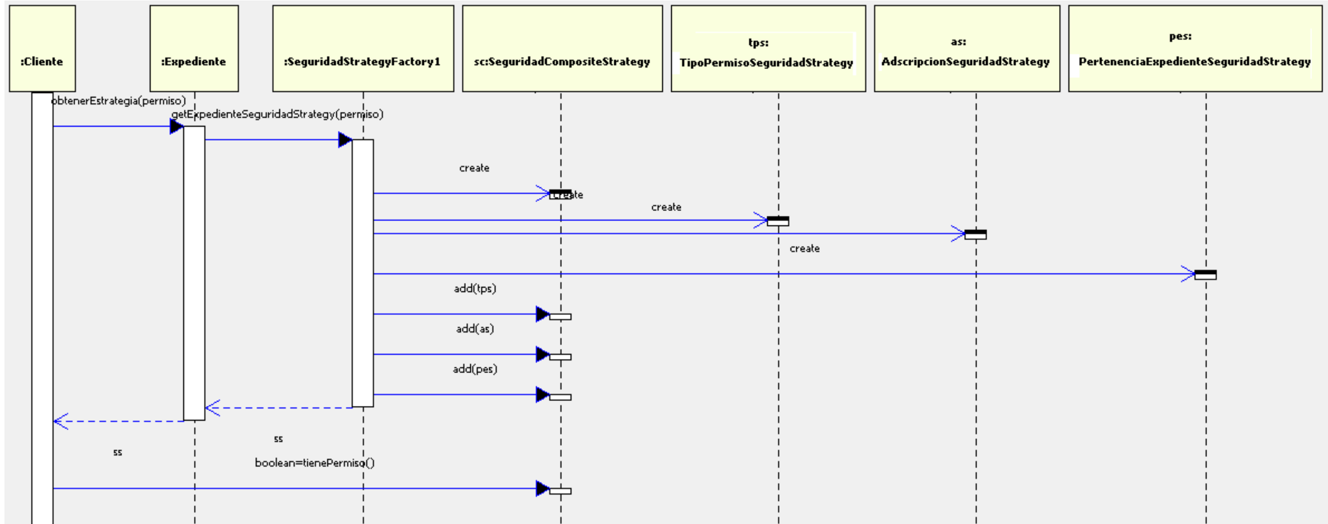


Figura 33: Colaboraciones para la estrategia de seguridad en Expediente.

Un cliente (posiblemente una clase controladora) envía el mensaje *obtenerEstrategia* a *Expediente* con *permiso* como parámetro. *Expediente* es responsable de obtener la estrategia correspondiente para verificar si el cliente puede realizar o no alguna operación sobre él. Para ello, delega a *SeguridadStrategyFactory* la responsabilidad de crear las estrategias.

Las estrategias que requiere *Expediente* son: *TipoPermisoSeguridadStrategy*, *AdscripcionSeguridadStrategy* y *PertenenciaExpedienteSeguridadStrategy*. Primero, *SeguridadStrategyFactory* crea la estrategia de composición (*sc*) y enseguida las demás estrategias (*tps*, *as*, *pes*), las agrega a *sc* y devuelve *sc* a *Expediente*. *Expediente* regresa la estrategia al Cliente. Finalmente, el cliente manda el mensaje *tienePermiso* a la estrategia, la cual devuelve el valor booleano que expresa si el cliente tiene permiso o no para realizar una operación dada.

Para la gestión *Servicio*, las colaboraciones para aplicar la estrategia de seguridad son análogas a la de *Expediente*, la diferencia radica en que para *Servicio, SeguridadStrategyFactory* no crea a *PertenenciaExpedienteSeguridadStrategy*, pues esta estrategia no aplica para esta gestión. De esa forma, cualquier gestión puede tener un conjunto de estrategias de seguridad que apliquen para cuando un usuario realice alguna operación sobre ella, tan sólo basta definir un método en *SeguridadStrategyFactory* que cree y devuelva las estrategias necesarias.

Pruebas e implementación para Generar documentos

Como ya mencioné anteriormente, el caso de uso *Generar documentos* es de gran relevancia para la CDHDF, por lo que requirió un esfuerzo de diseño mayor. A continuación describiré el mapeo a código del diseño para este caso de uso.

El desarrollo dirigido por puebas es una excelente práctica promovida en la programación extrema [Beck03], pues aclara para el programador, el detalle de cómo deben ser implementadas las interfaces y de cómo son los comportamientos de los métodos, además de brindar satisfacción y tranquilidad al momento de realizar cambios en el diseño o por refactorización.

El procedimiento para implementar las clases e interfaces fue el siguiente: primero escribí un método en una prueba de unidad correspondiente a la prueba del método de la clase a implementar. Corrí la prueba. La prueba falló. Escribí el código del método tal que pasara la prueba. Corrí la prueba. La prueba pasó exitosamente. Escribí una prueba para el siguiente método a probar y continué con los pasos realizados anteriormente hasta terminar con la implementación.

El orden de implementación fue: de las clases menos acopladas a las más acopladas[8].

En el listado 1 se muestra el código de las pruebas de unidad para el método *generar* de la clase *GeneradorArchivos*. *testGenerar* crea los objetos necesarios para la realización de la prueba. Se envía el mensaje *generar* a *GeneradorArchivos* para obtener un *Archivo*. Finalmente, se verifica que el archivo no sea nulo, que contenga el nombre especificado, y que exista una ruta que generada.

```
package test.java.domain.cdh.documentos.generador;

import static org.junit.Assert.*;

import main.java.cdh.domain.cdh.documentos.Archivo;
import main.java.cdh.domain.cdh.documentos.generador.GeneradorArchivos;
import main.java.cdh.domain.cdh.documentos.generador.Plantilla;
import main.java.cdh.domain.cdh.org.Adscripcion;

import org.junit.Test;

public class GeneradorArchivoTest {

	GeneradorArchivos gen = new GeneradorArchivos();

	@Test
	public void testGenerar() {
		Plantilla p = new Plantilla();
		Adscripcion ads = new Adscripcion("Primera visitaduría");
		short idAds = 1;
		ads.setId(idAds);
		String nombre = "nArchivo";
		String[] vals = {"", ""};
```

8 Ésta es una forma muy sencilla de comenzar a mapear el diseño en el código, tal que haya pocos errores de compilación desde el principio y que se clarifique las asociaciones y dependencias existentes [Larman05].

```
                Archivo a = gen.generar(p, ads, nombre, vals);
                assertNotNull(a);
                assertEquals(nombre, a.getNombre());
                assertNotSame("", a.getRuta());
        }
}
```

Listado 1.

En el listado 2 se muestra la creación de los generadores de archivos de texto a través del patrón de diseño *factory* que crea una instancia de sí mismo usando un *singleton*. *GeneradorArchivosTxtFactory* resuelve el problema de la creación del generador de archivos de texto y provee el generador adecuado para probar las dos API's de generación de archivos de texto especificadas en el caso de uso y en los requerimientos especiales.

```
package main.java.cdh.domain.cdh.documentos.generador;

public class GeneradorArchivosTxtFactory {
        private static GeneradorArchivosTxtFactory instance = null;
        public static int OOTEXT_ADAPTER = 1;
        public static int MSWORD_ADAPTER = 2;

        public static GeneradorArchivosTxtFactory getInstance() {
                if(instance == null) {
                        instance = new GeneradorArchivosTxtFactory();
                }
                return instance;
        }

        public IGeneradorArchivosTxtAdapter getGeneradorArchivosTxt(int adapter) {
                IGeneradorArchivosTxtAdapter genArchTxt = null;
                switch(adapter) {
                case 1:
                        genArchTxt = new GenOOWriterAdapter();
                        break;
                case 2:
                        genArchTxt = new GenMSWordAdapter();
                        break;
                }
                return genArchTxt;
        }
}
```

Listado 2.

En el listado 3 se muestra el código de la clase *GeneradorArchivos*. Tiene como atributo un generador de archivos de texto. En el método *generar* se realizan los mensajes del diagrama de secuencia de la figura 25. Las clases de utiliería ayudan a obtener la ruta donde se almacena el archivo. Por medio de inyección de dependencia manual [Hibernate09], se crea una instancia de *ArchivoDAO,* que inserta un *Archivo* en el almacenamiento persistente al enviar el mensaje *insertarArchivo* y regresa el *Archivo.* Finalmente, se envía el mensaje *generar* al generador de archivos de texto para crear el archivo de texto.

```
package main.java.cdh.domain.cdh.documentos.generador;
```

```java
import main.java.cdh.dao.ArchivoDAO;
import main.java.cdh.domain.cdh.documentos.Archivo;
import main.java.cdh.domain.cdh.org.Adscripcion;
import main.java.cdh.util.UtilFactory;

public class GeneradorArchivos {

        private IGeneradorArchivosTxtAdapter generadorArchivosTxt = GeneradorArchivosTxtFactory.
getInstance().getGeneradorArchivosTxt(GeneradorArchivosTxtFactory.OOTEXT_ADAPTER);

        public Archivo generar(Plantilla p, Adscripcion ads, String nombre, String[] vals) {
                String ruta = UtilFactory.getInstance().getNomenclaturaService().generarRutaArchivo(ads);

                ArchivoDAO aDAO = new ArchivoDAO();
                Archivo archivo = aDAO.insertarArchivo(nombre, ruta);

                generadorArchivosTxt.generar(p, ads, nombre, vals);

                return archivo;
        }
}
```

Listado 3.

En el listado 4 se observa el código para probar los métodos *generar* para cada clase concreta de *GeneradorDocumentosAbstract*. Por cada clase concreta habrá un método de prueba, sin embargo, por razones de espacio, en el listado muestro sólo los métodos de prueba para generar un *Acta circunstanciada* y para *Medidas precautorias*. Para esta prueba, es necesario un método de configuración (*setUp*) en donde se crean los objetos necesarios para la operación de la prueba. Se verifica que el documento que regreasa el método no sea nulo, que la longitud del nombre del documento sea mayor a cero, que el archivo que se generó no sea nulo y que contenga datos.

```java
package test.java.domain.cdh.documentos.generador;

import static org.junit.Assert.*;

import main.java.cdh.domain.cdh.documentos.Documento;
import main.java.cdh.domain.cdh.documentos.generador.GeneradorActaCircunstanciada;
import main.java.cdh.domain.cdh.documentos.generador.IGeneradorDocumentos;
import main.java.cdh.domain.cdh.documentos.generador.GeneradorMedidasPrecautorias;
import main.java.cdh.domain.cdh.gestion.GestionDefensa;
import main.java.cdh.domain.cdh.org.Adscripcion;
import main.java.cdh.domain.cdh.persona.Empleado;
import main.java.cdh.domain.cdh.persona.Persona;

import org.junit.Test;
import org.junit.Before;

public class GeneradorDocumentosTest {

        private IGeneradorDocumentos genMP = new GeneradorMedidasPrecautorias();
        private IGeneradorDocumentos genAC = new GeneradorActaCircunstanciada();
        private MockGestion gestion = new MockGestion();

        @Before
        public void setUp() throws Exception {
                Empleado e = new Empleado();
                e.setNombre("Craig");
                e.setApPaterno("Larman");
                e.setAdscripcion(new Adscripcion("Architecture dept."));

                Persona p = new Persona();
                p.setNombre("Ivar");
                p.setApPaterno("Jacobson");

                gestion.setTitulo("test");
                gestion.setNarracionHechos("narracion");
                gestion.setEmpleadoRegistra(e);
                gestion.setEmpleadoVisor(e);
```

```java
		gestion.setPeticionario(p);
	}

	@Test
	public void testGenerar_MedidasPrecautorias() {
		String[] datos = {	"contacto",
					"organismo",
					"plazo",
					"remitente",
					"destinatario"
				};

		Documento doc = genMP.generar(gestion, datos);
		assertNotNull(doc);
		assertTrue(doc.getNombre().length()>0);
		assertNotNull(doc.getArchivo());
		assertTrue(doc.getArchivo().getNombre().length()>0);
		assertTrue(doc.getArchivo().getRuta().length()>0);
	}

	@Test
	public void testGenerar_ActaCircunstanciada() {
		String[] datos = {	"",
					"",
					"",
					"",
					""						};

		Documento doc = genAC.generar(gestion, datos);
		assertNotNull(doc);
		assertTrue(doc.getNombre().length()>0);
		assertNotNull(doc.getArchivo());
		assertTrue(doc.getArchivo().getNombre().length()>0);
		assertTrue(doc.getArchivo().getRuta().length()>0);
	}

	private class MockGestion extends GestionDefensa {}
}
```

Listado 4.

El listado 5 muestra la interfaz *IGeneradorDocumentos* y el listado 6 muestra el código de *GeneradorDocumentosAbstract* que implementa tan sólo el método *generar* de dicha interfaz y define sus atributos. Los dos métodos restantes, son definidos como abstractos y se implementan en subclases. *generar* es el método plantilla (*template method)* responsable de generar documentos, por lo que regresa un tipo *Documento.* El cuerpo del método genera el nombre del documento y el archivo, y en seguida delega la responsabilidad de generar el documento al método ancla *generarD* (*hook method*), el cual es implementado en las clases que deriven de *GeneradorDocumentosAbstract*. Revisando el diagrama de secuencia de la figura 25, se nota la ausencia de la llamada a *estrategiaConteo* a *incrementarContador.* Sin embargo, es necesario incrementar el contador para que el siguiente documento tome el número correspondiente. En el método *generar* del listado 6 se observa la llamada al método *incrementarContador* de *IcontadorDocumentosStrategy.*

```java
package main.java.cdh.domain.cdh.documentos.generador;

import main.java.cdh.domain.cdh.documentos.Documento;
import main.java.cdh.domain.cdh.gestion.Gestion;

public interface IGeneradorDocumentos {

	public Documento generar(Gestion gestion, Object[] datos);
}
```

Listado 5.

```java
package main.java.cdh.domain.cdh.documentos.generador;

import main.java.cdh.domain.cdh.documentos.Archivo;
import main.java.cdh.domain.cdh.documentos.Documento;
import main.java.cdh.domain.cdh.gestion.Gestion;
import main.java.cdh.domain.cdh.org.Adscripcion;
import main.java.cdh.util.NomenclaturaService;
import main.java.cdh.util.UtilFactory;

public abstract class GeneradorDocumentosAbstract implements IGeneradorDocumentos{

	protected short tipoDocumento = 0;
	protected String prefijo = "";
	protected Plantilla plantilla = null;
	protected Adscripcion adscripcion = null;
	protected IContadorDocumentosStrategy estrategiaConteo = null;
	protected GeneradorArchivos genArchivos = new GeneradorArchivos();

	//template method
	public Documento generar(Gestion gestion, Object[] datos){
		//generar el nombre del documento
		long cont = estrategiaConteo.obtenerContador(adscripcion);
		NomenclaturaService nmcl = UtilFactory.getInstance().getNomenclaturaService();
		String nombre = nmcl.obtenerNomenclaturaDocumento(cont);

		//generar el archivo
		String[] vals = obtenerValores(gestion, nombre, datos);
		Archivo archivo = genArchivos.generar(plantilla, adscripcion, nombre, vals);

		//generar el documento (hook method)
		Documento documento = generarD(gestion, nombre, archivo, datos);

		estrategiaConteo.incrementarContador();

		return documento;
	}

	protected abstract Documento generarD(Gestion gestion, String nombre, Archivo archivo, Object[] datos);
	protected abstract String[] obtenerValores(Gestion gestion, String nombre, Object[] datos);
}
```

Listado 6.

Los listados 7 y 8 muestran las subclases *GeneradorDocumentosOrg* y *GeneradorDocumentosInternos* que derivan de *GeneradorDocumentosAbstract*. Lo interesante en estas clases es que definen el método ancla *generarD*, que básicamente se encargan de crear e insertar un documento en el almacenamiento persistente, cada una, con sus datos correspondientes. El código para *GeneradorDocumentosPeticionario* es análogo y no se muestra en los listados de código.

```java
package main.java.cdh.domain.cdh.documentos.generador;

import main.java.cdh.dao.DocumentoEnviadoDAO;
import main.java.cdh.domain.cdh.documentos.Archivo;
import main.java.cdh.domain.cdh.documentos.Documento;
import main.java.cdh.domain.cdh.documentos.DocumentoEnviado;
import main.java.cdh.domain.cdh.gestion.Gestion;

public abstract class GeneradorDocumentosOrg extends
		GeneradorDocumentosAbstract {

	@Override
	protected Documento generarD(Gestion gestion, String nombre, Archivo archivo, Object[] datos) {
		//Almacenar el documento en la BD
		DocumentoEnviadoDAO docDAO = new DocumentoEnviadoDAO();
		DocumentoEnviado doc = docDAO.insertar(nombre, String.valueOf(tipoDocumento),
				archivo, gestion.getEmpleadoVisor(), (String)datos[3], (String)datos[4]);

		return doc;
	}
}
```

Listado 7.

```java
package main.java.cdh.domain.cdh.documentos.generador;

import main.java.cdh.dao.DocumentoInternoDAO;
import main.java.cdh.domain.cdh.documentos.Archivo;
import main.java.cdh.domain.cdh.documentos.Documento;
import main.java.cdh.domain.cdh.documentos.DocumentoInterno;
import main.java.cdh.domain.cdh.gestion.Gestion;

public abstract class GeneradorDocumentosInternos extends
                GeneradorDocumentosAbstract {

        @Override
        protected Documento generarD(Gestion gestion, String nombre, Archivo archivo, Object[] datos) {
                //Almacenar el documento en la BD
                DocumentoInternoDAO docDAO = new DocumentoInternoDAO();
                DocumentoInterno doc = docDAO.insertar(nombre, String.valueOf(tipoDocumento),
                                archivo, gestion.getEmpleadoVisor());

                return doc;
        }

}
```

Listado 8.

Los listados 9 y 10 muestran la implementación para los generadores (clases concretas) *GeneradorMedidasPrecautorias* y *GeneradorActaCircunstanciada*. Cada clase concreta es un generador, y debe tener valores configurados que expresen el tipo de documento que van a generar, el prefijo con el cual se va a generar el nombre del documento, la plantilla que debe utilizar, la adscripción y la estrategia de conteo. Además, cada generador implementa el método *obtenerValores*, responsable de obtener los valores que se insertan en el archivo de texto. En adelante, si nuevos tipos de documentos son requeridos, sólo se tiene que crear y configurar una clase generadora que implemente algún generador de documentos (para organismos, para peticionarios o internos).

```java
package main.java.cdh.domain.cdh.documentos.generador;

import java.text.SimpleDateFormat;
import java.util.Date;

import main.java.cdh.domain.cdh.gestion.Gestion;
import main.java.cdh.domain.cdh.gestion.GestionDefensa;
import main.java.cdh.domain.cdh.org.Adscripcion;

public class GeneradorMedidasPrecautorias extends GeneradorDocumentosOrg {

        public GeneradorMedidasPrecautorias() {
                tipoDocumento = 1;
                prefijo = "MP";
                plantilla = null;
                short shIdAds = 1;
                adscripcion = new Adscripcion(shIdAds);
                estrategiaConteo = new ContadorDocumentoPorAdscripcionStrategy();
        }

        @Override
        protected String[] obtenerValores(Gestion gestion, String nombre, Object[] datos) {
                Date date = new Date();
                SimpleDateFormat sdf = new SimpleDateFormat("yy");
                String year = sdf.format(date);
                sdf.applyPattern("dd-MM-yyyy");
                String strDate = sdf.format(date);

                GestionDefensa g = (GestionDefensa)gestion;

                String[] valores = {
```

```
                        g.getEmpleadoVisor().getAdscripcion().getNombre(),                 //Adscripción del 
Usuario
                        year,            //año,
                        (String)datos[0],        //Contacto,
                        strDate,         //fecha,
                        nombre,          //Número de oficio,
                        g.getNarracionHechos(),  //narración de hechos,
                        (String)datos[1],        //Organismo,
                        g.getPeticionario().getNombreCompleto(),  //peticionario
                        (String)datos[2],        //plazo de respuesta en horas
                        g.getTitulo(),   //título de la Gestión,
                        g.getEmpleadoVisor().getNombreCompleto()  //Usuario,
                };

                return valores;
        }
}
```

Listado 9.

```
package main.java.cdh.domain.cdh.documentos.generador;

import java.text.SimpleDateFormat;
import java.util.Date;

import main.java.cdh.domain.cdh.gestion.Gestion;
import main.java.cdh.domain.cdh.gestion.GestionDefensa;
import main.java.cdh.domain.cdh.org.Adscripcion;

public class GeneradorActaCircunstanciada extends GeneradorDocumentosInternos{

        public GeneradorActaCircunstanciada() {
                tipoDocumento = 2;
                prefijo = "AC";
                plantilla = null;
                short shIdAds = 1;
                adscripcion = new Adscripcion(shIdAds);
                estrategiaConteo = new ContadorDocumentoPorTipoDocumento();
        }

        @Override
        public String[] obtenerValores(Gestion gestion, String nombre, Object[] datos){
                Date date = new Date();
                SimpleDateFormat sdf = new SimpleDateFormat("yy");
                String year = sdf.format(date);
                sdf.applyPattern("dd-MM-yyyy");
                String strDate = sdf.format(date);
                sdf.applyPattern("HH");
                String strHora = sdf.format(date);

                GestionDefensa g = (GestionDefensa)gestion;

                String[] valores = {
                        g.getEmpleadoVisor().getAdscripcion().getNombre(), //Adscripción
                        year,            //año,
                        strDate,         //fecha,
                        strHora,         //hora,
                        g.getPeticionario().getNombreCompleto(),  //peticionario
                        g.getTitulo(),   //título de la Gestión,
                        g.getEmpleadoVisor().getNombreCompleto()  //Usuario,
                };
                return valores;
        }
}
```

Listado 10.

En los listados 11 y 12 se observan las interfaces que envuelven la estrategia para generar el número de documento y el generador de archivos de texto, respectivamente.

```
package main.java.cdh.domain.cdh.documentos.generador;

import main.java.cdh.domain.cdh.org.Adscripcion;
```

```java
public interface IContadorDocumentosStrategy {
	public long obtenerContador(Adscripcion adscripcion);
	public void incrementarContador();
}
```

Listado 11.

```java
package main.java.cdh.domain.cdh.documentos.generador;

import main.java.cdh.domain.cdh.org.Adscripcion;

public interface IGeneradorArchivosTxtAdapter {
	public void generar(Plantilla p, Adscripcion ads, String nombre, String[] vals);
}
```

Listado 12.

Resultados y aportaciones

El Sistema Integral de Gestión de Información se concibió como una solución para enfrentar los retos de extensión y mejora de la gestión de la información en la CDHDF.

La fase de Elaboración del Proceso Unificado es aquella en donde se hace un análisis profundo de los requerimientos que sean relevantes para la arquitectura, que ayuden a mitigar los riesgos en el proyecto, que se enfoquen en los flujos o escenarios más importantes, y que sirvan para generar una arquitectura estable.

Mi participación profesional en esta fase, y desde la fase de inicio, tomando los roles de arquitecto, analista, diseñador y programador, tuvo como objetivo y resultado asegurar un buen comienzo y desarrollo del proyecto.

Al término de la fase de Elaboración, el Proyecto tuvo las siguientes características:

- se creó la base de una arquitectura estable
- se detallaron la mayoría de los casos de uso
- se realizaron los casos de uso más relevantes
- el análisis y buen diseño del caso de uso *Generar documentos,* hace flexible la creación de documentos y hará posible la extensión del framework en caso de ser necesario
- las bases para la seguridad en el SIIGESI están bien cimentadas y su aplicación es de gran alcance
- se cuenta con una buena estructura de clases para las gestiones.

Se requirió que el SIIGESI fuera flexible y extendible. Esto lo llevé a cabo mediante un buen diseño para algunos escenarios; con una codificación clara, simple y refactorizada; con comentarios en el código para hacerlo más comprensible al programador que en algún momento requiera realizar cambios; y con pruebas unitarias para brindar seguridad de que la funcionalidad se mantenga aún cuando haya modificaciones en el código fuente.

Sin embargo, todas aquellas consideraciones no suplen la necesidad de realizar un análisis de requerimientos y modelado del negocio, por lo que, aún cuando se hayan tomado medidas, faltó analizar los requerimientos para el Programa de Promoción. Aunque se intentó minimizar el riesgo, es

muy probable que haya algunas modificaciones no previstas que afecten de alguna forma el diseño actual y que espero, sean de poco impacto.

En su debido momento, hablé con mi jefe sobre la necesidad de capturar requerimientos para el Programa de Promoción y así tener un mejor panorama para lo que pretende el SIIGESI: conjuntar en un sistema las distintas áreas y gestiones de la Comisión. Pero debido a los trámites internos y poca disposición por parte de mi jefe para iniciar las sesiones con la gente del negocio (a pesar de que el equipo de desarrollo contaba con un enfoque ágil, en la cual la interacción con el cliente es prioritaria), no se realizó esta disciplina[9].

El SIIGESI es un sistema robusto que satisface la necesidad prioritaria de la Comisión de atender a las víctimas de las violaciones a los Derechos Humanos, y que permitirá la integración de los demás programas institucionales.

---

9 Todavía no se ha realizado el módulo correspondiente al Programa de Promoción. La actual administración termina su periodo de gestión en octubre de 2009, y no se nota mucha voluntad de las partes involucradas por realizar dicho módulo.

Conclusiones

El SIIGESI, por su naturaleza y relevancia, tuvo la virtud de ser el primer desarrollo de la CDHDF en seguir un proceso de desarrollo de software, siendo el Proceso Unificado la metodología que propuse y que se aplicó. Al seguir dicha metodología, las más grandes dudas que tuvimos fue en lo relativo a los requerimientos y la administración del proyecto, siendo éstos aspectos que, si bien me brindaron experiencia, pueden ser mejorados, sobre todo, en lo relativo a las sesiones con la gente del negocio.

El establecimiento de la arquitectura en las fases de inicio y elaboración, formó la base para construcción del sistema tomando en cuenta los requerimientos capturados. Actualmente, debido a las prioridades establecidas por los directivos, el Programa de Promoción por los derechos humanos no se ha integrado al SIIGESI. Sin embargo, se pretende que para la segunda mitad de 2010 ya forme parte del sistema, hasta entonces, y habiendo capturado los nuevos requerimientos, se podrá poner a prueba la arquitectura actual en medida que permita la integración de los nuevos módulos sin cambios trascendentales desde un punto de vista de la estructura base del sistema.

En el proceso de desarrollo, los trabajos previos de algunos científicos de la computación fueron importantes para la aplicación del mismo, por ejemplo, algunos de los patrones de diseño, descritos por *La banda de los 4* (*Gang-of-four*) en el libro *Desing* patterns desde 1994, que me sirvieron como base en el diseño; el proceso de desarrollo de software *Proceso Unificado,* descrito por primera vez en el libro *The Unified Software Development Process* por Ivar Jacobson, Grady Booch y James Rumbaugh en 1999, que fue la base que dirigió el proyecto, siendo éste un proceso dirigido por los casos de uso, influenció en gran medida el diseño y la programación del sistema; y finalmente, algunas prácticas XP (como la programación en parejas, pruebas de unidad, etc) descritas por Kent Beck y el manifiesto ágil firmado por varios profesionales del software en 2001.

Bibliografía

AManifiesto01 2001. *Manifesto for Agile Software Development*. agilemanifesto.org

Beck04 Beck, K. 2004. *Extreme Programming Explained–Embrace Change*. 2a. ed. Addison-Wesley.

CDHDF06-08 2008. *Ley y Reglamento Interno de la Comisión de Derechos Humanos del Distrito Federal*. Serie documentos oficiales. Comision de Derechos Humanos del Distrito Federal.

CDHDF09a 2009. *Informe anual 2008*. www.cdhdf.org.mx.

CDHDF10-08 2008. *Investigación de violaciones a los derechos humanos. Presupuestos y manual de métodos y procedimientos*. Serie documentos oficiales. Comision de Derechos Humanos del Distrito Federal.

GHJV95 Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hibernate09 2009. *Generic Data Access Objects*. www.hibernate.org

Larman05 Larman, C. 2005. *Applying UML and Patterns*, 2a. ed. Prentice Hall.

Martin04 2004. Martin, Robert C. *UML para programadores Java*.

OMG03a www.omg.org

RJB99 Rumbaugh, J., Jacobson, I., and Booch, G. 1999. *The Unified Software Development Process*. Addison-Wesley.

UML09 www.uml.org

Wiki09a 2009. *Clipper (lenguaje de programación)*. www.wikipedia.org

Anexo 1

Diagramas de casos de uso

Diagrama de casos de uso en la fase de inicio

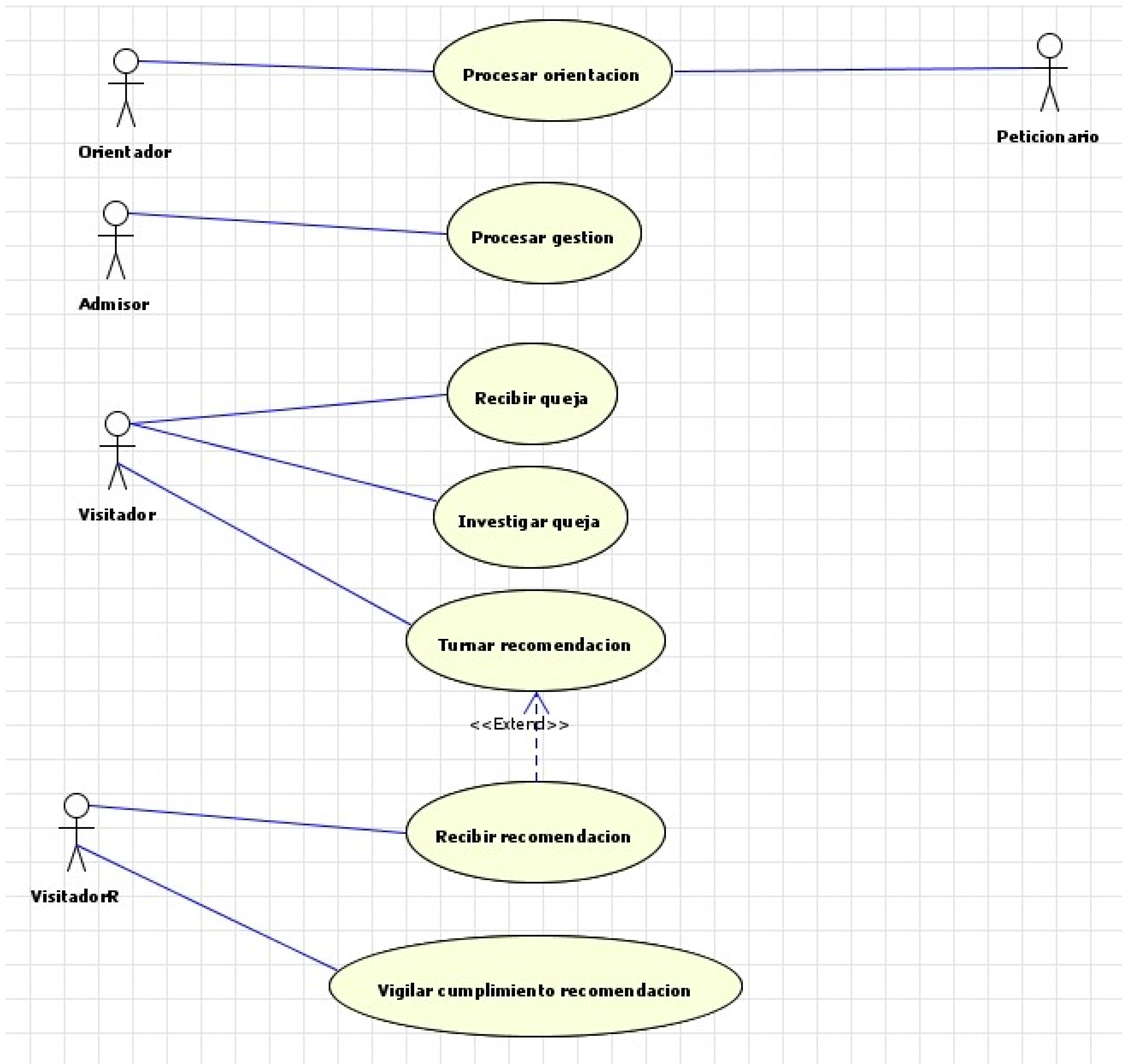
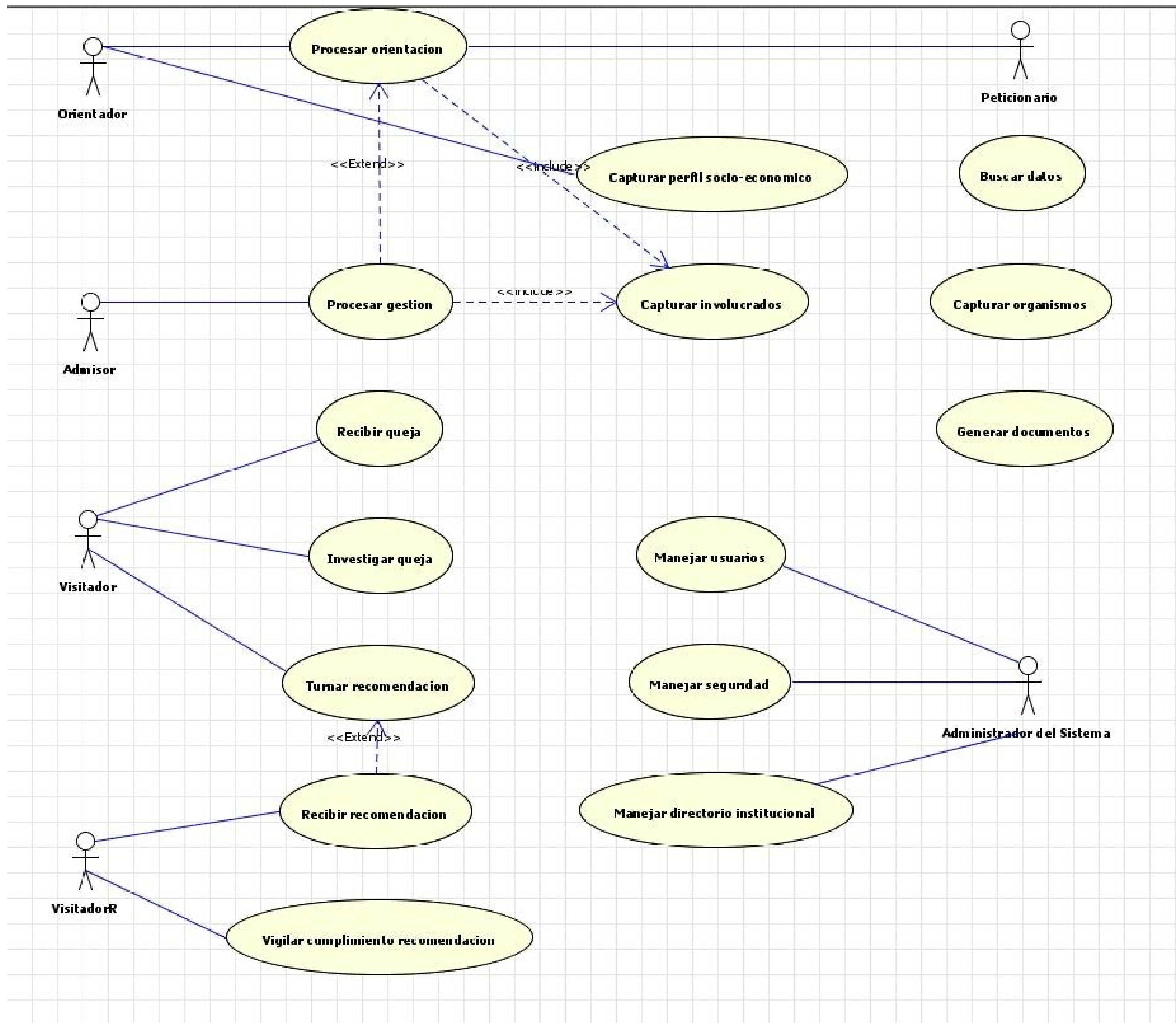


Diagrama de casos de uso en la fase de elaboración



Anexo 2

Casos de uso

Caso de Uso SIIGESI01. Procesar orientación.

Resumen: escenario desde el momento en el que un peticionario se comunica a la CDHDF para exponer un problema, el trato que se le da a dicho problema como gestión, y hasta que se concluye dicha gestión.

Alcance: SIIGESI
Nivel: usuario
Actor principal: Orientador
Intereses del cliente:
– Peticionario: plantea un problema y quiere obtener una respuesta adecuada.
– Orientador: quiere capturar de manera sencilla y eficaz los datos proporcionados por el Peticionario.
– CDHDF: quiere brindar un buen servicio que satisfaga el problema del Peticionario y datos consistentes para la generación de informes.

Precondiciones: El Orientador está identificado y autenticado.
Postcondiciones: Una Gestión Servicio es creada en un Caso con los datos personales del Peticionario, la narración de hehos y el Organismo que presuntamente ha violado los DDHH.

Escenario principal:
1. El Peticionario se comunica con un Orientador telefónica o personalmente.
2. El Orientador genera un Caso.
3. El Sistema genera el título del Caso con la nomenclatura CDHDFENSA-año-número_consecutivo y registra el Caso.
4. El Orientador genera una Gestión Servicio en el Caso.
5. El Orientador captura el tipo de vía de entrada de la Gestión Servicio (telefónica o personal).
6. El Orientador ingresa el Peticionario: Capturar involucrados.
7. El Orientador captura la narración de hechos descrita por el Peticionario.
8. El Orientador captura los lugares donde ocurrieron los hechos.
9. El Orientador ingresa los agraviados: Capturar involucrados.
10. El Orientador captura el Organismo que presuntamente violó los derechos humanos: Capturar organismos.
11. El Orientador captura el servicio que se le brinda al Peticionario: “suplencia de queja”
12. El Orientador genera un documento “Acta circunstanciada”: Generar documentos.
13. El Orientador concluye la Gestión Servicio.

Flujos altenativos:
1a. El Peticionario requiere información sobre la CDHDF.
1. El Orientador da información al Peticionario.
2. El Orientador genera un Caso.
3. El Orientador genera una Gestión Servicio.
4. El Orientador ingresa el Peticionario: Capturar involucrados.
5. El Orientador captura la información requerida por el Peticionario.
6. El Orientador captura el servicio que se le brinda al Peticionario: “Información sobre la CDHDF”.
7. Orientador concluye la Gestión Servicio.

1b. El Peticionario quiere aportar más elementos a una Queja expuesta anteriormente.
1. El Orientador busca el Caso donde se encuentra la Queja: Buscar datos.
2. El Orientador genera una Gestion Servicio en el Caso.
3. El Orientador ingresa el Peticionario: Capturar peticionario .
4. El Orientador captura la narración de hechos descrita por el Peticionario.
5. El Orientador captura el lugar donde ocurrieron los hechos.
6. El Orientador ingresa los agraviados: Capturar involucrados.
7. El Orientador captura el Organismo que presuntamente violó los derechos humanos: Capturar organismos.
8. El Orientador captura el servicio que se le brinda al Peticionario: “Aportación a Expediente de queja”.
9. El Orientador genera un documento “Acta circunstanciada”: Generar documentos.
10. El Orientador concluye la Gestión Servicio.

1c. El Peticionario requiere información sobre el avance de su Queja.
1. El Orientador a el Caso donde se encuentra la Queja.: Buscar datos.
2. El Orientador da información al Peticionario.
3. El Orientador genera una Gestion Servicio en el Caso.
4. El Orientador ingresa el Peticionario: Capturar involucrados.
5. El Orientador captura la información requerida por el Peticionario.
6. El Orientador captura el servicio que se le brinda al Peticionario como: “Curso de queja”
7. El Orientador concluye la Gestión Servicio.

1d. El Peticionario requiere asesoría para formular un escrito.
1. El Orientador orienta al Peticionario.
2. El Orientador busca el Caso donde se encuentra la Queja: Buscar datos.
3. El Orientador genera una Gestion Servicio en el Caso.
4. El Orientador ingresa el Peticionario: Capturar involucrados.
5. El Orientador captura el tipo de asesoría requerida por el Peticionario.
6. El Orientador captura el servicio que se le brinda al Peticionario como: “Asesoría para formular escrito”.
7. El Orientador concluye la Gestión Servicio.

7a. El Peticionario presenta un escrito ante el Orientador.
1. La captura de la narración de hechos se basa en el escrito presentado.
2. El Orientador captura el servicio que se le brinda al Peticionario como: “Revisión de escrito”.

5-12a. El Orientador determina que la gestión amerita un aviso inmediato al Organismo debido a la gravedad del asunto.
1. El Orientador captura el Organismo que presuntamente violó los derechos humanos: Capturar organismos.
2. El Orientador genera un documento de “Medidas Precautorias” para enviarlo al Organismo: Generar documentos.

11. Los elementos descritos por el Peticionario no son suficientes o no son claros.
1. El Orientador captura el servicio que se le brinda al Peticionario como: “suplencia de deficiencia de queja”.

12a. La vía de entrada es “Personal”.
1. El Peticionario firma el Acta circunstanciada.

Requerimientos especiales: Cada vez que se inicie un año, el número consecutivo del caso deberá reiniciarse automáticamente; el Orientador puede interrumpir la captura de los datos en cualquier momento para realizar alguna tarea distinta, por lo que se deberán almacenar los datos para su recuperarlos al reanudar la captura.
Tecnología utilizada:
Frecuencia: continuamente.
Temas abiertos:

Caso de Uso SIIGESI02. Procesar gestión

Resumen: escenario en el cual un Admisor registra a partir de una gestión Servicio, o por iniciativa de la CDHDF, o por otra vía de entrada, un Expediente de queja, una Remisión o un Expedientillo.

Alcance: SIIGESI
Nivel: usuario
Actor principal: Admisor
Intereses del cliente:
– Admisor: quiere turnar la Queja de manera eficaz.
– Visitador: quiere información correcta y adecuada a su criterio de trabajo, es decir, que el tipo de presunta violación a los derechos humanos competa al tipo de caso de estudio de su visitaduría.

Precondiciones: El admisor está identificado y autenticado.
Postcondiciones: La gestión es identificada con un tipo específico. Los documentos necesarios son creados.

Escenario principal:
1. El Admisor recibe el “Acta circunstanciada” que señala que la Gestión derivada del servicio de Orientación debe investigarse como Queja.
2. El Admisor busca la Gestion del Servicio: Buscar datos.
3. El Admisor lee la narración del Servicio de queja e identifica las personas y organismos involucrados.

4. El Admisor genera apartir de la Gestión del Servicio, una gestión Expediente de queja.
5. El Sistema gerenar el Expediente de queja en el caso.
6. El Admisor genera el título del Expediente con la siguiente nomenclatura:
   CDHDF/oficina/adscripción/queja_colectiva/delegación/año/tipo.número_consecutivo
   donde,

oficina	oficina donde se captura la queja	[df, uo, us, un, up]
adscripción	adscripción a donde se turna la queja	[1=1a visitaduría, 2=2a visitaduría, ...]
queja_colectiva	si el agraviado es una persona o un colectivo	[121=individual, 122=colectiva]
delegación	delegación donde ocurrieron los hechos	[AO=Alvaro Obregon, BJ=Benito Juárez, ...]
tipo	tipo de queja	[d=directa, p=penitenciaria, n=nacional]
consecutivo	número consecutivo de la queja	

7. El Admisor genera los documentos “Carátula de expediente de queja” y “Acuerdo de recepción de queja”: Generar documentos.
8. El Admisor turna el Expediente de queja a la Visitaduría correspondiente.

Flujos altenativos:

1a. El Admisor recibe una queja por escrito (e-mail, mensajería, correo, escrito, fax).
1. El Admisor genera un Caso.
2. El Admisor genera un Expediente de Queja.
3. El Admisor captura la vía de entrada del Expediente de Queja (e-mail, mensajería, correo, escrito, fax).
4. El Orientador ingresa el Peticionario: Capturar peticionario.
5. El Admisor captura la narración de los hechos ocurridos.
6. El Orientador captura los lugares donde ocurrieron los hechos.
7. El Orientador ingresa los agraviados: Capturar agraviados.
8. El Admisor captura las organismos involucrados: Capturar organismos.
9. El Admisor genera el título del Expediente.
10. El Admisor genera los documentos “Carátula de expediente de queja” y “Acuerdo de recepción de queja”: Generar documentos.
11. El Admisor turna el Expediente de queja a la Visitaduría correspondiente.

1b. La Queja se recibe por iniciativa del Visitador para iniciar la investigación de hechos presuntamente violatorios
1. El Admisor recibe el documento “Acuerdo de inicio de investigación de oficio”.
2. El Admisor genera un Caso.
3. El Admisor crea una Queja en el Caso y indicando que se inicia por investigación de oficio.
4. El Sistema registra la Queja y la CDHDF como Peticionario.
5. El Admisor inicia la captura de los datos de la Queja.
   4.1. El Admisor captura la narración de hechos.
   4.2. El Orientador captura los lugares donde ocurrieron los hechos.
   4.3. El Admisor captura los agraviados: Capturar involucrados.
   4.4. El Admisor captura los organismos involucrados: Capturar organismos.
   4.5. El Admisor genera el título del Expediente.
   4.6. El Admisor genera los documentos “Carátula de expediente de queja” y “Acuerdo de recepción de queja”: Generar documentos.
   4.7. El Admisor turna el Expediente de queja a la Visitaduría correspondiente.

1c. La Queja no es competencia de la Comisión.
1. El Admisor crea un Caso.
2. El Admisor crea una gestión de tipo Remisión.
3. El Admisor inicia la captura de los datos de la Remisión.
4. El Admisor genera los documentos “Carátula primaria de remisión” , “Remisión dirigida al peticionario” y “Remisión dirigida a la autoridad”: Generar documentos.
5. El Admisor turna la Queja alguna otra Comisión estatal o nacional que sea competente.

1d. El Admisor recibe una solicitud de colaboración por parte de un Organismo.
1. El Admisor crea una gestión de tipo Expedientillo.
2. El Admisor ingresa el Peticionario: Capturar peticionario.
3. El Admisor captura la solicitud requerida por el Organismo.
4. El Admisor ingresa los agraviados: Capturar agraviados.

5. El Admisor genera un documento “Carátula primaria de expedientillo”: Generar documentos.

Requerimientos especiales:
- Reglas de negocio interconectables para su inserción en el paso 8. Los criterios de asignación puede cambiar o intercambiarse entre las visitadurías.

Tecnología y variaciones de datos:
Frecuencia: continuamente.
Temas abiertos:

Caso de Uso SIIGESI03. Capturar involucrados.

Resumen: escenario en el cual un Usuario registra un involucrado (persona con carácter de peticionario o agraviado o ambos) en alguna gestión.

Alcance: SIIGESI
Nivel: Usuario
Actor principal: Usuario
Intereses del cliente:
- Usuario:

Precondiciones: Una Gestión está activa.
Postcondiciones: Un Involucrado es registrado y asociado a la Gestión.

Escenario principal:
1. El Usuario ingresa el nombre de una Persona.
2. El Sistema busca a la Persona y no encuentra resultados. La Persona es nueva en el Sistema.
3. El Usuario captura los datos de la Persona: nombre(s), fecha de nacimiento, edad, sexo y nacionalidad.
4. El Sistema registra a la Persona, sus datos y la relaciona con la Gestión.
5. El Usuario captura las direcciones, teléfonos y correos electrónicos de la Persona.
6. El Usuario marca a la Persona como Peticionario de la Gestión.
7. El Sistema registra a la Persona como Peticionario de la Gestión.

Flujos altenativos:

1a. El Usuario ingresa sólo el nombre.
   1. El Sistema alerta al Usuario de ingresar cuando menos el nombre y apellido paterno.

1b. El involucrado es un Grupo de gente agraviada.
   1. El Usuario captura el nombre del Grupo de gente.
   2. El Usuario marca el Grupo como Agraviado.
   3. El Sistema registra el Grupo como Agraviado de la Gestión.

2a. El Sistema encuentra resultados.
   1. El Sistema despliega los nombres y direcciones de las personas.
      1.a. Alguna Persona coincide con la buscada.
         1. El Usuario selecciona a la Persona.
         2. El Sistema asocia la Persona en la Gestión.
      1.b. Ninguna Persona coincide con la buscada.
         1. El Usuario registra una persona nueva.

3a. El Usuario captura la fecha de nacimiento.
   1. El Sistema calcula y despliega la edad.

3b. El Usuario desconoce la edad de la Persona.
   1. El Usuario marca que se desconoce la edad de la Persona.
   2. El Sistema registra no registra la edad ni la fecha de nacimiento. Registra que se desconoce la edad de la Persona.

6a. La persona es un Agraviado.
   1. El Usuario marca a la Persona como Agraviado.
   2. El Sistema registra a la Persona como un Agraviado de la Gestión.

6b. La gestión ya tiene un Peticionario.
   1. El Sistema alerta al Usuario de que la Gestión ya tiene Peticionario.

Requerimientos especiales: al buscar las personas, si la persona a buscar tiene más de un nombre, el Sistema deberá considerar la búsqueda con las combinaciones posibles. Por ejemplo: si el Usuario ingresa como nombre Juan José López López, el Sistema deberá buscar coincidencias para José Juan López López, puesto que se reporta que es común que no se encuentren personas debido al orden de los nombres.
Tecnología utilizada:
Frecuencia: continuamente.
Temas abiertos:

Caso de Uso SIIGESI04. Capturar organismos.

Resumen: Escenario en el cual un Usuario registra un Organismo a una Gestión.

Alcance: SIIGESI
Nivel: Usuario
Actor principal: Usuario
Intereses del cliente:
– Usuario:

Precondiciones: Hay una Gestión activa.
Postcondiciones: La Gestión tiene relacionada un Organismo.

Escenario principal:
1. El Usuario ingresa el nombre de un Organismo o su identificador.
2. El Sistema despliega los Organismos que coincidan con la búsqueda; despliega por cada Organismo: el nombre, si es genérico o específico.
3. El Usuario selecciona un Organismo.
4. El Sistema relaciona el Organismo con la Gestión.

Flujos altenativos:
2a. El Organismo no está registrado en el Sistema.
1. El Sistema notifica que el Organismo no existe.
2. El Usuario envía una notificación de alta del Organismo al Administrador del directorio.

3a. El Organismo ya está relacionado con la Gestión.
1. El Sistema despliega una notificación expresando que el Organismo ya está relacionado con la gestión.

Requerimientos especiales: para el punto 2 del flujo alternativo 2a, la notificación deberá enviarse por medio de un correo electrónico al Administrador del directorio.
Tecnología utilizada: API para el intercambio de mensajes de correo electrónico que soporte SMTP y POP3.
Frecuencia: continuamente.
Temas abiertos:

Caso de Uso SIIGESI05. Generar documentos.

Resumen: Escenario para el cual un Usuario selecciona un documento para que lo genera el Sistema.

Alcance: SIIGESI
Nivel: Subfunción
Actor principal: Sistema
Intereses del cliente:
- Usuario: quieren generar un documento que le proporcione la información básica, dependiendo del tipo de documento seleccionado.
- CDHDF: quiere que la generación de los documentos proporcione un número de oficio adecuado para poder enviar tanto como a organismos externos como a peticionarios.

Precondiciones: hay una gestión activa.
Postcondiciones: un documento es creado y asociado a la gestión. El documento contiene la información básica (narración de hechos, personas, organismos etc.) para continuar con su redacción.

Escenario principal:

1. El Sistema presenta los tipos de documentos que el Usuario puede generar de acuerdo con el Área a la que pertenezca, agrupándolos si están dirigidos a un Organismo, a un Peticionario, o son internos.
2. El Usuario selecciona el documento que va a generar.
3. El Usuario ingresa observaciones del documento y confirma la creación del documento.
4. El Sistema crea el documento, genera y registra un número, registra la fecha de creación y el Empleado que lo creó.
5. El Sistema relaciona el documento con la gestión

Flujos altenativos:

1a. El Usuario es de la Dirección de Orientación.

1. El Sistema presenta: "Medidas precautorias", "Oficio de canalización", "Acta circunstanciada".

1b. El Usuario es de la Dirección Admisibilidad y Registro de Quejas.

1. El Sistema presenta: "Remisión dirigida al peticionario", "Remisión dirigida a la autoridad", "Acuerdo de recepción de queja", "Carátula de expediente remisión", "Carátula de expediente de queja", "Carátula de expediente de expedientillo".

1c. El Usuario es de alguna Visitaduría General.

1. El Sistema presenta: "Medidas precautorias", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario", "Oficio de comisión", "Oficio de canalizacion", "Acta circunstanciada", "Acuerdo de conclusión", "Acuerdo de reapertura".

1d. El Usuario es de la Dirección Ejecutiva de Seguimiento.

1. El Sistema presenta: "Medidas precautorias", "Oficio de comisión", "Solicitud de acceso al expediente", "Acta circunstanciada", "Cumplimiento del punto recomendatorio", "Acuerdo de conclusión de recomendación".

2a. El documento es dirigido a un Organismo.

1. El Usuario selecciona un Organismo de la Gestión.
   1.a. El Organismo no está en la Gestión.
      1. El Usuario busca el Organismo en el Directorio: <u>Buscar Organismo</u>.
1. El documento requiere un plazo de respuesta.
   2.a. El Usuario ingresa el plazo de respuesta en días (4,6,8,10,12,15).
   2.b. El documento es "Medidas precautorias"
      1. El Usuario ingresa el plazo de respuesta en horas (2,4,8,12,24,48).
2. El Sistema registra al Usuario como remitente, al Organismo como destinatario, .

4a. El documento es dirigido al Peticionario.

1. Hay un Peticionario en la Gestión.
   1.a. El Sistema registra al Peticionario como destinatario del documento y al Usuario como remitente.
2. No hay un Peticionario en la Gestión.
   2.a. El Sistema despliega un mensaje de error expresando que el Usuario debe agregar un Peticionario a la Gestión.

4b. El documento seleccionado es "Acta circunstanciada".

1. El Sistema agrega los siguentes datos al documento: Usuario, año, fecha, hora, título de la Gestión, peticionario y Adscripción del Usuario.
2. El Sistema genera el número del documento con la siguiente nomenclatura AC-ADSCRIPCIÓN-NÚMERO_CONSECUTIVO-AÑO

4c. El documento seleccionado es "Medidas precautorias".

1. El Sistema agrega los siguentes datos al documento: Número de oficio, Usuario, año, Organismo, Contacto, fecha, título de la Gestión, narración de hechos, peticionario y Adscripción del Usuario.
2. El Sistema genera el número del documento con la siguiente nomenclatura NÚMERO_CONSECUTIVO_CDHDF-AÑO

4d. El documento seleccionado es "Carátula de expediente remisión" o "Carátula de expediente de queja" o "Carátula de expediente de expedientillo".

1. El Sistema agrega los siguentes datos al documento: de los agraviados Agraviados: nombres, domicilios, telefónos; del Peticionario: nombre, domicilio, teléfonos, correos electrónicos, CURP, edad, sexo, nacionalidad; Usuario, título de la Gestión, fecha, lugar donde ocurrieron los hechos, narración de hechos, servicio brindado y vía de entrada (en caso de existir).
2. El Sistema genera el número del documento con la siguiente nomenclatura CP-TIPO_GESTIÓN-NÚMERO_CONSECUTIVO-AÑO

4e. El documento seleccionado es "Acuerdo de recepción de queja".

1. El Sistema agrega los siguentes datos al documento: nombres de los Agraviados, nombre del Peticionario y título del Expediente de queja.
2. El Sistema genera el número del documento con la siguiente nomenclatura ARQ-ADSCRIPCIÓN-NÚMERO_CONSECUTIVO-AÑO

4f. El documento seleccionado es "Acuerdo de calificación" o "Acuerdo de recalificación".

1. El Sistema agrega los siguentes datos al documento: Usuario, agraviados, fecha, Adscripción, Organismos, denuncias a los Organismos, servidores públicos involucrados, calificación, título del expediente, narración de hechos.
2. El Sistema genera el número del documento con la siguiente nomenclatura A[CAL/RCAL]-ADSCRIPCIÓN-NÚMERO_CONSECUTIVO-AÑO.

4g. El documento seleccionado es "Acuerdo de conclusión".

1. El Sistema agrega los siguentes datos al documento: título del expediente.
2. El Sistema genera el número del documento con la siguiente nomenclatura ACON-ADSCRIPCIÓN-NÚMERO_CONSECUTIVO-AÑO.

4h. El documento seleccionado es "oficio de comisión".

1. El Sistema agrega los siguentes datos al documento: Usuario, título de la Gestión, fecha, número de oficio, Organismo, contacto, cargo del contacto, peticionario, Adscripción.
2. El Sistema genera el número del documento con la siguiente nomenclatura NÚMERO_CONSECUTIVO_CDHDF-AÑO.

4i. El documento seleccionado es "Remisión dirigida al peticionario".

1. El Sistema agrega los siguentes datos al documento: número de oficio, Peticionario.
2. El Sistema genera el número del documento con la siguiente nomenclatura REMP-NÚMERO_CONSECUTIVO-AÑO.

4j. El documento seleccionado es "Remisión dirigida al organismo".

1. El Sistema agrega los siguentes datos al documento: número de oficio, Organismo, contacto.
2. El Sistema genera el número del documento con la siguiente nomenclatura: REMO-NÚMERO_CONSECUTIVO-AÑO.

4k. El documento seleccionado es "Oficio de canalización".

1. El Sistema agrega los siguentes datos al documento: Usuario, número de oficio, fecha, Organismo, contacto, domicilio del organismo, teléfono del organismo, peticionario, domicilio del peticionario, teléfono del peticionario.
2. El Sistema genera el número del documento con la siguiente nomenclatura OC-ADSCRIPCIÓN-NÚMERO_CONSECUTIVO-AÑO.

4l. El documento seleccionado es "Acuerdo de reapertura".

1. El Sistema agrega los siguentes datos al documento: TODO.
2. El Sistema genera el número del documento con la siguiente nomenclatura AR-ADSCRIPCIÓN-NÚMERO_CONSECUTIVO-AÑO.

5a. El Usuario decide cancelar el documento.

1. El Usuario selecciona el documento e ingresa el motivo.
2. El Sistema cancela el documento, registra el motivo y el Usuario que cancela.

Requerimientos especiales:
Tecnología utilizada: API para generar los documentos. Probar MSOfice y OpenOfice.
Frecuencia: continuamente.
Temas abiertos:

Caso de Uso SIIGESI06. Recibir queja.

Resumen: Escenario para el cual un Administrador de visitadurías recibe una queja y la asigna a uno o más visitadores para su investigación.

Alcance: SIIGESI
Nivel: usuario
Actor principal: Administrador Visitaduría
Intereses del cliente:
– Visitador: quiere recibir rápidamente la Queja para asignarla a uno o más Visitadores Investigadores.
– Admisor:

– Peticionario: quiere que su queja sea atendida rápida y eficazmente.

Precondiciones: debe haber una Queja turnada a la Visitaduría.
Postcondiciones: la Queja es asignada a un Visitador Investigador para su atención.

Escenario principal:
1. El Administrador es notificado de terner una Queja entrante.
2. El Administrador recibe y asigna la Queja a uno o varios Visitadores Investigadores.

Flujos altenativos:
1a. La Queja es asignada a una Visitaduría incorrecta.
1. El Administrador genera un documento "Acuerdo de reasignación".
2. El Administrador reasigna el Expediente a la Visitaduría de reasignación.
3. El Sistema envía una notificación al Administrador de la Visitaduría de reasignación.
   3.a El Administrador de la Visitaduría de reasignación acepta el Expediente.
   3.b El Administrador de la Visitaduría de reasignación rechaza el Expediente.

2a. El Admistrador reasigna la Queja.
1. El Administrador agrega uno o más Visitadores.
   1a. El Sistema registra los Visitadores seleccionados.
2. El Administrador remueve uno o más Visitadores.
   2a. El Sistema remueve los Visitadores seleccionados.
   2b. El Sistema registra la información histórica de los Visitadores removidos.

Requerimientos especiales:
Tecnología:
Frecuencia: continuamente.
Temas abiertos:

Caso de Uso SIIGESI07. Investigar queja.

Resumen: Escenario para el cual, un visitador recibe una Queja para su investigación, hasta el momento en que la concluye.

Alcance: SIIGESI
Nivel: usuario
Actor principal: Visitador
Intereses del cliente:
– Visitador:
– CDHDF: quiere datos consistentes para la generación de informes.

Precondiciones: una Queja está asignada a la Visitaduría
Postcondiciones: la Queja es concluída.

Escenario principal:
1. El Visitador califica la Queja como "presunta violación"
2. El Visitador genera el documento "Acuerdo de calificación": Generar documentos.
3. El Visitador inicia la investigación de la Queja.
4. El Visitador registra las violaciones a los derechos humanos en que los organismos involucrados en la Queja incurrieron.
5. El Visitador concluye la investigación por cada violación imputada a un Organismo.
6. El Visitador concluye la investigación por cada Organismo.
7. El Visitador concluye la Queja por "solución durante el tramite".
8. El visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

Flujos altenativos:
*a. En cualquier momento, el Peticionario no desea continuar con la investigación.
1. El Visitador concluye la Queja por "desistimiento del peticionario".
2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el

organismo", "Notificación de conclusión para el peticionario": Generar documentos.
*b. En cualquier momento, el Peticionario muere y su presencia es necesaria para continuar la investigación.
1. El Visitador concluye la Queja por "muerte de la parte quejosa".
2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo": Generar documentos.

*c. En cualquier momento, el Peticionario decide no seguir adelante con la investigación por presión externa, miedo o angustia motivada por una amenaza real o potencial del Organismo.
1. El Visitador concluye la Queja por "muerte de la parte quejosa".
2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo": Generar documentos.

*d. En cualquier momento, el Visitador determina que un Organismo amerita un aviso inmediato debido a la gravedad del asunto.
1. El Visitador genera un documento de "Medidas Precautorias" para enviarlo al Organismo: Generar documentos.

*e. En cualquier momento, el Visitador genera un documento (oficio de comisión, acta circunstanciada, oficio de canalización).
1. Generar documentos.

1a. La Queja no compete a la Comisión.
1. El Visitador califica la Queja como "incompetencia".
2. El Visitador concluye la Queja por "incompetencia".
3. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

1b. La Queja no procede.
1. El Visitador califica la Queja como "improcedencia".
2. El Visitador concluye la Queja por "improcedencia".
3. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

1c. La información aportada en la narración de hechos es poco clara, confusa o imprecisa y el Visitador no logra aclarar los hechos en el plazo de calificación.
1. El Visitador califica la Queja como "Pendiente".
2. El Visitador se comunica con el Peticionario para que aclare los hechos.
   2a. El Peticionario aclara los hechos en un plazo de diez días.
      1. El Visitador recalifica la Queja.
      2. El Visitador continua con la investigación de la Queja.
   2b. El Peticionario no aclara los hechos en un plazo de diez días.
      1. El Sistema notifica al Visitador que el plazo ha vencido.
      2. El Visitador concluye la Queja por "falta de interés".
      3. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

3a. Los hechos no son violatorios de los derechos humanos.
1. El Visitador concluye la Queja por "hechos no violatorios de derechos humanos".
2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

3b. La vía de entrada de la Queja fue por escrito, fax o correo electrónico y se requiere la comparecencia del Peticionario para ratificarla.
1. El Peticionario acude a la CDHDF dentro de un plazo de cinco días.
   1a. El Visitador continua con la investigación de la Queja.
2. El Peticionario no acude a la CDHDF en el plazo de cinco días.
   2a. Se concluye la Queja.
   1. El Visitador concluye la Queja por "falta de interés".
   2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

3c. La imputación a un Organismo es falsa.
1. El Visitador genera el documento "Acuerdo de no responsabilidad": Generar documentos.
2. El Presidente de la CDHDF firma el documento.
3. El Visitador concluye la Queja por "acuerdo de no responsabilidad".
4. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

4a. No es posible identificar al Organismo que cometió la violación.

1. El Visitador concluye la Queja por "no es posible identificar al Organismo que cometió la violación".
2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

5a. No hay elementos suficientes para acreditar la violación.

1. El Visitador concluye la Queja por "no hay elementos suficientes para acreditar la violación".
2. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

5-8a. El Organismo no demuestra su voluntad de devolver las cosas al estado en que se encontraban antes de la violación o de reparar el daño de manera satsfactoria para el Peticionario.

1. El Visitador genera el documento "Recomendación": Generar documentos.
2. El Presidente de la CDHDF firma el documento.
3. El Visitador concluye la Queja por "recomendación".
4. El Visitador genera los documentos "Acuerdo de conclusión", "Notificación de conclusión para el organismo", "Notificación de conclusión para el peticionario": Generar documentos.

7a. El Visitador decide reabrir el Expediente de Queja.

1. El Sistema restablece la conclusión de la Queja, la conclusión por cada violación y por cada organismo, y registra la información histórica de la conclusión.
2. El Visitador genera el documento "Acuerdo de reapertura".

Requerimientos especiales:
Tecnología:
Frecuencia: continuamente
Temas abiertos:

Anexo 3

Especificaciones suplementarias

Especificación suplementaria

Historial de revisiones

Versión	Fecha	Descripción	Autor
inicio		introducción y componentes requeridos.	Marco A. Camacho.
elaboración		se agrega Funcionalidad (Bitácora y errores y Seguridad). Se agrega en componentes de software libre y código abierto API para SMTP y POP3.	Marco A. Camacho.

Introducción
Este documento es el repositorio de todos los requerimientos del SIIGESI que no se capturaron en los casos de uso.

Funcionalidad
Bitácora y errores
Almacenar en un medio persistente tanto las operaciones que se consideren críticas, así como los errores sucitados.

Seguridad
Se requiere seguridad en la autenticación al Sistema. Adicionalmente, se requiere seguridad en diferentes niveles:

- Verificar que el Usuario tenga permiso de realizar una operación dada.
- En el caso de Expedientes de queja, verificar que al Usuario le pertenezca el expediente para poder realizar operaciones en él.
- Verificar que el Usuario modifique sólo gestiones que correspondan a su Adscripción.
- Para la administración, que el Usuario pertenezca a el grupo de Administradores.

Componentes Comprados

- API para crear documentos de texto desde plantillas.
- Se sugiere comprar un conector JDBC que maneje un pool de conexiones para agilizar las transacciones en la BD.

Componentes de software libre y código abierto
En general, por flexibilidad y por que la CDHDF quiere minimizar el costo del proyecto, se recomienda aprovechar el uso de componentes de software libre de Java. Para el diseño preeliminar, se sugiere el uso de los siguientes componentes:

- El Framework Struts para la construcción de la aplicación web basado en el paradigma de diseño modelo-vista-controlador.
- el Framework Hibernate para el manejo de la persistencia en el mapeo objeto-relacional.
- log4j
- API para el intercambio de mensajes de correo electrónico.