Universidad de las Ciencias Informáticas
Facultad 3

Componentes Reclamación y Requisición de pago del Sistema de Gestión para la Empresa de Seguros Nacionales.

Autor: Diego Javier Álvarez Díaz

Tutores: Ing. William González Obregón

Ing. Andy Daniel Meriño Coronado

La Habana. Julio del 2016

DECLARACIÓN DE AUTORÍA

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

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

Diego Javier Álvarez Díaz

_____

Firma del Autor

Ing. William González Obregón

_____

Firma del Tutor

Ing. Andy Daniel Meriño Coronado

_____

Firma del Tutor

“La recompensa se encuentra en el esfuerzo y no en el resultado. Un esfuerzo total es una victoria completa”

Mahatma Gandhi.

AGRADECIMIENTOS

Quisiera resaltar que la realización del presente trabajo ha demandado de mi total entrega y dedicación, lo que me enorgullece aún más el haberlo terminado. Aunque siendo sincero no lo he logrado solo, y me siento con la necesidad de agradecer a aquellas personas que me han ayudado a alcanzar mis metas.

En primer lugar, a mis padres que siempre han sido mi ejemplo y paradigma a seguir, ambos me han demostrado que son insaciables de conocimiento y que aprender y superarse son procesos que nunca terminan.

En segundo lugar, agradecer a mi hermano Daniel por brindarme su apoyo incondicional en todo momento, sin él no lo hubiera logrado. Como también agradecer a mi familia siempre preocupada por mí, y a mis hermanos Felipe, Omar y José que se van a estar ahí el día de mi graduación, porque la vida nos ha hecho inseparables.

Agradecer también a mis tutores William y Andy, así como el resto de integrantes del proyecto que me ayudaron en todo lo que pudieron y me aconsejaron en todo momento.

Sabiendo que no voy a terminar de leer los agradecimientos por las lágrimas, termino aquí, aunque mi gratitud será infinita para y con los antes mencionados.

DEDICATORIA

"Este trabajo lo dedico a mis padres que no van a poder estar en mi graduación, pero siempre estarán en mi corazón."

RESUMEN

El presente trabajo tiene como objetivo desarrollar los componentes Reclamación y Requisición de pago, ambos enmarcados en el proceso de reclamaciones que desempeña la Empresa de Seguros Nacionales, de manera que corrijan las deficiencias que provocan la pérdida de información, dificultades en el acceso a la misma y la comisión de errores humanos actualmente presentes en dicho proceso. En el transcurso del mismo se describe el diseño, implementación y validación de los componentes antes mencionados, para lo cual se realizó un estudio de otros sistemas que llevan a cabo la gestión de pólizas de seguros tanto en el ámbito nacional como internacional, concluyendo que era necesario construir una solución propia. Como resultado de la investigación se obtienen los componentes Reclamación y Requisición de pago, cumpliendo con los objetivos propuestos y las normas establecidas en el proyecto.

Palabras claves: gestión, pólizas, seguro, reclamación, requisición de pago.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

INTRODUCCIÓN

El seguro es un negocio que coloca énfasis en el contacto directo de las partes interesadas en la firma de un contrato, a fin de afianzar una relación con los agentes y corredores de seguros que son las partes que constituyen aún hoy en día la fuerza de ventas en este mercado. Por ello, todas las herramientas y mecanismos empleados en este proceso son cruciales a la hora de persuadir a los clientes a tener contactos más frecuentes con las aseguradoras (Galanti, 2013).

A diferencia de otras empresas presentes en el mercado, las compañías de seguros no se escapan de los riesgos a los que se ven expuestos los bienes o las personas. A cambio del pago de una cantidad monetaria (denominada prima), la compañía de seguros pagará la suma acordada (indemnización) en el caso de que ocurra un determinado evento (siniestro) cuyo riesgo de que ocurra es objeto de cobertura por el seguro. Una competencia fundamental de los seguros es la capacidad de clasificar sus coberturas frente a los efectos de sucesos aleatorios. Por esta razón, la eficacia del sector de los seguros es muy importante para las personas, los hogares y las empresas (Galanti, 2013).

En Cuba existen dos grandes empresas aseguradoras, ambas regidas por la Superintendencia de Seguros. Ellas son la Empresa de Seguros Internacionales de Cuba (ESICUBA) y la Empresa de Seguros Nacionales (ESEN). Esta última siendo parte fundamental del objetivo en el cual se enfocará este trabajo de diploma, más específicamente en el proceso de reclamaciones que se inicia dentro de la misma al ser un asegurado, víctima de un siniestro[1], y culmina con la elaboración de la requisición de pago, como documento oficial que permite la liberación de fondos monetarios a favor del siniestrado. Dicho proceso será abarcado en el presente trabajo dividiéndose en dos componentes nombrados: Reclamación y Requisición de pago, los cuales facilitaran la gestión del mismo.

La ESEN, tiene como objeto empresarial desarrollar operaciones de seguros y reaseguros en moneda nacional y divisa tanto a personas naturales y jurídicas cubanas como extranjeras, así como realizar actividades preparatorias y complementarias al seguro, dirigidas a la evaluación de riesgos y prevención de daños. También le compete ofrecer servicios de inspección, tasación y ajustes de averías, cálculos actuariales y prevención del

[1] Siniestro: en la terminología de empresas de seguros, es el acontecimiento futuro e incierto del cual depende la obligación de la entidad de seguros de indemnizar o de pagar la prestación convenida.

riesgo en bienes asegurados en ambas monedas. Esta entidad cuenta con una Oficina Central y varias Unidades Empresariales de Base (UEB) distribuidas por todo el país. En la actualidad, el proceso de reclamaciones que desarrolla la empresa se ve afectado por la siguiente situación:

La existencia y utilización de dos sistemas informáticos diferentes (INFOSEG y SIGES) para la gestión de su información en distintas provincias, provoca:

- Que todos los informáticos tengan acceso a su código fuente y base de datos, pudiendo alterar su programación o incluso la información almacenada, poniendo en riesgo su integridad, al estar desplegados en cada UEB.

- La falta de estandarización de la información trae como consecuencia la imposibilidad de conocer de una manera ágil, por parte de la Oficina Central, datos necesarios como la cantidad de pólizas, reclamaciones o asegurados a nivel de país, para la toma de decisiones y el seguimiento y control de sus procesos.

- Bases de datos descentralizadas y distintas, lo que posibilita que una persona se asegure en el mismo producto[2] y con las mismas condiciones en dos provincias diferentes, lo cual constituye una violación a los procedimientos establecidos, además de permitirle al mismo hacer reclamaciones en ambas unidades empresariales de base, incurriendo así en ilegalidades.

Ambos sistemas cuentan como dos clientes distintos a aquel asegurado que tiene dos contratos de seguro, cuando realmente es uno solo con dos pólizas.

Los sistemas utilizados no cubren todas las funcionalidades del proceso de reclamaciones, tales como:

- Gestión de beneficiarios y terceros afectados.
- Búsqueda de pólizas.
- Realización de requisiciones de pago.

[2] Producto: en la terminología de empresas de seguros, es una opción que ofrece el asegurador, destinada a la cobertura de un conjunto de riesgos.

La situación obliga a realizar estos procesos manualmente, pudiendo introducirse errores en los cálculos. Lo que constituye un escenario favorable para las ilegalidades, provocando además tardanza entre la confección de la documentación y su aprobación a nivel central.

Por la situación antes expuesta se plantea como **problema a resolver:** Las herramientas informáticas que se utilizan en la actualidad en la ESEN para la gestión del proceso de reclamaciones presentan deficiencias que provocan la pérdida de información, dificultades en el acceso a la misma y favorecen la comisión de errores humanos.

Con el objetivo de solucionar el problema planteado anteriormente, el **objeto de estudio** queda enmarcado en el proceso Reclamaciones de seguros en la ESEN, y delimitado el **campo de acción** en la informatización del proceso Reclamaciones de seguros como parte de un sistema de gestión de seguros.

Para llevar a cabo este trabajo se planteó como **objetivo general:** Desarrollar los componentes Reclamación y Requisición de pago cumpliendo con los requisitos identificados de manera que se registre toda la información del proceso de Reclamaciones, permita el acceso a la misma y limite la posibilidad de ocurrencia de errores humanos.

Para dar cumplimiento al objetivo general se definieron los siguientes **objetivos específicos:**

- Elaborar el marco teórico de la investigación que permita identificar los principales logros y limitaciones en cuanto a la gestión de las reclamaciones y requisiciones de pago de seguros, así como las tecnologías y herramientas a utilizar.
- Diseñar los componentes Reclamación y Requisición de pago basado en los requisitos previamente identificados.
- Validar el diseño de los componentes mediante el uso de métricas.
- Implementar los componentes Reclamación y Requisición de pago teniendo en cuenta el diseño realizado.
- Verificar los componentes obtenidos mediante la realización de pruebas de caja blanca y caja negra.
- Validar que la solución propuesta cumpla con los requisitos identificados.

Las **tareas a cumplir** son las siguientes:

- Definición de los principales conceptos asociados a la gestión de los reclamos de seguros y requisiciones de pago como parte del proceso Reclamaciones.

- Análisis de los sistemas existentes para la gestión de las reclamaciones de seguro identificando sus características y deficiencias fundamentales.
- Caracterización de las tecnologías, lenguajes y herramientas propuestas para el desarrollo de los componentes.
- Diseño de los componentes Reclamación y Requisición de pago a partir de los requisitos previamente identificados y pactados con el cliente.
- Implementación de los componentes Reclamación y Requisición de pago teniendo en cuenta el diseño realizado.
- Aplicación de buenas prácticas establecidas en el proyecto al código fuente para facilitar su comprensión y reutilización.
- Realización de pruebas de caja blanca a las principales funcionalidades implementadas.
- Verificación de los componentes obtenidos mediante la realización de pruebas de caja negra.
- Validación de la solución propuesta.

Métodos de la investigación empleados para el desarrollo del trabajo

Para llevar a cabo el desarrollo del presente trabajo de diploma, se utilizaron los métodos teóricos:

- Histórico-lógico: realizándose un estudio de cómo se ha comportado el desarrollo de los sistemas de gestión de pólizas que emplea la ESEN con el transcurso del tiempo.
- Modelación: creando abstracciones mediante diagramas para explicar la estructura de la solución propuesta.

Como métodos empíricos utilizados se encuentran:

- Entrevistas de tipo abierta: realizadas al equipo de desarrollo, analistas y al jefe de proyecto, con el objetivo de recolectar datos fundamentales para una mejor comprensión del proceso de reclamaciones.
- Observación asistemática: del trabajo llevado a cabo por los desarrolladores, identificándose las cualidades y aptitudes necesarias, así como el empleo correcto de técnicas para la implementación y diseño de la solución propuesta.

La estructura del trabajo de diploma será la siguiente:

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA: Describe el estado del arte de los sistemas que gestionan pólizas de seguros, los principales conceptos relacionados, así como la fundamentación del uso de los lenguajes, herramientas y metodología de desarrollo.

CAPÍTULO 2: DISEÑO DE LA SOLUCIÓN: En este capítulo se diseña la solución propuesta mediante los productos de trabajo, modelo de datos y diagrama de clases del diseño, representando las clases de la implementación y las relaciones entre las mismas. Es definida la arquitectura de la aplicación mediante los patrones arquitectónicos. Además, se valida el diseño mediante el uso de métricas.

CAPÍTULO 3: IMPLEMENTACIÓN, PRUEBAS Y VALIDACIÓN: Se muestran los resultados obtenidos de la implementación de los componentes, su verificación, mediante el empleo de pruebas de calidad, así como la validación de la solución propuesta.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

Introducción

En este capítulo se abordan los fundamentos teóricos asociados a la investigación, se enuncian las definiciones de los conceptos relacionados con la gestión de pólizas de seguros y el manejo de las reclamaciones y requisiciones, así como la relación existente entre ellos. Se realiza un estudio de diferentes sistemas existentes en Cuba y el mundo que llevan a cabo este proceso, así como una descripción de las tecnologías utilizadas para el desarrollo de la solución propuesta.

1.1 Principales conceptos

A continuación se exponen los principales conceptos citados durante el desarrollo del presente trabajo, los cuales han sido extraídos de (Castro Ruz 1997)(Castro Ruz 2009), decretos leyes que rigen el negocio de gestión de pólizas de seguros en Cuba:

- La **entidad de seguros** es la persona jurídica, constituida conforme a las leyes de la República de Cuba, dedicada a la comercialización y ejecución de seguros, previamente autorizada por la Superintendencia de Seguros para ejercer como tal, de acuerdo con lo establecido legalmente a estos efectos, y que cuente con patrimonio suficiente en el territorio nacional para responder por las obligaciones que asume.

- El **contrato de seguro** es aquel por el cual la entidad de seguros se obliga, mediante el cobro de una **prima**, a garantizar el interés del asegurado o del beneficiario en cuanto a las consecuencias que resulten del riesgo cubierto por el contrato.

- La **prima** es el pago que se le exige al asegurado, según sea la suma asegurada y tiempo de contrato establecido.

- La **póliza** es el documento en el que se hacen constar las condiciones del contrato de seguro y en la que se establecen los derechos y obligaciones de las personas que intervienen en él.

El **contrato de seguro**, introduce otros términos como:

- **Asegurado** es la persona titular del interés asegurado y, por consiguiente, aquella cuyos bienes, persona y responsabilidades, están expuestos al riesgo y que ejerce los derechos y responde por las obligaciones de la relación contractual constituida.

- **Tomador** es la persona que no es el titular del interés asegurado, pero contrata el seguro, a nombre de un tercero, con la entidad de seguros, el tomador y el asegurado pueden ser o no la misma persona.

- **Beneficiario** es aquel que es titular de los beneficios del contrato de seguro y que tiene acción directa contra la entidad de seguros, una vez ocurrido el siniestro, pasando así al **proceso de reclamación.**

- **Proceso de reclamación** es aquel en que el beneficiario hace uso de su derecho sobre lo acordado en el contrato de seguro, para exigir el cobro de la cantidad en efectivo acordada según sea el tipo de siniestro y si está o no comprendido en el contrato. En caso de aprobar la reclamación, se culmina dicho proceso con la **requisición de pago**.

- **Requisición de pago** es un documento oficial que se genera con el fin de aprobar la entrega del efectivo acordado al beneficiario.

1.2 Sistemas que incluyen el proceso de reclamaciones

En la actualidad existen diferentes sistemas informáticos que se encargan de la realización del proceso de la gestión de pólizas de seguros, por lo que es importante valorar sus particularidades y analizar su reutilización:

Nekül:

Nekül es un sistema web de gestión de seguros. Cuenta con la funcionalidad de administración de cartera de clientes y pólizas, y un multicotizador[3] que se encuentra conectado vía Web Services con las principales compañías aseguradoras del mercado. Estas funcionalidades son gratis y de uso ilimitado. Siendo el primer sistema integral que permite a los clientes multicotizar de manera online con sus compañías elegidas, de manera totalmente gratuita y confiable. Por ser un sistema online, los usuarios pueden acceder al

[3] Multicotizador: Permite al usuario que este asegurado con más de una compañía de seguros, realizar los pagos de prima a través del sistema, con solo registrar las compañías que lo respaldan.

mismo desde cualquier computadora o dispositivo móvil con conexión a Internet, en cualquier momento, desde cualquier lugar (Kimn Consulting 2013).

A la fecha, ya se encuentran integrados al mismo los servicios de emisión y cotización de 16 de las compañías aseguradoras más importantes del mercado: Allianz, Berkley, Boston, Chubb, Generali, La Caja, Liberty, Mapfre, Meridional, Provincia, QBE, RSA, Sancor, San Cristóbal, SMG, Zúrich (Kimn Consulting 2013).

NOA Gestión de Seguros 5.9:

La versión 5.9 es la última conocida del sistema desarrollado por SIMBIOTIC SYSTEM. Con NOA Gestión de Seguros se puede gestionar todo tipo de pólizas de seguros para la pequeña y mediana empresa, aunque también puede servir para agentes y corredores de seguros. El programa permite llevar un control exhaustivo de todas las pólizas, vigilando fechas de vencimiento, cobros, renovaciones, historial de siniestros y renovaciones de carné de conducir (ABCdatos, 2009).

Millennium:

Millenium es la solución modular para una gestión integral de compañías de seguros abarcando los diversos aspectos del negocio. En la actualidad es el sistema empleado por la empresa ESICUBA, para llevar a cabo su gestión de pólizas. Dicho sistema a pesar de satisfacer las necesidades de la ESEN, la misma no desea adquirirlo por su alto costo en licencia y mantenimiento. Su estructura está diseñada de forma que permite convivir con otros sistemas ya existentes en la compañía, utilizando toda la aplicación o solamente los módulos precisos. Entre las características que presenta se encuentran (Sia Services, 2015):

- La gestión de los principales procesos de la compañía.
- Integración con módulos de proveedores externos.
- Conexión con software de contabilidad externo.
- Control de canales de venta (agentes y mediadores).
- Generador de productos con control de versiones.
- Entorno de trabajo cliente/servidor o vía web.

Las funcionalidades que ofrece son (Sia Services, 2015):

- Planificación de los objetivos, en la que pueden definirse los objetivos de seguimiento en función de diferentes parámetros, como puede ser objetivos por

producto, por número de pólizas, por comisiones, por primas medias, etc. Incluso combinar varios de ellos.

- Elaboración de informes, que permite la visión de los datos en gráficos y tablas, con diferentes puntos de vista: organizativo, económico y asegurador. Igualmente se pueden extraer en formato Excel y similares.
- Control de resultados, en la que el análisis de resultados se controla por producto, por total de la agencia o individualmente por cada punto de venta. Los resultados se extraen periódicamente con una actualización de las previsiones a final del ejercicio.
- Gráficas de rendimiento, en la que un sistema de gráficos muy elaborado y de fácil uso permite visualizar rápidamente el rendimiento económico de la agencia, comparando la situación actual con periodos anteriores o con la media del sector.

Otros sistemas para la gestión de pólizas existentes en el mundo son SIRIS (Systems, 2015), Poliza-Win (Roo, 2015) y ADMINSYF (Developer Center, 2016). Los cuales a pesar de haber sido analizados y estar entre los sistemas más utilizados por agentes aseguradores pequeños, no son tomados en cuenta en las comparaciones posteriores por presentar problemas comunes como: que su desarrollo está mayormente enfocado en empresas pequeñas y no presentan una gestión completa de captura de pólizas. Los mismos a pesar de no cumplir con los requisitos de la presente investigación son de vital importancia para agentes aseguradores cuentapropistas y muchas empresas pequeñas a nivel mundial.

Resultados del estudio de los sistemas

Luego del estudio realizado de los sistemas anteriores se refleja en la Tabla 1 las características que estos deben cumplir para ser utilizados por la ESEN. Donde las características deseadas aparecen por las filas y por las columnas los sistemas analizados.

Tabla 1. Resultado del estudio de los sistemas.

Características	Nekül	NOA Gestión de Seguros	Millennium
Licencia gratuita	Sí	No	No
Código libre	No	No	No
Plataforma web	Sí	Sí	Sí

Ejecución de todas las funcionalidades necesarias	No	Se desconoce	Sí

Entre las funcionalidades que brindan los sistemas anteriormente mencionados en relación con el componente de reclamaciones, se pueden listar algunas como: Registrar reclamaciones, Modificar reclamaciones, Denegar reclamaciones, Aprobar reclamaciones, Buscar reclamaciones. La mayoría de ellas presentan relación con los requisitos levantados por el equipo de desarrollo de la UCI para la confección de los componentes Reclamación y Requisición de pago, por lo que fueron tomadas algunas sugerencias visuales para la implementación de los mismos, el resto no se adecuan a las particularidades de la economía cubana descritas en (Castro Ruz, 1997)(Castro Ruz, 2009), ni son consecuentes con el paradigma de independencia tecnológica por el cual apuesta el país.

Las condiciones fundamentales con las que debe cumplir el software deseado para dar solución al problema de la investigación planteado serían:

- Licencia gratuita para no depender de pagos y servicios de segundos países.
- Código libre en caso de ser necesarias modificaciones para ajustarlo a las necesidades específicas de la empresa.
- Plataforma web para lograr la centralización de los datos, lo cual agilizaría de forma sustancial el desarrollo de los procesos entre las diferentes UEB y la sede de la ESEN, mejorando el acceso a la información.
- Ejecución de todas las funcionalidades que se llevan a cabo actualmente en la empresa.

Teniendo en cuenta lo anteriormente expuesto se puede afirmar que sería necesario el desarrollo de una solución propia que responda a las características identificadas, mediante la utilización de herramientas y tecnologías libres.

1.3 Metodología

La metodología de desarrollo empleada se conoce como Metodología de Desarrollo para la Actividad Productiva de la UCI. La es una variación de AUP (Agile Unified Process, por sus siglas en inglés) (Sánchez Rodríguez, 2014).

Dicha metodología cuenta con tres fases principales (Inicio, Ejecución y Cierre) de las cuales solo se llevará a cabo parte de la fase de Ejecución, ya que el presente trabajo parte de los requisitos identificados previamente, empleando el escenario 3 de la metodología, que plantea como técnica de encapsulación de requisitos DRP(Descripción de requisitos por procesos)(Cobas, 2015a). En esta fase se ejecutan las actividades requeridas para desarrollar el software, incluyendo el ajuste de los planes del proyecto considerando los requisitos y la arquitectura. En la misma se realizará el diagrama de clases del diseño, el diseño de la base de datos y los diseños de casos de prueba. Para ello se emplearán las disciplinas de análisis y diseño, implementación y pruebas internas, con el fin de obtener los componentes de Reclamación y Requisición de pago correctamente diseñados e implementados (Sánchez Rodríguez, 2014).

1.4 Herramientas, lenguajes y tecnologías para el desarrollo

Cuando se inicia el desarrollo de un sistema se hace un estudio de las principales soluciones que existen y que van a darle solución a una determinada problemática. Luego del estudio realizado por el equipo de desarrollo se han identificado cuáles serían las tecnologías y herramientas que mejor se integran para llevar a cabo la solución propuesta, quedando plasmadas en (Escobar, 2015) y enunciadas a continuación:

Lenguajes de Programación

JavaScript

Es un lenguaje de programación interpretado. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Se utiliza principalmente en su forma del lado del cliente implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas, aunque existe una forma de JavaScript del lado del servidor. Permite el desarrollo de una interfaz de usuario mejorada, aumentando la experiencia del usuario en la web (Jones, et al, 2009).

Hibernate Query Lenguage (HQL)

Es el lenguaje de consultas que usa Hibernate para obtener los objetos desde la base de datos. Su principal particularidad es que las consultas se realizan sobre los objetos java que forman el modelo de negocio, es decir, las entidades que se persisten en Hibernate. Condicionando que HQL tenga las siguientes características (Rydahl, et al, 2010):

- Los tipos de datos son los de Java.

- Las consultas son independientes del lenguaje de SQL específico de la base de datos.
- Las consultas son independientes del modelo de tablas de la base de datos.
- Es posible tratar con las colecciones de Java.
- Es posible navegar entre los distintos objetos en la propia consulta.

Herramientas y Tecnologías

Java-EE

Java Platform, Enterprise Edition o Java EE (anteriormente conocido como Java 2 Platform, Enterprise Edition o J2EE hasta la versión 1.4; traducido informalmente como Java Empresarial), es una plataforma de programación para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. Permite utilizar arquitecturas de N capas distribuidas y se apoya ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones (Haase, et al, 2010).

Hibernate 4.3.9

Hibernate es una herramienta de Mapeo Objeto-Relacional (ORM) para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los *beans*[4] de las entidades que permiten establecer estas relaciones (Rydahl, et al, 2010).

Spring 4.2

Es un marco de trabajo de código abierto. Se creó para abordar la complejidad del desarrollo de aplicaciones empresariales. Cualquier aplicación Java se puede beneficiar de Spring en términos de simplicidad, capacidad de prueba y acoplamiento. Proporciona una potente gestión de configuración basada en JavaBeans, además de una capa genérica de abstracción para la gestión de transacciones (Breidenbach, et al, 2011).

IntelliJ IDEA 14.1

Es un ambiente de desarrollo integrado (IDE) para el desarrollo de programas informáticos. Es desarrollado por jetbrains (anteriormente conocido como intellij), y está disponible en dos ediciones: community edition, y edición comercial. Brinda un conjunto de funcionalidades que agilizan el trabajo con las tecnologías seleccionadas para el desarrollo. Incluye completamiento inteligente de código, navegación dentro del código,

[4] Bean: componente software que tiene la particularidad de ser reutilizable.

refactorización, integración con maven, subversion, postgresql, spring framework, hibernate (Rydahl, et al, 2010).

Web Storm 11

Es un entorno de desarrollo integrado (IDE por sus siglas en inglés) desarrollado por Jetbrains que provee excelente soporte para PHP (incluyendo las últimas versiones del lenguaje y marcos de trabajo), HTML, JavaScript, CSS, Sass, Less, CoffeeScript, Node.js, AngularJS, entre otros. Tienen dentro de sus características la detección de errores, inspecciones y correcciones de código al vuelo. Poseen además funciones de búsquedas rápidas para saltar a cualquier parte de una clase, archivo, símbolo, o incluso acción del entorno en el IDE (JetBrains, 2015).

Postgres SQL 9.4.5

Es un servidor de base de datos relacional libre. Tiene soporte total para transacciones, disparadores, vistas, procedimientos almacenados, almacenamiento de objetos de gran tamaño. Se destaca en ejecutar consultas complejas, consultas sobre vistas, subconsultas y joins de gran tamaño. Permite la definición de tipos de datos personalizados e incluye un modelo de seguridad completo. Cuenta con una gran comunidad de desarrollo en Internet, su código fuente está disponible sin costo alguno y es multiplataforma (Lockhart, 2011).

Maven

Es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Posee un modelo de configuración de construcción más simple, basado en un formato XML. Estuvo integrado inicialmente dentro del proyecto Jakarta, pero ahora ya es un proyecto de nivel superior de la Apache Software Fundation. Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado (Software Foundation, 2015).

EXTJS 6:

Es una biblioteca de JavaScript para el desarrollo de aplicaciones web interactivas usando tecnologías como AJAX, DHTML y DOM. Fue desarrollada por Sencha. Constituye un marco único para la creación de aplicaciones que se ejecutan en todos los tipos de

dispositivos, desde teléfonos, tabletas y escritorios. Esta última versión de Extjs provee al usuario, además, de una nueva herramienta como Sencha Cmd que ofrece un conjunto completo de características de administración de ciclo de vida del proyecto (Andresen, et al, 2014).

Tomcat 7.0.47:

Tomcat funciona como un contenedor de servlets (Clase en el lenguaje de programación Java, utilizada para ampliar las capacidades de un servidor) desarrollado bajo el proyecto Jakarta. Esta versión incluye mejoras de limpieza de código, de detección y prevención de fugas de memoria[5] en las aplicaciones web, además de soporte para la inclusión de contenidos externos directamente en una aplicación web (Jones, et al, 2003).

Visual Paradigm 8:

Es una herramienta CASE: Ingeniería de Software Asistida por Computación. La misma propicia un conjunto de ayudas para el desarrollo de programas informáticos, desde la planificación, pasando por el análisis y el diseño, hasta la generación del código fuente de los programas y la documentación. Estas herramientas están destinadas a aumentar la productividad en el desarrollo de software reduciendo el coste de las mismas en términos de tiempo y de capital (Pressman, 2010).

Subversion 1.9.3:

Subversion (abreviado frecuentemente como SVN) es una herramienta de control de versiones de código abierto basada en un repositorio cuyo funcionamiento se asemeja al de un sistema de ficheros. Utiliza el concepto de revisión para guardar los cambios producidos en el repositorio. Entre dos revisiones sólo se guarda el conjunto de modificaciones, optimizando así al máximo el uso de espacio en disco. SVN permite al usuario crear, copiar y borrar carpetas con la misma flexibilidad con la que lo haría si estuviese en su disco duro local.

Subversion puede acceder al repositorio a través de redes, lo que le permite ser usado por personas que se encuentran en distintas computadoras. A cierto nivel, la posibilidad de que varias personas puedan modificar y administrar el mismo conjunto de datos desde sus

[5] Fugas de memoria: error de software que ocurre cuando un bloque de memoria reservada no es liberado en un programa de computación. Comúnmente ocurre porque se pierden todas las referencias a esa área de memoria antes de haberse liberado.

respectivas ubicaciones fomenta la colaboración. Se puede progresar más rápidamente sin un único conducto por el cual deban pasar todas las modificaciones. Si se realiza un cambio incorrecto a los datos, estos pueden ser restaurados a su versión anterior (Fitzpatrick, et al, 2012).

1.5 Patrones de arquitectura

Los patrones arquitectónicos, o patrones de arquitectura, también llamados arquetipos ofrecen soluciones a problemas de arquitectura de software en ingeniería de software. Dan una descripción de los elementos y el tipo de relación que tienen junto con un conjunto de restricciones sobre cómo pueden ser usados. Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. En comparación con los patrones de diseño, los patrones arquitectónicos tienen un nivel de abstracción mayor (Pressman, 2010).

Uno de los aspectos más importantes de los patrones arquitectónicos es que encarnan diferentes atributos de calidad. Por ejemplo, algunos patrones representan soluciones a problemas de rendimiento y otros pueden ser utilizados con éxito en sistemas de alta disponibilidad. A principio de la fase de diseño, un arquitecto de software escoge qué patrones arquitectónicos ofrecen las calidades deseadas para el sistema. A continuación, se listan los patrones a emplear en el desarrollo de la solución propuesta (Bass, et al, 2005).

Patrón Modelo-Vista-Controlador (MVC):

De acuerdo con (Sommerville, 2005) el Modelo-Vista-Controlador (MVC) es un patrón arquitectónico para el desarrollo de aplicaciones. Dicho patrón está compuesto por tres partes fundamentales:

- Modelo: Es el nivel más bajo de este patrón y es el responsable de mantener los datos.
- Vista: Es responsable de mostrar o bien todo o una porción de los datos al usuario.
- Controlador: El código de la aplicación que controla las interacciones entre el Modelo y la Vista.

MVC separa la lógica de la aplicación de la capa de interfaz de usuario. El controlador recibe todas las peticiones de la aplicación y entonces trabaja con el modelo para preparar los datos que necesita la vista. La vista entonces usa los datos preparados por el controlador

para generar una respuesta final (Sommerville, 2005). La abstracción del modelo MVC puede ser gráficamente representada de la siguiente forma:

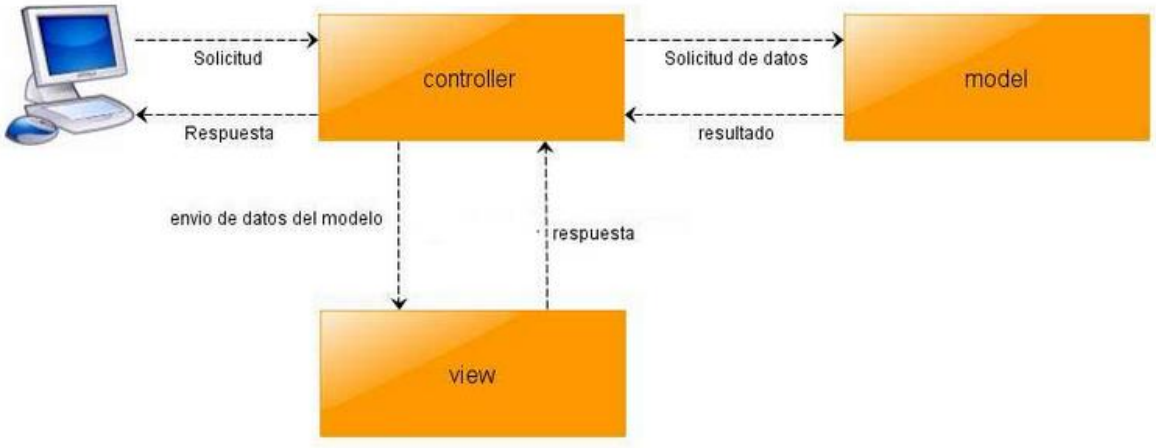

Figura 1. Patrón Modelo-Vista-Controlador (Smith, 2013).

Patrón Arquitecturas en Capas:

La programación por capas es una arquitectura cliente-servidor en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño; un ejemplo básico de esto consiste en separar la capa de datos de la capa de presentación al usuario (Johnson, et al, 2003).

La ventaja principal de este estilo es que el desarrollo se puede llevar a cabo en varios niveles y, en caso de que sobrevenga algún cambio, solo se ataca al nivel requerido sin tener que revisar entre código mezclado. Además, permite distribuir el trabajo de creación de una aplicación por niveles; de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, de forma que basta con conocer la API[6] que existe entre niveles (Wilson, 2001).

En el diseño de sistemas informáticos actual se suelen usar las arquitecturas multinivel o Programación por capas. En dichas arquitecturas a cada nivel se le confía una misión simple, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten) (Johnson, y otros 2003). El más utilizado actualmente es el diseño en tres niveles (o en tres capas), empleado en la solución propuesta y mostrado a continuación como se estructura según (Wilson, 2001).

[6] API: es el conjunto de subrutinas, funciones y que ofrece cierta biblioteca para ser utilizado por otro *software* como una capa de abstracción.

Capas o niveles

- **Capa de presentación:** la que ve el usuario (también se la denomina "capa de usuario"), presenta el sistema al usuario, le comunica la información y captura la información del usuario en un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). También es conocida como interfaz gráfica y debe tener la característica de ser "amigable" (entendible y fácil de usar) para el usuario. Esta capa se comunica únicamente con la capa de negocio.
- **Capa de negocio:** es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Se denomina capa de negocio (e incluso de lógica del negocio) porque es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos almacenar o recuperar datos de él. También se consideran aquí los programas de aplicación.
- **Capa de datos:** es donde residen los datos y es la encargada de acceder a los mismos. Está formada por uno o más gestores de bases de datos que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Para una mejor comprensión del patrón se muestra a continuación la Figura 2.

Figura 2. Modelo de Arquitectura en 3 capas (Liaget, 2013).

1.6 Métricas de validación del diseño

De acuerdo a (Pressman, 2010) medir la calidad del software es una tarea propensa al debate pues no existen métodos absolutos para lograrlo. Por su parte, (Grady, et al, 1992) y (Card, et al, 1990) proponen un conjunto de listas de chequeo con el objetivo de medir la

calidad de los atributos de software como: facilidad de mantenimiento, facilidad de prueba y reusabilidad.

Con respecto al diseño a nivel de componentes algunos autores han propuesto con similar objetivo un conjunto de métricas centrándose en las características internas de los componentes de software.(Bieman, et al., 1994) propone procedimientos para medir específicamente la cohesión entre los elementos de los componentes. (Dhama, 2000) y (McCabe, et al, 1994) proponen procedimientos para medir el acoplamiento y la complejidad como atributos de calidad del software.

En el desarrollo del presente trabajo se propone aplicar las métricas tamaño operacional de clases (TOC), pues permite medir el total de atributos y operaciones encapsulados en una clase para valorar la sobrecarga de responsabilidades asignadas y la métrica relaciones entre clases (RC) para evaluar el grado de acoplamiento entre las clases. Fueron seleccionadas dichas métricas a partir de un estudio realizado de diferentes fuentes de la comunidad científica de la universidad, en las cuales se reflejaba que las métricas TOC y RC son las que más atributos de calidad miden de conjunto para validar un correcto diseño de la solución.

1.6.1 Tamaño operacional de la clase (TOC)

Evalúa los siguientes atributos de calidad (Rodríguez Landín, 2012):

- **Responsabilidad:** consiste en la responsabilidad asignada a una clase en un marco de modelado de un dominio o concepto, de la problemática propuesta.
- **Complejidad de implementación:** consiste en el grado de dificultad que tiene implementar un diseño de clases determinado.
- **Reutilización:** consiste en el grado de reutilización presente en una clase o estructura de clase, dentro de un diseño de software.

Tabla 2. Atributos que afecta TOC (Rodríguez Landín, 2012).

Atributo que afecta	Modo en que lo afecta
Está dada por el número de métodos asignados a una clase.	
Responsabilidad	Un aumento del TOC implica un aumento de la responsabilidad asignada a la clase.

Complejidad de implementación	Un aumento del TOC implica un aumento de la complejidad de implementación de la clase.
Reutilización	Un aumento del TOC implica una disminución en el grado de reutilización de la clase.

Tabla 3. Rango de valores para la evaluación técnica de los atributos de calidad relacionados con la métrica (Rodríguez Landín, 2012).

Atributo	**Categoría**	**Criterio**
Responsabilidad	Baja	<= Promedio
	Media	Entre Promedio y 2* Promedio
	Alta	> 2* Promedio
Complejidad de Implementación	Baja	<= Promedio
	Media	Entre Promedio y 2* Promedio
	Alta	> 2* Promedio
Reutilización	Baja	> 2* Promedio
	Media	Entre Promedio y 2* Promedio
	Alta	<=Promedio

1.6.2 Relaciones entre clases (RC)

Mide los siguientes atributos de calidad (Rodríguez Landín, 2012):

Reutilización: consiste en el grado de reutilización presente en una clase o estructura de clase, dentro de un diseño de software.

Acoplamiento: consiste en el grado de dependencia o interconexión de una clase o estructura de clase, con otras, está muy ligada a la característica de “reutilización”.

Complejidad del mantenimiento: consiste en el grado de esfuerzo necesario a realizar para desarrollar un arreglo, una mejora o una rectificación de algún error de un diseño de software. Puede influir indirecta, pero fuertemente en los costos y la planificación del proyecto.

Cantidad de pruebas: consiste en el número o el grado de esfuerzo para realizar las pruebas de calidad del producto diseñado.

Tabla 4. Atributos que afecta RC (Rodríguez Landín, 2012).

Atributo que afecta	**Modo en que lo afecta**
Está dada por el número de relaciones de uso de una clase con otras.	
Acoplamiento	Un aumento del RC implica un aumento del Acoplamiento de la clase.
Complejidad del mantenimiento	Un aumento del RC implica un aumento de la complejidad del mantenimiento de la clase.
Reutilización	Un aumento del RC implica una disminución en el grado de reutilización de la clase.
Cantidad de pruebas	Un aumento del RC implica un aumento de la Cantidad de pruebas de unidad necesarias para probar una clase.

Tabla 5. Rango de valores para la evaluación técnica de los atributos de calidad relacionados con la métrica RC (Rodríguez Landín, 2012).

Atributo	**Categoría**	**Criterio**
Acoplamiento	Ninguna	0
	Baja	1
	Media	2

	Alta	>2
Complejidad de mantenimiento	Baja	<=Promedio
	Media	Entre promedio y 2*Promedio
	Alta	>2*Promedio
Reutilización	Baja	>2*Promedio
	Media	Entre promedio y 2*Promedio
	Alta	<=Promedio
Cantidad de pruebas	Baja	<=Promedio
	Media	Entre promedio y 2*Promedio
	Alta	>2*Promedio

1.7 Pruebas de software

Las pruebas de software son un elemento crítico para la garantía de la calidad de la aplicación y representan una revisión final de las especificaciones, del diseño y de la codificación. Como parte de la fase de ejecución de la metodología seleccionada se valida la propuesta de solución mediante la estrategia de prueba. Un software se prueba para descubrir los errores cometidos. De ser realizada la acción sin ningún plan seguramente se desperdicia tiempo, un esfuerzo innecesario, y puede que no sean detectados los errores. Las pruebas miden además el grado en que el software cumple con los requerimientos definidos (Pressman, 2010).

Objetivos de las pruebas de software

Entre los objetivos de las pruebas de software pueden ser encontrados:

- Detectar errores en el software.
- Verificar la integración adecuada de los componentes.
- Verificar que todos los requisitos se han implementado correctamente.
- Identificar y asegurar que los defectos encontrados se han corregido antes de entregar el software al cliente.
- Diseñar casos de prueba que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

De acuerdo a lo planteado en (Pressman, 2010) las pruebas se rigen por una serie de principios. Una buena comprensión de los mismos facilita el posterior uso de los métodos en un efectivo diseño de casos de prueba. Dentro de estos principios se pueden encontrar:

Principios de las pruebas de software

- La prueba puede ser usada para mostrar la presencia de errores, pero nunca su ausencia.
- La principal dificultad del proceso de prueba es decidir cuándo parar.
- Evitar casos de pruebas no planificados, no reusables y triviales a menos que el programa sea verdaderamente sencillo.
- Una parte necesaria de un caso de prueba es la definición del resultado esperado.
- Los casos de pruebas tienen que ser escritos no solo para condiciones de entrada válidas y esperadas sino también para condiciones no válidas e inesperadas.
- El número de errores sin descubrir es directamente proporcional al número de errores descubiertos.

1.7.1 Pruebas Caja blanca

De acuerdo a (Pressman, 2010) la prueba de caja blanca, denominada en ocasiones prueba de caja de cristal, es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que:

- Garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo.
- Ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa.
- Ejecuten todos los bucles en y con sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.

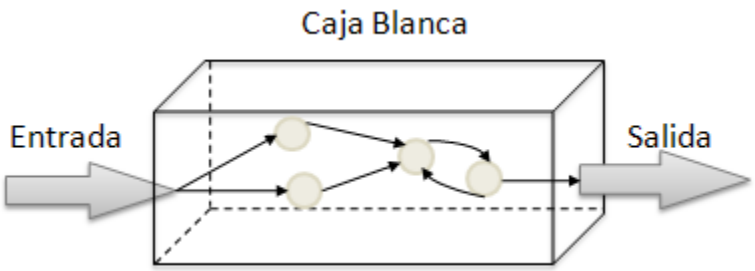

Figura 3.Representacion de técnica de pruebas de caja blanca (Pressman, 2010).

Dentro de la prueba de caja blanca, la técnica utilizada fue Camino básico. Para aplicar esta técnica se debe introducir la notación para la representación del flujo de control. El mismo puede representarse por un Grafo de flujo en el cual (Pressman, 2010):

- Cada nodo del grafo corresponde a una o más sentencias de código fuente.
- Todo segmento de código de cualquier programa se puede traducir a un Grafo de Flujo.
- Para construir el grafo se debe tener en cuenta la notación para las instrucciones.

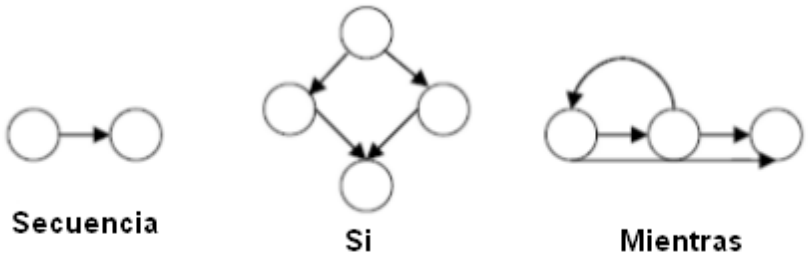

Figura 4. Notación de grafos de flujo para las instrucciones: Secuenciales, Si, Mientras (Pressman 2010).

Un grafo de flujo está formado por tres componentes fundamentales que ayudan a su elaboración y comprensión, estos brindan información para confirmar que el trabajo se está haciendo adecuadamente (Pressman, 2010).

Componentes del grafo de flujo (Pressman, 2010):

- **Nodo:** Los nodos son los círculos representados en el grafo, el cual contiene una o más secuencias del procedimiento donde un nodo corresponde a una secuencia de procesos o a una sentencia de decisión. Los nodos que no están asociados se utilizan al inicio y final del grafo.
- **Aristas:** Las aristas son las flechas del grafo. Las mismas son iguales a las representadas en un diagrama de flujo y constituyen el flujo de control del

procedimiento. Las aristas terminan en un nodo, aun cuando el nodo no representa la sentencia de un procedimiento.

- **Regiones:** Las regiones son las áreas delimitadas por las aristas y nodos donde se incluye el área exterior del grafo como una región más. Las regiones se enumeran, siendo la cantidad de regiones equivalente a la cantidad de caminos independientes del conjunto básico de un procedimiento.

1.7.2 Pruebas de Caja negra

Según plantea (Pressman, 2010) las pruebas de caja negra hacen referencia a las pruebas que se llevan a cabo sobre la interfaz del software sin tener en cuenta el código, por lo que los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene.

El diseño de estas pruebas tiene el propósito de detectar (Pressman, 2010):

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en accesos a bases de datos externas.
- Errores de rendimiento.
- Errores de inicialización y de terminación.

Figura 5. Representación de técnica de prueba de Caja negra.

Existen varias técnicas propuestas por (Pressman, 2010) para validar las funcionalidades del sistema sin entrar a analizar su ejecución interna. En el presente trabajo se aplica la técnica Partición de Equivalencia, la cual permite examinar los valores válidos e inválidos

de las entradas en el software. Para la aplicación de esta técnica se realizan los diseños de casos de prueba, los cuales se basan en la evaluación de las clases de equivalencia.

1.8 Conclusiones parciales

Luego de haberle dado cumplimiento a los objetivos trazados para el primer capítulo, se arribó a las siguientes conclusiones:

- El análisis de los sistemas de gestión de pólizas de seguros realizado, permitió identificar la necesidad de desarrollar una solución propia que dé respuesta al problema de la investigación planteado.
- Con la selección de las tecnologías a emplear, se fortalece el paradigma de independencia tecnológica por el que apuesta Cuba.
- El análisis de diferentes métricas y procedimientos para medir la calidad de atributos de software permitió seleccionar TOC y RC como métricas de validación del diseño, pues estas abarcan gran parte de los atributos de calidad de software.

CAPÍTULO 2: DISEÑO DE LA SOLUCIÓN

Introducción

En el presente capítulo se realiza una breve descripción de los requisitos funcionales y no funcionales previamente identificados por el equipo de desarrollo. Se obtienen los artefactos: diagramas de clases del diseño, y el modelo de datos donde se representan las entidades vinculadas a la propuesta de solución, así como las relaciones existentes entre ellas. Se describen los patrones de diseño y arquitectónicos empleados, finalizando con la aplicación de métricas que validan el diseño.

2.1 Requisitos analizados

Para el desarrollo del presente trabajo se parte del análisis de los requisitos funcionales y no funcionales previamente identificados en (Cobas, 2015a)(Cobas, 2015b), firmados por los clientes, y enunciados a continuación.

Requisitos Funcionales

Los requisitos funcionales (RF) permiten identificar las condiciones que debe cumplir un sistema para satisfacer un contrato, estándar, especificación u otra documentación formalmente impuesta (Barrios Iglesias, et al, 2013)(Cobas, 2015a).

Tabla 6. Requisitos Funcionales.

No	Requisito	Complejidad
1	Buscar póliza	Media
2	Consultar póliza	Baja
3	Registrar reclamación	Alta
4	Modificar reclamación	Alta
5	Buscar reclamación	Media
6	Listar reclamación	Media

7	Aprobar reclamación	Alta
8	Denegar reclamación	Alta
9	Cerrar reclamación	Alta
10	Reabrir reclamación	Alta
11	Generar requisición de pago	Alta
12	Cancelar requisición de pago	Alta
13	Buscar requisición de pago	Media
14	Listar requisición de pago	Media
15	Imprimir requisición de pago	Alta
16	Imprimir relevo de responsabilidad	Alta
17	Imprimir registro de reclamaciones	Alta

Requisitos no funcionales

Los requisitos no funcionales (RNF) permiten definir las propiedades que el producto debe tener. Estas características son las que hacen el producto más atractivo, usable, rápido y confiable. Una vez que se conozca lo que el componente debe hacer será posible determinar cómo ha de comportarse y qué cualidades debe tener. Dichas propiedades son importantes para los clientes y usuarios a la hora de valorar cuán aceptable es el producto (Barrios Iglesias, et al, 2013)(Cobas, 2015b).

Tabla 7. Requisitos no funcionales.

No	**Requisito**
1	Permitir generar reportes estándares en formatos PDF y XLS.
2	Permitir generar reportes a partir de la selección de parámetros por el usuario.

3	Permitir desplazamiento por los campos de los formularios mediante el uso del tabulador.
4	Responder al 100% de los criterios seleccionados en las búsquedas.
5	Mostrar en los mensajes de error los datos identificativos del lugar, operación o dato con error.
6	Permitir solo la entrada de datos correctos.
7	Prevenir la eliminación de información de la base de datos que se haya asociado a alguna operación.
8	Mostrar mensajes de error ante el fallo de una funcionalidad del sistema indicando la causa.
9	Representar los conceptos del dominio de la aplicación con un único ícono distintivo.
10	Representar la información de forma estándar.
11	Permitir acceder a todas las funcionalidades para entrar los datos y procesar la información a través de un menú.
12	Permitir confirmar mediante el uso del mouse o el teclado.

2.2 Diseño del sistema

El diseño es un modelo del sistema o producto que se va a construir. Debe ser suficiente para que la implementación de dicho sistema se realice sin ambigüedades, encontrando la forma para que soporte todos los requisitos y restricciones que se le suponen. Una entrada esencial en el diseño es el resultado del análisis de los requisitos identificados, mediante el cual se obtienen los artefactos enunciados a continuación (Martínez, 2012).

2.3 Arquitectura del software

La arquitectura del software es el diseño de más alto nivel de la estructura de un sistema. Consiste en un conjunto de abstracciones que forman el “marco” del software. La arquitectura se diseña en la fase posterior a la de requisitos, la llamada fase de diseño (Gómez, 2015).

Aplicación de los patrones de arquitectura en Tres Capas y MVC

Diseñar en 3 capas trata sobre no poner todo tu código en las interfaces de usuario de tu sistema. Para subsanar lo antes mencionado, la idea es tener 3 niveles de funcionalidad bien definidos, los cuales se explican en el estudio realizado en el Capítulo 1, ejemplificándose en la *Figura 2. Modelo de Arquitectura en 3 capas (Liaget, 2013).* Encontrando además en la *Figura 1. Patrón Modelo-Vista-Controlador (Smith, 2013).* la representación del patrón MVC que aunque está asociado a la idea de 3 capas, su objetivo es aún más fino. El mismo se centra en la secuencia de ejecución, desde que se produce un evento en la capa de presentación hasta que el mismo es atendido de forma completa (Ercoli, 2007). A continuación, se representa en la Figura 6 como se emplean ambos patrones en el diseño arquitectónico de la solución propuesta.

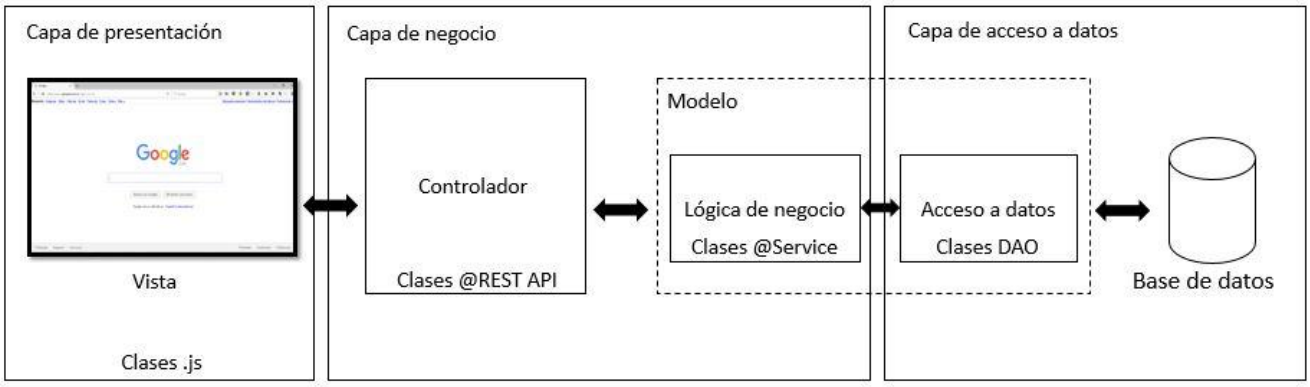

Figura 6. Representación de los patrones arquitectónicos MVC y Tres Capas.

Se puede apreciar que existen tres rectángulos exteriores cada uno representando una capa del Patrón en Tres capas, donde la capa de acceso a datos está conformada por la base de datos y las clases DAO (Sufijo que presentan las clases de acceso a datos). La capa de negocio está conformada por las clases etiquetadas con @Service y @Rest API, que no son más que las clases dedicadas a los servicios (Clases de lógica del negocio) y las controladoras respectivamente. Encontrando la capa de presentación la cual está compuesta por las clases .js que representan la vista de la solución propuesta, implementadas en el lenguaje java script sobre el marco de trabajo ExtJs 6.0.

Se puede observar además en la Figura 6 como se emplea el patrón MVC. Encontrando que el modelo está compuesto por las clases DAO y las clases etiquetadas con @Service. El controlador está conformado por las clases controladoras, que en el caso de la solución

propuesta se encuentran etiquetadas con @REST API. Estando finalmente, la vista conformada por las clases .js, como se mencionaba en el párrafo anterior.

2.3.1 Diagramas de clase del diseño.

Un diagrama de clases del diseño es un diagrama estático que describe la estructura de un sistema mostrando sus clases, operaciones y las relaciones existentes entre ellas, los mismos son una parte fundamental en el diseño de software (Pressman, 2010). A continuación, se presenta la estructura que tiene el componente Reclamación perteneciente a la solución propuesta.

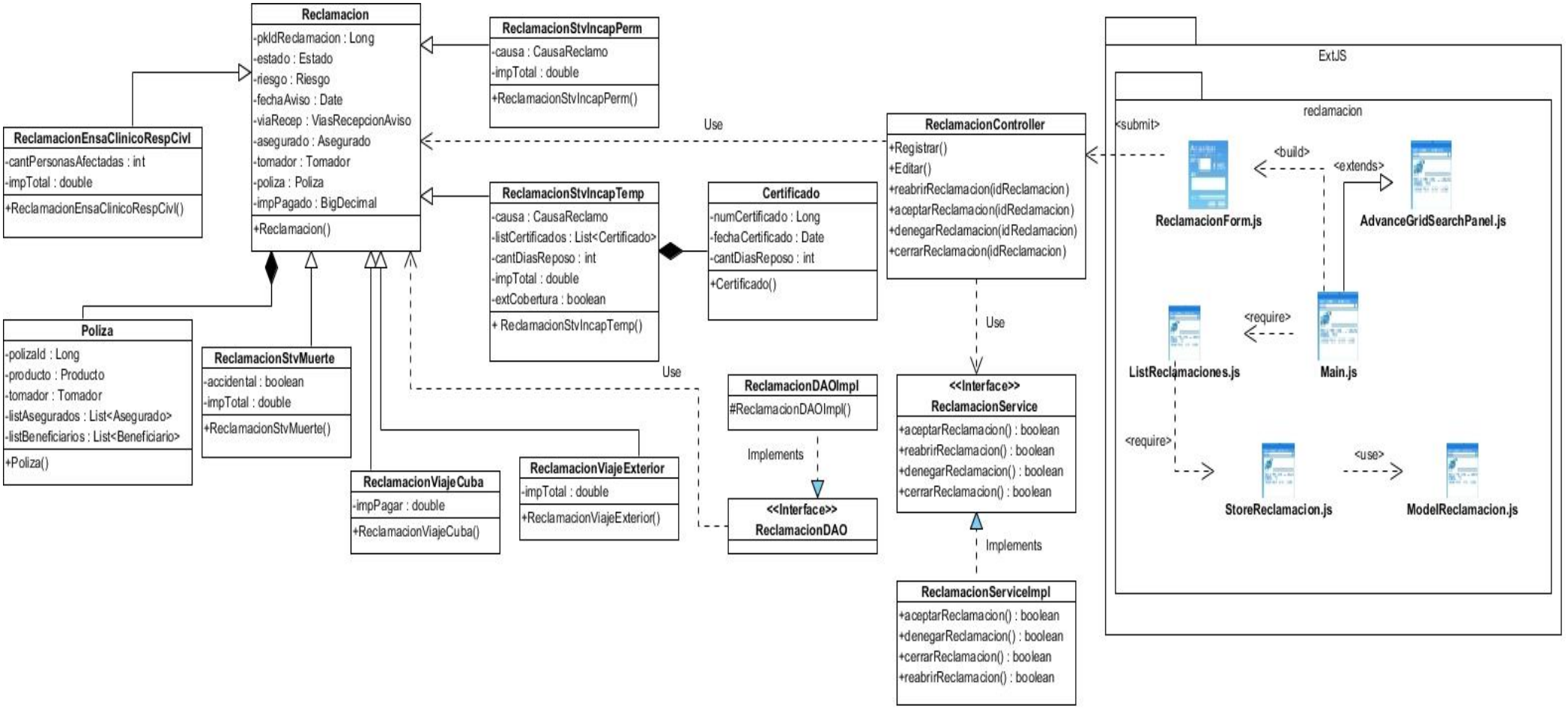

Figura 7. Diagrama de clases de diseño del componente Reclamación.

El mismo cuenta con un conjunto de clases implementadas en el lenguaje Java, y complementadas con el marco de trabajo Spring y el ORM Hibernate, las cuales se encargan de manejar la lógica del negocio, así como el acceso a la base de datos. Por otro lado, se aprecian clases implementadas en el lenguaje ExtJS, que son las encargadas de presentarle al usuario la información referente al componente de manera visual, interactuando mediante el intercambio de objetos JSONs con la clase ReclamacionController, la cual atiende las peticiones, procesa los datos a través de las clases Services y DAOs y brinda una respuesta al usuario inicial. De igual manera ocurre en el componente Requisición, cuyo diagrama se encuentra en el anexo 1.

2.4 Patrones de diseño empleados en la solución

Patrón Experto en información

El patrón experto en información es el principio básico de asignación de responsabilidades. Nos indica, por ejemplo, que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo. De este modo se obtiene un diseño con mayor cohesión y así la información se mantiene encapsulada (disminución del acoplamiento). La utilización de este patrón hace que se mantenga el encapsulamiento, los objetos utilizan su propia información para llevar a cabo sus tareas. Se distribuye el comportamiento entre las clases que contienen la información requerida y son más fáciles de entender y mantener (Buschmann, et al, 2003). El empleo de este patrón se evidencia en la implementación de las clases del dominio puesto que son expertas en la información que manejan. Pueden por lo tanto implementar las funcionalidades necesarias para darle tratamiento a las peticiones que recibe el controlador ReclamacionController.

Patrón Controlador

El patrón controlador es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado. Este patrón sugiere que la lógica de negocios debe estar separada de la capa de presentación, esto para aumentar la reutilización de código y a la vez tener un mayor control. Se recomienda dividir los eventos del sistema en el mayor número de controladores para poder aumentar la cohesión y disminuir el acoplamiento (Shaw, 2007). Una vez más la utilización de este patrón se evidencia con la implementación de las clases ReclamacionController y

RequisicionController, ambas responsables e intermediarias en el proceso de intercambio de datos entre las interfaces de la vista y las clases del negocio.

Patrón Alta cohesión

Según (Douglas Schmidt, et al, 2010), este patrón indica que la información que almacena una clase debe de ser coherente y debe estar relacionada con la misma. Un mayor grado de cohesión implica uno menor de acoplamiento. Maximizar el nivel de cohesión intramodular en todo el sistema resulta en una minimización del acoplamiento intermodular. En otras palabras, que cada clase atienda solo los datos que están directamente relacionada con la misma, evidenciándose esto en todas las clases de la solución propuesta, donde haciendo uso del patrón en cuestión se declaran los atributos de las mismas de la forma más atómica posible evitando la dependencia de otras, garantizando que los datos que contengan solo sean los necesarios y coherentes de acuerdo a la información que maneje la clase.

Patrón bajo acoplamiento

Según se define en (Douglas Schmidt, el at, 2010), es la idea de tener las clases lo menos relacionadas entre sí que se pueda. De tal forma que, en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases. El empleo de este patrón se evidencia en la Figura 4 cuando se aprecia que existe como máximo dos relaciones de uso entre las clases del diseño.

2.5 Modelo de datos (MD)

Un modelo de datos es la descripción de una Base de Datos (BD). Típicamente un modelo de datos permite describir las estructuras de datos de la base, su tipo, descripción y la forma en que se relacionan, restricciones de integridad, entre otros, es factible pensar que un modelo de datos permite describir los elementos de la realidad que intervienen en un problema dado y la forma en que se relacionan esos elementos entre sí (Jalexiscv, 2014).

El objetivo de construir un MD es identificar y representar las tablas (entidades) de importancia para el funcionamiento del negocio, sus propiedades (atributos), y la forma en que estas tablas se comunican entre sí (relaciones). Este modelo se desarrolla para facilitar el diseño de la BD y mostrar los datos que contendrá el sistema (Jalexiscv, 2014). A continuación, se observa el modelo de datos correspondiente a la solución propuesta:

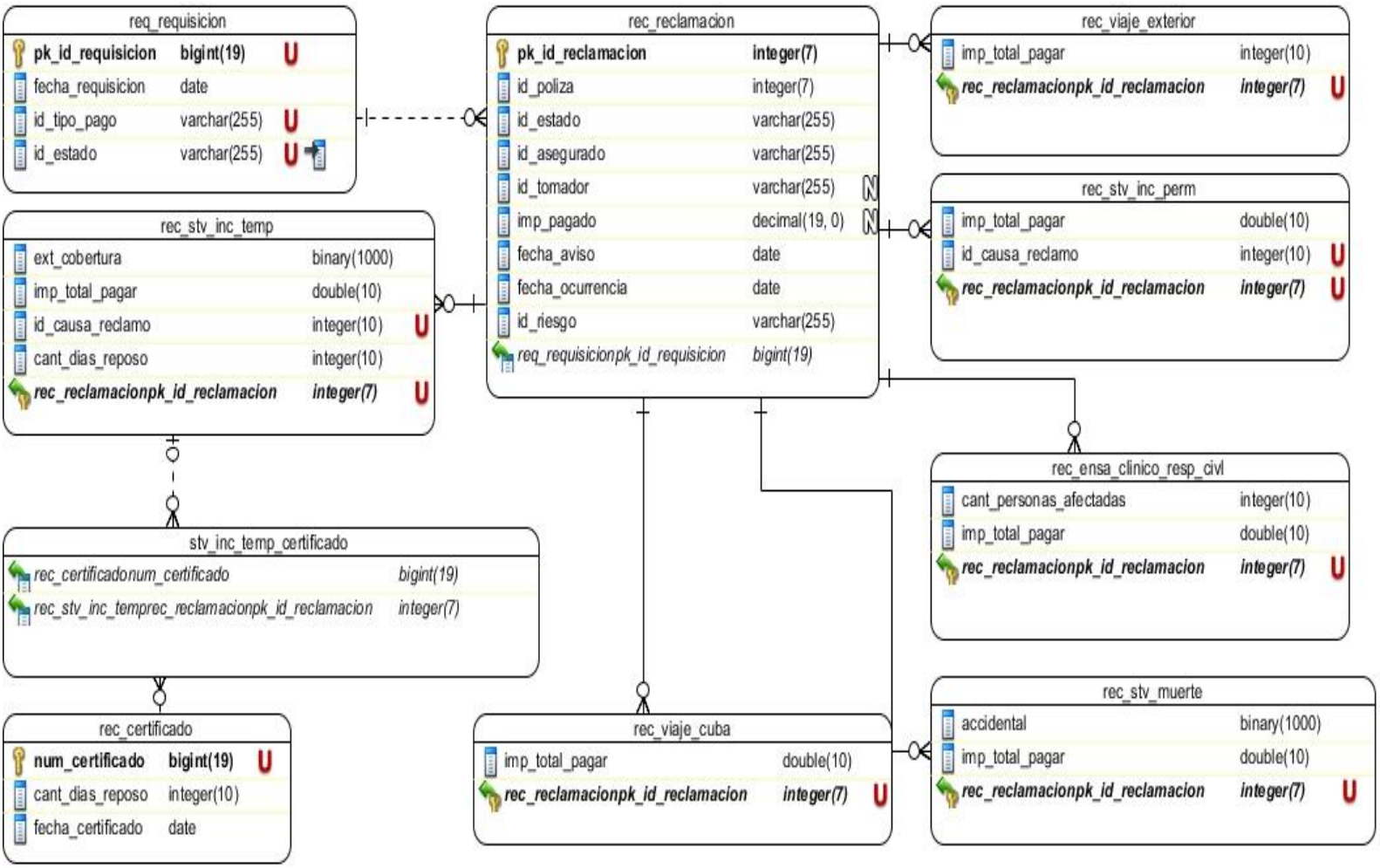

Figura 8. Modelo de datos general.

En el MD se puede apreciar como la tabla rec_reclamacion contiene los datos comunes a registrar de una reclamación, manteniendo una relación de UnoAMuchos con las tablas rec_stv_muerte, rec_stv_inc_temp, rec_stv_inc_perm, rec_viaje_cuba y rec_viaje_exterior las cuales se encargan de registrar los datos específicos de cada tipo de reclamación. Además, se puede observar la relación UnoAMuchos de la tabla req_requisicion con la tabla rec_reclamacion, donde la primera es la encargada de guardar una lista de reclamaciones aceptadas con sus correspondientes beneficiarios a ser indemnizados y la segunda los datos más significativos correspondientes al proceso de reclamación que se desencadena cuando un asegurado es víctima de un siniestro.

2.6 Validación del diseño

Con el objetivo de medir de manera cuantitativa la calidad de los atributos de software se aplican las métricas de validación del diseño (Lorenz, et al, 1994). Para llevar a cabo este proceso de validación se emplearon las métricas Tamaño Operacional de la Clase y Relación entre Clases, presentando a continuación los resultados obtenidos.

2.6.1 Métrica Tamaño Operacional de Clase

La tabla que se muestra a continuación ofrece las clases del sistema a las que se le aplicó la métrica y los resultados obtenidos para cada atributo evaluado. Para determinar el valor de los atributos, se calcula el promedio de la columna cantidad de procedimientos y este promedio (en este caso el resultado obtenido fue 22,5) es el que se tienen en cuenta para la evaluación según la columna criterio de la *Tabla 3. Rango de valores para la evaluación técnica de los atributos de calidad relacionados con la métrica (Rodríguez Landín, 2012).*

Tabla 8. Clases medidas por la métrica TOC.

No	Clase	Cantidad de Procedimientos	Responsabilidad	Complejidad	Reutilización
1	Reclamacion	21	Baja	Baja	Alta
2	ReclamacionEnsaClinicoRespCivl	25	Media	Media	Media
3	ReclamacionStvMuerte	25	Media	Media	Media
4	ReclamacionStvIncapTemp	31	Media	Media	Media
5	ReclamacionStvIncapPerm	25	Media	Media	Media
6	Certificado	7	Baja	Baja	Alta
7	ReclamacionViajeCuba	23	Media	Media	Media
8	ReclamacionViajeExterior	3	Baja	Baja	Baja

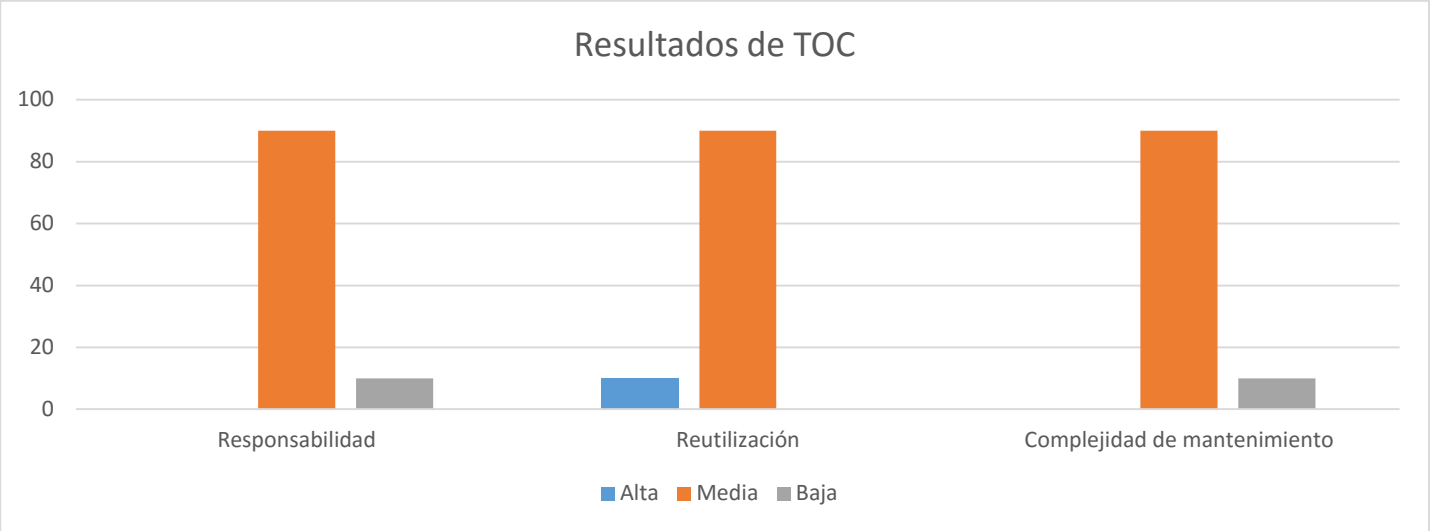

Figura 9. Resultados de TOC.

Luego de realizar un análisis de los resultados obtenidos para los atributos de la métrica TOC en la evaluación del diseño, se puede observar que la mayoría de las clases que conforman el sistema para los atributos responsabilidad y complejidad están dentro de la categoría Media y Baja para un 90% y 10% respectivamente mientras que el atributo Reutilización cuenta con iguales por cientos en las categorías Media y Alta mostrando así que el componente cuenta con un buen grado de reutilización, baja complejidad y responsabilidad en el diseño propuesto. Por lo que se concluye que los resultados obtenidos según esta métrica son positivos.

2.6.2 Métrica Relaciones entre Clases

La Tabla 10 ofrece las clases del sistema a las que se le aplicó la métrica y los resultados obtenidos para cada atributo evaluado. El valor utilizado para evaluar los atributos, es el resultado del cálculo del promedio de la columna Cantidad de relaciones de uso *Tabla 5. Rango de valores para la evaluación técnica de los atributos de calidad relacionados con la métrica RC (Rodríguez Landín, 2012).*Donde el valor obtenido fue 1.

Tabla 9. Aplicación de la métrica Relación entre clases.

No	Clase	Cantidad de relaciones de uso	Acoplamiento	Complejidad de mantenimiento	Reutilización	Cantidad de pruebas

1	Reclamacion	6	Alto	Alta	Baja	Alta
2	ReclamacionEnsaClcinicoRespCivl	0	Ninguno	Baja	Alta	Baja
3	ReclamacionStvMuerte	0	Ninguno	Baja	Alta	Baja
4	ReclamacionStvIncapTemp	1	Bajo	Baja	Alta	Baja
5	ReclamacionStvIncapPerm	1	Bajo	Baja	Alta	Baja
6	Certificado	0	Ninguno	Baja	Alta	Baja
7	ReclamacionViajeCuba	0	Ninguno	Baja	Alta	Baja
8	ReclamacionViajeExterior	0	Ninguno	Baja	Alta	Baja

Resultados de la aplicación de la métrica RC

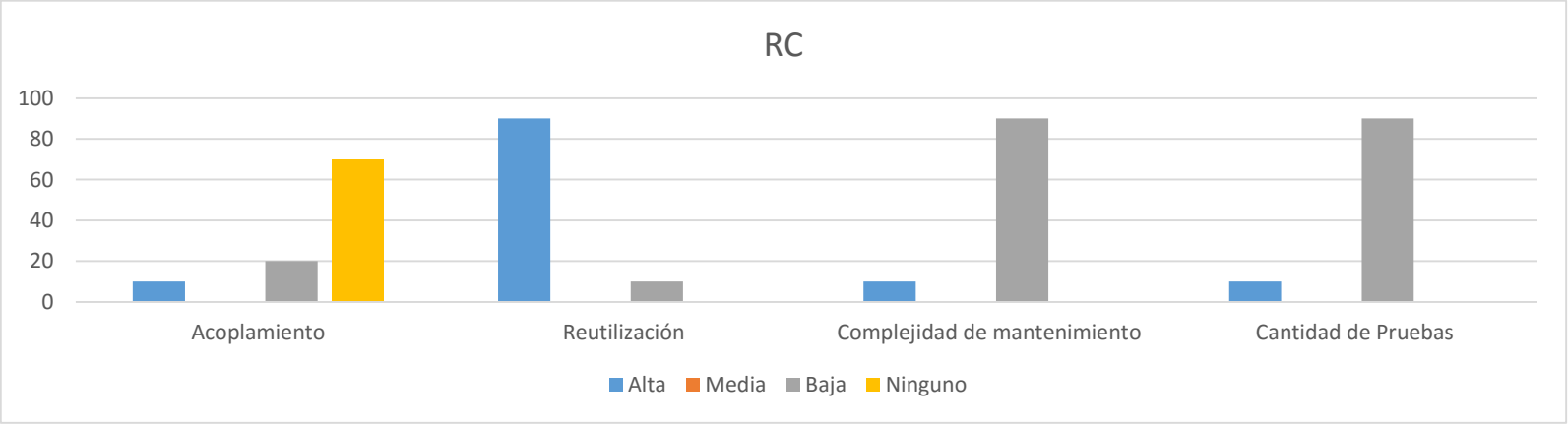

Figura 10. Resultados de RC.

Los resultados obtenidos durante la aplicación de la métrica RC demuestran que las clases del diseño posee un bajo acoplamiento, ya que para este atributo las categorías Ninguno y Bajo sumaron un 90% del total, mostrando además un 90% y 10% en las categorías Alta y Baja del atributo reutilización respectivamente. Los atributos complejidad de mantenimiento y cantidad de pruebas, obtuvieron un 90% en la categoría Baja, lo que demuestra que no es necesario un elevado esfuerzo en el momento de realizar cambios, rectificaciones y pruebas al software. Concluyendo que el resultado final de la métrica es positivo, así como el de TOC, implicando que el diseño en general de la aplicación es lo suficientemente bueno para obtener un software de alta calidad.

2.7 Conclusiones parciales

Luego de haberle dado cumplimiento a los objetivos trazados para el segundo capítulo, se arribaron a las siguientes conclusiones:

- El diseño de los componentes: Reclamación y Requisición de pago permite una mejor definición de la implementación.

- Los resultados de la validación del diseño realizada, tributan al desarrollo de una aplicación de alta calidad.

CAPÍTULO 3: IMPLEMENTACIÓN, PRUEBAS Y VALIDACIÓN

Introducción

En el presente capítulo se describen los estándares de codificación empleados durante la implementación de los componentes, así como la estructura de los mismos. Se muestran los resultados de las pruebas efectuadas al software y se presenta la estrategia que se sigue para la validación de la investigación.

3.1 Estándar de codificación

Para un proyecto de desarrollo de software se definen estándares de codificación puesto que un estilo de programación homogéneo permite que todos sus participantes puedan entenderlo en menos tiempo, facilitando la legibilidad y mantenibilidad del código. Para la implementación de la solución propuesta se empleó como guía de estándares el documento (Escobar, 2015). A continuación, se muestran ejemplos de su uso:

Comentarios iniciales de clases o interfaces:

Todos los ficheros fuente deben contener un comentario donde se lista la documentación de la clase y la información del autor y la fecha de creación (Escobar, 2015), como se muestra en la Figura 11:

```
/* La clase Requisicion es la encargada de liberar los fondos a favor de los beneficiarios y terceros afectados.
 *
 * Created by Diego on 11/05/2016.
 */
```

Figura 11. Comentario de inicio de la clase Requisicion.

Declaraciones de clases e interfaces:

La siguiente lista describe las partes de la declaración de una clase o interfaz, en el orden en que deberían aparecer (Escobar, 2015):

- Sentencia class o interface
- Comentario de implementación de la clase o interface si fuera necesario (/*...*/): Este comentario debe contener cualquier información aplicable a toda la clase o interfaz que no era apropiada para estar en los comentarios de documentación.

- Variables de clase (static): Primero las variables de clase public, después las protected, después las de nivel de paquete (sin modificador de acceso), y después las private.
- Variables de instancia: Primero las public, después las protected, después las de nivel de paquete (sin modificador de acceso), y después las private.
- Métodos: Los métodos se deben agrupar por funcionalidad más que por visión o accesibilidad. Por ejemplo, un método de clase privado puede estar entre dos métodos públicos de instancia. El objetivo es hacer el código más legible y comprensible.

```
public class Requisicion implements Serializable{
    //Atributos de la clase
    private long pkIdRequisicion;
    private List<Reclamacion> listReclamaciones;
    private Date fechaRequisicion;
    private TipoPago tipoPago; //(Total o parcial)
    private List<TerceroAfectado> listTercAfectados;
    private Estado estadoRequisicion;
```

Figura 12. Ejemplo de declaración de clase y orden de los atributos.

Estructura de las clases controladoras:

Las clases controladoras se nombran a partir del nombre del recurso que manejan y el sufijo Controller y heredan de las clases abstractas AbstractController o RestCRUDController dependiendo de las particularidades de cada clase controladora. Las clases deben contener las anotaciones @RestController y @RequestMapping ("/api/nombre_recurso"). Todos los métodos que forman parte del api REST deben tener la anotación @RequestMapping y emplearán el atributo method de la siguiente manera (Escobar, 2015):

- GET: acciones de obtención
- POST: acciones de inserción
- PUT: acciones de actualización
- DELETE: acciones de eliminación

```
public class ReclamacionController extends AbstractController {
```

Figura 13. Declaración de una clase controladora.

```
@RequestMapping(
    value = {"/buscaraseguradoo"},
    method = {RequestMethod.GET}
)
public Map<String, Object> buscarAseguradoPorCI(@RequestParam("nocarnet") String nocarnet) {
    HashMap map = new HashMap(0);
    Asegurado asegurado = this.aseguradoService.buscarAseguradoPorCI(nocarnet);
    map.put("existe", Boolean.valueOf(asegurado != null));
    map.put("data", asegurado);
    return map;
}
```

Figura 14. Declaración de métodos, empleando la anotación @RequestMapping.

3.2 Diagrama de componentes

Se utilizan para modelar la vista estática de un sistema. Muestra la organización y las dependencias entre un conjunto de componentes. No es necesario que un diagrama incluya todos los componentes del sistema, normalmente se realizan por partes. Cada diagrama describe un apartado del sistema (Programacion.net, 2013).

La solución final consta de los componentes Reclamación y Requisición, los cuales consumen servicios de otros como: Concertación, Configuraciones, Provisiones, Contabilidad y Nomencladores y brindan servicios a Reportes y Seguimiento. Además, el componente Reclamación, encargado de registrar los reclamos provenientes de los asegurados al ser víctimas de un siniestro, brinda servicios al componente de Requisición, el cual se encarga de registrar las reclamaciones aceptadas que serán liquidadas. Quedando el Diagrama de componentes estructurado de la siguiente manera:

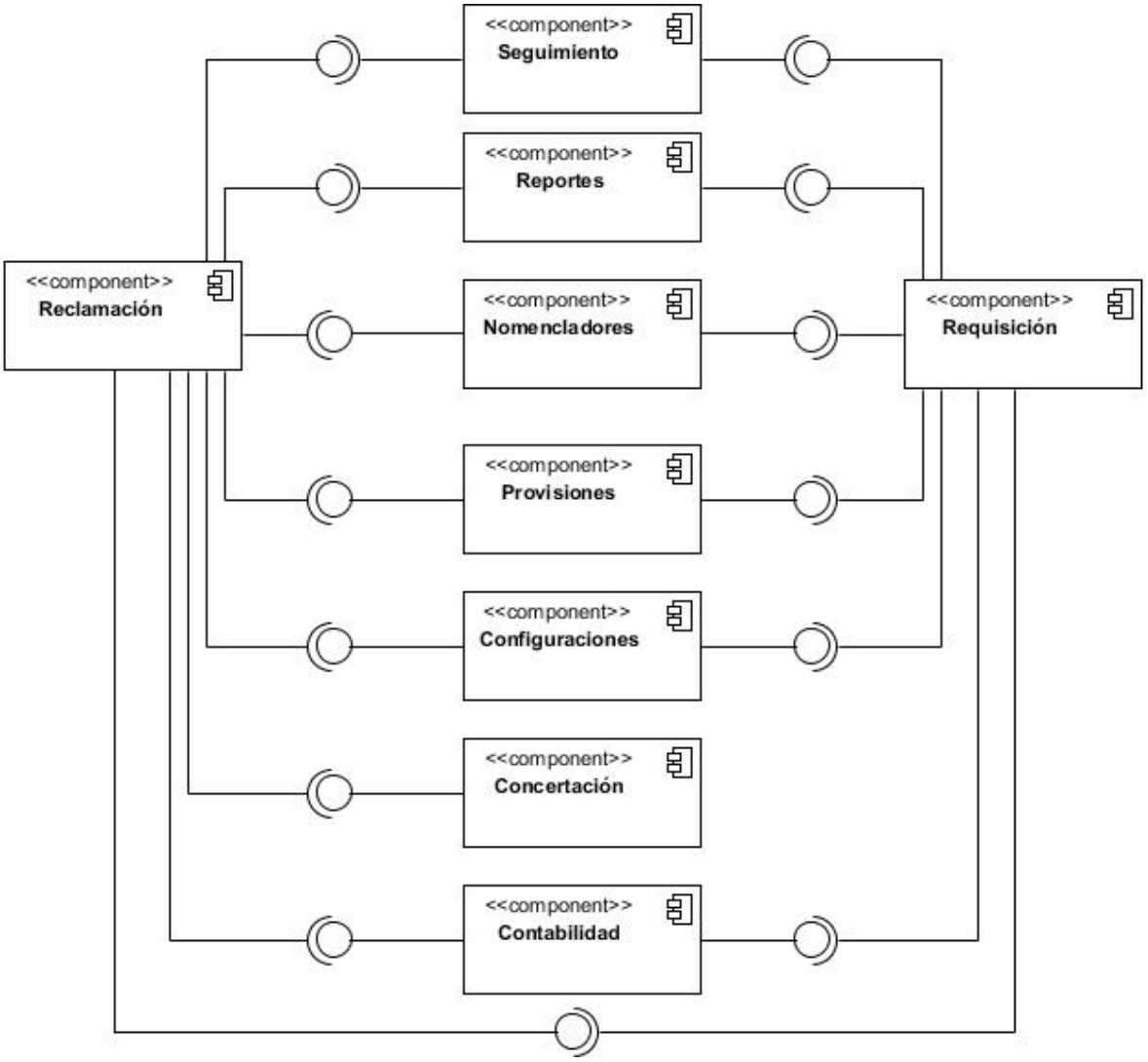

Figura 15. Diagrama de componentes.

La Figura 16 muestra la estructura de carpetas para el código java propuesta por el proyecto de producción empleando el entorno de desarrollo Intellij IDEA, donde se pueden apreciar las siguientes carpetas:

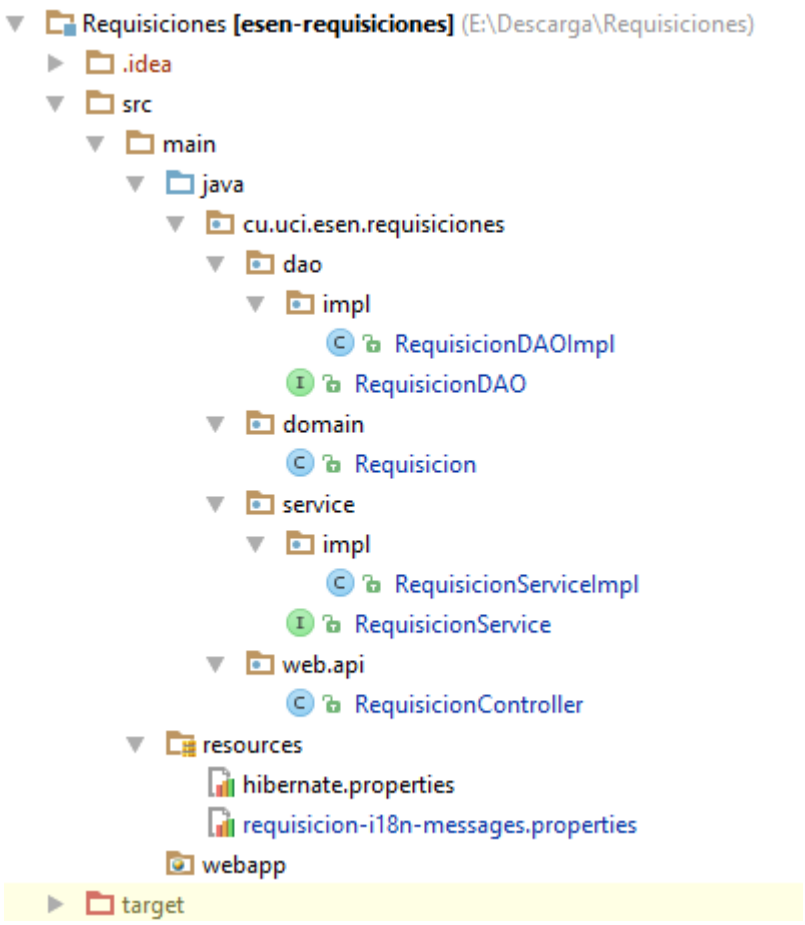

Figura 16. Estructura de carpetas del componente Requisición.

- .idea: contiene la configuración base del proyecto, y algunas librerías necesarias para el funcionamiento del componente.

- src/main/java/cu/uci/esen/requisiciones/dao: contiene las clases de acceso a datos o DAOs, encargadas de interactuar directamente con la base de datos, las cuales se dividen en interfaces, nombradas con el sufijo DAO, definiendo dentro de la carpeta impl, las clases que implementan dichas interfaces nombradas igual que la interfaz que implementan seguidas por el sufijo DAOImpl.

- src/main/java/cu/uci/esen/requisiciones/domain: contiene las clases del dominio, las cuales representan los objetos que interactúan en el proceso requisición, implementando el comportamiento de los mismos.

- src/main/java/cu/uci/esen/requisiciones/service: contiene las clases del negocio, encargadas de implementar los métodos que interactúan con la vista a través de los

controladores, enviando los datos solicitados. Al igual que las DAOs, las interfaces Service, se componen por el nombre de la clase del dominio que representan seguido por el sufijo Service, y las clases que implementan dichas interfaces, se nombran según la interfaz que implementan seguido del sufijo ServiceImpl.

- src/main/java/cu/uci/esen/requisiciones/web/api: contiene las clases controladoras, las cuales funcionan como intermediarias entre la vista y el modelo. Se nombran de acuerdo a la clase del dominio que referencian, seguido por el sufijo Controller.

- Src/main/resource: contiene los archivos hibernate.properties con la configuración para acceder a la base de datos, y requisicion-i18n-messages.properties en donde se plasman todos los mensajes que se lanzarán a la vista.

- target: carpeta que es creada en tiempo de ejecución. La misma almacena el proyecto compilado, permitiendo agilizar las futuras compilaciones del mismo.

A continuación, se muestra la estructura de la capa de presentación del componente Requisición implementada en Java Script empleando el entorno de desarrollo Web Storm, donde se pueden apreciar las siguientes carpetas:

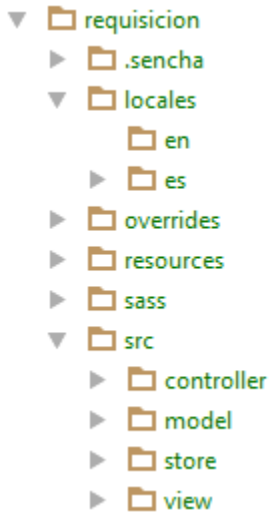

Figura 17. Estructura de carpetas del componente Requisición (Capa de presentación).

- .sencha: carpeta que contiene los archivos de configuración de Sencha cmd pertenecientes al componente.
- locales: contiene los distintos ficheros de internacionalización que se deseen añadir al componente.

- overrides: contiene las clases que serán automáticamente requeridas por el componente.

- resources: contiene los recursos estáticos que emplea el componente, ejemplo, imágenes.

- sass: contiene archivos sass[7] de varios tipos, empleados en el tema visual de las ventanas.

- src: contiene las clases que conforman el código base del componente agrupadas en subcarpetas con la siguiente organización:

    - controller: contiene las clases controladoras, nombradas haciendo uso del sufijo Controller. Encargadas de enviar las peticiones de la vista a los controladores del lenguaje Java y recibir las respuestas de los mismos.

    - model: contiene los modelos o clases, que se crean con el propósito de obtener datos de objetos que se encuentran en el código java, siguiendo la estructura de sus atributos.

    - store: contiene clases encargadas de funcionar como almacenes de modelos, generalmente se crea uno por cada modelo con que cuente el componente.

    - view: contiene las clases encargadas de dibujar lo que visualiza el usuario. Ejemplo: ventanas, formularios y paneles.

3.3 Aplicación de pruebas de software.

En el presente acápite se presentarán los resultados obtenidos de la aplicación de las pruebas de caja blanca y caja negra, a través de las técnicas camino básico y particiones de equivalencia respectivamente.

3.3.1 Pruebas de caja blanca

Para realizar la prueba del Camino básico es preciso calcular la complejidad ciclomática del algoritmo o fragmento de código a analizar. A continuación, se muestra un fragmento de

[7] Sass: Es un lenguaje de script que es traducido a CSS.

código del método registrarReclamacion() encargado de insertar la reclamación en la base de datos, según del tipo que sea la misma.

```
switch (((JSONObject) jsonObject.get("general")).getInt("producto")) {
    case 1:  //Ensayos clinicos
        ReclamacionEnsaClinicoRespCivl recl = new ReclamacionEnsaClinicoRespCivl(
                Long.parseLong("0"),
                new Estado(12),
                new Riesgo(((JSONObject) jsonObject.get("general")).getInt("riesgoRec")),
                DateUtils.parseDate(((JSONObject) jsonObject.get("general")).getString("fechaAviso"), "dd-MM-yyyy"),
                DateUtils.parseDate(((JSONObject) jsonObject.get("general")).getString("fechasiniestro"), "dd-MM-yyyy"),
                new ViasRecepcionAviso(((JSONObject) jsonObject.get("general")).getInt("viaRecepAviso")),
                aseguradoService.buscarAseguradoPorCI(((JSONObject) jsonObject.get("general")).getString("noCarnetAsegurado")),
                poliza.getTomador(),
                poliza,
                BigDecimal.ZERO,
                ((JSONObject) jsonObject.get("producto")).getJSONObject("ensayosClinicos").getInt("cantpersonas"),
                ((JSONObject) jsonObject.get("producto")).getJSONObject("ensayosClinicos").getInt("importepagar")

        );
        reclamacionService.registrar(recl);
        break;
```

Figura 18. Fragmento de código del método registrarReclamacion().

El método registrarReclamacion() consiste en un switch() que consta de cuatro casos según sea el producto de la reclamación a registrar. Contando además con que el caso 4 presenta una sentencia if(), donde se pregunta qué riesgo presenta la reclamación, ya que para dicho producto se pueden registrar 3 tipos de reclamaciones según sea el riesgo asociado al producto. A continuación, se describe el proceso seguido para aplicar la técnica de Camino básico.

El primer paso es calcular la complejidad ciclomática, para lo que es necesario representar el grafo de flujo asociado al método antes presentado a través de nodos, aristas y regiones, quedando como se muestra en la Figura 17:

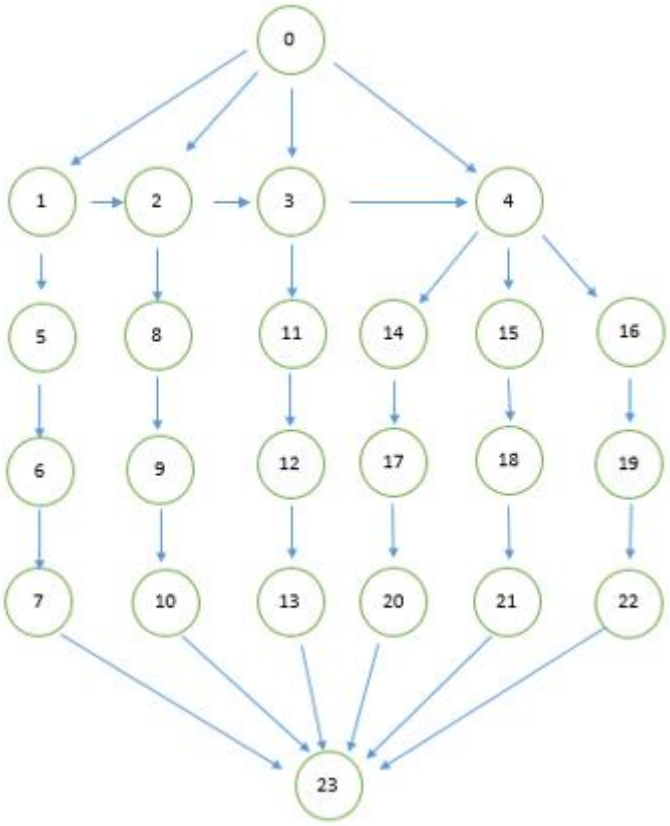

Figura 19. – Grafo de flujo asociado al método registrarReclamacion().

Una vez construido el grafo de flujo asociado al procedimiento anterior se determina la complejidad ciclomática, el cálculo es necesario efectuarlo mediante tres vías o fórmulas, se debe utilizar el mismo grafo en cada caso:

Fórmula 1 V(G) = (A – N) + 2

- Siendo “A” la cantidad total de aristas y “N” la cantidad total de nodos.

Resultado:

V (G) = (28-24) + 2

V (G) = 6

Fórmula 2 V(G) = P + 1

- Siendo “P” la cantidad total de nodos predicados (son los nodos de los cuales parten dos o más aristas).

Resultado:

Nodos predicados: 0, 1, 2, 3, 4.

V (G) = 6

Fórmula 3 V(G) = R

- Siendo “R” la cantidad total de regiones, se incluye el área exterior del grafo, contando como una región más.

Resultado:

V (G) = 6

Seguidamente es necesario especificar los caminos básicos que puede tomar el algoritmo durante su ejecución.

Camino básico # 1: 0, 1, 5, 6, 7, 23.

Camino básico # 2: 0, 2, 8, 9, 10, 23.

Camino básico # 3: 0, 3, 11, 12, 13, 23.

Camino básico # 4: 0, 4, 14, 17, 20, 23.

Camino básico # 5: 0, 4, 15, 18, 21, 23.

Camino básico # 6: 0, 4, 16, 19, 22, 23.

Después de haber extraído los caminos básicos del flujo, se procede a ejecutar los casos de pruebas. Para este procedimiento, se debe realizar al menos un caso de prueba por cada camino básico. Para realizarlos es necesario cumplir con las siguientes exigencias:

- Descripción: Se hace la entrada de datos necesaria, validando que ningún parámetro obligatorio pase nulo al procedimiento o no se entre algún dato erróneo.
- Condición de ejecución: Se especifica cada parámetro para que cumpla una condición deseada para ver el funcionamiento del procedimiento.
- Entrada: Se muestran los parámetros que entran al procedimiento.
- Resultados esperados: Se expone el resultado que se espera que devuelva el procedimiento.
- Resultados: Se muestra el resultado obtenido.
- Salida: Se presenta el valor final.

Tabla 10. Caso de prueba para el Camino básico 1.

Camino básico #1: 0, 1, 5, 6, 7, 23.	
Descripción	A partir del JSON enviado de la vista que contiene una reclamación, se obtiene el id de su

	producto y si es “1” se registra la reclamación. De ser otro caso, el algoritmo busca el próximo camino.
Condición de ejecución	Se debe tener el producto de la reclamación (idProducto).
Entrada	jsonObject.get(**"general"**).getInt(**"producto"**)= 1.
Resultados esperados	El id del producto de la reclamación es “1”, por lo que se debe registrar la misma.
Resultados	La reclamación es registrada.
Salida	N/A

Tabla 11. Caso de prueba para el camino básico 2.

Camino básico #2: 0, 2, 8, 9, 10, 23.	
Descripción	A partir del JSON enviado de la vista que contiene una reclamación, se obtiene el id de su producto y si es “2” se registra la reclamación. De ser otro caso, el algoritmo busca el próximo camino.
Condición de ejecución	Se debe tener el producto de la reclamación (idProducto).
Entrada	jsonObject.get(**"general"**).getInt(**"producto"**)= 2.
Resultados esperados	El id del producto de la reclamación es “2”, por lo que se debe registrar la misma.

Resultados	La reclamación es registrada.
Salida	N/A

Tabla 12. Caso de prueba para el camino básico 3.

Camino básico #3: 0, 3, 11, 12, 13, 23.	
Descripción	A partir del JSON enviado de la vista que contiene una reclamación, se obtiene el id de su producto y si es “3” se registra la reclamación. De ser otro caso, el algoritmo busca el próximo camino.
Condición de ejecución	Se debe tener el producto de la reclamación (idProducto).
Entrada	jsonObject.get(**"general"**).getInt(**"producto"**)= 3.
Resultados esperados	El id del producto de la reclamación es “3”, por lo que se debe registrar la misma.
Resultados	La reclamación es registrada.
Salida	N/A

Tabla 13. Caso de prueba para el camino básico 4.

Camino básico #4: 0, 4, 14, 17, 20, 23.	
Descripción	A partir del JSON enviado de la vista que contiene una reclamación, se obtiene el id de su producto y si es “4” se pregunta si el id de riesgo asociado a la reclamación es 4, entonces se registra la reclamación. De ser otro caso, el algoritmo busca el próximo camino.

Condición de ejecución	Se debe tener el producto de la reclamación y el riesgo. (idProducto)(idRiesgo).
Entrada	jsonObject.get(**"general"**).getInt(**"producto"**)= 4. jsonObject.get(**"general"**)).getInt(**"riesgoRec"**)=4
Resultados esperados	El id del producto de la reclamación es “4” y el de riesgo igual, por lo que se debe registrar la misma.
Resultados	La reclamación es registrada.
Salida	N/A

Tabla 14. Caso de prueba para el camino básico 5.

Camino básico # 5: 0, 4, 15, 18, 21, 23.	
Descripción	A partir del JSON enviado de la vista que contiene una reclamación, se obtiene el id de su producto y si es “4” se pregunta si el id de riesgo asociado a la reclamación es 5, entonces se registra la reclamación. De ser otro caso, el algoritmo busca el próximo camino.
Condición de ejecución	Se debe tener el producto de la reclamación y el riesgo. (idProducto)(idRiesgo).
Entrada	jsonObject.get(**"general"**).getInt(**"producto"**)= 4. jsonObject.get(**"general"**)).getInt(**"riesgoRec"**)=5
Resultados esperados	El id del producto de la reclamación es “4” y el de riesgo es 5., por lo que se debe registrar la misma.
Resultados	La reclamación es registrada.
Salida	N/A

Tabla 15. Caso de prueba para el camino básico 6.

Camino básico # 6: 0, 4, 16, 19, 22, 23.	
Descripción	A partir del JSON enviado de la vista que contiene una reclamación, se obtiene el id de su producto y si es “4” se pregunta si el id de riesgo asociado a la reclamación es 6, entonces se registra la reclamación. De ser otro caso, el algoritmo busca el próximo camino.
Condición de ejecución	Se debe tener el producto de la reclamación y el riesgo. (idProducto)(idRiesgo).
Entrada	jsonObject.get(**"general"**).getInt(**"producto"**)= 4. jsonObject.get(**"general"**)).getInt(**"riesgoRec"**)=6
Resultados esperados	El id del producto de la reclamación es “4” y el de riesgo es 6, por lo que se debe registrar la misma.
Resultados	La reclamación es registrada.
Salida	N/A

Como resultado de la aplicación de pruebas de caja blanca al sistema fueron analizadas las funcionalidades del mismo y no se detectan errores. Esto permite afirmar que el flujo de trabajo de las funciones es correcto.

3.3.2 Pruebas de caja negra

Para la aplicación de la prueba de caja negra se realizaron 17 diseños de casos de prueba (DCP) uno por cada requisito funcional definido, los cuales se basan en una evaluación de las clases de equivalencia para una condición de entrada.

Como resultado de la aplicación de las pruebas de caja negra fueron identificadas un total de 21 No Conformidades (NC), las cuales fueron corregidas a lo largo de tres iteraciones. La relación de NC y la clasificación de las mismas se representa de la siguiente forma:

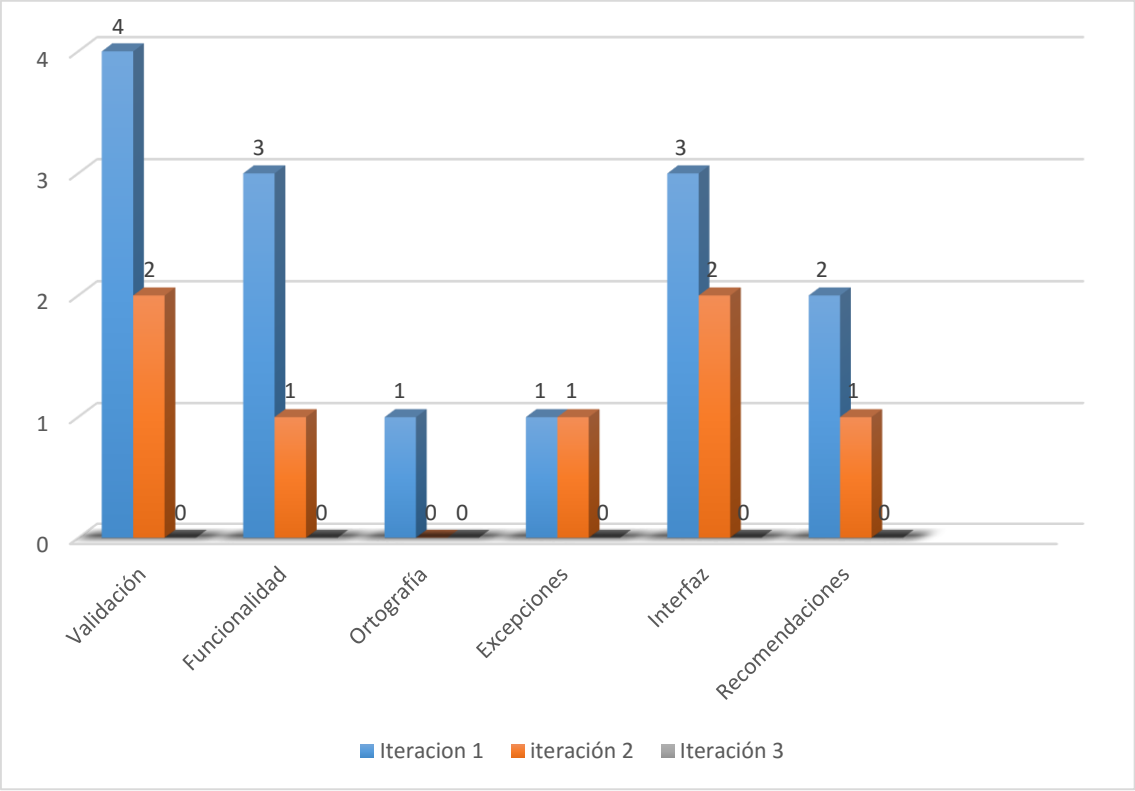

Figura 20. No conformidades detectadas aplicando pruebas de caja negra.

La ejecución de las pruebas de caja negra permitió demostrar que las funciones del software son operativas a través de los diseños de casos de pruebas, y que las interfaces se encuentran libres de errores.

3.4 Validación de la solución

En el presente acápite se estará analizando cómo a través de la obtención de los componentes Reclamación y Requisición de pago se da solución al problema de investigación. Las variables dependientes contenidas en el mismo son:

- Pérdida de la información.
- Dificultades en el acceso a la información.
- Comisión de errores humanos.

Pérdida y dificultad de acceso a la información: Los componentes obtenidos brindan funcionalidades que permiten al usuario gestionar la impresión de documentos, así como su almacenamiento en bases de datos. Evitando el deterioro al que eran expuestos los

mismos, y agilizando a su vez el proceso de acceso a la información mediante la aplicación de búsquedas avanzadas. Con la normalización de la base de datos a la tercera forma, se resuelven además problemas de redundancia, actualización e integridad de la información.

Comisión de errores humanos: la interfaz visual de la solución propuesta rellena de forma automática la gran parte de los campos de los formularios mostrados, limitando al usuario a que solo pueda modificar la menor cantidad de datos. Además, durante la implementación de los componentes se aplicaron mecanismos de validación con el objetivo de que no sea posible enviar datos erróneos al servidor, aunque no se pudo mitigar la incurrencia de los usuarios en errores voluntarios, los cuales constituyen ilegalidades.

Pruebas de aceptación

Como parte fundamental del proceso de la validación de un software se encuentran las pruebas de aceptación. Dichas pruebas tienen como fin validar que el sistema cumpla con los requisitos básicos de funcionamiento esperado y permitir que el cliente determine su aceptación (Pressman, 2010).

Para su realización el cliente prueba el sistema introduciendo datos reales con el objetivo de revelar errores, omisiones en la definición de requerimientos del sistema o problemas donde los recursos del software no cumplan las necesidades del usuario. Que un software sea aceptado implica que debe cumplir con un 85% de satisfacción del cliente. Para el 15% restante se establecerá un plan de corrección, el que una vez terminado obligará a ejecutar nuevamente el plan de pruebas de aceptación (Pressman, 2010).

En el caso de la solución propuesta se cumple con los requerimientos del cliente satisfactoriamente, obteniendo así su aceptación.

3.5 Conclusiones parciales

Por lo antes expuesto en el presente capítulo se puede arribar a las siguientes conclusiones:

- Los estándares de programación definidos permitieron una mayor organización y comprensión del código de la solución.

- Con las pruebas de caja negra realizadas se hizo posible detectar un conjunto de No Conformidades que tributaron a elevar la calidad de la solución propuesta.

- La realización de pruebas de caja blanca efectuadas al código, permite afirmar que el flujo de trabajo de las funciones es correcto.

- Se realizó la validación de la solución, obteniendo como resultado una aplicación funcional que satisface la propuesta de solución planteada en el presente trabajo.

Conclusiones generales

Con la realización del presente trabajo se logró la obtención de los componentes Reclamación y Requisición de pago cumpliendo con los requisitos propuestos inicialmente, con ello se contribuye al proceso de informatización del país y se posibilita una mejor gestión de los datos dentro de la Empresa de Seguros Nacionales.

- Se realizó el análisis de varios sistemas que gestionan el proceso de reclamaciones, con el cual se concluyó que era más factible el desarrollo de una solución propia que respondiera a las particularidades de la economía cubana y cumpliera con el paradigma de independencia tecnológica por el que aboga el país.
- Se efectuó el modelado del sistema, contribuyendo a un ahorro considerable de tiempo en la construcción del mismo y la reutilización de código.
- Se realizó la validación del diseño aplicando métricas donde los resultados arrojados demuestran la presencia de valores positivos en los indicadores de calidad medidos.
- Siguiendo los estándares de programación definidos se alcanzó una mejor organización y comprensión del código de la solución propuesta.
- Como parte del proceso verificación se realizaron pruebas que permitieron minimizar la ocurrencia de errores en los componentes antes de su implantación.
- Se realizó la validación de la solución, obteniendo como resultado una aplicación funcional que satisface la propuesta de solución planteada en el presente trabajo.

BIBLIOGRAFÍA

ABCDATOS.COM, 2009, ABCdatos. [online]. 2009. [Accessed 10 June 2016]. Available from: http://ABCdatos.com

ANDRESEN, Jacob K., GARCIA, Jesus and GRISOGONO, Grgur, 2014, *ExtJS in Action*. 2nd. Manning Publications Co.

BARRIOS IGLESIAS, Claudia and PÉREZ CINTRÓN, Fernando, 2013, *Sistema de análisis web para el entorno virtual del Sistema de Apoyo a la Municipalización en el Ministerio del Poder Popular para la Educación Universitaria en Venezuela.*

BASS, L, CLEMENTS, P and KAZMAN, R, 2005, *Software Architecture in Practice.* Segunda.

BIEMAN, J M and OTT, L M, 1994, *Measuring Functional Cohesion.*

BREIDENBACH, Ryan and WALLS, Craig, 2011, *Spring in Action* [online]. Tercera. Manning Publications Co. [Accessed 10 February 2016]. Available from: http://www.gocit.vn/files/Spring.in.Action.3rd.Edition-www.gocit.vn.pdf

CARD, D and GLASS, N R, 1990, *Measuring Software Design Quality.* Prentice-Hall.

CASTRO RUZ, Fidel, 1997, *Decreto Ley No.177*. 5 September 1997.

CASTRO RUZ, Raúl, 2009, *Decreto Ley No.263* [online]. Enero 2009. [Accessed 10 March 2016]. Available from: http://www.gacetaoficial.cu/

COBAS, Ing. Leydis, 2015a, *CEIGE_ESEN_Descipcion_Requisitos_Reclamaciones*. 2015.

COBAS, Ing. Leydis, 2015b, *CEIGE_ESEN_Especificacion_de_requisitos_de_software*.

2015.

DANIEL GALANTI, 2013, SysOne. [online]. 2013. Available from: http://www.sysone.com/la-importancia-de-las-companias-de-seguros/

DEVELOPER CENTER, Softonic, 2016, Poliza-Win. [online]. 2016. [Accessed 30 May 2016]. Available from: http://poliza-win.softonic.com/

DHAMA, H, 2000, *Quantitative Models of Cohesion and Coupling in Software.* Journal of Systems and Software.

DOUGLAS SCHMIDT, Y OTROS, 2010, *Pattern-Oriented Software Architecture*. Patterns for Concurrent and Networked Objects.

ERCOLI, Jorge, 2007, Arquitectura de Sistemas Informáticos. [online]. 2007. [Accessed 3 May 2016]. Available from: http://metodologiasdesistemas.blogspot.com/2007/05/diseo-en-3-capas-fisicas-lgicas-es.html

ESCOBAR, Eduardo, 2015, *CEIGE_ESEN_Estandar_de_Codificacion_Java*. 2015.

ESCOBAR, Ing. Eduardo Rojas, 2015, *CEIGE_ESEN_Arquitectura_vista_de_entorno_de desarrollo_tecnológico*. 2015.

FITZPATRICK, Brian W. and COLLINS, Ben, 2012, *Control de versiones con Subversion*.

FRANK BUSCHMANN Y REGINE MEUNIER, 2003, *Pattern-oriented software architecture*.

GÓMEZ, Victor, 2015, Arquitectura en Tres Capas. [online]. 2015. [Accessed 3 May 2016]. Available from: http://instintobinario.com/arquitectura-en-tres-capas/

GRADY, R B and CASWELL, D L, 1992, *Sotware Metrics: Establishing a Company-Wide Program.* Prentice-Hall.

HAASE, Kim, JENDROCK, Eric, BALL, Jennifer and EVANS, Ian, 2010, *The Java EE Tutorial*.

JALEXISCV, 2014, Modelo de Datos. . 2014.

JETBRAINS, SRO, 2015, JetBrains. [online]. 2015. [Accessed 5 December 2015]. Available from: http://www.jetbrains.com.

JOHNSON, Ralph, GANMA, Erich and HELM, Richard, 2003, *Design Patterns: Elements of reusable objectoriented software*. 3ra.

JONES, Harley, ORCHARD, Leslie, PEHLIVANIAN, Ara and KOON, Scott, 2009, *Professional JavaScript Frameworks: Prototype, YUI, ExtJS, Dojo and MooTools*. Primera.

JONES, Rupert, CHOPRA, Vivek and LI, Sing, 2003, *Beginning JavaServer Pages*. Wiley Publishing,Inc.

KIMN CONSULTING, 2013, NEKUL-Soft de Seguros. [online]. 2013. Available from: http://nekul.com.ar/nekulweb/index.html

LOCKHART, Thomas, 2011, *PostgreSQL Programmer's Guide.*

LORENZ, M and KIDD, J, 1994, *Object Oriented Metrics*. Prentice Hall : Englewood.

MARTÍNEZ, Alejandro and MARTÍNEZ, Raul, 2012, *Guía a Rational Unified Process* [online]. Available from: http://www.dsi.uclm.es/asignaturas/42551/trabajosAnteriores/Trabajo-Guia%20RUP.pdf

MCCABE, T J and WATSON, A H, 1994, *Software Complexity*.

PRESSMAN, Roger, 2010, *Ingenieria de Software. Un enfoque práctico.* Quinta.

PROGRAMACION.NET, 2013, Diagramas de componentes. [online]. 2013. [Accessed 29 June 2016]. Available from: http://programacion.net/articulo/introduccion_a_uml_181/6

RODRÍGUEZ LANDÍN, Liset, 2012, *Integración del Archivo Universitario con el módulo Estructura y Composición del Sistema de Gestión Universitaria*. La Habana : Universidad de las Ciencias Informáticas.

ROO, Quintana, 2015, Adminsyf. [online]. 2015. [Accessed 10 June 2016]. Available from: http://adminsyf.com/

RYDAHL, Max, KING, Gavin and BAUER, Christian, 2010, *Hibernate. Persistencia relacional para java idiomático.*

SÁNCHEZ RODRÍGUEZ, Tamara, 2014, *Metodología de desarrollo para la Actividad productiva de la UCI.* 2014.

SHAW, Mary, 2007, *Some Patterns for Software Architecture*.

SIA SERVICES, 2015, La suit modular para las compañías de seguros. [online]. 2015. Available from: http://siaservices.es/index.php/productos/gestion-de-seguros/millennium

SMITH, Jorge, 2013, MVC-Iniciando el proyecto. [online]. 2013. [Accessed 29 March 2016]. Available from: http://www.jc-mouse.net/php/blog-mvc-iniciando-el-proyecto-p2

SOFTWARE FOUNDATION, The Apache, 2015, Apache Maven Project. . diciembre 2015.

SOMMERVILLE, Ian, 2005, *Ingienería de Software*. Séptima.

STEPHEN SAUNDERS,DUANE K. FIELDS,EUGENE BELAYEV, 2006, *IntelliJ IDEA in Action*. 1st.

SYSTEMS, Axxis, 2015, SIS10. [online]. 2015. [Accessed 10 June 2016]. Available from: http://www.axxis-systems.com/?gclid=CMLCqva0z8oCFdgHgQod7cUH-A

WILSON, Scott F, 2001, *Analyzing Requirements and Defining Solution Architectures.* Segunda.

ANEXOS

Anexo 1 – Diagrama de clases del Componente Requisición.