Universidad de las Ciencias Informáticas

Facultad 3

Diseño e implementación del módulo Contabilidad del sistema Administración Financiera.

Trabajo de Diploma para optar por el título de
Ingeniero en Ciencias Informáticas

Autor: Enrique Alberto Pérez Anchia

Tutor: Ing. Yunieski Fábregas Santos

Tutor Asesor: MSc. José Raúl Rodríguez Galera

Ciudad Habana, Cuba

Junio - 2008

DECLARACIÓN DE AUTORÍA

Declaro que soy el único 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 _____.

“*Cuanto más talento tiene el hombre más se inclina a creer en el ajeno.”*

Pascal.

AGRADECIMIENTOS

A mis padres, a mis abuelos, a mis hermanos, a mis tíos y demás familiares. A mis amigos de la universidad y a los del pre-universitario con los que he compartido los mejores ocho años de mi vida. A mi tutor el ingeniero Yunieski Fábregas Santos y a mi tutor-asesor el máster José Raúl Rodríguez Galera, al equipo de desarrollo de software del proyecto RN en especial a los ingenieros Henrik Pestano Pino y René Queizán Pérez. A todo el que de una forma u otra contribuyó a la creación de este trabajo.

DEDICATORIA

A mi madre por ser la persona que más me quiere y la que más yo quiero, por ser la persona que me tuvo nueve largos meses en su vientre y que más amor me ha dado en los últimos veinticuatro años, con la plena seguridad de que ese amor será para siempre.

A mi abuelo José Luis, mi gran abuelo, mi ejemplo como persona, como hombre y como amigo, lo que hoy soy, se lo debo en gran medida a mi abuelo José Luis.

A la memoria de mi bisabuela "Belica", que en paz descanse.

Resumen

Como parte de la estrategia del Gobierno de la República Bolivariana de Venezuela de mejorar el servicio y la seguridad del ciudadano venezolano en lo referente a los procesos registrales en el territorio, se plantea el Proyecto de Modernización de los Registros y Notarías.

Perteneciente a este proyecto se encuentra el módulo automatizado de Contabilidad, el cual brinda un soporte indispensable para los demás módulos, esta aplicación se encarga de registrar, confirmar y resumir todas las operaciones contables del Sistema Administración Financiera.

En el presente trabajo se describen las metodologías, así como las técnicas y herramientas utilizadas tanto en el diseño como en la implementación.

El documento incluye las etapas de diseño e implementación del módulo, sobre la base de la arquitectura que soporta el sistema, así como la calificación de los artefactos generados en cada una de estas etapas.

Palabras Claves:

Contabilidad, procesos contables, diseño, implementación, aplicación, eventos, componentes, métricas, pruebas, herramientas, metodologías, plataformas.

Índice

Introducción.

Los lazos de amistad entre los países de Cuba y Venezuela se han ido estrechando cada vez más y con ello los convenios y contratos de trabajo, un ejemplo de ellos es la Universidad de las Ciencias Informáticas, específicamente el grupo de proyecto Registros y Notarias de la facultad tres.

Este proyecto, guiado por su antecedente la aplicación Versat-Sarasola y sobre la base de un antiguo módulo llamado administración contable tiene la misión de automatizar el proceso de administración financiera del Ministerio del Poder Popular para Relaciones Interiores y Justicia de La República Bolivariana de Venezuela (MPPRIJ) en el cual existe una dirección, que se encarga de gestionar, organizar y dirigir los registros y notarías de todo el país. La función de estos registros es prestar servicio de procesos legales a la población venezolana, a través de la realización de trámites.

Actualmente existen tarifas según conceptos de pago y recaudo pero dada la descentralización de los procesos y la independencia de estos registros, no existe un control por parte de la dirección de Registros y Notarías de cuanto se ingresa en cada uno de ellos, cuanto se gasta y cuanto cobran los trabajadores según su nómina y cargo que ocupan.

El pueblo es el que sufre las alteraciones de los precios en los trámites a realizar y el ministerio no ingresa el dinero correspondiente a estos servicios.

Muchos funcionarios de estos registros se enriquecen debido a que el dinero queda en sus bolsillos y no se ingresa a las cuentas del MPPRIJ. No se tiene la relación de trámite contra ingreso que existe en cada uno de los registros.

Dada esta situación el ministerio de justicia de Venezuela decidió crear un sistema que canalice de forma automatizada todas estas operaciones de carácter financiero, en este sistema es indispensable la integración de un módulo de contabilidad, el cual se encargue del control de todos los procesos contables necesarios.

Problema científico de investigación

¿Cómo automatizar los procesos contables en los Registros y Notarías?

Objeto de estudio

Procesos contables que serán automatizados en los Registros y Notarías.

Campo de acción

Diseño e implementación del módulo de Contabilidad para registrar, confirmar y resumir todas las operaciones contables.

Objetivo General

Realizar el diseño e implementación de un módulo capaz de mantener un control eficiente sobre todas las operaciones contables que se automaticen, garantizando el correcto funcionamiento contable en la aplicación que tenga lugar en la dirección del MPPRIJ de la república Bolivariana de Venezuela.

Hipótesis

Si se hace un eficiente diseño e implementación de una solución de software para la gestión de los procesos contables en los registros y notarías del MPPRIJ de la república Bolivariana de Venezuela, entonces se logrará tener un control sobre las operaciones contables automatizadas.

Tareas de la Investigación

- Sistematización del estado del arte.
- Diseño e implementación de una solución de software para la gestión de los procesos contables en los registros y notarías del MPPRIJ.
- Aplicar métricas de diseño a la solución propuesta.
- Aplicar pruebas de software a la solución propuesta.

Métodos científicos de investigación

Teóricos

- Inductivo-deductivo: Partiendo de la idea de comprender los problemas más generalizadores y a partir de estos resultados interpretar conceptos con menor nivel de generalización.

- Histórico-lógico: Teniendo en cuenta lo trascendental se ha podido asimilar el conocimiento lógico para elaborar los diferentes temas.
- Análisis- síntesis: A partir del estudio y análisis de los diferentes temas que componen este trabajo y la recopilación de información se fue creando ideas que contemplaban e integraban la información que se tenía.

Empíricos

- Observación
- Entrevista

Resultados Esperados

- Lograr a partir de las reglas del negocio, el diseño e implementación de los casos de uso del sistema con los requisitos asociados.
- Construir los artefactos inherentes al diseño.
- Construir los artefactos inherentes a la implementación.

Capitulo 1 Fundamentación teórica

Introducción

En el presente capítulo se expone primeramente los conceptos y evolución de la contabilidad como herramienta de gestión y los principios que rigen este campo, así como el estado actual y evolución de los sistemas informáticos contables.

Posteriormente se presenta un estudio sobre los temas relacionados con el diseño e implementación de software.

Finalmente se presenta un estudio acerca de las metodologías, plataformas y herramientas de modelado de software más usadas a nivel mundial.

1.1 Contabilidad.

1.1.1 Conceptos y evolución

El inicio de la literatura contable se enmarca por primera vez de manera formal, en la obra de Francisco Fray Lucca Paccioli de 1494 titulada *"Suma de aritmética, geométrica, proporciones y proporcionalidad"*, material donde por primera vez se emplea el término de partida doble.

A comienzos del siglo XIX, comienza la Revolución Industrial y con ella Adam Smith, David Ricardo y otros economistas clásicos de la época, crean las bases del liberalismo y comienza una revolución en el pensamiento comercial.

La contabilidad comienza a experimentar modificaciones y cambios en sus concepciones, con el nombre de "Principios de Contabilidad", en 1887 se funda la "*American Association of Public Accountants*[1]", antes, en 1854 "*The Institute of Chartered Accountants of Scotland*", en 1880 "*The Institute of Chartered Accountants of England and Wales*", otros organismos similares se constituyen en Francia en 1881, Austria en 1885, Holanda en 1895 y Alemania en 1896. A raíz de la crisis 1930, en Estados Unidos, se crea el Instituto Americano de Contadores Públicos. (Pestano Pino, 2007)

Sobre la base de estas experiencias y con el devenir de los años, este concepto ha evolucionado, de manera que el grado de especialización de la disciplina ha

[1] "American Association of Public Accountants": Asociación norteamericana de contadores públicos.

tomado fuerza. La contabilidad ahora es el lenguaje que se utiliza en el mundo de los negocios, cualquier empresa o institución pública que pretenda prosperar y tener resultados positivos, tiene que saber comunicarse con otras e interpretar la información que recibe, procedente de diferentes ámbitos. Sobre ella existen diversas definiciones con diferentes enfoques, a continuación relacionamos algunas de las más importantes: (Pestano Pino, 2007)

- “Es el arte de registrar, clasificar y resumir en forma significativa y en términos de dinero, las operaciones y los hechos que son cuanto menos de carácter financiero, así como el de interpretar sus resultados.” (AICPA)
- “Es el proceso de identificar, medir y comunicar la información económica que permite formular juicios basados en información y la toma de decisiones, por aquellos que se sirven de la información.” (AICPA)
- “Es la ciencia que se encarga del estudio cualitativo y cuantitativo del patrimonio, tanto en su aspecto estático como dinámico, con la finalidad de lograr la dirección adecuada de las riquezas que la integran.” (Pestano Pino, 2007)
- “Es una técnica en constante evolución, basada en conocimientos razonables y lógicos que tiene como objetivo fundamental, registrar y sintetizar las operaciones financieras de una entidad e interpretar sus resultados.” (Pestano Pino, 2007)
- "La contabilidad es el sistema que mide las actividades del negocio, procesa esa información convirtiéndola en informes y comunica estos hallazgos a los encargados de tomar las decisiones." (Horngren, 1991)
- "La contabilidad es el arte de interpretar, medir y describir la actividad económica." (Meigs, 1992)
- "La contabilidad es el lenguaje que utilizan los empresarios para poder medir y presentar los resultados obtenidos en el ejercicio económico, la situación financiera de las empresas, los cambios en la posición financiera y/o en el flujo de efectivo." (CATACORA, 1998)

1.1.2 Principios

Los principios de la contabilidad se pueden definir como guías de acuerdo con las cuales se ha de manejar la información. El resultado de aplicar sistemáticamente

estos principios, es que la información que se obtiene muestra la imagen fiel de la realidad a que se refiere.

Los principios de contabilidad generalmente aceptados, son los conceptos básicos que se reconocen como esenciales para la cuantificación y el adecuado registro de los estados contables, sus informes financieros y de gestión complementarias, de manera tal que los mismos registren en el tiempo en forma uniforme las variaciones patrimoniales y el resultado de las operaciones, siendo necesario entonces, el conocimiento de los criterios seguidos para su preparación, lo cual facilita entre otros aspectos, el fluido accionar de los órganos de control público. (Pestano Pino, 2007)

Los principios de contabilidad deben aplicarse de manera conjunta y relacionada entre sí. Las bases conceptuales que los conforman guardan relación tanto con el proceso económico financiero, como con el flujo continuo de operaciones a fin de, identificarlas y cuantificarlas, de manera tal que satisfagan la necesidad de información de los responsables de la conducción de la entidad, así como a terceros interesados y por lo tanto, les permitan adoptar decisiones sobre la gestión del mismo. (Pestano Pino, 2007)

Los diferentes principios contables públicos recogidos y definidos en los Principios de Contabilidad Generalmente Aceptados (PCGA.) son los siguientes:

- Principio de entidad contable.
- Principio de uniformidad.
- Principio de prudencia.
- Principio de gestión continuada.
- Principio de importancia relativa.
- Principio de devengo.
- Principio de registro.
- Principio de precio de adquisición.
- Principio de correlación de ingreso y gastos.
- Principio de imputación de la transacción.

- Principio de desafectación.
- Principio de no compensación.

1.2 Los sistemas informáticos contables.

En la época de las grandes civilizaciones las necesidades informativas estaban dadas en el conocimiento de los gastos e ingresos, las posibilidades tecnológicas no pasaban del uso de papiro y escritura cuneiforme y así surge en la contabilidad la técnica de partida simple; con el inicio del comercio la necesidad fundamental estaba en registrar cada movimiento, los cuales se materializaban con el uso del papel y es en este momento cuando surge la técnica de partida doble, conjuntamente con los primeros libros contables.

Posteriormente comienza la revolución industrial con la cual las necesidades informativas eran más complejas, ahora se calculaba la importancia de los activos y beneficios mediante el uso de papel e imprenta, la contabilidad responde a esto con el perfeccionamiento de la partida doble.

Ya en 1960 las necesidades estaban más orientadas a la informática y radicaban específicamente en cómo manejar más información y en el menor tiempo posible, así fue que empezó la automatización de los primeros sistemas contables; poco más de dos décadas después las necesidades estaban dadas en cómo se podía obtener información financiera útil para la toma de decisiones, todo esto en medio de una popularización de la informática, pues la respuesta a estas necesidades estuvieron en la creación de sistemas de información contables integrados en bases de datos, informes, ratios y gráficos.

Pleno siglo XXI y las necesidades informáticas se han derivado en otras aún más complejas; información en tiempo real, comercio electrónico y la medición de activos intangibles para gestionar el conocimiento abarcan las principales necesidades. Hoy en día, con ordenadores en red (internet) y tecnologías de comunicación, la contabilidad ha ido resolviendo estas necesidades a través de la automatización de la captura de datos, intercambio de documentos, en general la informatización de todo tipo de documento restándole protagonismo al papel. (Serrano Cinca, 2006)

Actualmente el mundo de los negocios avanza a pasos agigantados, y este movimiento arrollador va de la mano con los cambios que surgen en la tecnología, y las nuevas demandas de la información, los cambios sociales, culturales y económicos existentes en este nuevo entorno.

Todo esto deriva en el nuevo oriente que debe seguir la Contabilidad y el profesional contable, pues es una de las actividades más importantes dentro del campo de los negocios, dada su naturaleza de informar acerca del crecimiento de las riquezas, la productividad y el posicionamiento de las empresas en los ambientes competitivos.

La importancia de los sistemas de información contable radica en la utilidad que tienen tanto para la toma de decisiones de los socios de las empresas como para aquellos usuarios externos de la información.

Las nuevas demandas de la información abren campo a la introducción de nuevos conceptos que pueden llegar a potencializar la empresa dentro del mercado si se le da el adecuado manejo, reconocimiento y medición.

1.3 Diseño de software.

1.3.1 Evolución

La evolución del diseño del software es un proceso continuo que ha abarcado las últimas cuatro décadas. El primer trabajo de diseño se concentraba en criterios para el desarrollo de programas modulares y métodos para refinar las estructuras del software de manera descendente. Los aspectos procedimentales de la definición de diseño evolucionaron en una filosofía denominada programación estructurada. Un trabajo posterior propuso métodos para la conversión del flujo de datos o estructura de datos en una definición de diseño. Enfoques de diseños más recientes hacia la derivación de diseño proponen un método orientado a objetos. Hoy en día, se ha hecho hincapié en un diseño de software basado en la arquitectura del software. (Pestano Pino, 2007)

Independientemente del modelo de diseño que se utilice, un ingeniero de software deberá aplicar un conjunto de principios fundamentales y conceptos básicos para el diseño a nivel de componentes, de interfaz, arquitectónico y de datos.

1.3.2 Estado actual del diseño de software

Las diferentes vertientes del diseño de software en la actualidad han ido evolucionando principalmente en sus estilos arquitectónicos, buscando acercar cada vez más los problemas reales de la vida a la computadora.

El diseño del software, al igual que los enfoques de diseño de ingeniería en otras disciplinas, va cambiando continuamente a medida que se desarrollan métodos nuevos, análisis mejores y se amplía el conocimiento. Las metodologías de diseño del software carecen de la profundidad, flexibilidad y naturaleza cuantitativa que se asocian normalmente a las disciplinas de diseño de ingeniería más clásicas. Sin embargo, si existen métodos para el diseño del software; también se dispone de calidad de diseño y se pueden aplicar notaciones de diseño. (Pressman, 1998)

El diseño del software se encuentra en el núcleo técnico de la ingeniería del software y se aplica independientemente del modelo de diseño de software que se utilice. Una vez que se analizan y especifican los requisitos del software, el diseño del software es la primera de las tres actividades técnicas (diseño, generación de código y pruebas) que se requieren para construir y verificar el software. Cada actividad transforma la información de manera que de lugar por último a un software de computadora validado. (Pressman, 1998)

El diseño es el lugar en donde se fomentan la calidad en la ingeniería del software. El diseño proporciona las representaciones del software que se pueden evaluar en cuanto a calidad. El diseño es la única forma de convertir exactamente los requisitos de un cliente en un producto o sistema de software finalizado. El diseño del software sirve como fundamento para todos los pasos siguientes del soporte del software y de la ingeniería del software. Sin un diseño, se corre el riesgo de construir un sistema inestable un sistema que falla cuando se lleven a cabo cambios; un sistema que puede resultar difícil de comprobar; y un sistema cuya calidad no puede evaluarse hasta muy avanzado el proceso, sin tiempo suficiente y con mucho dinero gastado. (Pressman, 1998)

1.3.3 Principios del diseño de software

El diseño de software es tanto un proceso como un modelo. El proceso de diseño es una secuencia de pasos que hacen posible que el diseñador describa todos los aspectos del software que se va construir. Sin embargo, es importante destacar que el proceso de diseño simplemente no es un recetario. Un conocimiento creativo, experiencia en el tema, un sentido de lo que hace que un software sea bueno y un compromiso general con calidad son factores críticos de éxito para un diseño competente.

El modelo de diseño es el equivalente a los planes de un arquitecto para una casa. Comienza representando la totalidad de todo lo que se va a construir (por ejemplo, una representación en tres dimensiones de la casa) y refina lentamente lo que va a proporcionar la guía para construir cada detalle (por ejemplo, el diseño de fontanería). De manera similar, el modelo de diseño que se crea para el software proporciona diversas visiones diferentes de software de computadora.

Los principios básicos de diseño hacen posible que el ingeniero del software navegue por el proceso de diseño. Algunos autores sugieren un conjunto de principios para el diseño del software, los cuales han sido adaptados y ampliados en la lista siguiente: (Pressman, 1998)

- **En el proceso de diseño no deberá utilizarse "orejeras".** Un buen diseñador deberá tener en cuenta enfoques alternativos, juzgando todos los que se basan en los requisitos del problema, los recursos disponibles para realizar el trabajo y los conceptos de diseño.
- **El diseño deberá poderse rastrear hasta el modelo de análisis.** Dado que un solo elemento del modelo de diseño suele hacer un seguimiento de los múltiples requisitos, es necesario tener un medio de rastrear como se han satisfecho los requisitos por el modelo de diseño.
- **El diseño deberá minimizar la distancia intelectual entre el software y el problema como si de la misma vida real se tratara.** Es decir, la estructura del diseño del software, siempre que sea posible, imita la estructura del dominio del problema.
- **El diseño deberá presentar uniformidad e integración.** Las reglas de estilo y de formato deberán definirse para un equipo de diseño antes de

comenzar el trabajo sobre el diseño. Un diseño se integra si se tiene cuidado a la hora de definir interfaces entre los componentes del diseño.

- **El diseño deberá estructurarse para admitir cambios.** Los conceptos de diseño estudiados hacen posible un diseño que logra este principio.
- **El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes.** Un software bien diseñado no deberá nunca explotar, deberá diseñarse para adaptarse a circunstancias inusuales y si debe terminar de funcionar, que lo haga de forma suave.
- **El diseño no es escribir código y escribir código no es diseñar.** Incluso cuando se crean diseños procedimentales para componentes de programas, el nivel de abstracción del modelo de diseño es mayor que el código fuente. Las únicas decisiones de diseño realizadas a nivel de codificación se enfrentan con pequeños datos de implementación que posibilitan codificar el diseño procedimental.
- **El diseño deberá evaluarse en función de la calidad mientras se va creando, no después de terminarlo.** Para ayudar al diseñador en la evaluación de la calidad se dispone de conceptos de diseño y de medidas de diseño.
- **El diseño deberá revisarse para minimizar los errores conceptuales.** A veces existe la tendencia de centrarse en minucias cuando se revisa el diseño, olvidándose del bosque por culpa de los árboles. Un equipo de diseñadores deberá asegurarse de haber afrontado los elementos conceptuales principales antes de preocuparse por la sintaxis del modelo del diseño.

1.4 Implementación de software.

1.4.1 Evolución de los lenguajes de programación

Tras el desarrollo de las primeras computadoras surgió la necesidad de programarlas para que realizaran tareas deseadas en el momento deseado. Los lenguajes más primitivos son los llamados lenguajes máquina, como el hardware se desarrollaba antes que el software, estos lenguajes se basaban en el

hardware, con lo que cada máquina tenía su propio lenguaje y por ello la programación era un trabajo muy costoso, válido solo para la máquina en cuestión.

El primer avance fue el desarrollo de las primeras herramientas automáticas generadoras de código fuente. Pero con el permanente desarrollo de las computadoras y el aumento de complejidad de las tareas surgieron a partir de los años 50 los primeros lenguajes de programación de alto nivel.

Con la aparición de los distintos lenguajes solían aparecer versiones de un mismo lenguajes por lo que surgió la necesidad de estandarizarlos buscando universalidad. Las organizaciones que se encargan de regularizar los lenguajes son ANSI (Instituto de las Normas Americanas) e ISO (Organización de Normas Internacionales).

1.4.2 Lenguajes de programación

Los lenguajes de programación pueden clasificarse por diversos criterios, siendo el más común su nivel de semejanza con el lenguaje natural, y su capacidad de manejo de niveles internos de la máquina.

Los principales tipos de lenguajes utilizados son tres:

- **Lenguaje máquina**

Son aquéllos que están escritos en lenguajes directamente entendibles por la máquina (computadora), ya que sus instrucciones son cadenas binarias (cadenas o series de caracteres de dígitos 0 y 1) que especifican una operación y las posiciones (dirección) de memoria implicadas en la operación se denominan instrucciones de máquina o código máquina. El código máquina es el conocido código binario. Las instrucciones en lenguaje máquina dependen del hardware de la computadora y, por tanto, diferirán de una computadora a otra.

Ventajas:

Posibilidad de cargar (transferir un programa a la memoria) sin necesidad de traducción posterior, lo que supone una velocidad de ejecución superior a cualquier otro lenguaje de programación.

Desventajas:

- Dificultad y lentitud en la codificación.
- Poca fiabilidad.
- Gran dificultad para verificar t poner a punto los programas.
- Los programas solo son ejecutables en el mismo procesador.

En la actualidad, las desventajas superan a las ventajas, lo que hace prácticamente no recomendables a los lenguajes máquinas.

- **Lenguaje de bajo nivel**

Son más fáciles de utilizar que los lenguajes máquina, pero al igual que ellos, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el ensamblador. Las instrucciones en lenguaje ensamblador son instrucciones conocidas como nemotécnicos. Por ejemplo, nemotécnicos típicos de operaciones aritméticas son: en inglés: ADD, SUB, DIV, etc. en español: SUM, RES, DIV, etc.

Una instrucción típica de suma sería:

ADD M, N, P.

Esta instrucción significa "sumar el contenido en la posición de memoria M al número almacenado en la posición de memoria N y situar el resultado en la posición de memoria P". Evidentemente es más sencillo recordar la instrucción anterior con un nemotécnico que su equivalente en código máquina.

0110 1001 1010 1011

Un programa escrito en lenguaje ensamblador, requiere de una fase de traducción al lenguaje máquina para poder ser ejecutado directamente por la computadora.

El programa original escrito en lenguaje ensamblador se denomina programa fuente y el programa traducido en lenguaje máquina se conoce como programa objeto, el cual ya es directamente entendible por la computadora.

Ventajas:

- Mayor facilidad de codificación y, en general, su velocidad de cálculo.

Desventajas:

- Dependencia total de la máquina lo que impide la transportabilidad de los programas (posibilidad de ejecutar un programa en diferentes máquinas). El lenguaje ensamblador del PC es distinto del lenguaje ensamblador del Apple Macintosh.
- La formación de los programadores es más compleja que la correspondiente a los programadores de alto nivel, ya que exige no solo las técnicas de programación, sino también el conocimiento del interior de la máquina.
- Los lenguajes ensamblador tienen sus aplicaciones muy reducidas, se centran básicamente en aplicaciones de tiempo real, control de procesos y de dispositivos electrónicos.

➢ **Lenguaje de alto nivel**

Estos lenguajes son los más utilizados por los programadores. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y ensambladores. Un programa escrito en lenguaje de alto nivel es independiente de la máquina (las instrucciones no dependen del diseño del hardware o de una computadora en particular), por lo que estos programas son portables o transportables. Los programas escritos en lenguaje de alto nivel pueden ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras.

Ventajas:

- El tiempo de formación de los programadores es relativamente corto comparado con otros lenguajes.
- La escritura de programas se basa en reglas sintácticas similares a los lenguajes humanos. Nombres de las instrucciones tales como READ, WRITE, PRINT, OPEN, etc.
- Las modificaciones y puestas a punto de los programas son más fáciles. Reducción del coste de los programas.
- Transportabilidad.

Desventajas:

- Incremento del tiempo de puesta a punto al necesitarse diferentes traducciones del programa fuente para conseguir el programa definitivo.
- No se aprovechan los recursos internos de la máquina que se explotan mucho mejor en lenguajes máquina y ensambladores.
- Aumento de la ocupación de memoria.
- El tiempo de ejecución de los programas es mucho mayor.

1.4.3 Estilos de programación

- **Programación Estructurada**

Se llama programación estructurada a la aplicación de los métodos básicos de descomposición de problemas, para establecer una estructura jerárquica fácilmente utilizable, a través de un proceso progresivo. Es un método de construcción y diseño de programar en el que participan características como la modificación y la facilidad en uso.

- **Programación no Estructurada**

Esta técnica es muy simple, y se usa fundamentalmente cuando se está empezando a aprender a programar, su mayor uso se encuentra en la elaboración de programas pequeños y sencillos, son simples algoritmos cuyas instrucciones de control operan directamente sobre la data que es de acceso global. Una vez que los programas se hacen suficientemente grandes esta técnica se convierte en una desventaja, por ejemplo, si se necesitaba hacer una operación en diferentes momentos del programa, habría que repetir el código para dicha operación, esto condujo a la idea de agrupar este código que pudiera repetirse y ofrecer una técnica para acceder a ellos desde el programa principal y regresar al mismo, esta nueva técnica se llamó “Programación procedimental”. (Müller, 1997)

- **Programación procedimental**

Es un paradigma de programación basado en el concepto de “llamado de procedimientos”, los procedimientos, también conocidos como rutinas, sub rutinas, métodos o funciones, simplemente consisten en series de pasos

computacionales. La mayor parte de los lenguajes de alto nivel la soportan, permiten la creación de procedimientos, que son trozos de código que realizan una tarea determinada. (USR.CODE, 2004)

Un procedimiento podrá ser invocado muchas veces desde otras partes del programa con el fin de aislar la tarea en cuestión y, una vez que finaliza su ejecución, retorna al punto del programa desde donde se realizó la llamada.

De esta forma, podemos dividir nuestro problema en problemas más pequeños, y así, llevar la complejidad a un nivel manejable. Si un procedimiento ya es correcto, cada vez que es usado produce resultados correctos. (Müller, 1997)

A medida que los programas fueron complejizándose surgió la necesidad de agrupar estos procedimientos según su comportamiento, y a esta nueva técnica se le denominó “Programación Modular”.

- **Programación Modular**

En la programación modular, los procedimientos con funcionalidades semejantes se agruparon en módulos separados, como consecuencia un programa, ya no está formado solo de una sección. Ahora está dividido en varias secciones más pequeñas que interactúan a través de llamadas a procedimientos y que integran el programa en su totalidad. (USR.CODE, 2004)

Cada módulo puede tener sus propios datos, lo cual permite que manejen un estado interno que pueda ser modificado por las llamadas a sus procedimientos internos, existe solo un estado por módulo, y un solo módulo por programa. (Müller, 1997)

Esta técnica de programación presenta algunas desventajas como son la repetición del mismo código referente al estado interno en más de un módulo, lo cual implicaría que al hacer un cambio en el estado interno por pequeño que fuese en alguno de los módulos, habría que modificar varios de los otros módulos y probarlos de nuevo, otra desventaja la constituye también la forma en sí de agrupar los procedimientos por funcionalidades semejantes. Se siguió profundizando en la teoría y aparecieron nuevos conceptos que revolucionarían la forma de programar.

- **Programación orientada a objetos**

Con el uso de esta técnica de programación se pretende simplificar la modificación y extensión del software para lograr una mayor reutilización del mismo, permite una fácil comprensión debido a que la estructura del software y la del problema a resolver están relacionadas directamente.

En la programación orientada a objetos, el concepto de módulo es profundizado y se transforma en un objeto. “Los objetos representan componentes de un sistema descompuesto modularmente o bien unidades modulares de representación del conocimiento.” (Booch, 1998)

Además, allí, un programa no es otra cosa que una colección de objetos que se comunican entre sí mandándose mensajes unos a otros para lograr un objetivo común. (Müller, 1997)

Aplicando diseño orientado a objetos, se crea software resistente al cambio y escrito con economía de expresión. Se logra un mayor nivel de confianza en la corrección del software a través de una división inteligente de su espacio de estados. (Booch, 1998)

Esta técnica es la más usada actualmente por programadores, se ha mejorado para lograr una heterogeneidad de plataformas con la creación de servicios webs basados en el estándar XML[2]. Lo cual ha inferido el surgimiento de nuevas formas de concebir un software tomando como base los servicios.

- **Programación orientada a servicios**

De forma resumida se podría decir que un Servicio Web es un componente de software que se comunica con otras aplicaciones codificando los mensajes en XML y enviando estos mensajes a través de protocolos estándares de Internet, intuitivamente es similar a un sitio web pero sin interfaz de usuario, brinda servicio a las aplicaciones en vez de a las personas. (González, 2007)

La programación orientada a servicios constituye un complemento de la programación orientada a objetos debido a que puede representarse como una capa adicional en el modelado de una solución, esta capa podría llamarse “Capa de Servicios” la cual se encargará de publicar aquellas funcionalidades que

[2] XML: Lenguaje extensible de marcas.

puedan resultar comunes para diferentes problemas, también ofrece facilidad para el desarrollo de aplicaciones distribuidas, que están dadas producto de la experiencia acumulada en la última década, sobre todo en las áreas de la computación distribuida, instalación de una solución e interoperabilidad entre sistemas heterogéneos. (Schwindt, 2007)

Una de las diferencias fundamentales entre esta técnica y la Programación orientada a objetos es la manera en la que ambas definen una aplicación. La Programación orientada a objetos determina que una aplicación está compuesta de clases interdependientes, mientras que la Programación orientada a servicios considera que una aplicación está compuesta por un conjunto de servicios autónomos. (Schwindt, 2007)

Los sistemas orientados a servicios, en cambio, son diseñados con un bajo nivel de acoplamiento que facilita la implementación de cambios y estos servicios pueden ser desarrollados en cualquier lenguaje corriendo en diferentes plataformas. (Schwindt, 2007)

Dado el creciente aumento en complejidad de las soluciones de software planteadas actualmente, ha surgido una nueva técnica denominada “Programación Orientada a Aspectos”.

- **Programación orientada a aspectos**

También conocida por AOP, por las siglas de (*Aspect-Oriented Programming)* o AOSD, por (*Aspect-Oriented Software Development*) buscan resolver un problema identificado hace tiempo en el desarrollo de software. Se trata del problema de la separación de incumbencias o conceptos y de la minimización de las dependencias entre ellos. (Schwindt, 2007)

Al adentrarnos en el significado de un aspecto, se representa como una unidad que se define en términos de información parcial de otras unidades. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa. (Congote Edgar, 2006.).

Es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor

separación de conceptos, sobre todo cuando para la programación orientada a objetos se le complejizan mucho las relaciones entre los objetos definidos.

El uso de una o más de estas técnicas está en dependencia de la complejidad del problema que se necesite resolver, aunque vale la pena aclarar que cada una de estas técnicas usa a sus predecesoras, ya sea de una forma o de otra.

1.5 Plataformas de desarrollo.

Desde el enfoque de la realización de sistemas informáticos, se le denomina plataformas a los diferentes ambientes creados para el desarrollo de software. Actualmente en el mundo existen diferentes tipos de plataformas de desarrollo para aplicaciones electrónicas y de escritorio.

En los últimos años los ambientes de programación han ido dirigiéndose especialmente al bienestar del desarrollador y a la unificación en un mismo entorno de desarrollo, la creación de aplicaciones con sus prestaciones para diferentes campos de acción como los son, la Web y las aplicaciones de escritorio. En los años noventa solo existían como la representación del desarrollo de software, leguajes de programación, con su ambiente visual en algunos casos, para desarrollar sistemas entre los que se encuentran Borland Delphi, Borland C++ y el Visual Basic.

En estos momentos con la aparición de la plataforma .Net de Microsoft los conceptos del desarrollo de aplicaciones han sufrido un cambio radical por el simple hecho de unificar en un único ambiente de desarrollo múltiples lenguajes de programación representados por un framework[3]. De las diferentes plataformas existentes en el mundo para el desarrollo de aplicaciones de escritorio se encuentran: (Almenares Herrera, 2007)

[3] Framework: Denota la infraestructura sobre la cual se reúnen un conjunto de lenguajes, herramientas y servicios que simplifican el desarrollo de aplicaciones en entorno de ejecución distribuido.

1.5.1 Java y su Máquina Virtual[4]

Una de las principales características que favoreció el crecimiento y difusión del lenguaje Java es su capacidad de que el código funcione sobre cualquier plataforma de software y hardware. Esto significa que un programa escrito para Linux puede ser ejecutado en Windows. Además es un lenguaje orientado a objetos que resuelve los problemas en la complejidad de los sistemas.

El código que generan los compiladores de Java no es particular de una maquina física, sino de una máquina virtual. Aún cuando existen múltiples implantaciones de la Máquina Virtual Java, cada una específica de la plataforma sobre la cual subyace, existe una única especificación de la máquina virtual, que proporciona una vista independiente del hardware y del sistema operativo sobre el que se esté trabajando.

El concepto de máquina virtual es antiguo. Fue usado por IBM en 1959 para describir uno de los primeros sistemas operativos que existieron en la historia de la computación. En 1970, el ambiente de programación de *SmallTalk* llevó la idea a un nuevo nivel y construyó una máquina virtual para soportar abstracciones orientadas a objetos de alto nivel, sobre las máquinas subyacentes. (Almenares Herrera, 2007)

1.5.2 Mono[5]

Es un proyecto de código abierto impulsado por Novell para crear un grupo de herramientas libres, basadas en GNU/Linux[6] y compatibles con .NET según lo especificado por el ECMA14[7]. Entre los principales componentes de los que dispone como plataforma están los siguientes:

- Una máquina virtual de lenguaje común de infraestructura (CLI) que contiene un cargador de clases, un compilador en tiempo de ejecución (JIT), y unas rutinas de recolección de memoria.

[4] Maquina Virtual: Programa nativo, plataforma específica capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial.

[5] Mono: Implementación del framework .NET para ambientes de Linux.

[6] GNU/LINUX: uno de los variados sistemas operativos que existen.

[7] ECMA14: organización internacional basada en membrecías de estándares para la comunicación y la información.

- Una biblioteca de clases que puede funcionar en cualquier lenguaje que funcione en el CLR (Common Language Runtime).
- Un compilador para el lenguaje C#, MonoBasic (la versión para mono de Visual Basic), Java y Python.
- Es un proyecto independiente de la plataforma. Actualmente Mono corre sobre Linux, FreeBSD, UNIX, Mac OS X, Solaris y plataformas Windows.

Esta plataforma presenta una importante ventaja, y es que se ha posicionado como un entorno que permite ejecutar en Linux aplicaciones diseñadas para Microsoft .Net en entorno Windows, facilitando la migración de aplicaciones a Linux y aumentando su base de desarrolladores y usuarios.

1.5.3 .Net Framework[8]

Se decidió escoger esta plataforma para el desarrollo de la aplicación por sus características y su estructura.

Es una plataforma de software que conecta información, sistemas, personas y dispositivos. La plataforma .NET conecta una gran variedad de tecnologías de uso personal y de negocios, de teléfonos celulares a servidores corporativos, permitiendo el acceso a información importante, donde y cuando se necesiten. Desarrollado con base en los estándares de Servicios Web XML, .NET permite que los sistemas y aplicaciones, ya sea nuevos o existentes, conecten sus datos y transacciones independientemente del sistema operativo, tipo de computadora o dispositivo móvil que se utilice, o del lenguaje de programación empleados para crearlo.

La idea fundamental de Microsoft .NET es un cambio de enfoque en lo que es la informática, pasando de un mundo de aplicaciones, sitios Web y dispositivos aislados a una infinidad de computadoras, dispositivos, transacciones y servicios que se conectan directamente y trabajan en conjunto para ofrecer soluciones más amplias y ricas en contenido.

[8] .NET framework: Plataforma de desarrollo para las tecnologías Microsoft.net.

Esta plataforma cuenta con ciertas ventajas a tener en cuenta:

- **Código administrado**: El CLR[9] (*Command Language Runtime*) realiza un control automático del código para que este sea seguro, es decir, controla los recursos del sistema para que la aplicación se ejecute correctamente.
- **Interoperabilidad multilenguaje**: El código puede ser escrito en cualquier lenguaje compatible con .Net ya que siempre se compila en código intermedio (MSIL[10]).
- **Compilación just-in-time**: El compilador JIT[11] incluido en el Framework compila el código intermedio (MSIL) generando el código máquina propio de la plataforma. Se aumenta así el rendimiento de la aplicación al ser específico para cada plataforma.
- **Recolector de basura**: El CLR proporciona un sistema automático de administración de memoria denominado recolector de basura (Garbage Collector[12]). El CLR detecta cuándo el programa deja de utilizar la memoria y la libera automáticamente. De esta forma el programador no tiene por que liberar la memoria de forma explícita aunque también sea posible hacerlo manualmente.
- **Seguridad de acceso al código**: Se puede especificar que una pieza de código tenga permisos de lectura de archivos pero no de escritura. Es posible aplicar distintos niveles de seguridad al código, de forma que se puede ejecutar código procedente del Web sin tener que preocuparse si esto va a estropear el sistema.
- **Despliegue**: Por medio de los ensamblados resulta mucho más fácil el desarrollo de aplicaciones distribuidas y el mantenimiento de las mismas. El Framework realiza esta tarea de forma automática mejorando el rendimiento y asegurando el funcionamiento correcto de todas las aplicaciones.

[9] CLR: Lenguaje común de ejecución para los diferentes lenguajes de .NET framework.

[10] MSIL: Código de bites que utiliza la tecnología .NET para lograr independencia de la plataforma y seguridad de ejecución.

[11] JIT: Técnica para mejorar el rendimiento de sistemas de programación en la tecnología .NET.

[12] Garbage Colletor: Mecanismo implícito de gestión de memoria implementado en algunos lenguajes de programación.

1.6 Metodologías.

El modelado de sistemas informáticos de altos grados de complejidad requiere del uso de la Ingeniería de Software para llevar a cabo de una manera organizada y bien definida las tareas del Software en cuestión. Existen en el mundo diferentes metodologías para llevar a cabo del desarrollo de estos sistemas, y cada una de ellas contienen ciertos pasos que identifican el proceso de ingeniería de software, y estos son los siguientes:

- **Análisis de requisitos**

Extraer los requisitos de un producto de software es la primera etapa para crearlo. Se requiere de habilidad y experiencia en la ingeniería de software para reconocer requisitos que el usuario en ocasiones no es capaz de identificar o los que puedan ser incompletos, ambiguos ó contradictorios.

- **Especificación**

Es la tarea de describir detalladamente el software a ser escrito, en una forma matemáticamente rigurosa. En la realidad, la mayoría de las buenas especificaciones han sido escritas para entender y afinar aplicaciones que ya estaban desarrolladas. Las especificaciones son más importantes para las interfaces externas, que deben permanecer estables.

- **Diseño y arquitectura**

Se refiere a determinar cómo funcionará de forma general sin entrar en detalles. Consiste en incorporar consideraciones de la implementación tecnológica, como el hardware, la red, etc.

- **Programación**

Reducir un diseño a código puede ser la parte más obvia del trabajo de ingeniería de software, pero no es necesariamente la porción más larga.

- **Prueba**

Consiste en comprobar que el software realice correctamente las tareas indicadas en la especificación. Una técnica de prueba es probar por separado cada módulo del software, y luego probarlo de forma integral.

- **Documentación**

Realización del manual de usuario, y posiblemente un manual técnico con el propósito de mantenimiento futuro y ampliaciones al sistema.

- **Mantenimiento**

Mantener y mejorar el software para enfrentar errores descubiertos y nuevos requisitos. Esto puede llevar más tiempo incluso que el desarrollo inicial del software. Alrededor de 2/3 de toda la ingeniería de software tiene que ver con dar mantenimiento. Una pequeña parte de este trabajo consiste en arreglar errores. La mayor parte consiste en extender el sistema para hacer nuevas cosas. De manera similar, alrededor de 2/3 de toda la ingeniería civil, arquitectura y trabajo de construcción es dar mantenimiento.

1.6.1 Rational Unified Process (RUP)

Es una infraestructura flexible de desarrollo de software que proporciona prácticas recomendadas probadas y una arquitectura configurable

Esta metodología divide en cuatro fases el proceso de desarrollo de un software.
Inicio: Se determina la visión del proyecto en general.

Elaboración: Se plantea la arquitectura óptima.

Construcción: Se centra en la obtención de la capacidad operacional inicial.

Transición: Se logra obtener una versión funcional del software.

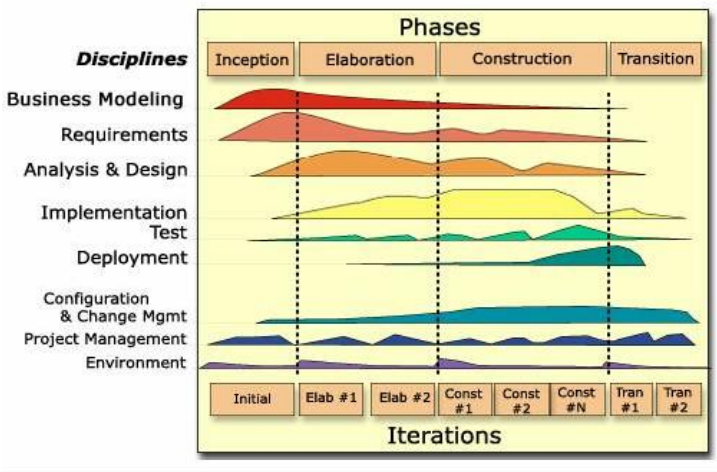


FIG 1. Vista en dos dimensiones de RUP.

Estas etapas se llevan a cabo usando un ciclo de iteraciones, y los objetivos de una iteración se establecen en función de la evaluación de las iteraciones anteriores. Además, vale destacar que el ciclo de vida que se desarrolla por cada iteración es aplicado bajo dos disciplinas, una es de desarrollo y la otra es la de soporte.

1.6.2 eXtreme Programming[13] (XP)

Es la metodología más utilizada de los procesos ágiles del desarrollo de software. Esta, a diferencia de los métodos tradicionales pone más énfasis en la adaptabilidad que en la previsibilidad. El principio de este proceso es el hecho de tener en cuenta la posible y natural aparición de cambios de requisitos durante el desarrollo del software, por lo cual esta metodología presenta los mecanismos necesarios para adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto.

El ciclo de desarrollo consiste, a grandes rasgos, en los siguientes pasos:

- El cliente define el valor de negocio a implementar.
- El programador estima el esfuerzo necesario para su implementación.
- El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
- El programador construye ese valor de negocio.
- Se regresa al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden.

No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.

[13] eXtreme programming: Programación extrema.

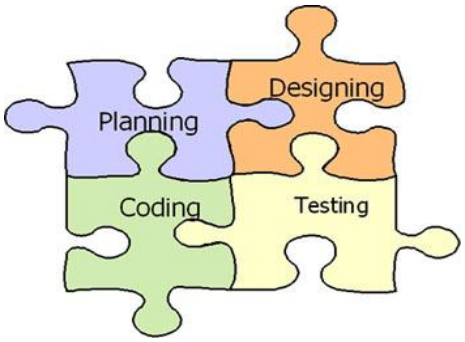


FIG 2. Vista de XP.

- Los roles definido de acuerdo con la propuesta original del patriarca de la metodología ágil XP, Kent Beck son:
- **Programador**

Escribe las pruebas unitarias y produce el código del sistema.

- **Cliente**

Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuales se implementan en cada iteración, centrándose en aportar mayor valor al negocio.

- **Encargado de pruebas**

Ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.

- **Encargado de seguimiento**

Proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.

- **Entrenador**

Es responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.

- **Consultor**

Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.

- **Gestor**

Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

¿Qué es lo que propone XP?

- Empieza en pequeño y añade funcionalidad con retroalimentación continua
- El manejo del cambio se convierte en parte sustantiva del proceso
- El costo del cambio no depende de la fase o etapa
- No introduce funcionalidades antes que sean necesarias
- El cliente o el usuario se convierte en miembro del equipo

1.6.3 Microsoft Solution Framework (MSF)

Esta es una metodología flexible e interrelacionada con una serie de conceptos, modelos y prácticas de uso, que controlan la planificación, el desarrollo y la gestión de proyectos tecnológicos. MSF se centra en los modelos de proceso y de equipo dejando en un segundo plano las elecciones tecnológicas. (Mendoza Sanchez, 2004)

FIG 3. Vista de MSF

MSF es:

- **Adaptable**

Es parecido a un compás, usado en cualquier parte como un mapa, del cual su uso es limitado a un específico lugar.

- **Escalable**

Puede organizar equipos tan pequeños entre tres o cuatro personas, así como, proyectos que requieren cincuenta personas o más.

- **Flexible**

Adaptable al ambiente de desarrollo de cualquier cliente.

- **Tecnología agnóstica**

Puede ser usada para desarrollar soluciones basadas sobre cualquier tecnología.

MSF se compone de varios modelos encargados de planificar las diferentes partes implicadas en el desarrollo de un proyecto: Modelo de Arquitectura del Proyecto, Modelo de Equipo, Modelo de Proceso, Modelo de Gestión del Riesgo, Modelo de Diseño de Proceso y finalmente el modelo de Aplicación.

- **Modelo de Arquitectura del Proyecto:** Diseñado para acortar la planificación del ciclo de vida. Este modelo define las pautas para construir proyectos empresariales a través del lanzamiento de versiones.
- **Modelo de Equipo:** Este modelo ha sido diseñado para mejorar el rendimiento del equipo de desarrollo. Proporciona una estructura flexible para organizar los equipos de un proyecto. Puede ser escalado dependiendo del tamaño del proyecto y del equipo de personas disponibles.
- **Modelo de Proceso:** Diseñado para mejorar el control del proyecto, minimizando el riesgo, y aumentar la calidad acortando el tiempo de entrega. Proporciona una estructura de pautas a seguir en el ciclo de vida del proyecto, describiendo las fases, las actividades, la liberación de versiones y explicando su relación con el Modelo de equipo.
- **Modelo de Gestión del Riesgo:** Diseñado para ayudar al equipo a identificar las prioridades, tomar las decisiones estratégicas correctas y controlar las emergencias que puedan surgir. Este modelo proporciona un entorno estructurado para la toma de decisiones y acciones valorando los riesgos que puedan provocar.
- **Modelo de Diseño del Proceso:** Diseñado para distinguir entre los objetivos empresariales y las necesidades del usuario. Proporciona un modelo centrado en el usuario para obtener un diseño eficiente y flexible a través de un enfoque iterativo. Las fases de diseño conceptual, lógico y

físico proveen tres perspectivas diferentes para los tres tipos de roles: los usuarios, el equipo y los desarrolladores.

- **Modelo de Aplicación:** Diseñado para mejorar el desarrollo, el mantenimiento y el soporte, proporciona un modelo de tres niveles para diseñar y desarrollar aplicaciones software. Los servicios utilizados en este modelo son escalables, y pueden ser usados en un solo ordenador o incluso en varios servidores.

La metodología RUP (Rational Unified Process) fue la seleccionada por la dirección del proyecto Registro y Notarias como la adecuada para llevar a cabo la confección de dicho proyecto, en el cuál se incluye el Módulo Contabilidad al cual nos referimos en este documento.

Esta metodología además de varias de sus características, antes mencionadas, se identificada como perteneciente a los procesos de desarrollo del tipo pesados. Se le llama de esta forma por la extensión en tiempo de desarrollo, así como el gran número de especialistas necesarios para la confección del Software. Muy difícilmente las diferentes metodologías pueden ser comparadas en el mayor número de los casos, ya que cada una de ellas es aplicada a uno u otro ambiente de desarrollo determinado por variables como lo son: magnitud de lo que se desea desarrollar, tiempo con el cual se cuente, número del personal con el cual se trabajara, complejidad de los diferentes módulos.

Se considera que el uso de la metodología RUP para la confección de este proyecto fue correcto, según la magnitud y extensión del funcionamiento de su negocio, el gran número de especialistas involucrados, así como su proceso de desarrollo a distancia, no se determino aplicar una metodología ágil y si una correcta y robusta captura de requisitos, lo cual es exigido como premisa fundamental por la metodología RUP.

1.7 Herramientas.

Las herramientas CASE, C*omputer Aided Software Engineering*[14], Ingeniería de Software Asistida por Ordenador, son las aplicaciones informáticas que tienen

[14] Computer Aided Software Engineering: Ingenieria de software asistida por computadoras.

como tarea principal lograr una mayor productividad en el desarrollo de software y una reducción de costos de desarrollo. Estas herramientas contribuyen de manera directa en todos los aspectos del ciclo de vida de desarrollo del software en tareas como el proceso de realizar un diseño del proyecto, calculo de costes, implementación de parte del código automáticamente con el diseño dado, compilación automática, documentación o detección de errores entre otras. Cada una de estas herramientas persigue nueve objetivos principales: (Almenares Herrera, 2007)

- Mejorar la productividad en el desarrollo y mantenimiento del software.
- Aumentar la calidad del software.
- Mejorar el tiempo y coste de desarrollo y mantenimiento de los sistemas informáticos.
- Mejorar la planificación de un proyecto.
- Aumentar la biblioteca de conocimiento informático de una empresa ayudando a la búsqueda de soluciones para los requisitos.
- Automatizar, desarrollo del software, documentación, generación de código, pruebas de errores y gestión del proyecto.
- Ayuda a la reutilización del software, portabilidad y estandarización de la documentación.
- Gestión global en todas las fases de desarrollo de software con una misma herramienta.
- Facilitar el uso de las distintas metodologías propias de la ingeniería del software.

1.7.1 Rational Rose.

Es una de las más poderosas herramientas de modelado visual para el análisis y diseño de sistemas basados en objetos. Se utiliza para modelar un sistema antes de proceder a construirlo. Cubre todo el ciclo de vida de un proyecto:

- Concepción y formalización del modelo.
- Construcción de los componentes.
- Transición a los usuarios.
- Certificación de las distintas fases.

1.7.2 Visual Paradigm para UML.

- Visual Paradigm es una herramienta UML profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue.
- El software de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste.
- Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación.
- La herramienta UML CASE también proporciona abundantes tutoriales de UML, demostraciones interactivas de UML y proyectos UML.

1.7.3 Enterprise Architect.

La herramienta de desarrollo escogida por el equipo de proyecto fue Enterprise Architect.

Enterprise Architect combina el poder de la última especificación UML 2.1 con alto rendimiento, interfaz intuitiva, para traer modelado avanzado al escritorio, y para el equipo completo de desarrollo e implementación. Esta herramienta presenta las siguientes características:

- **Alta capacidad**

Enterprise Architect es una herramienta comprensible de diseño y análisis UML, cubriendo el desarrollo de software desde el paso de los requerimientos a través de las etapas del análisis, modelos de diseño, pruebas y mantenimiento. EA es una herramienta multi-usuario, basada en Windows, diseñada para ayudar a construir software robusto y fácil de mantener. Ofrece salida de documentación flexible y de alta calidad.

- **Velocidad, estabilidad y buen rendimiento**

El Lenguaje Unificado de Modelado provee beneficios significativos para ayudar a construir modelos de sistemas de software rigurosos y donde es posible mantener la trazabilidad de manera consistente. Enterprise Architect soporta este proceso en un ambiente fácil de usar, rápido y flexible.

- **Trazabilidad de extremo a extremo**

Enterprise Architect provee trazabilidad completa desde el análisis de requerimientos hasta los artefactos de análisis y diseño, a través de la implementación y el despliegue. Combinados con la ubicación de recursos y tareas incorporados, los equipos de Administradores de Proyectos y Calidad están equipados con la información que ellos necesitan para ayudarles a entregar proyectos en tiempo.

- **Construido sobre las bases de UML 2.1**

Las bases de Enterprise Architect están construidas sobre la especificación de UML 2.0. Usa Perfiles UML para extender el dominio de modelado, mientras que la Validación del Modelo asegura integridad.

Conclusiones parciales:

Se ha mostrado de manera argumental y conceptual temas imprescindibles para el entendimiento amplio y profundo del negocio en cuestión. La Contabilidad como herramienta de gestión es útil e imprescindible en cualquier entidad empresarial que pretenda ser competente. Se adquirió el conocimiento técnico y metodológico, básicos para el desarrollo del software.

Capitulo 2 Solución propuesta

Introducción

A partir de los conceptos mencionados en el capítulo anterior se propone una solución informática con vistas a resolver los problemas que presenta el Ministerio del Poder Popular para las Relaciones Interiores y de Justicia de la República Bolivariana de Venezuela. En el presente capítulo se describen las características arquitectónicas enfocadas principalmente a los patrones de diseño aplicados y a los componentes de interfaz de usuario utilizados.

2.1 Arquitectura.

2.1.1 Concepciones Generales.

La solución de software en general está dirigida a la automatización de los procesos administrativos y financieros según la distribución adoptada actualmente a tal efecto en SAREN[15] en la República Bolivariana de Venezuela.

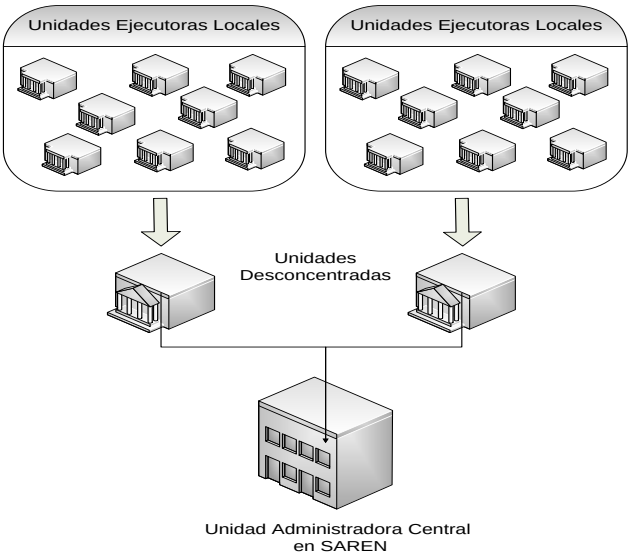


FIG 4. Estructura Financiera de SAREN.

El sistema de Administración Financiera consta de diez módulos para garantizar la gestión de los procesos funcionales: Contabilidad, Presupuesto, Tesorería, Recaudación, Inventario de Bienes, Inventario de Materiales y Suministros, Obras,

[15] SAREN: Servicio Autónomo de Registros y Notarias

Compras, Administración y un módulo Comunes para administrar y encapsular los recursos de programación comunes a los nueve restantes.
La solución de Administración Financiera está concebida para desempeñarse como una aplicación de escritorio y desarrollada sobre la plataforma .Net. Cada uno de los módulos y funcionalidades que lo componen estarán disponible en la aplicación dependiendo de los requerimientos y competencia que tenga la unidad donde se esté implantando la solución a excepción de Comunes; puesto que no es parte de los requerimientos administrativos y contables de SAREN, sino un recurso utilizado en la arquitectura que será justificado en posteriores secciones de este documento.

En las Unidades Ejecutoras Locales, Unidades Desconcentradas o Unidad Centralizada, la aplicación brindará y garantizará la configuración e infraestructura necesaria para establecer y definir los elementos de la solución que son competencia y dominio de la misma, así como establecer posibles integraciones con otros módulos o sistemas que puedan estar presentes; todos estos elementos estarán implementados en el módulo de Administración.

- **Representación arquitectónica.**

Horizontal:
El sistema está dividido en nueve módulos principales cada uno de los cuales responde a un conjunto de funcionalidades y casos de uso específicos del cliente.
Todos estos módulos interactúan y comparten datos de interés en dependencia de la funcionalidad de cada uno y siguiendo un estricto régimen de seguridad de la información. Los módulos son:
Administración.
Contabilidad.
Presupuesto.
Tesorería.
Recaudación.
Inventario de Bienes.
Inventario de Materiales y Suministros.
Obras.
Compras.

Vertical:

El desarrollo de cada una de los 9 módulos responde a un modelo multicapas. Este tipo de sistema puede dar cabida a varias configuraciones diferentes. En una arquitectura de n niveles como esta, las acciones de la aplicación están lógicamente divididas por funciones.

Capa de Presentación.

Capa Lógica del Negocio.

Capa de Fachada.

Capa Acceso a Datos.

Capa de Datos.

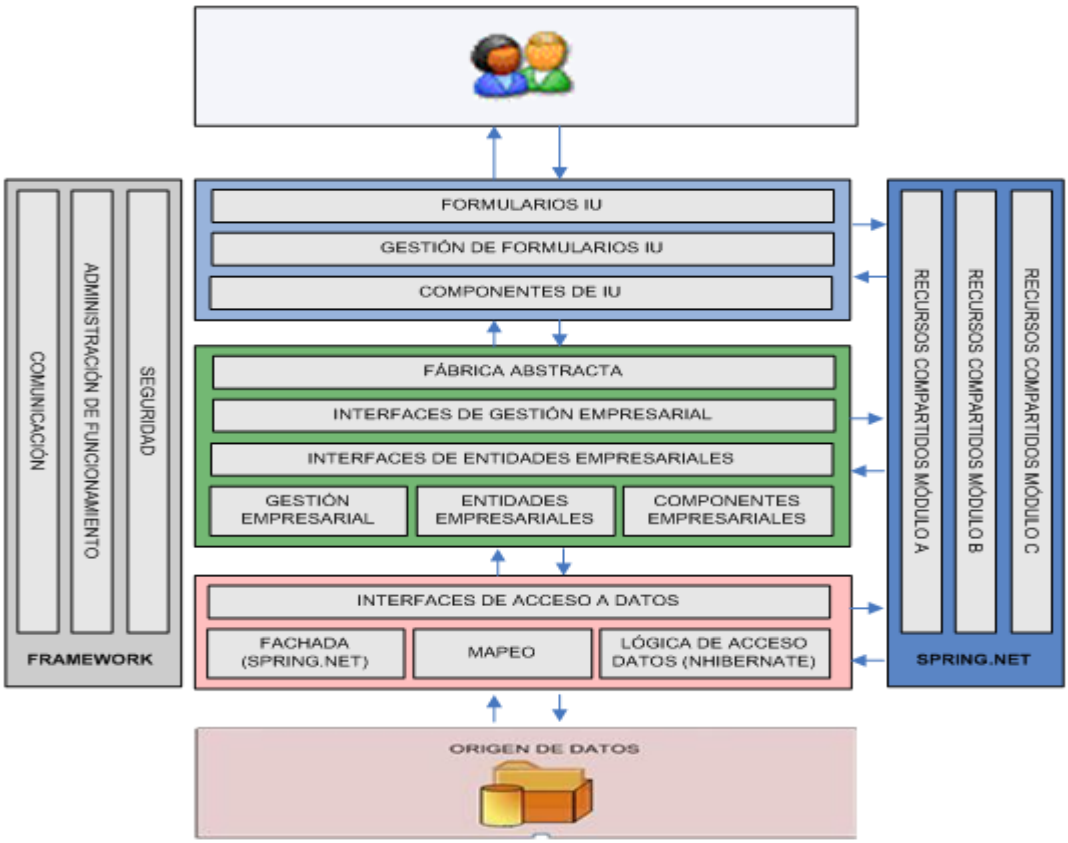


FIG.5 Representación Arquitectónica.

Leyenda

- Capa de Presentación.
- Capa de Lógica de Negocio.
- Capa de Acceso a Datos.
- Capa de Datos.
- Capa de Fachada.
- Framework.

Resultan elementos significativos en la arquitectura el uso de:

Spring.net

Es un framework que ayuda a construir aplicaciones empresariales sobre la plataforma .NET, basado en el framework Spring de java del cual hereda los principales conceptos, tales como Inyección de Dependencia y Programación Orientada a Aspectos.

La mayoría de las aplicaciones empresariales están compuestas por una variedad de capas físicas y por un conjunto de funcionalidades. Independientemente de la arquitectura empleada y atendiendo a la gran variedad de objetos internos y externos, que se manejan en cada nivel, la utilización de spring.net resulta una buena opción.

Ventajas

Precisamente al utilizar spring.net en una aplicación aseguramos una serie de puntos importantes en toda aplicación, puntos que comprende el framework y que facilitan el trabajo de los desarrolladores.

- Implementación de Patrones de Diseño: La utilización de patrones en una arquitectura resulta una decisión muy acertada y eficiente puesto que constituyen buenas prácticas y soluciones a problemas muy comunes en el desarrollo de cualquier aplicación. Spring.net implementa una gran variedad de estos patrones de diseño de una forma bastante sencilla y práctica.
- Inyección de dependencia: Delega la responsabilidad de instanciar los objetos en un archivo de configuración XML, esto resulta muy útil puesto que evita cualquier dependencia que pueda existir entre los objetos, más si son de capas o aplicaciones externas. El nombre viene precisamente de que los objetos generalmente son instanciados antes de ser utilizados en el momento en que se carga el XML.
- Basado en Programación Orientada a Aspectos: Nos ayuda a modificar dinámicamente nuestro modelo estático para incluir el código requerido para cumplir los requerimientos secundarios sin tener que modificar el modelo

- Estático original. Mejor aún, normalmente podremos tener este código adicional en una única localización en vez de tenerlo repartido por el modelo existente, como se haría si se estuviera trabajando Orientado a Objetos.

NHibernate

Utilizar un framework de Object Relational Mapping[16] (ORM) para resolver la lógica de persistencia es una técnica madura que ha demostrado ser extremadamente superior a las técnicas tradicionales basadas en el uso de APIs[17] como ADO.NET.

Esta sección pretende argumentar esta idea a partir el uso de Nhibernate, un framework de ORM open-source para .NET basado en el excelente framework Hibernate surgido en la comunidad Java en el año 2002.

Ventajas

Una estrategia elegante y compatible con las buenas prácticas de diseño pregonadas en estos últimos 10 años, es la de diseñar un modelo de objetos del dominio que represente el 100% de la información que maneja una aplicación y utilizar un framework de Object Relational Mapping que resuelva en forma transparente la persistencia de estos objetos contra una base de datos relacional.

- **Transparente:** Los objetos del dominio desconocen todo lo concerniente a la base de datos donde son persistidos, el framework lo resuelve en forma automática utilizando archivos de mapeo expresados en XML.
- **Soporte de polimorfismo:** Se puede cargar jerarquías de objetos en forma polimórfica.
- **Soporte de los 3 niveles de mapeo de herencia:** Se puede mapear toda una jerarquía de clases a una sola tabla, crear una tabla por cada clase concreta o crear una tabla por cada escalón de la jerarquía.
- **Soporte completo de asociaciones:** Los frameworks de ORM soportan el mapeo de todos los tipos de relaciones que pueden existir en un modelo de

[16] Object Relational Mapping: Mapeo de objetos relacionales.

[17] APIs: Interfaz de programación de aplicaciones.

objetos del dominio (asociaciones 1...1, 1...N, N...M, unidireccionales y bidireccionales).

- **Soporte de carga de objetos Proxy:** Permite cargar objetos que solo contengan la clave del objeto completo.
- **Soporte de caching[18]:** En el contexto de una transacción, se puede disminuir la cantidad de veces que se accede base de datos cacheando en memoria los objetos que son accedidos varias veces.
- **Soporte de múltiples dialectos SQL:** Se puede independizar completamente del tipo de base de datos utilizada. Una aplicación puede persistir sus datos en SQL Server, en Oracle, en MySQL, etc. simplemente cambiando la configuración correspondiente.

Módulo Común

Se decide agregar este módulo a la solución como mecanismo de control y organización, donde nacerán todos los elementos comunes a los restantes módulos, es decir, aquí se gestionan, desarrollan y prueban los componentes y recursos comunes del sistema en general.

Presenta una estructura que satisface las necesidades planteadas y que difiere horizontalmente del los demás, básicamente se tendrá una Capa de Servicios, otra Capa de Componentes y finalmente una de Prueba.

Ventajas

- Se mantienen los elementos comunes en un solo lugar independiente del código de los módulos.
- Se evitan códigos y elementos repetidos en todo el sistema.
- Mayor control de los recursos que se van creando.

Mayores facilidades para desarrollar estos recursos puesto que se cuenta con toda una infraestructura a tal efecto. *Véase anexo1.*

[18] Caching: Almacenamiento de información en memoria aleatoria (RAM).

2.2 Patrones de diseño

Los Patrones de Diseño nos hablan de cómo construir software, de cómo utilizar las clases y los objetos de forma conocida.

Los precedentes a los patrones de diseño vienen del campo de la Arquitectura, Christopher Alexander a finales de los 70 escribe varios libros acerca de urbanismo y construcción de edificios, y se plantea reutilizar diseños ya aplicados en otras construcciones que cataloga como modelos a seguir.

Nuestro sistema se plantea el uso de patrones de diseño con el objetivo de aprovechar las ventajas que estos nos brindan a continuación se caracterizan los patrones usados:

- **Patrón Solitario**

Tipo:

Creación, a nivel de objetos.

Propósito:

Garantizar que una clase sólo tiene una única instancia, proporcionando un punto de acceso global a la misma.

Estructura:

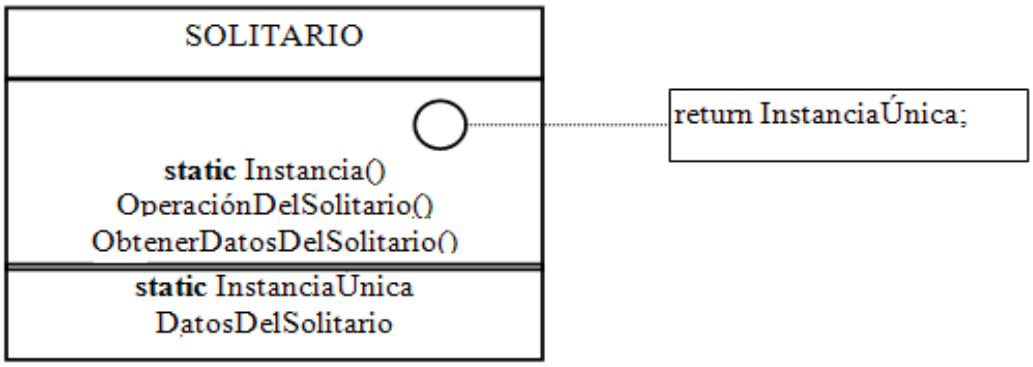


FIG 6. Estructura del patrón Solitario.

Ventajas:

- El acceso a la “Instancia Única” está más controlado.
- Se reduce el espacio de nombres (frente al uso de variables globales).

- Permite refinamientos en las operaciones y en la representación, mediante la especialización por herencia de “Solitario”.
- Es fácilmente modificable para permitir más de una instancia y, en general, para controlar el número de las mismas (incluso si es variable).

Implementación:

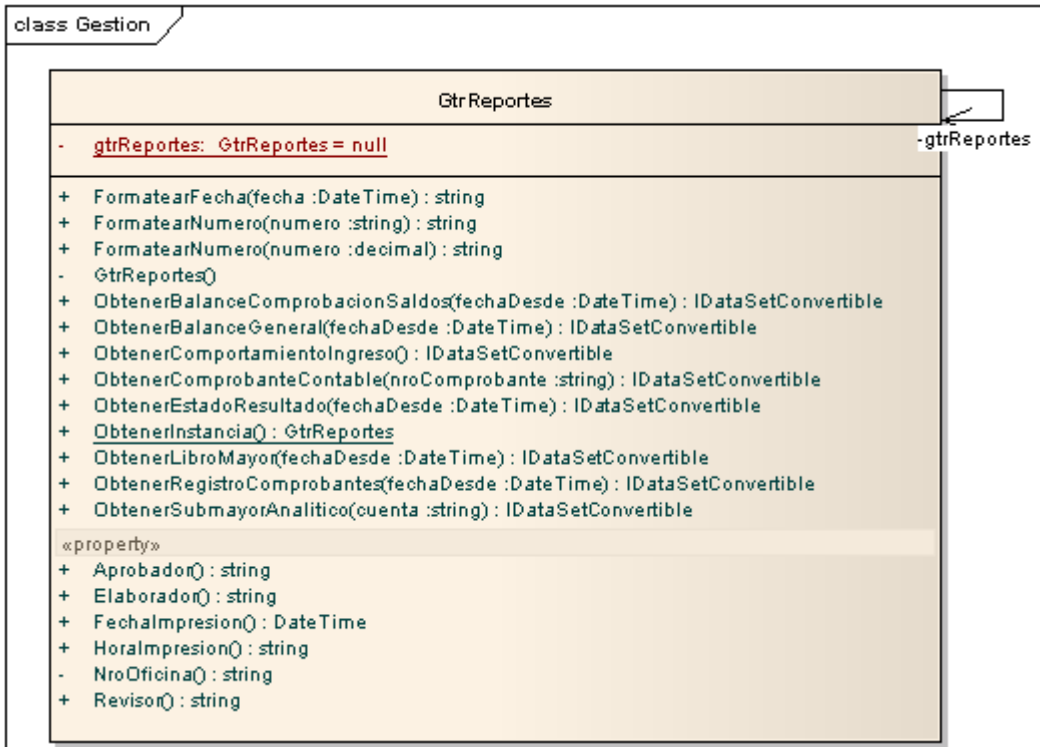


FIG 7. Diseño del Patrón Solitario.

El atributo “gtrReportes” es de tipo clase “GtrReportes” y está inicializado en NULL[19] por defecto, a través de este atributo se obtiene una instancia de la clase en cuestión.

El constructor de la clase “GtrReportes()” es el encargado de darle valor al atributo, este constructor se declara privado para que nadie pueda acceder a él desde fuera de la clase en cuestión.

El método “ObtenerInstancia()” tiene la responsabilidad de devolver una instancia de la propia clase.

[19] NULL: Valor de dato no definido.

De esta forma queda implementado el patrón solitario, garantizando que el acceso global a la clase “GtrReportes” sea a través de un solo punto.

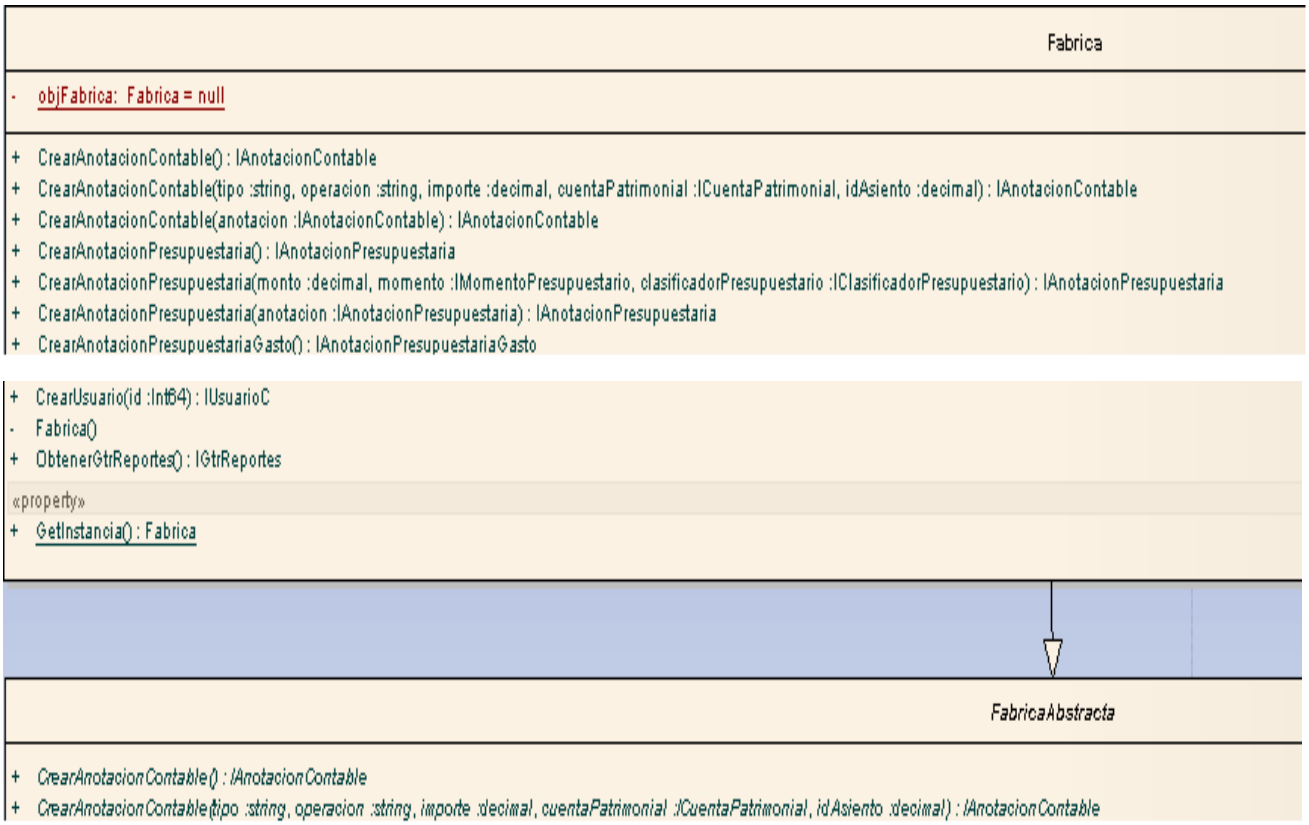


FIG 8. Diseño del Patrón Solitario.

Esta clase tiene la responsabilidad de crear las instancias de todas las entidades, además cuenta con el método “ObtenerReportes()” el cual devuelve una instancia de la clase “GtrReportes”. Esta clase implementa el patrón Fabrica Abstracta al mismo tiempo que incluye el patrón Solitario.

El atributo “objFabrica” es de tipo “Fabrica”, a través de este atributo se obtiene una instancia de la propia clase.

El constructor de la clase “Fabrica()” es el encargado de inicializar el atributo.

El método “GetInstancia()” devuelve una instancia de la propia clase.

- **Patrón Fachada**

Tipo:

Estructura, a nivel de objetos.

Propósito:

Proporcionar una interfaz unificada de alto nivel que, representando a todo un subsistema, que facilite su uso. La "fachada" satisface a la mayoría de los clientes, sin ocultar las funciones de menor nivel a aquellos que necesiten acceder a ellas.

Estructura:

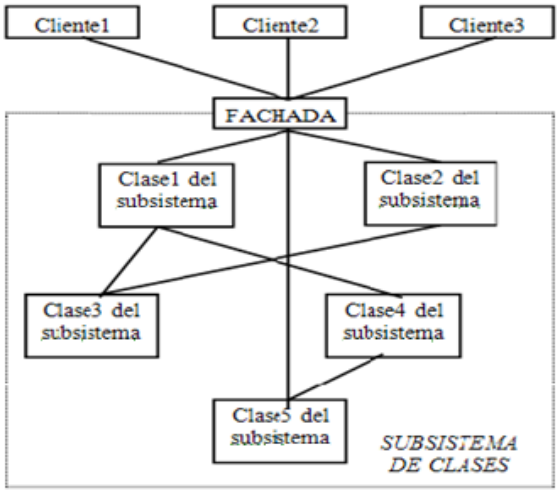


FIG 9. Estructura del patrón Fachada.

Ventajas:

Al separar al cliente de los componentes del subsistema, se reduce el número de objetos con los que el cliente trata, facilitando así el uso del subsistema.

Se promueve un acoplamiento débil entre el subsistema y sus clientes, eliminándose o reduciéndose las dependencias.

No existen obstáculos para que las aplicaciones usen las clases del subsistema que necesiten. De esta forma podemos elegir entre facilidad de uso y generalidad.

Relacionado con:

Patrón Fábrica Abstracta:

Puede usarse conjuntamente con ***Fachada*** para proporcionar una interfaz que permita crear objetos del subsistema de manera independiente a éste. Además, puede ser una alternativa para ocultar las clases específicas de una plataforma.

La implementación que se propone en nuestra arquitectura comprende esta relación a través de spring.net como ya se vio, siendo relativamente sencillo puesto que spring.net ya trae implementado el patrón Fabrica Abstracta. Más información acerca de los patrones aplicados en el sistema. *Más información acerca de los elementos de la arquitectura véase anexo1.*

2.3 Pautas de implementación.

El estándar de codificación fue definido por el grupo de proyecto se define el formato para los siguientes puntos:

2.3.1 Convenciones de código para el lenguaje de programación c#.

- **Identación.**

Cuando una expresión ocupa más de una línea, debemos dividir la misma atendiendo a los siguientes aspectos:
Después de una coma.
Después de un operador.
Alinear la nueva línea al inicio de la expresión.

- **Espacios en blanco.**

Un estándar de identación con espacios en blanco nunca ha existido.
No utilizar espacios en blanco para identar, utilizar tabs de cuatro caracteres (por defecto en el Visual Studio .NET).

- **Comentarios.**

Bloques de comentarios deben ser evitados. Para descripciones se recomienda hacer uso del estándar de C# (///).

- **Líneas.**

Se debe usar // para comentar una línea simple (Alt + / en Visual Studio .NET), aunque puede ser usado para comentar bloques también. Deben estar identados al mismo nivel de la línea que comentan.

- **Documentación.**

En el .NET Framework se ha introducido un sistema para generar documentación basado en comentarios XML.

- **Declaraciones.**

Una declaración por línea es recomendada, ejemplo:
No poner más de una variable o variables de diferentes tipos en una misma línea; tener en cuenta que los nombres deben ser lo más declarativo posible.

- **Convenciones de declaración.**

Estilo camello

Capitaliza la primera letra de cada palabra, excepto para la primera palabra.

Mayúsculas:

Usar este convencionalismo sólo para el caso de identificadores que consisten en abreviaturas de uno o dos caracteres de largo.

El uso de underscore es considerado una mala práctica. Fue utilizado un tiempo atrás para describir el tipo de una variable, pero ya debe considerarse obsoleto.
Un buen nombre de variable describe la semántica, no el tipo.
Con la excepción del código relacionado con la interfaz gráfica de la aplicación. Todos los nombres de variables y campos que contengan elementos de interfaz como botones, se les debe agregar al principio la abreviatura del tipo.

- **Prácticas de programación.**

No hacer instancia alguna o campo de una clase público, deben ser privadas.

Utilizar propiedades, se pueden emplear campos públicos estáticos (o constantes) como excepción de la regla, pero no se recomienda.
No utilizar constantes numéricas directamente en el código, declaramos una variable constante y la utilizamos en su lugar.
Utilizar el estilo de Pascal para los nombres de los formularios.
Utilizar el estilo de Pascal para los nombres de las librerías.
Utilizar enumerados para los nomencladores.

2.3.2 Estándar de codificación.

- **Identado.**

Se utilizan 2 espacios para identar.

- **Estructuras de control.**

Las sentencias de las estructuras de control deben tener un espacio entre la palabra clave y el paréntesis que abre, con el objetivo de distinguirlas de las llamadas a funciones.

Debe usarse siempre las llaves que delimitan los bloques de código, incluso cuando estas son opcionales, con el objetivo de ganar en legibilidad del código.

- **Llamada a funciones.**

Las funciones deben ser llamadas sin espacios entre el nombre de la función y el paréntesis abre. Los parámetros deben ser separados por comas y un espacio después de la coma.

- **Declaración de funciones.**

Los nombres de las funciones deben ser todos en minúsculas.

El nombre de toda función debe comenzar con el nombre del modulo al que pertenece y seguido por un nombre que describa el comportamiento de la función.

- **Arreglos.**

Los arreglos deben ser formateados separando cada elemento del operador de asignación.

Las buenas prácticas de programación y la uniformidad en el código permiten que un programa sea legible y que parezca que fue escrito por un solo programador. *Más información acerca de los elementos de la arquitectura véase anexo2.*

2.4 Pautas para el diseño de interfaces.

- **Propiedades generales de las formas:**

Font (Fuente, Estilo de fuente, Tamaño): Verdana; Normal; 8,25pt

GridSize: 5; 5

Size: 845; 713

pnlAreaTrabajo:

GridSize: 5; 5

Posición de los botones:

btnAyuda: 19; 642
btnAceptar: 668; 642
btnCancelar: 751; 642
btn…(en el caso que la forma requiera otro botón a la izquierda del "btnAceptar**"):** 586; 642

- **Otras Pautas a tener en cuenta:**
  - Todos los elementos visuales de las formas deben quedar contenidos en GroupBox.
  - Los GroupBox tendrán un ancho de 807, su alto estará en dependencia del grupo de componentes visuales que queden contenidos en el mismo.
  - El primer GroupBox estará ubicado en la posición: 19**; 53.**
  - No deben ser usados las abreviaturas, aunque los nombres en las etiquetas deben lo más corto posible, ejemplo**: UEL**, no debe ser usado, a no ser que sea una nomenclatura aprobada por la dirección del proyecto, usar **Unidad Ejecutora Local** siempre y cuando sea posible de acuerdo al diseño de esa forma en particular.
- **En los casos similares deberán usarse las siguientes pautas:**

Las etiquetas **Código, Fecha de Inicio, Duración en días, Fecha Fin** quedan a:

Location (X, Y):

X = 6

Los TextBox correspondientes a cada una de estas etiquetas quedan a:

Location (X, Y):

X = 9

Nota: La diferencia con respecto a su posicionamiento entre las etiquetas y los TextBox correspondientes es de 3 pixeles.

- Los TextBox tendrán un ancho de 23 píxeles, solo en casos como el de **"Breve Descripción"** tendrá un ancho acorde al texto que pueda escribirse en el mismo.

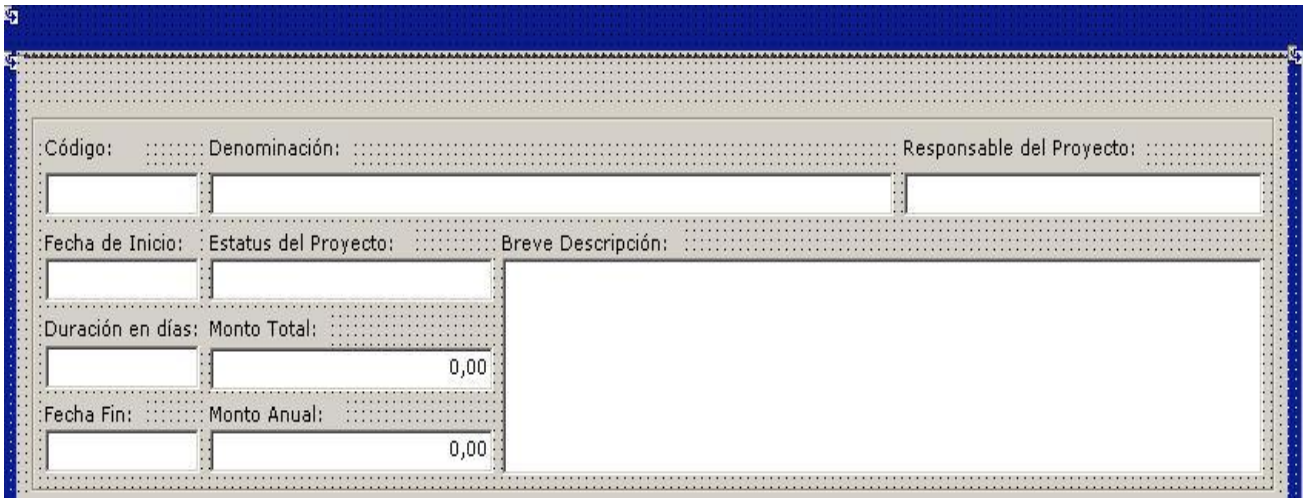


FIG 10. Interfaz de usuario.

2.5 Componentes.

El uso de componentes interfaz de usuario es imprescindible para el desarrollo de sistemas, el equipo de trabajo cuenta con algunos componentes que facilitan considerablemente el desarrollo del sistema.

Entre las ventajas principales se pueden enunciar:

- Disminución del tiempo de desarrollo del software.
- Se reduce el número de líneas de código de la aplicación.
- Aumenta el rendimiento de los programadores.
- Brinda mayor claridad y mantenibilidad del código.

A continuación se describen los principales componentes usados en el sistema:

- **CListView**

Agrega al componente clásico de .NET las funcionalidades que el sistema necesita, este componente, por estar orientado a objeto, necesita una configuración o sea es necesario que se le especifique en cada una de sus columnas el nombre de la propiedad del objeto que se va a mostrar.

Funcionalidades:

- Objeto

Propiedad que admite cualquier tipo de objeto que se quiera mostrar.

- Objetos

Propiedad que admite una lista de cualquier tipo de objetos que se quiera mostrar.

- ControlAsociado

Propiedad que se asocia a otro componente visual que permite editar la información de la celda activa.

Eventos:

- antesAdicionarEventHandler

Permite determinar si se puede o no adicionar un nuevo elemento al CListView.

- subItemBeginEditing

Ocurre antes de editar algún campo con control asociado que tenga el CLsitView.

- subItemEndEditing

Ocurre cuando se finaliza la edición de algún campo con control asociado que tenga el CListView.

- **CManager**

Este componente no tiene una interfaz definida, está orientado al trabajo sobre los demás componentes como son CTexBox y CCheckBox y CListView. Cada uno de

estos componentes debe tener especificado la propiedad que muestra del objeto que se va manejar.

Funcionalidades:

- SetValor(Object object)

Le asigna a cada componente asociado el valor de la propiedad que le corresponde del objeto que recibe por parámetro.

- GetValor(Objcet object)

Captura de cada componente visual los datos y los almacena en el objeto especificado.

- **CCondicion**

Este componente no tiene definida una interfaz, el mismo está orientado a establecer reglas sobre los CTexBox. Las reglas pueden ser variadas en dependencia de la lógica que se desee definir en cada CTexBox.

Permite validar el rango de valores válidos o permisibles en los CTextBox asociados.

- **Nomenclador**

Tiene las funcionalidades del ComboBox de .Net, se le agregan nuevas funcionalidades que dan factibilidad al sistema.

Las principales funcionalidades de este componente están orientadas al trabajo con los datos nomencladores los cuales son cargados desde la base de datos utilizando las principales propiedades que permiten configurar el nomenclador en tiempo de diseño.

Funcionalidades:

- TipoOrigenDatos:

Se asocia con una vista o un procedimiento de almacenado.

- NombreOrigenDatos:

Se selecciona el nombre de la vista o el procedimiento al que se quiera asociar.

- ColumnaIdentificador

Se especifica que parámetro es el identificador en los datos de retorno.

- ColumnaMostrar

Se especifica que parámetro se mostrara al usuario en los datos de retorno.

- CapturarParametros

Se definen los parámetros de entrada al método de acceso a datos si los requiere.

- IdSeleccionado

Devuelve el parámetro definido en la ColumnaIdentificador identificador.

2.6 Programación orientada a eventos.

El rol de programador en el equipo de desarrollo del sistema administración financiera tiene mayor responsabilidad en la capa de negocio y principalmente en la capa de presentación.

En esta capa se decidió manejar las responsabilidades de cada componente de interfaz a través de una clase independiente, abstrayendo de esta forma los formularios de sus funciones. Los formularios se controlan aplicando una programación orientada a eventos.

Son eventos típicos el Click sobre un botón, el hacer doble Click sobre el nombre de un fichero para abrirlo, el arrastrar un icono, el pulsar una tecla o combinación de teclas, el elegir una opción de un menú, el escribir en una caja de texto, o simplemente mover el ratón.

Para mejor entendimiento se muestran algunos ejemplos que muestran de forma práctica como se aplicó dicho estilo de programación en el módulo de Contabilidad.

- **Botones**

El evento más usado es el CLICK, en el cual se programa la lógica definida en la interfaz. Ejemplo:

Se ordena guardar información en la base de datos.

Se ordena cargar información desde la base de datos.

Se ordena mostrar los resultados de una búsqueda.

- **Formularios**

El evento más usado en un formulario es el LOAD, en el cual se muestra la información inicial del mismo.

- **Navegación de información a través de formularios**

Frecuentemente a los desarrolladores se les hace necesario manejar la misma información (instancia de una clase) de un formulario a otro, esto es implementado apoyándose en la programación orientada a eventos.

El formulario secundario notifica al formulario principal (el principal ejecuta al secundario) que ejecute una acción determinada, por ejemplo:

El formulario secundario ordena al principal que guarde los datos en la base de datos.

Los lenguajes visuales que facilitan la programación orientada a eventos y con manejo de componentes dan al usuario que no cuenta con mucha experiencia en desarrollo, la posibilidad de construir sus propias aplicaciones utilizando interfaces gráficas sobre la base de ocurrencia de eventos.

Conclusiones parciales:

Por la arquitectura propuesta, los patrones arquitectónicos que la componen así como los elementos más importantes dentro de esta y por los patrones de diseño aplicados, las pautas de programación establecidas, los componentes utilizados en aras de agilizar y facilitar el desarrollo; podemos concluir, que se ha obtenido un producto de software listo para ser probado.

Capitulo 3 Análisis de los resultados

Introducción

En este capítulo se presentan los artefactos del mecanismo de diseño e implementación obtenidos teniendo en cuenta la responsabilidad del rol desempeñado dentro del equipo de desarrollo. Se describen además los resultados obtenidos en la aplicación de métricas y pruebas de software.

3.1 Modelo de Diseño

El diseño de sistemas se ocupa de desarrollar las directrices propuestas durante el análisis en términos de aquella configuración que tenga más posibilidades de satisfacer los objetivos planteados tanto desde el punto de vista funcional como del no funcional. El proceso de diseño de un sistema complejo se suele realizar de forma descendente:

- Diseño de alto nivel (o descomposición del sistema a diseñar en subsistemas menos complejos).
- Diseño e implementación de cada uno de los subsistemas.
- Especificación consistente y completa del subsistema de acuerdo con los objetivos establecidos en el análisis.
- Desarrollo según la especificación.
- Prueba.
- Integración de todos los subsistemas.
- Validación del diseño.

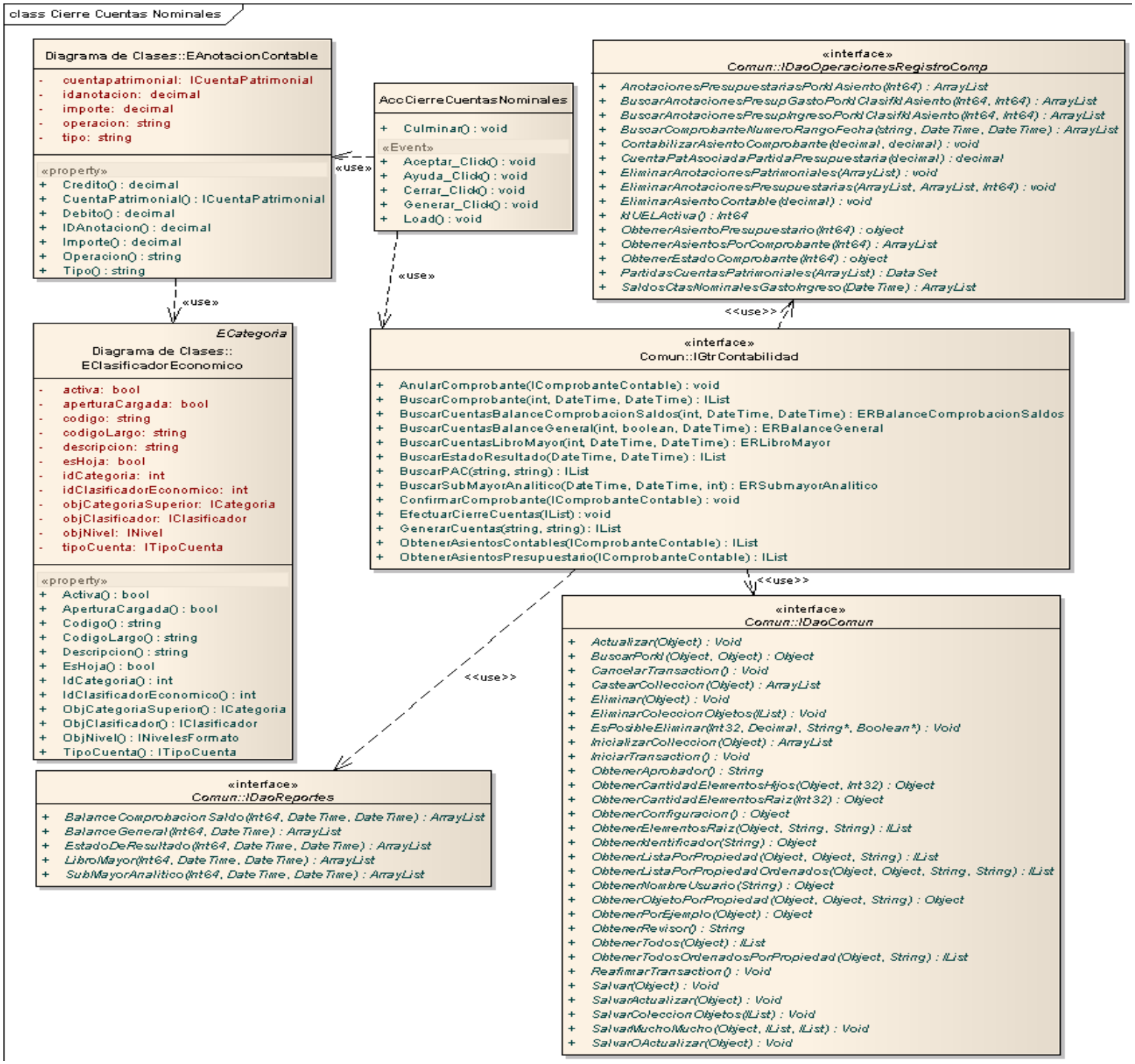
Como parte del Modelo de Diseño se procede a la realización de los casos de uso lo cual consiste en la realización de los diagramas de Clases, y diagramas de Interacción (diagramas de Secuencia y diagramas de Colaboración).

3.1.1 Diagramas de clases

Los diagramas de clases proporcionan una perspectiva estática del sistema, un diseño estructural.

- **Caso de uso Cierre de Cuentas Nominales.**

Este caso de uso consiste en generar un comprobante contable para llevar el saldo de las cuentas nominales a cero y visualizar el estado de resultados según el período establecido, estas cuentas pueden ser de ingreso ó de gasto.

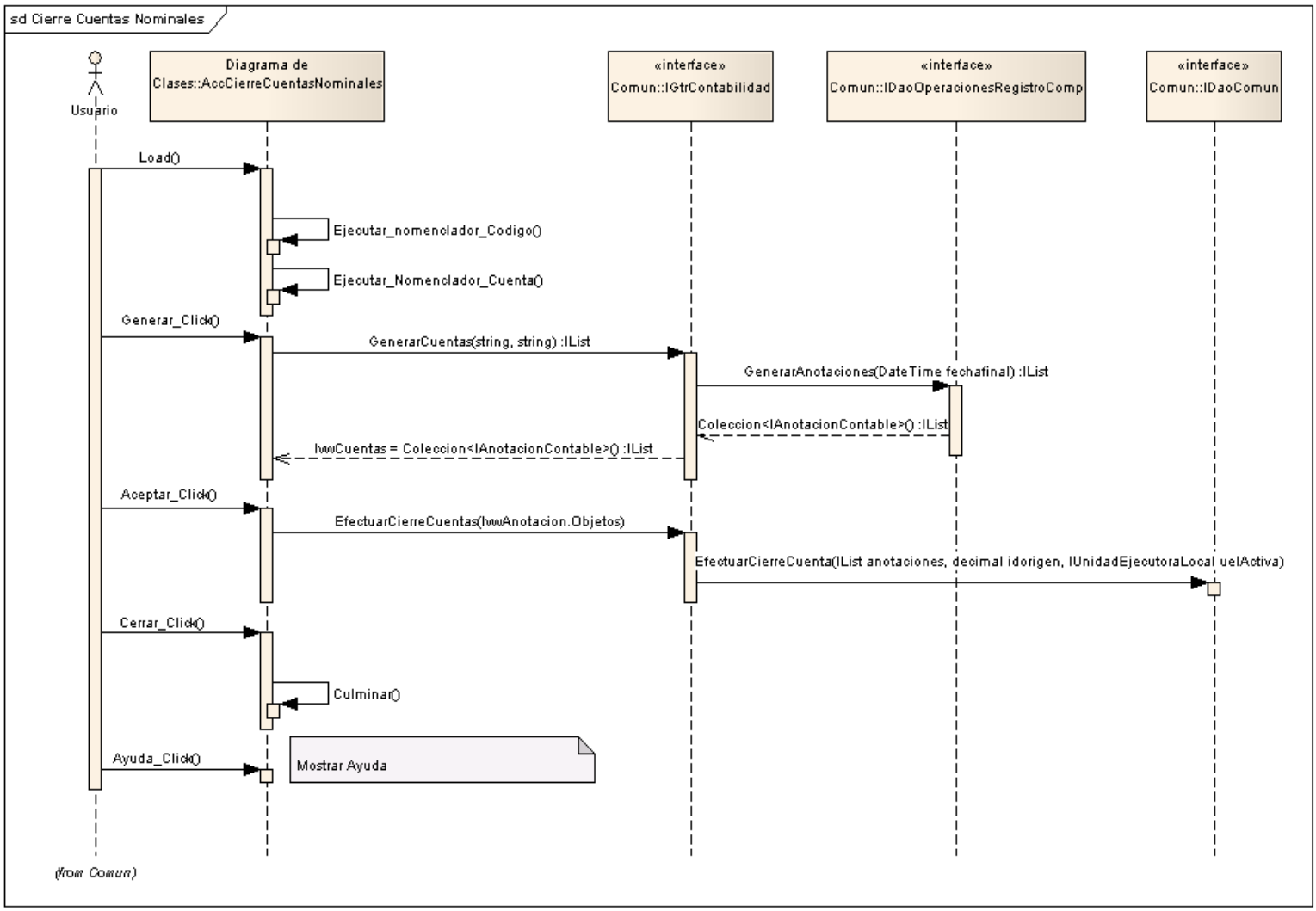


3.1.2 Diagramas de interacción

Los diagramas de interacción se utilizan para modelar los aspectos dinámicos de un sistema, lo que conlleva a modelar instancias concretas o prototípicas de clases interfaces, componentes y nodos, junto con los mensajes enviados entre ellos, todo en el contexto de un escenario que ilustra un comportamiento.

En el contexto de las clases se describe la forma en que grupos de objetos colaboran para proveer un comportamiento.

Un diagrama de secuencia es un diagrama de interacción que destaca la ordenación temporal de los mensajes; un diagrama de colaboración es un diagrama de interacción que destaca la organización estructural de los objetos que envían y reciben mensajes.



- **Caso de uso Reporte Libro Mayor.**

Este caso de uso consiste en mostrar a través de un reporte a nivel de cuentas contables el saldo inicial, debe, haber y saldo final de las mismas. Partiendo de la información que van generando los comprobantes. Anexo3.

- **Caso de uso Reporte Balance General.**

Este caso de uso consiste en realizar un Reporte Contable donde se muestran las cuentas que pertenecen a los activos, pasivos, y patrimonio. El saldo de los activos debe ser igual a la suma del saldo de los pasivos más las del patrimonio, partiendo de la información que van generando los comprobantes. Anexo4.

- **Caso de uso Reporte Sub-Mayor Analítico.**

Este caso de uso consiste en realizar un Reporte Contable por cuenta seleccionada, mostrando el saldo de la misma y los comprobantes que influyeron en este saldo. Partiendo de la información que van generando los comprobantes. Anexo5.

- **Caso de uso Reporte Estado de Resultado.**

Este caso de uso consiste en realizar un Reporte Contable donde se registren los ingresos y los gastos cuya diferencia resulta ser ganancia o pérdida en el período y en el acumulado del Ejercicio Fiscal, partiendo de la información que van generando los comprobantes contables. Anexo6.

- **Caso de uso Reporte Balance de Comprobación de Saldo.**

Este caso de uso consiste en generar un Reporte Contable mostrando en cualquier nivel de las cuentas los saldos de las mismas en el período contable analizado y acumulado en el Ejercicio Fiscal, partiendo de la información que van generando los comprobantes. Anexo7.

- **Caso de uso Gestionar Registro de Comprobantes.**

Este caso de uso consiste en gestionar los comprobantes contables, visualizarlos en un registro tanto los manuales como los automatizados.Anexo8.

3.2 Modelo de implementación

En este modelo se implementa el sistema diseñado en términos de componentes (ficheros de código fuente, scripts, ficheros de código binario, ejecutables, entre otros). Los propósitos de la implementación son los siguientes:

- Planificar las integraciones del sistema necesarias en cada iteración. Se sigue un enfoque incremental dando lugar a un sistema que se implementa en una sucesión de pasos pequeños y manejables.
- Distribuir el sistema asignando componentes ejecutables a nodos en el Diagrama de Despliegue.
- Implementar las clases y subsistemas encontrados durante el diseño. Las clases se implementan como componentes de fichero que contienen código fuente.
- Probar los componentes individualmente, y a continuación integrarlos y enlazarlos en uno o más ejecutables, antes de ser enviados para ser integrados y llevar a cabo las comprobaciones del sistema.

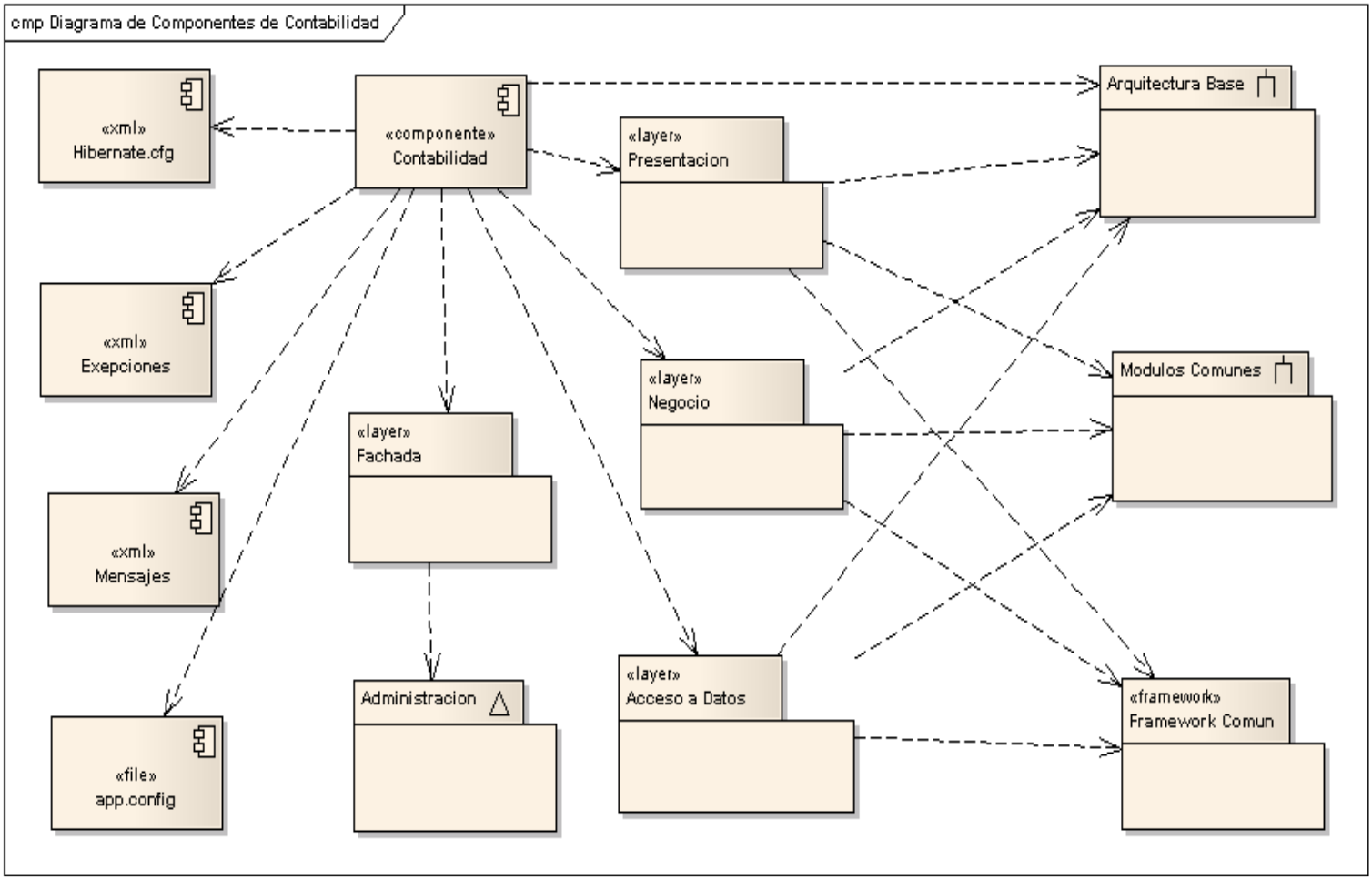
3.2.1. Diagrama de Componentes

Un diagrama es una vista de implementación que muestra las organizaciones y dependencias lógicas entre componentes software, sean éstos componentes de código fuente, binarios o ejecutables. Desde el punto de vista del diagrama de componentes se tienen en consideración los requisitos relacionados con la facilidad de desarrollo, la gestión del software, la reutilización, y las restricciones impuestas por los lenguajes de programación y las herramientas utilizadas en el desarrollo. Los elementos de modelado dentro de un diagrama de componentes serán componentes y paquetes. En cuanto a los componentes, sólo aparecen tipos de componentes, ya que las instancias específicas de cada tipo se encuentran en el diagrama de despliegue.

Dado que los diagramas de componentes muestran los componentes software que constituyen una parte reusable, sus interfaces, y sus interrelaciones, en muchos aspectos se puede considerar que un diagrama de componentes es un diagrama de clases a gran escala. Cada componente en el diagrama debe ser

documentado con un diagrama de componentes más detallado, un diagrama de clases, o un diagrama de casos de uso.

Un paquete en un diagrama de componentes representa una división física del sistema. Los paquetes se organizan en una jerarquía de capas donde cada capa tiene una interfaz bien definida.



3.2.2 Diagrama de despliegue

Un diagrama de despliegue muestra las relaciones físicas entre los componentes hardware y software en el sistema final, es decir, la configuración de los elementos de procesamiento en tiempo de ejecución y los componentes software (procesos y objetos que se ejecutan en ellos).

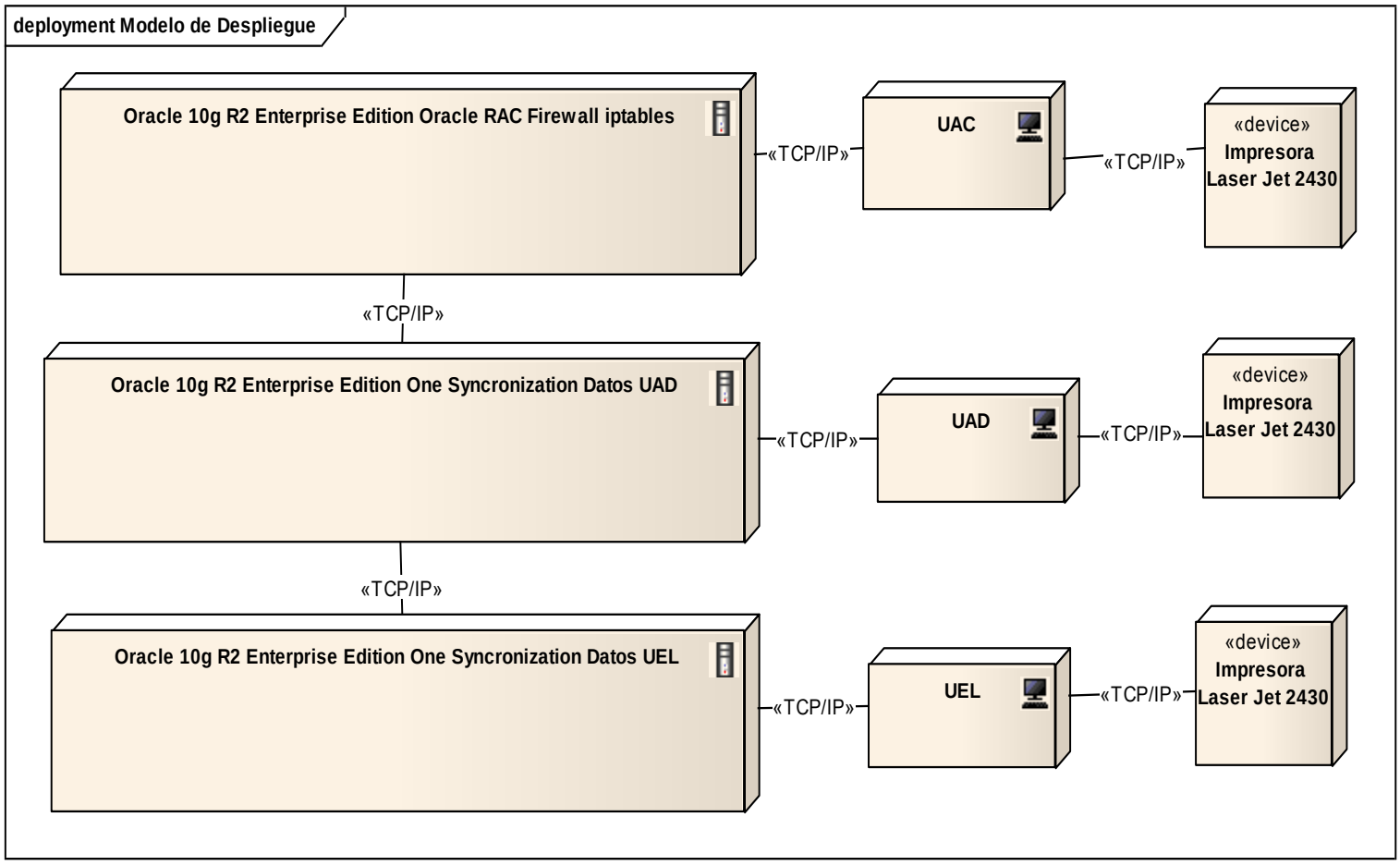
Estarán formados por instancias de los componentes software que representan manifestaciones del código en tipo de ejecución (los componentes que sólo sean

utilizados en tiempo de compilación deben mostrarse en el diagrama de componentes).

En general un nodo será una unidad de computación de algún tipo, desde un sensor a un mainframe[20]. Las instancias de componentes software pueden estar unidas por relaciones de dependencia, posiblemente a interfaces ya que un componente puede tener más de una interfaz.

La mayoría de las veces el modelado de la vista de despliegue estática implica modelar la topología del hardware sobre el que se ejecuta el sistema y el diagrama de despliegue del módulo Contabilidad no es la excepción.

Los diagramas de despliegue son fundamentalmente diagramas de clases que se ocupan de modelar los nodos de un sistema.



[20] Mainframe: Programa principal.

3.3 Métricas del diseño de software

Las métricas de diseño están creadas para medir la calidad desde cualquier punto vista desde el que se quiera saber si un diseño determinado responde a ciertas cualidades determinadas.

Estas métricas están definidas a nivel mundial con el objetivo de estandarizar las cualidades y potencialidades de un diseño de software, cabe aclarar que estas métricas no producen resultados exactos ya que es muy engorrosa la tarea de determinar si un diseño está al máximo en cuanto a calidad.

Muchos expertos aseguran que se necesita más experimentación hasta que se puedan emplear las métricas de diseño y sin embargo no se concibe un diseño sin medición.

La calidad del diseño del módulo Contabilidad se ha medido a través de las métricas orientadas a clases dentro de las cuales están la serie de métricas ***CK*** y las métricas propuestas por Lorenz y Kidd.

3.3.1 Serie de métricas CK

Uno de los conjuntos de métricas orientadas a objetos ampliamente referenciados, ha sido el propuesto por Chidamber y Kemerer. Normalmente conocidas como la serie de métricas ***CK***, los autores han propuesto seis métricas basadas en clases para sistemas orientados a objetos. (Pressman, 1998)

- **Árbol de profundidad de herencia (APH)**

Esta métrica se define como <<la máxima longitud del nodo a la raíz del árbol >>. A medida que el ***APH*** crece, es posible que clases de más bajos niveles hereden muchos métodos. Esto conlleva a dificultades potenciales, cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clases profunda (el ***APH*** es largo) también conduce a una complejidad de diseño mayor. Por el lado positivo, los valores ***APH*** grandes implican un gran número de métodos que se reutilizarán. (Pressman, 1998)

Por su parte, algunos autores sugieren un umbral de seis niveles como indicador de un abuso en la herencia en distintos lenguajes de programación.

Resultados:

A partir de la aplicación de la métrica ***APH*** al sistema se obtuvo como resultado que los niveles más altos de herencia son de 2 lo cual deduce que se hace un correcto uso de la herencia. La figura muestra la herencia entre la clase entidad EAsientoContable y la clase entidad EAsientoPatrimonial.

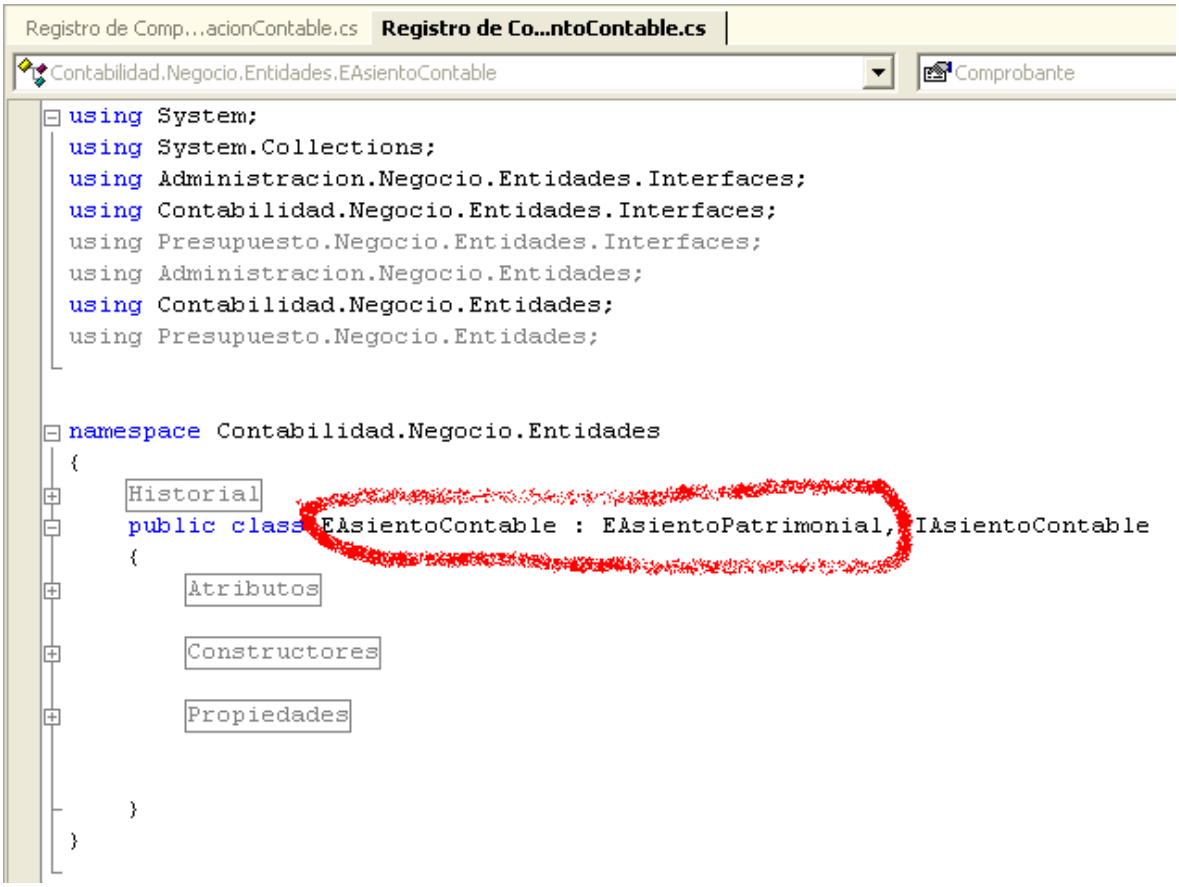


FIG 11. Entidad EAsientoContable.

3.3.2 Métricas propuestas por Lorenz y Kidd

- **Tamaño de clase (TC):**

Esta métrica consiste en determinar el tamaño de una clase a partir de los resultados que se obtengan en el cálculo de los siguientes parámetros:

- Total de operaciones (operaciones tanto heredadas como privadas de la instancia) que se encapsulan dentro de la clase.
- Número de atributos (atributos tanto heredados como privados de la instancia) encapsulados por la clase.
- Promedio general de los dos parámetros anteriores para el sistema en general.

Un **TC** grande afecta los indicadores de calidad definidos para esta métrica por los especialistas.

Parámetros de calidad	A valores grande de TC
Reutilización	Reduce la reutilización de la clase
Implementación	Complica la implementación
Complejidad de las pruebas	Hace compleja las pruebas del sistema
Responsabilidad	La clase debe tener bastante responsabilidad

Tabla 1. Parámetros de calidad para valores grandes de TC.

Las medidas o metas definidas para los parámetros han sido a lo largo de la historia un tema de discusión a nivel mundial entre los expertos de la materia, muchos de estos especialistas plantean como umbrales los siguientes valores:

TC	Valor obtenido
Pequeño	<=20
Medio	>20 y <=30
Grande	>30

Tabla 2. Nro. atributos y/o operaciones.

Todas las clases del sistema se encuentran en una tabla que muestra el nombre de la clase, total de atributos y operaciones que contiene y un tamaño estimado a partir de los umbrales establecidos. Anexo10.

Resultados:

Los resultados obtenidos al aplicar esta métrica en el diseño realizado fueron los siguientes:

La aplicación cuenta con un total de 46 clases con un promedio de 4.7 atributos por clases y 4.3 operaciones como se muestra en la tabla 4.

Total de Clases	Promedio de Atributos	Promedio de Operaciones
46	4.7	4.3

Tabla 4. Totales de la capa de negocio.

De manera más detallada se obtuvo que la aplicación está compuesta por 44 clases pequeñas, 2 clase media y 0 clase grande.

Umbral	Tamaño	Cantidad de Clases
<=20	Pequeño	44
>20 y <=30	Medio	2
>30	Grande	x

Tabla 5. Cantidad de clase por tamaño.

Otra forma representativa es el por ciento de clase según los tamaños a partir de los umbrales definidos en la **tabla 2** se puede observar que existe en la capa de negocio un 95.66% de clase pequeñas, un 4.34% de clases medianas y un 0% de clases grandes.

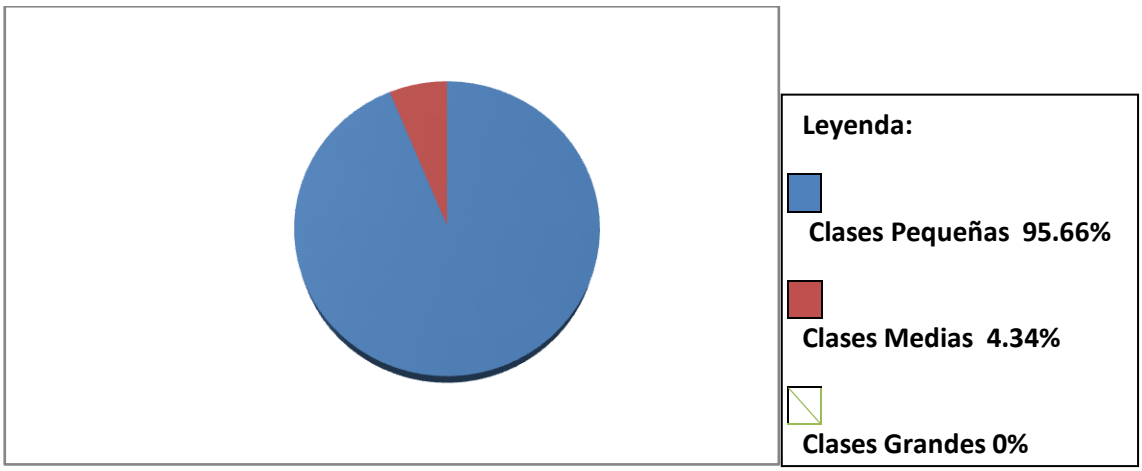


Fig12. Por ciento de clases por tamaño.

Como se puede observar en las tablas que muestran los resultados la mayor cantidad de clases están clasificadas entre pequeñas y medianas para un resultado positivo según los parámetros de calidad propuesto por la métrica **TC.**

- **Número de operaciones redefinidas para una sub-clase (NOR):**

Existen casos en que una subclase reemplaza una operación heredada de su superclase por una versión especializada para su propio uso. A esto se le llama redefinición. Los valores grandes para el **NOR**, generalmente indican un problema de diseño. (Pressman, 1998)

Esta es una de las métricas que se aplican para medir la calidad del diseño propuesto para los registros y notarías de Venezuela.

Resultado: A partir de los datos obtenidos después de aplicar al sistema la métrica **NOR** se obtuvo lo siguiente:

Cantidad de clases del sistema	Total de subclases que reemplazan operaciones
58	2

Tabla 6. Total de clases que reemplazan operaciones.

El número de subclases que remplazan operaciones de las superclases es mínimo en relación a la cantidad de clases que conforman el sistema. Se puede ver que existe una jerarquía adecuada, esto permite que el software pueda ser fácilmente probado y modificado en el tiempo.

3.4 Métodos de prueba de software

Las pruebas son de gran importancia en la garantía del software. Por lo general están orientadas a comprobar la funcionalidad (el sistema debe comportarse a la altura de los requisitos especificados por el cliente) y lógica de negocio.

En programas pequeños la verificación de todas sus posibles alternativas del flujo de control es relativamente fácil, pero en programas mayores o de gran

complejidad es imposible efectuar pruebas exhaustivas. Sin embargo, una selección cuidadosa de los datos de prueba puede darnos mucha confianza en el desempeño que tengan esos programas. Esto, aunado a un determinado mecanismo de comprobación de errores, puede producir un software más confiable.

Es importante destacar que los objetivos principales de la realización de una prueba son:

- Detectar un error.
- Tener un buen diseño de caso de prueba.
- Descubrir un error no descubierto antes.

3.4.1 Caja negra (prueba unitaria)

El Método de la Caja Negra se centra en los requisitos fundamentales del software y permite obtener entradas que prueben todos los requisitos funcionales del programa.

Con este tipo de pruebas se intenta encontrar:

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

- **El caso de uso Balance de comprobación de Saldos muestra los resultados de las pruebas de caja negra.**

Pruebas realizadas al caso de uso.

- Caso de prueba para Visualizar el Libro Mayor.
- Caso de prueba para Visualizar en un Reporte.

CPR1. Visualizar el balance de comprobación de saldo

1.1 Descripción de la funcionalidad.

Es iniciada por el responsable de la Unidad de Contabilidad y permite generar un Reporte Contable mostrando en cualquier nivel de las cuentas los saldos de las mismas en el período contable analizado y acumulado en el Ejercicio Fiscal, partiendo de la información que van generando los comprobantes.

1.2 Flujo central.

- El responsable ordena Visualizar el Balance de comprobación de Saldos.
- El sistema muestra la interfaz con los datos correspondientes.
- El responsable selecciona el Nivel de la Cuenta, Fecha Desde y la Fecha hasta.
- El responsable ordena buscar.
- El sistema visualiza el Balance de comprobación de Saldos con los datos.
- El responsable ordena aceptar la operación.
- El sistema acepta la operación.
- El responsable ordena cerrar la interfaz.
- El sistema cierra la interfaz. Terminando así el Caso de Uso.

1.3 Iteraciones.

Clases Válidas	Clases Inválidas	Resultado Esperado	Resultado de la Prueba	Observaciones
El botón Buscar debe estar activado para realizar la búsqueda correspondiente		El botón Buscar permanece activado	Satisfactorio	
El botón Cerrar debe cerrar la interfaz		El botón Cerrar cierra la interfaz	Satisfactorio	

CPR2. Visualizar en un reporte.

2.1 Descripción de la funcionalidad.

Es iniciada por el responsable de la Unidad de Contabilidad y permite visualizar en un Reporte.

2.2 Flujo Central.

- El responsable ordena visualizar el Reporte Balance de Comprobación de Saldos.
- El sistema visualiza el reporte.

2.3 Iteraciones.

Clases Válidas	Clases Inválidas	Resultado Esperado	Resultado de la Prueba	Observaciones
El botón Visualizar debe mostrar un reporte con los datos del balance		Muestra el reporte	Satisfactorio	

El registro de defectos y dificultades encontrados no mostró argumento alguno por lo que concluimos este caso de uso con estado satisfactorio. El Anexo11 muestra la interfaz de usuario sobre la que se aplicó el caso de prueba.

- **El caso de uso Cierre de Cuentas Nominales muestra los resultados de las pruebas de caja negra.**

Pruebas realizadas al caso de uso.

- Caso de prueba para cerrar Cuentas Nominales.

CPR1. Cerrar Cuentas nominales

1.1 Descripción de la funcionalidad

Es iniciada por el responsable de la Unidad de Contabilidad y permite generar un comprobante para llevar el saldo de las cuentas nominales a cero.

1.2 Flujo central

- El responsable ordena cerrar las Cuentas Nominales.
- El sistema muestra la interfaz correspondiente.
- El responsable selecciona el código o la cuenta de cierre.
- El responsable ordena generar el comprobante.
- El sistema visualiza el comprobante con los datos.

- El responsable ordena aceptar la operación.
- El sistema acepta la operación.
- El responsable ordena cerrar la interfaz.
- El sistema cierra la interfaz. Terminando así el Caso de Uso.

1.3 Condiciones de ejecución

Deben existir cuentas patrimoniales. Debe existir un ejercicio fiscal y los períodos contables asociados.

1.4 Iteraciones

Clases Válidas	Clases Inválidas	Resultado Esperado	Resultado de la Prueba	Observaciones
El botón Aceptar debe permanecer inhabilitado hasta que se genere el comprobante.		El botón Aceptar permanece inhabilitado hasta que se genere e comprobante.	Satisfactorio	
Al seleccionar el código de la cuenta, el sistema debe mostrar la denominación de la misma y viceversa.		El sistema actualiza el campo Cuenta de Cierre al seleccionar el código de la cuenta y viceversa.	Satisfactorio	
AL oprimir el botón Generar el sistema debe mostrar los datos del comprobante.		El sistema muestra los datos del comprobante.	Satisfactorio	
Al oprimir el botón Cerrar antes de Aceptar el sistema no debe efectuar el cierre de las cuentas, debe cerrar la interfaz.		El sistema cierra la interfaz y no efectúa el cierre.	Satisfactorio	
Al oprimir el botón Aceptar el sistema debe mostrar un mensaje de confirmación antes de cerrar las cuentas, si se le da cancelar al mensaje, el sistema debe cancelar la operación. Si se le da Aceptar al mensaje el sistema debe mostrar otro mensaje diciendo que la operación se realizo satisfactoriamente.		El sistema muestra todos los mensajes.	Satisfactorio	
Si ya se cerraron las cuentas el sistema muestra un mensaje diciendo que las cuentas ya están cerradas.		El sistema muestra el mensaje	Satisfactorio	

El Anexo11 muestra la interfaz de usuario sobre la que se aplicó el caso de prueba.

3.4.2 Caja blanca (prueba unitaria)

La prueba de caja blanca del software se basa en el minucioso examen de los detalles procedimentales del software.

- Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y\o ciclos.
- Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el esperado o mencionado.

Las pruebas de caja blanca que se realizaron estuvieron orientadas a los componentes de interfaz visual que se desarrollaron por el equipo de desarrollo sobre la base del framework NUnit de .NET.

- Método que declara el punto de inicio de prueba. El método InicializandoListView inicializa un componente CListView, el método InicializandoCTexbox inicializa un componente CTextBox.

```
[SetUp]
public void Inicializador()
{
    InicializandoListView();
    InicializandoCTextBox();
}
```

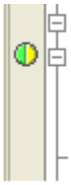
- Esta prueba valida que el CListView agrega un elemento correctamente a su lista de objetos.

```
[Test]
public void PruebaAdicionando()
{
    Assert.AreEqual(personaEsperada.Nombre,(listView.Objetos[0] as Persona).Nombre,"Probando Adicionar Nombre");
    Assert.AreEqual(personaEsperada.Edad,(listView.Objetos[0] as Persona).Edad,"Probando Adicionar Edad");
}
```

- Esta prueba valida que el CListView elimine un elemento de su lista de objetos según la posición especificada por parámetro.

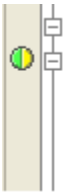
```
public void PruebaEliminar()
{
    listView.Del(0);
    Assert.AreEqual(0, listView.Objetos.Count);
}
```

- Esta prueba valida que el CListview no se adicione un elemento repetido.



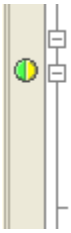
```
[Test, ExpectedException(typeof(Exception))]
public void PruebaInsertandoRepetido()
{
    listView.Add(persona);
}
```

- Esta prueba valida que el CListView lance una excepción si se ordena eliminar un elemento en una posición que no existe.



```
[Test, ExpectedException(typeof(Exception))]
public void PruebaEliminarPosInvalidad()
{
    listView.Del(5);
}
```

- Esta prueba valida que el CTextBox no acepte un dato que no sea del tipo especificado.

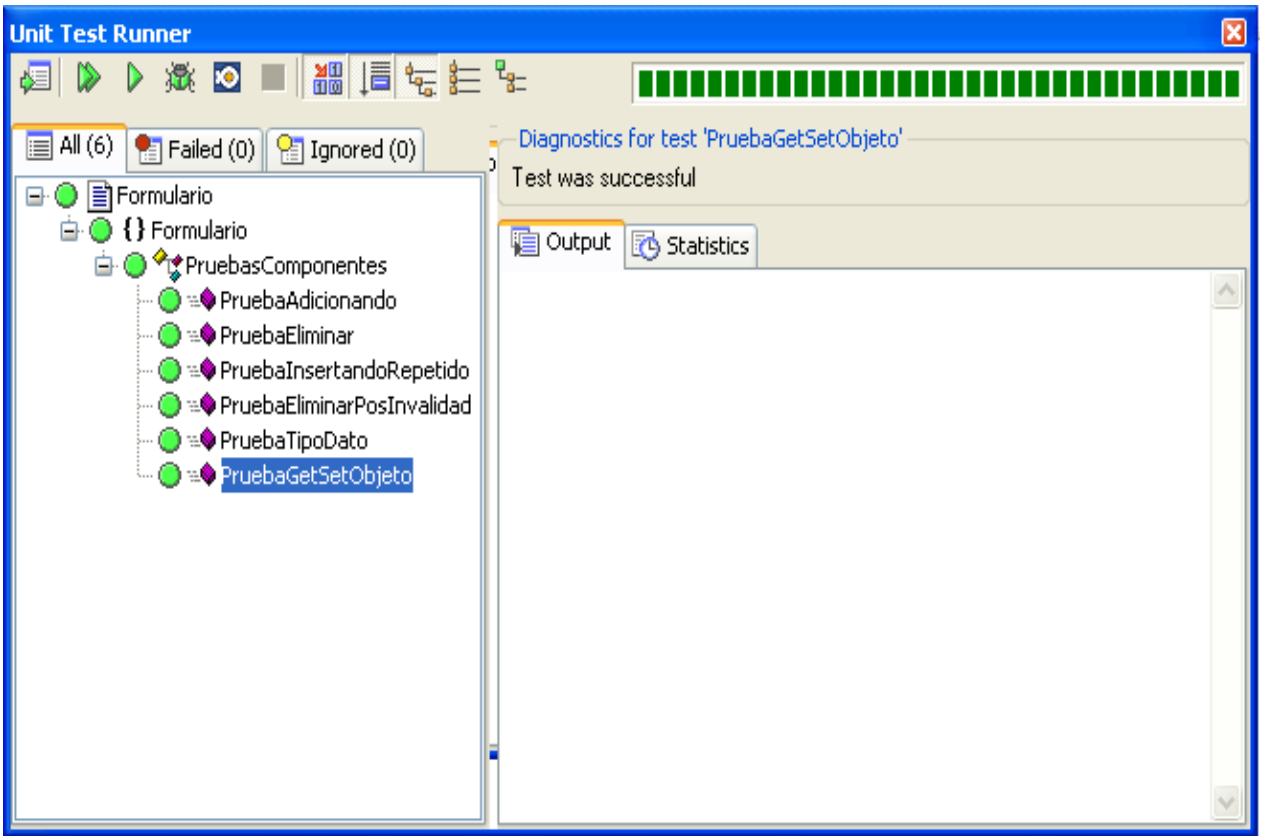


```
[Test]
public void PruebaTipoDato()
{
    textBox.Text = "tipoDato";
    Assert.AreNotEqual("tipoDato", textBox.Text);
}
```

- Esta prueba valida que el CTextBox extraiga del objeto la información que contiene según la propiedad especificada ó le introduzca al objeto la información que contiene en la propiedad especificada.

```
[Test]
public void PruebaGetSetObjeto()
{
    objpersona = new Persona("nombre",55,true,"calle",120);
    textBox.Objeto = objpersona;
    textBox.SetValor();
}
```

Este framework define un color amarillo para las pruebas que hayan sido ignoradas (no realizadas), para las pruebas que muestren resultado fallido se define un color rojo y para las pruebas que hayan mostrado los resultados esperados se muestran en color verde.



Leyenda:

Los resultados fueron satisfactorios

Resultados fallidos.

Resultados ignorados.

Conclusiones parciales:

Teniendo en cuenta la responsabilidad asignada dentro del rol desempeñado en el equipo de trabajo, se obtuvieron satisfactoriamente los artefactos requeridos. Tanto el diseño como la implementación fueron sometidos a validaciones con vistas a obtener un producto de calidad.

Conclusiones

- Se profundizó en el estudio de los temas relacionados con la Contabilidad y los procesos contables, útiles para el mejor entendimiento del negocio en cuestión.
- Se logró automatizar los procesos contables vigentes en las oficinas del MPPRIJ de la República Bolivariana de Venezuela.
- Se obtuvo un producto de calidad en cuanto a diseño e implementación a partir de las métricas y pruebas de software aplicadas.

Recomendaciones

- Continuar profundizando en los estudios sobre la contabilidad y los procesos contables.
- Agregar nuevas iteraciones en las pruebas de caja negra.
- Realizar pruebas de caja blanca a la capa de negocio y acceso a datos.

Bibliografía

AICPA. American Institute of Certified Public Accountants. *American Institute of Certified Public Accountants.* [En línea]

Almenares Herrera, Kiosmy. 2007. *Diseño e Implementación del proceso de inscripción del Módulo Mercantil en las Oficinas Registrales de la República Bolivariana de Venezuela.* C.Habana : s.n., 2007.

Booch, Grady. 1998. *OBJECT-ORIENTED ANALYSIS AND DESIGN.* Santa Clara, California : s.n., 1998.

CATACORA, F. 1998. *Contabilidad. La base para las decisiones gerenciales.* Caracas : s.n., 1998.

Congote Edgar, John. 2006.. Programación Orientada a Aspectos. *Programación Orientada a Aspectos.* [En línea] 2006.

González, Benjamín. 2007. Introducción al entorno de desarrollo de Microsoft .NET. *Introducción al entorno de desarrollo de Microsoft .NET.* [En línea] Marzo de 2007.

Horngren, Charles. 1991. *Accounting.* 1991.

ISO 9126-3, Querétaro. 2006. Métricas Internas de la Calidad del Producto de Software. *Métricas Internas de la Calidad del Producto de Software.* [En línea] marzo de 2006.

Meigs, R. 1992. IN46A CONTABILIDAD Y CONTROL DE GESTIÓN. *IN46A CONTABILIDAD Y CONTROL DE GESTIÓN.* [En línea] 1992.

Mendoza Sanchez, María A. 2004. [En línea] 2004.

Müller, Peter. 1997 . *Introducción a la Programación Orientada a Objetos.* 1997 .

Pestano Pino, Henrik. 2007. *Diseño del módulo para la Administración Contable y Financiera de los Registros y Notarías de la República Bolivariana de Venezuela.* C. Habana : s.n., 2007.

Pressman, R.S. 1998. *Ingenieria de software. Un enfoque practico. Vol. Vol. 1. .* 1998.

Schwindt, Ariel. 2007. Construcción de Sistemas Multiplataforma basados en Servicios. MSDN. *Construcción de Sistemas Multiplataforma basados en Servicios. MSDN.* [En línea] 2007.

Serrano Cinca, Carlos. 2006. Evolución de la contabilidad. [En línea] 2006.

USR.CODE. 2004. Programación Orientada a Objetos. *Programación Orientada a Objetos.* [En línea] 2004.

Anexos

Anexo1:

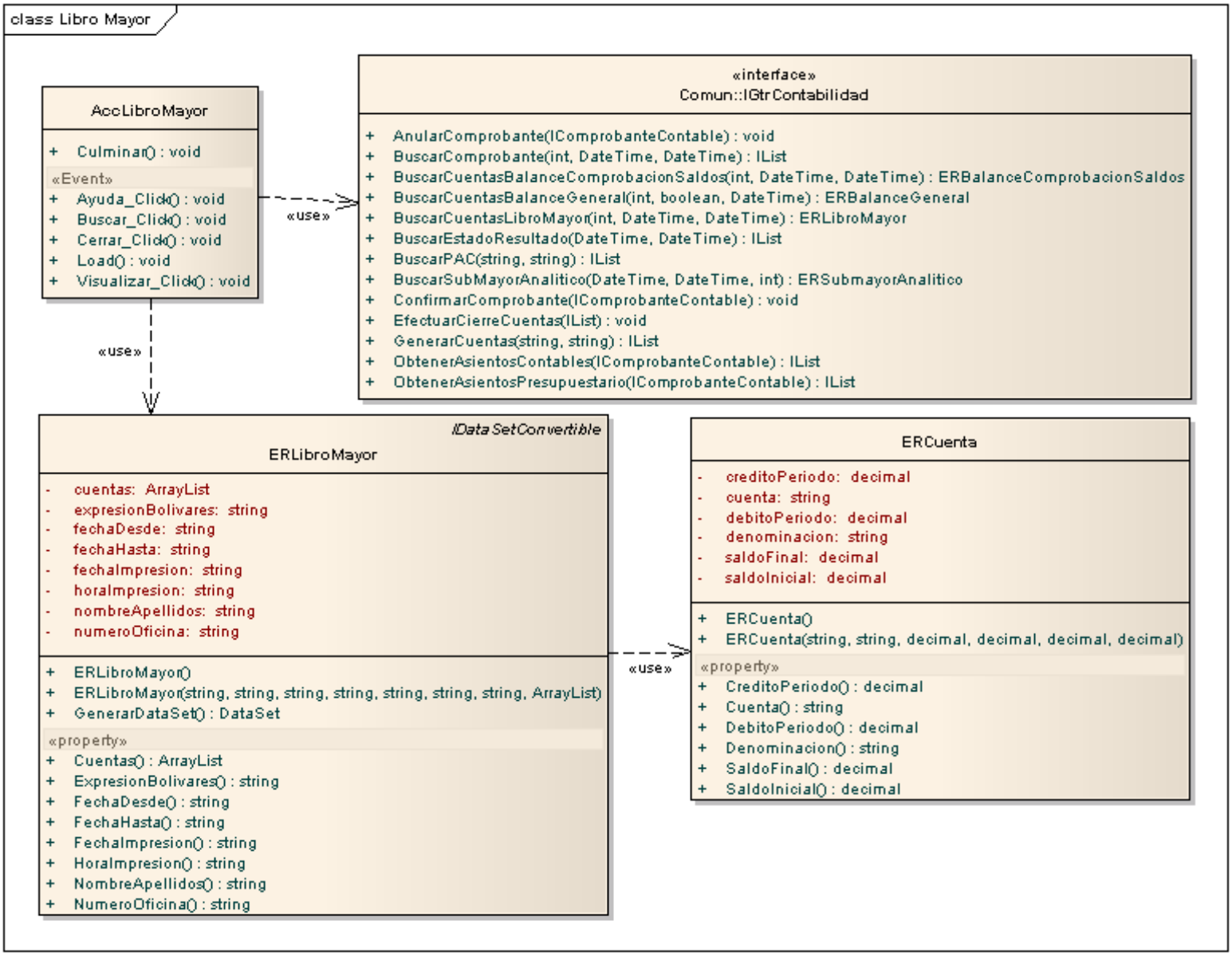
Documento Línea Base de la Arquitectura, se encuentra en el repositorio del proyecto Registros y Notarias de la Facultad 3.

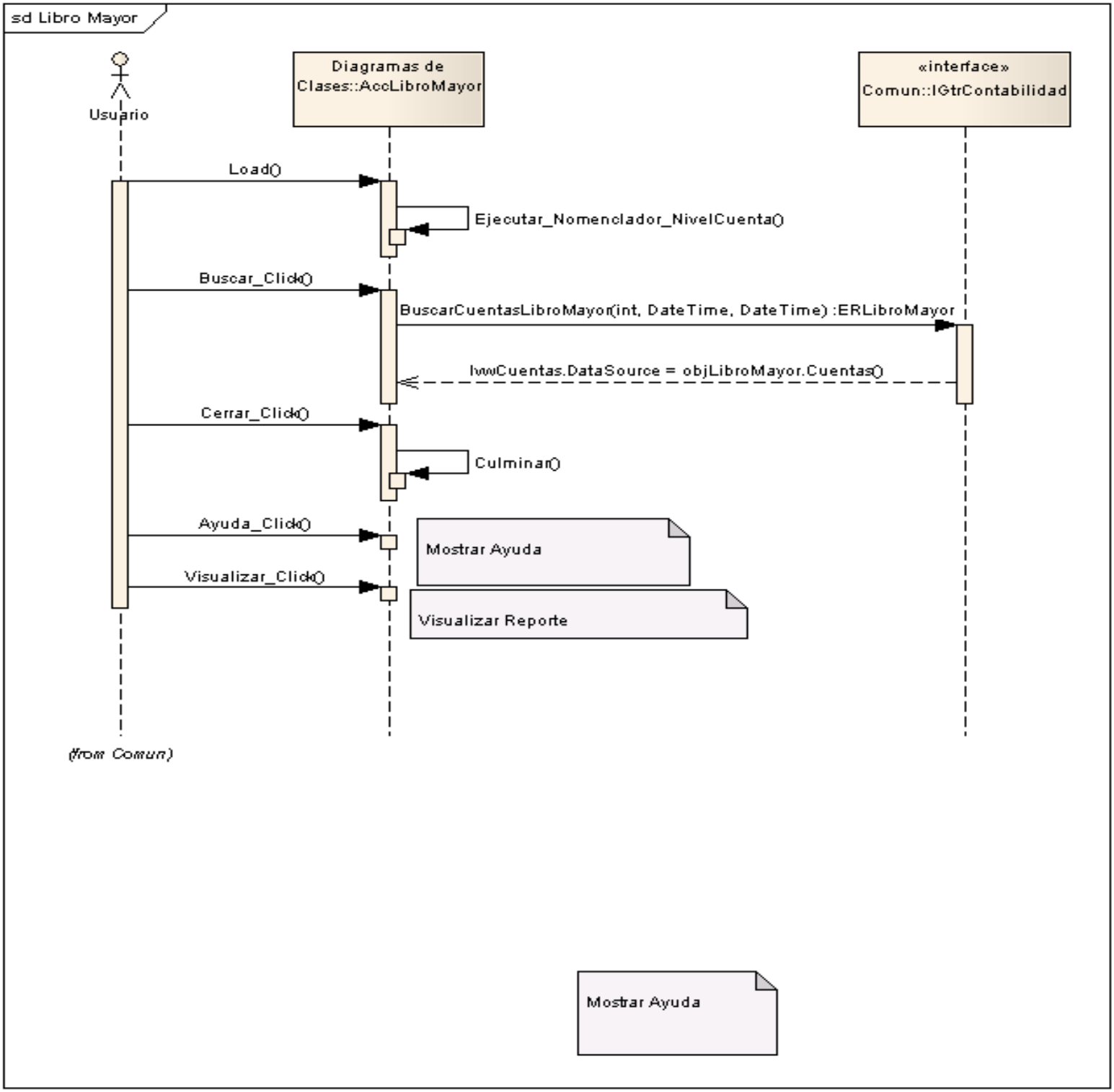
Anexo2:

Documento Estándares de codificación, se encuentra en el repositorio del proyecto Registros y Notarias de la Facultad 3.

Anexo 3:

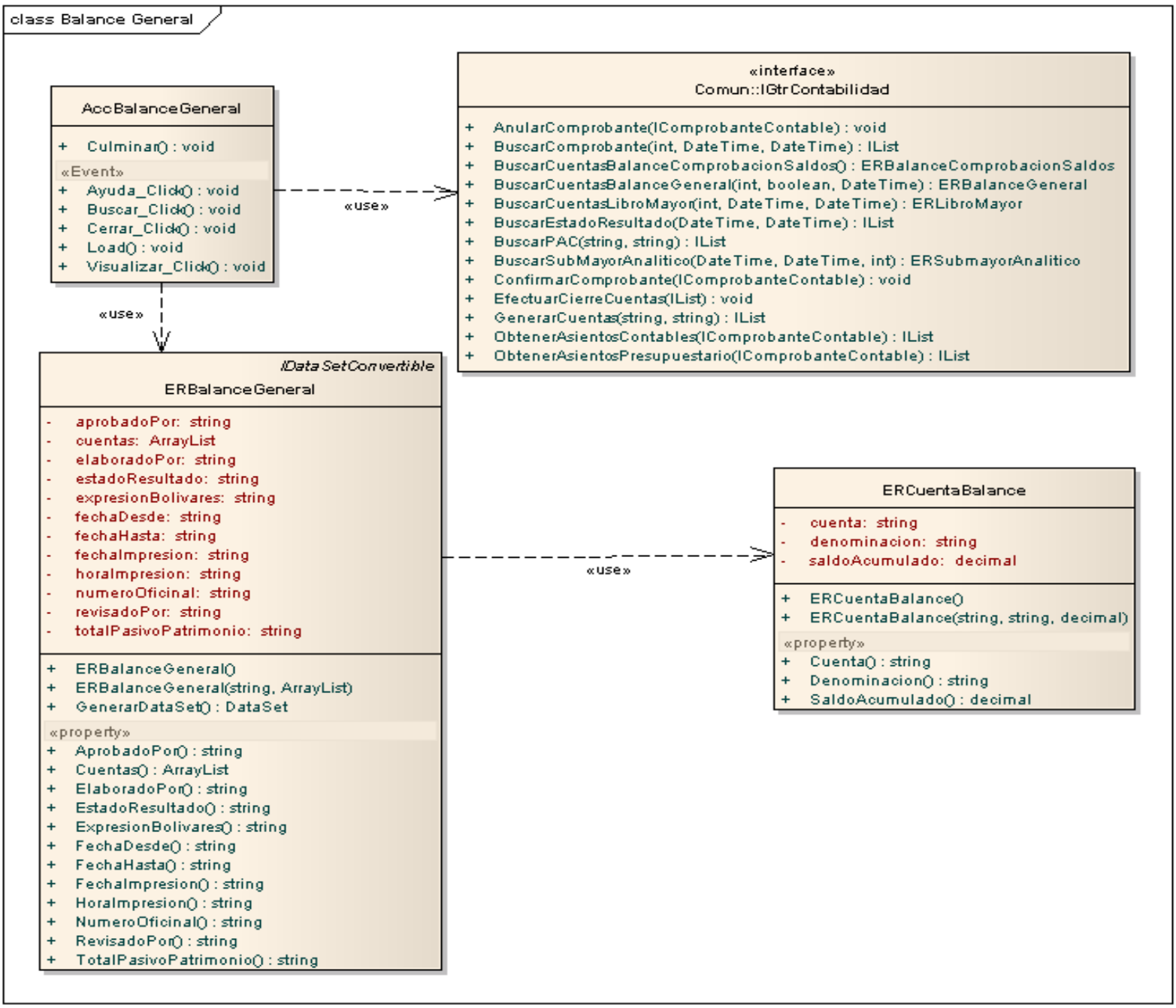
- **Caso de uso Reporte Libro Mayor.**

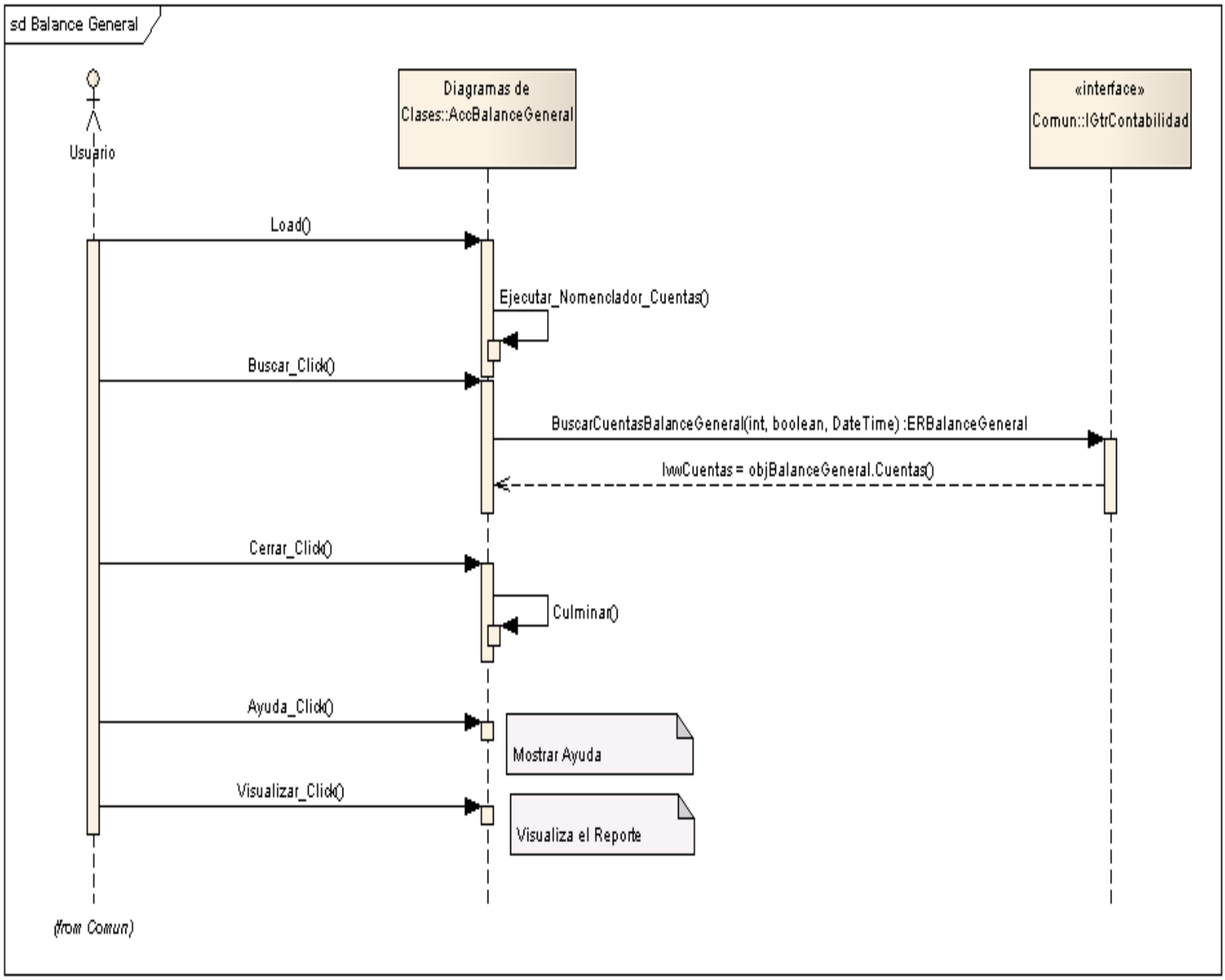



sd Libro Mayor
Usuario
Diagramas de Clases::AccLibroMayor
«interface» Comun::IGtrContabilidad
Load()
Ejecutar_Nomenclador_NivelCuenta()
Buscar_Click()
BuscarCuentasLibroMayor(int, DateTime, DateTime) :ERLibroMayor
lwwCuentas.DataSource = objLibroMayor.Cuentas()
Cerrar_Click()
Culminar()
Ayuda_Click()
Mostrar Ayuda
Visualizar_Click()
Visualizar Reporte
(from Comun)
Mostrar Ayuda

Anexo4:

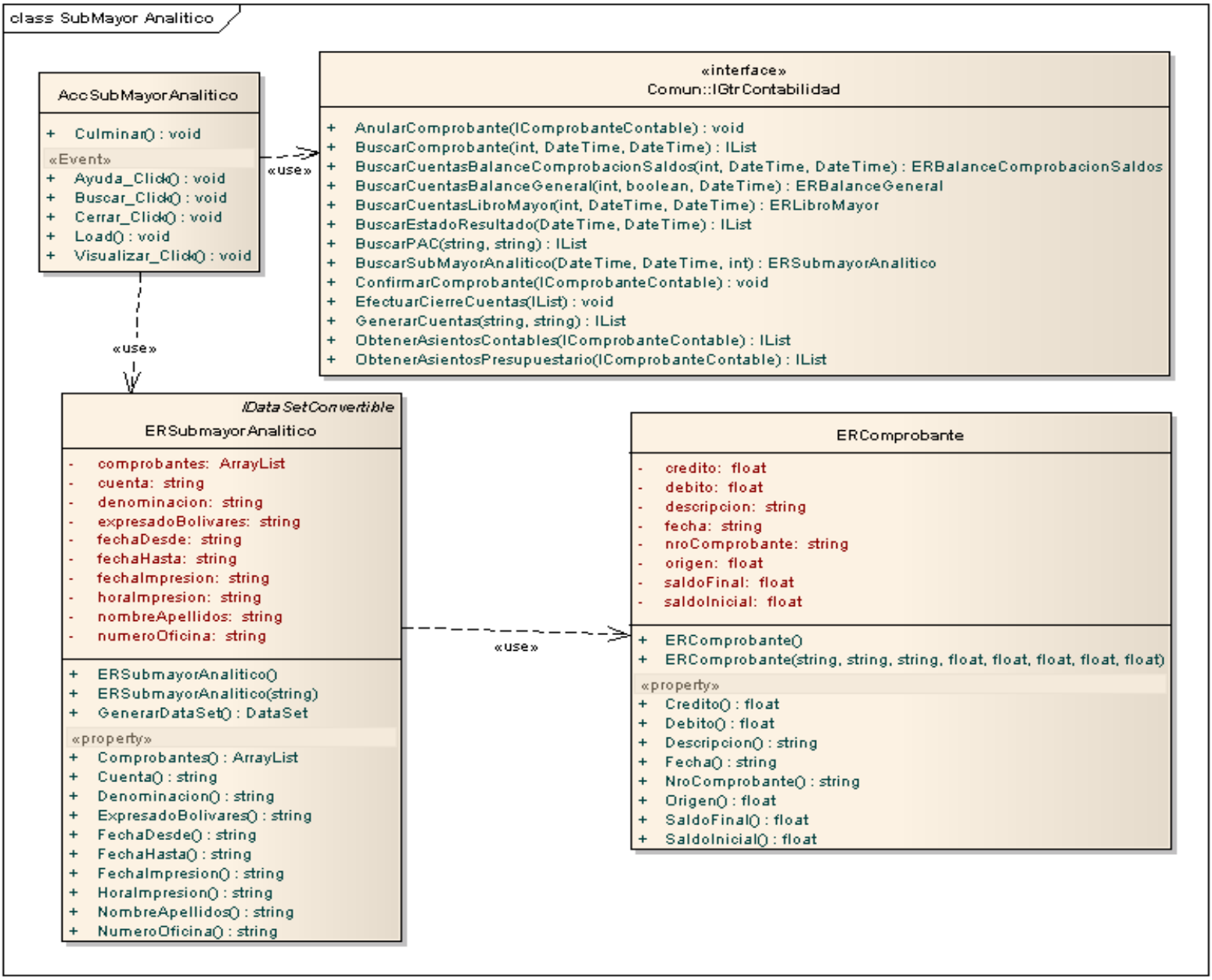
- **Caso de uso Reporte Balance General.**

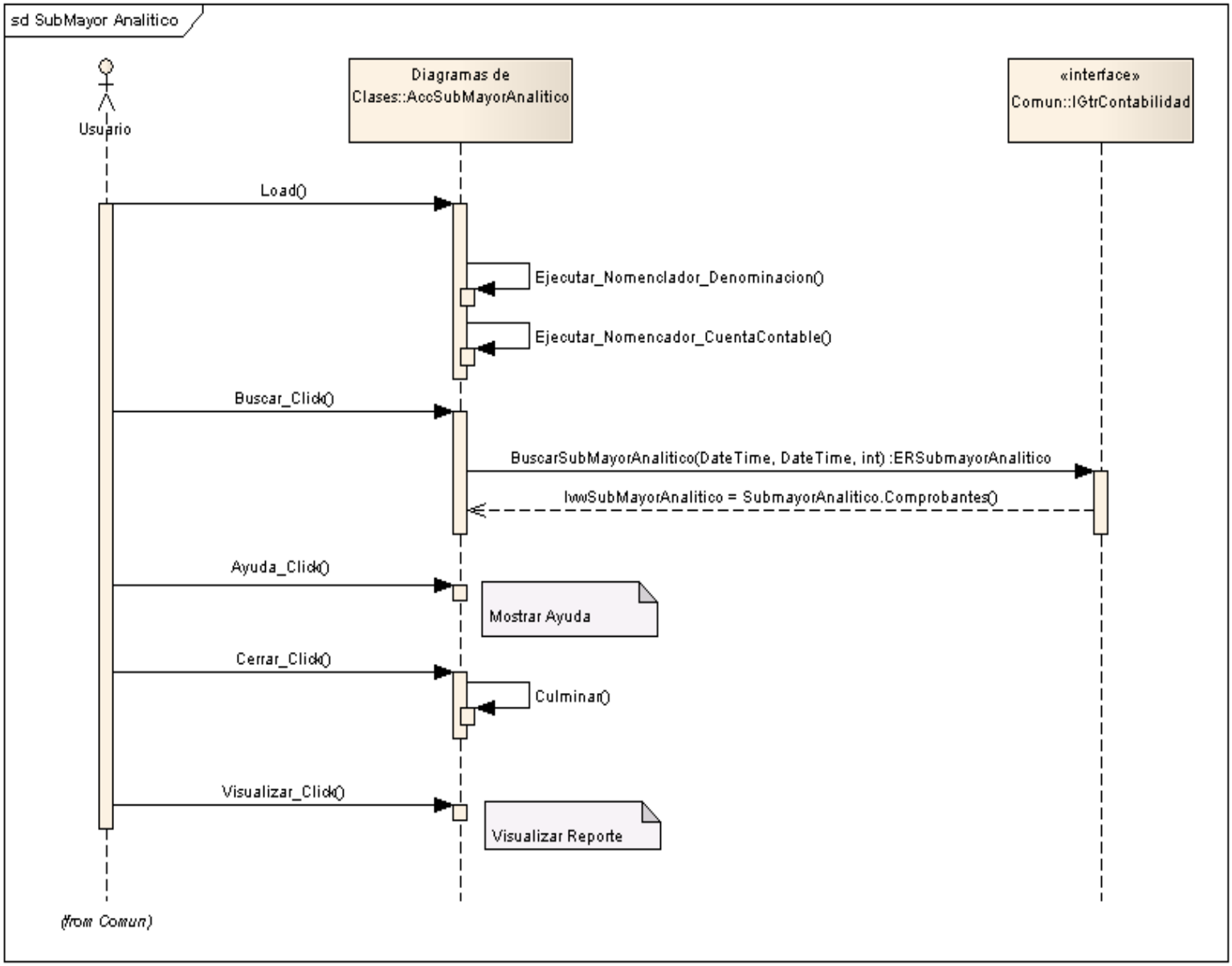



sd Balance General
Usuario
Diagramas de Clases::AccBalanceGeneral
«interface» Comun::IGtrContabilidad
Load()
Ejecutar_Nomenclador_Cuentas()
Buscar_Click()
BuscarCuentasBalanceGeneral(int, boolean, DateTime) :ERBalanceGeneral
lvwCuentas = objBalanceGeneral.Cuentas()
Cerrar_Click()
Culminar()
Ayuda_Click()
Mostrar Ayuda
Visualizar_Click()
Visualiza el Reporte
(from Comun)

Anexo5:

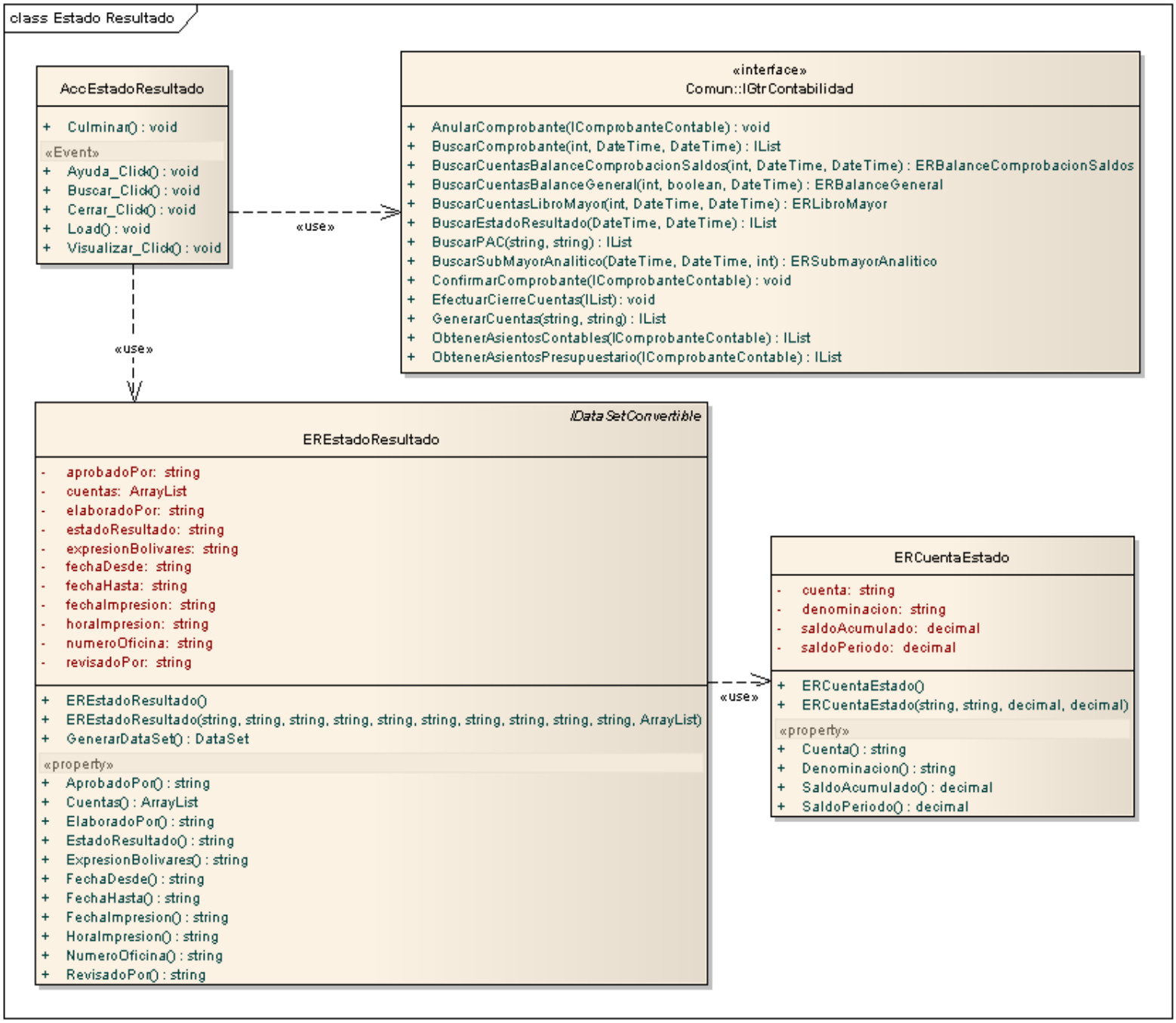
- **Caso de uso Reporte Sub-Mayor Analítico.**

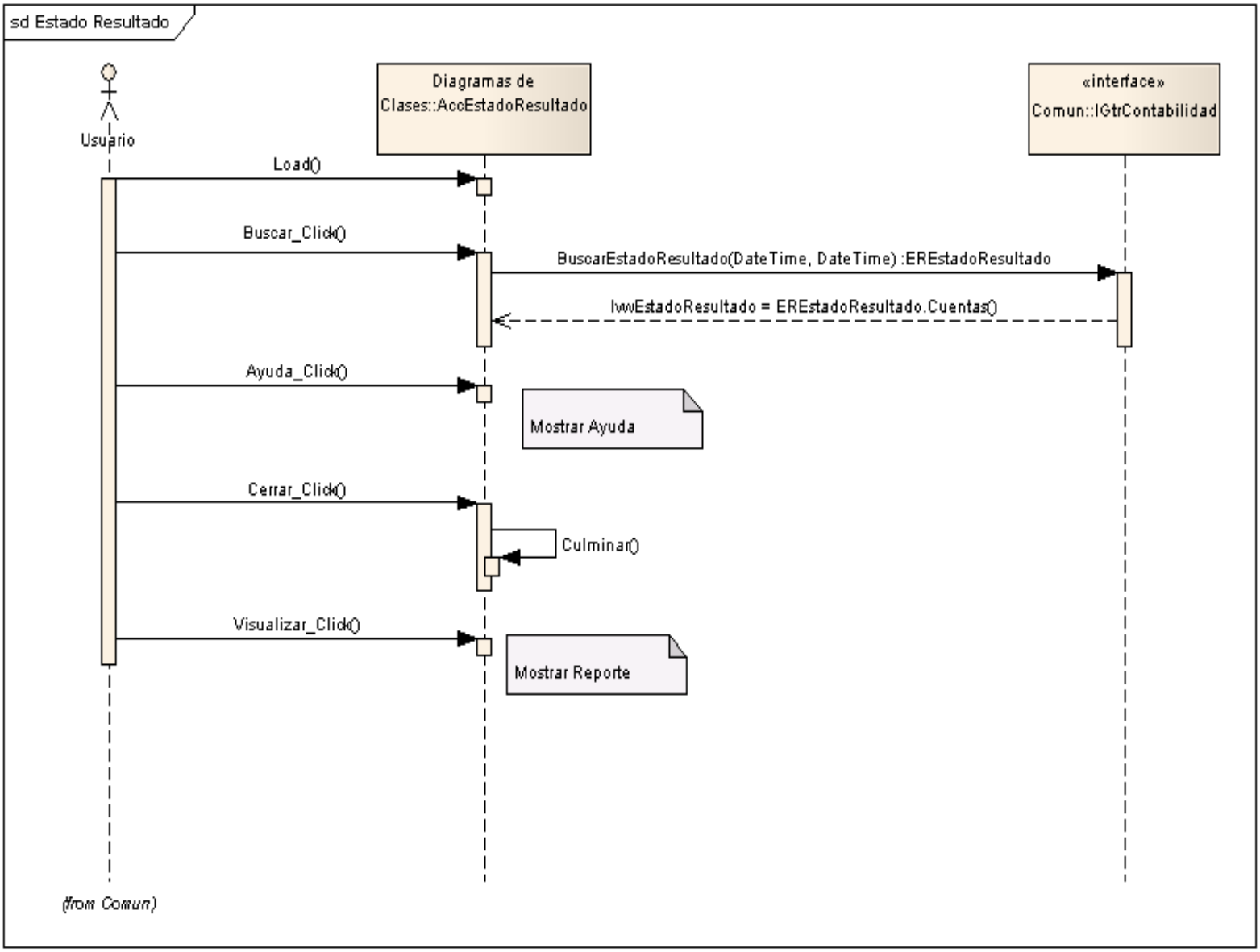



sd SubMayor Analitico
Usuario
Diagramas de Clases::AccSubMayorAnalitico
«interface» Comun::IGtrContabilidad
Load()
Ejecutar_Nomenclador_Denominacion()
Ejecutar_Nomencador_CuentaContable()
Buscar_Click()
BuscarSubMayorAnalitico(DateTime, DateTime, int) :ERSubmayorAnalitico
lvwSubMayorAnalitico = SubmayorAnalitico.Comprobantes()
Ayuda_Click()
Mostrar Ayuda
Cerrar_Click()
Culminar()
Visualizar_Click()
Visualizar Reporte
(from Comun)

Anexo6:

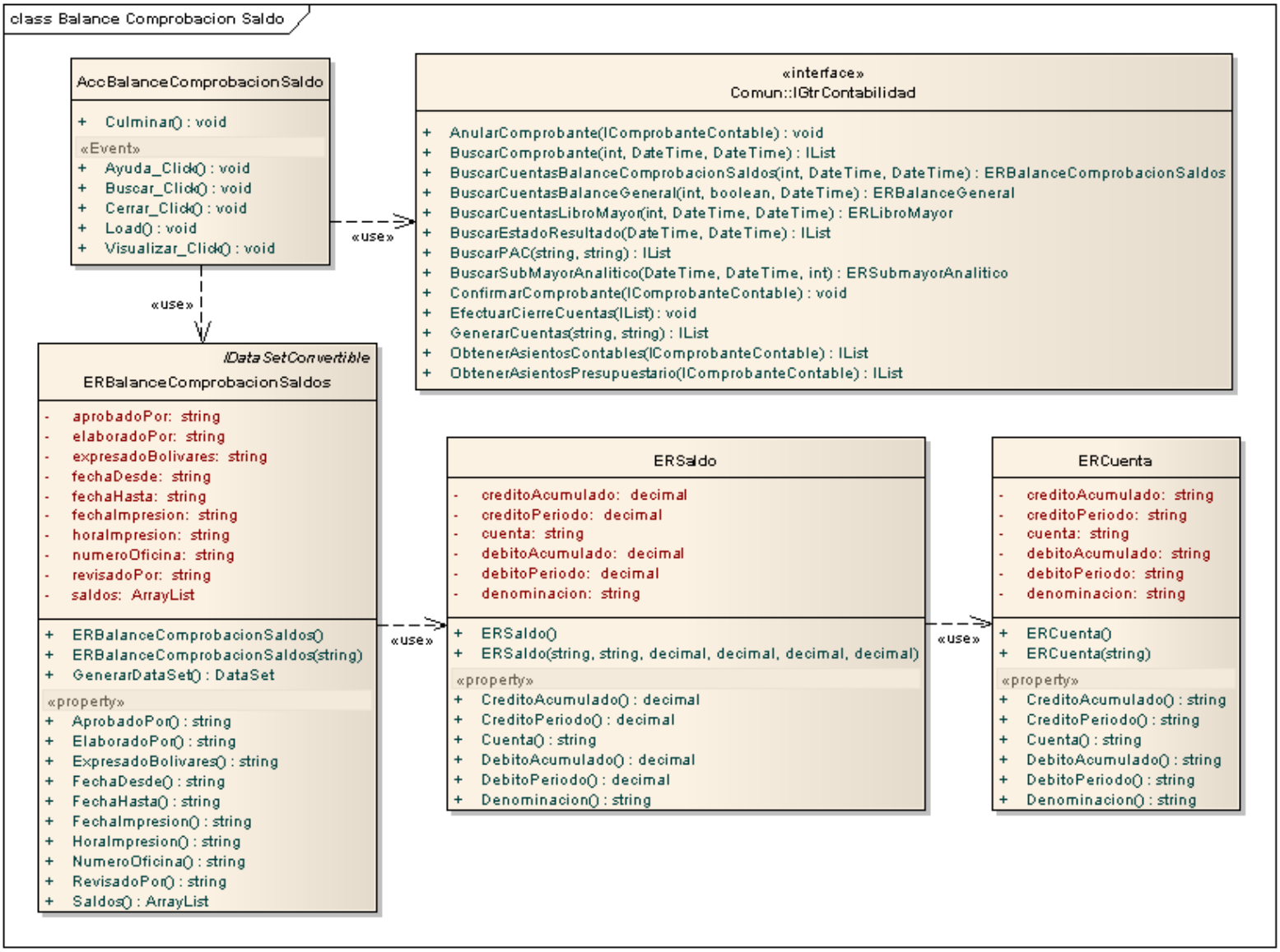
- **Caso de uso Reporte Estado de Resultado.**

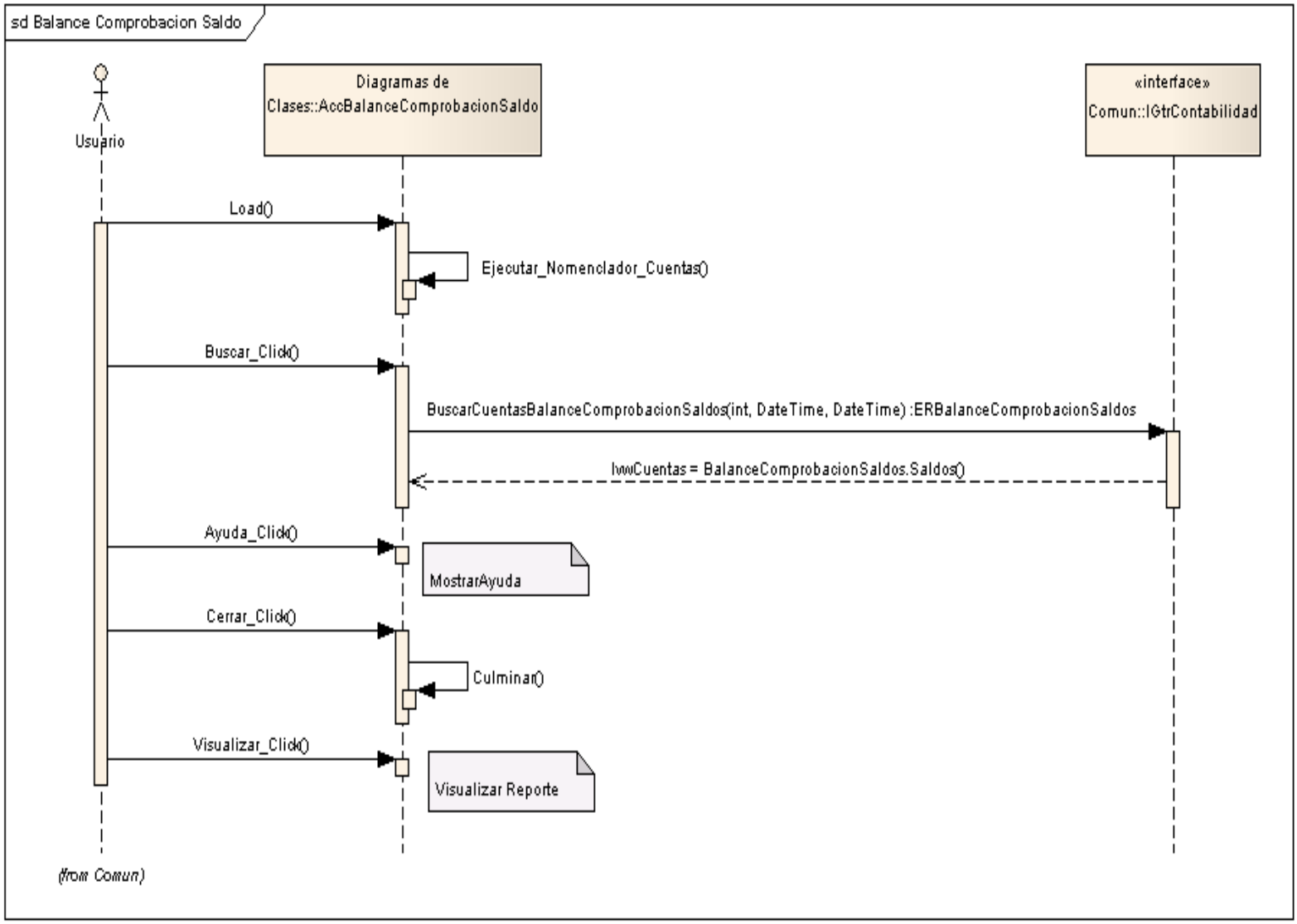



sd Estado Resultado
Usuario
Diagramas de Clases::AccEstadoResultado
«interface» Comun::IGtrContabilidad
Load()
Buscar_Click()
BuscarEstadoResultado(DateTime, DateTime) :EREstadoResultado
lvwEstadoResultado = EREstadoResultado.Cuentas()
Ayuda_Click()
Mostrar Ayuda
Cerrar_Click()
Culminar()
Visualizar_Click()
Mostrar Reporte
(from Comun)

Anexo7:

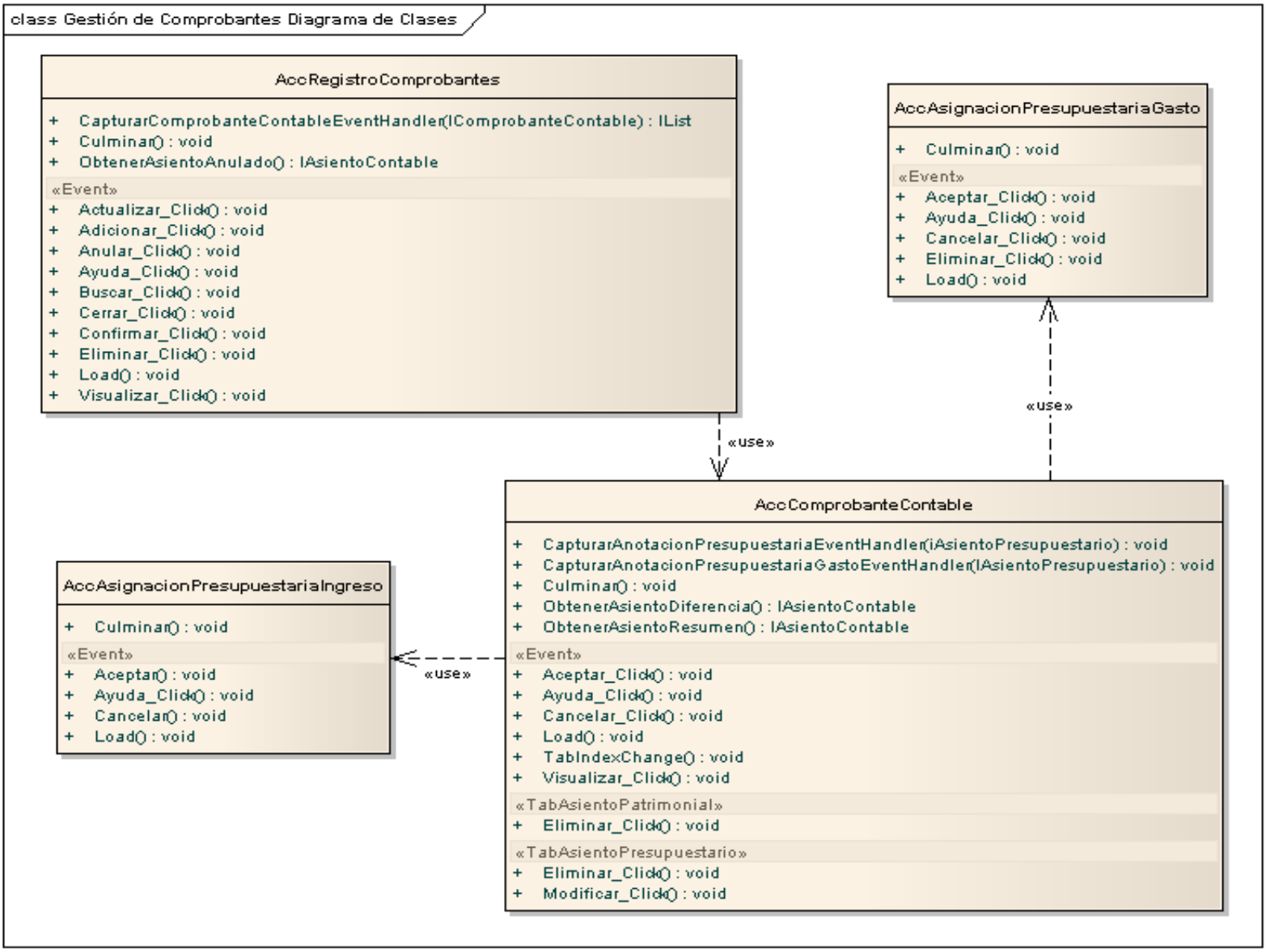
- **Caso de uso Reporte Balance de Comprobación de Saldo.**



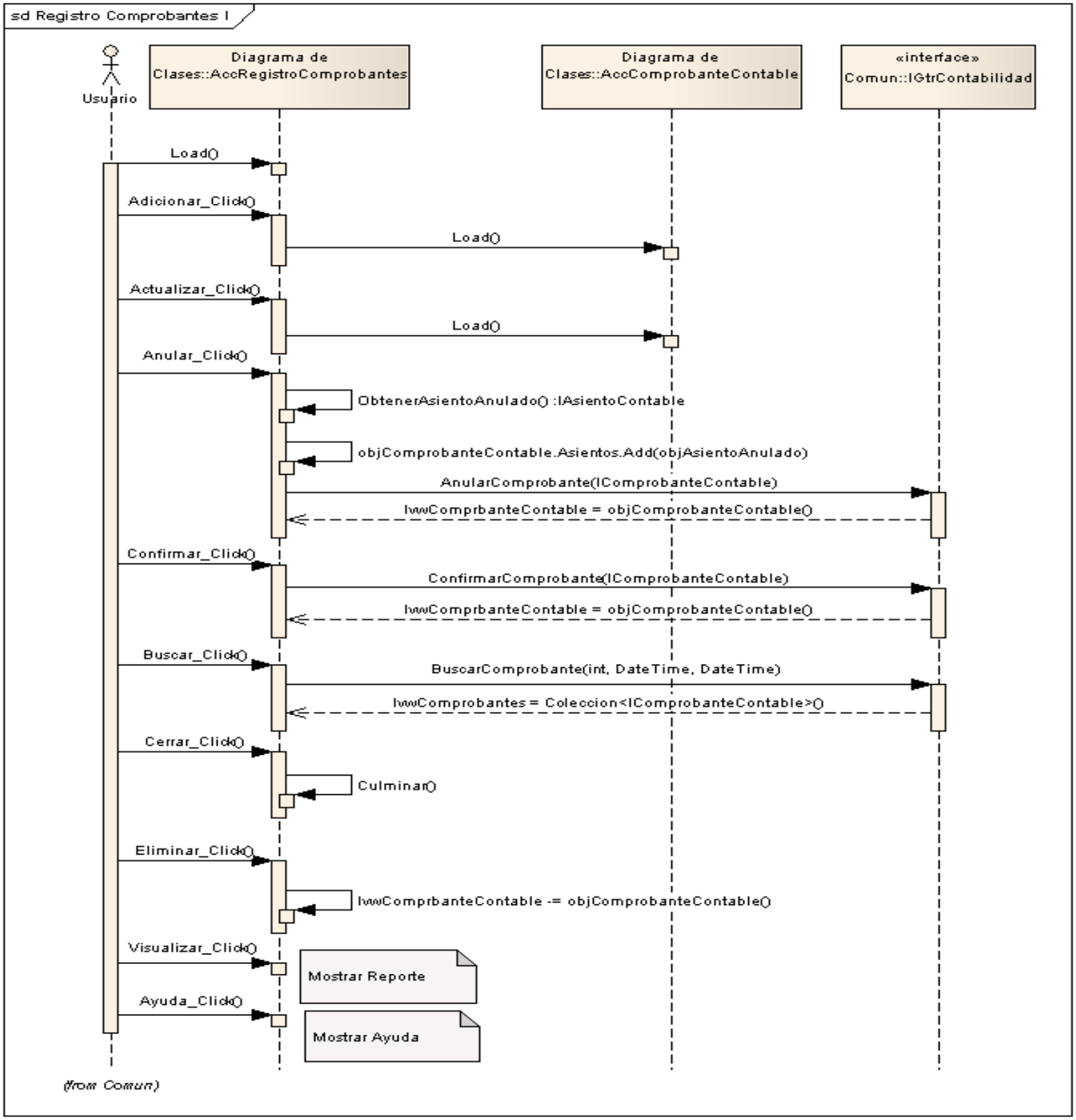

sd Balance Comprobacion Saldo
Usuario
Diagramas de Clases::AccBalanceComprobacionSaldo
«interface» Comun::IGtrContabilidad
Load()
Ejecutar_Nomenclador_Cuentas()
Buscar_Click()
BuscarCuentasBalanceComprobacionSaldos(int, DateTime, DateTime) :ERBalanceComprobacionSaldos
lvwCuentas = BalanceComprobacionSaldos.Saldos()
Ayuda_Click()
MostrarAyuda
Cerrar_Click()
Culminar()
Visualizar_Click()
Visualizar Reporte
(from Comun)

Anexo8:

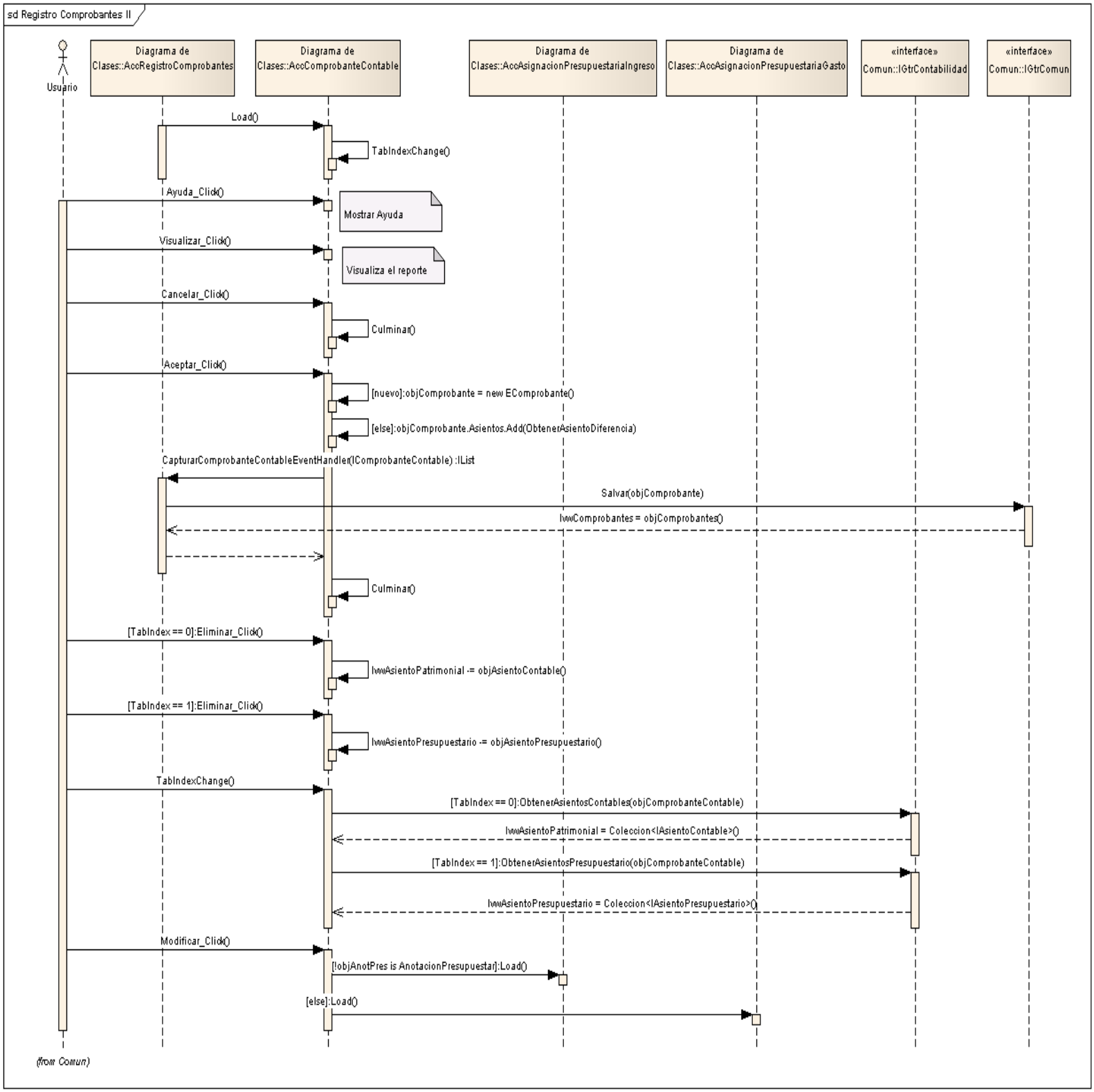
- **Caso de uso Gestionar Registro de Comprobantes.**



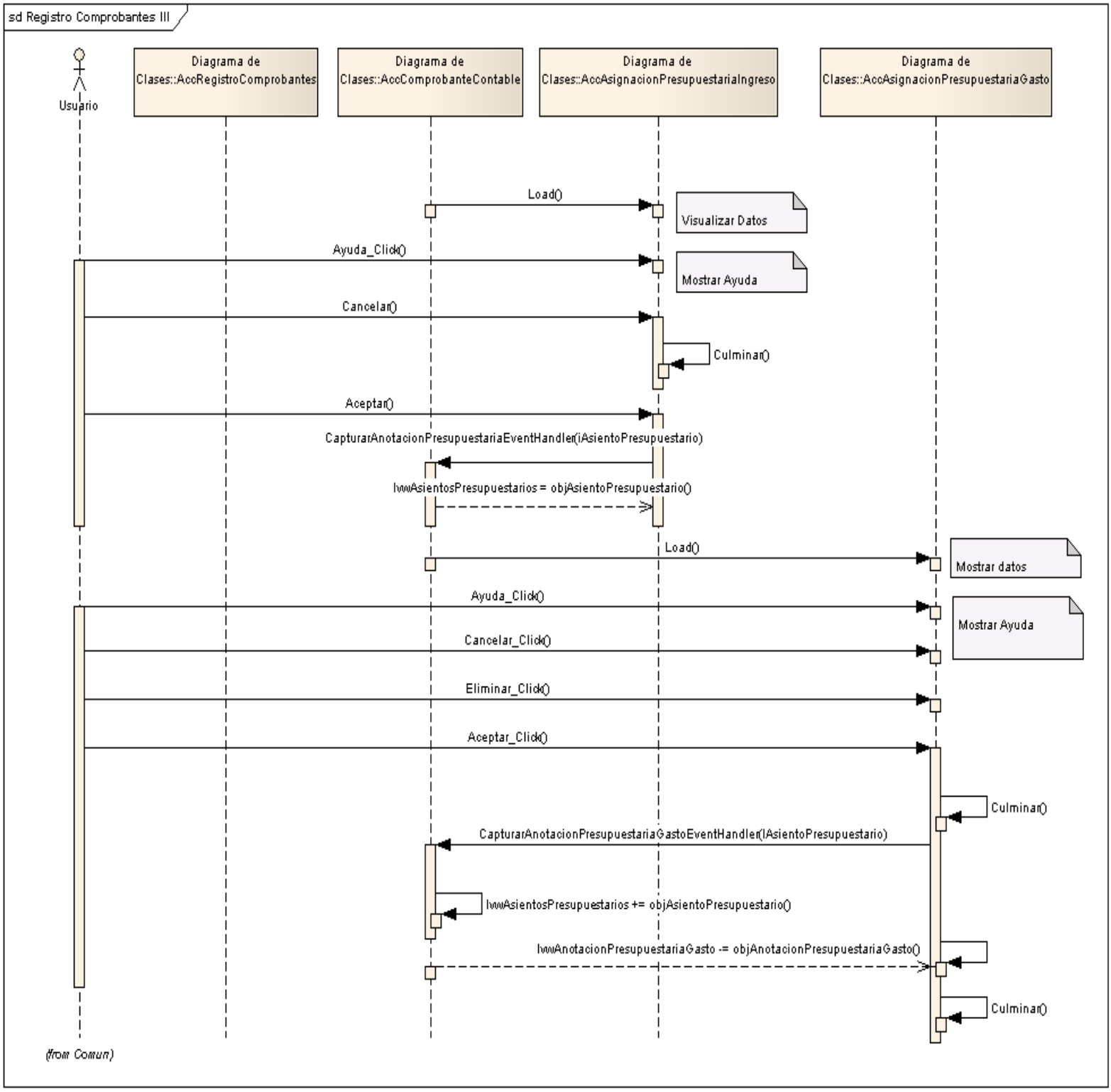
(Parte 1).


sd Registro Comprobantes I
Usuario
Diagrama de Clases::AccRegistroComprobantes
Diagrama de Clases::AccComprobanteContable
«interface» Comun::IGtrContabilidad
Load()
Adicionar_Click()
Load()
Actualizar_Click()
Load()
Anular_Click()
ObtenerAsientoAnulado() :IAsientoContable
objComprobanteContable.Asientos.Add(objAsientoAnulado)
AnularComprobante(IComprobanteContable)
IvwComprbanteContable = objComprobanteContable()
Confirmar_Click()
ConfirmarComprobante(IComprobanteContable)
IvwComprbanteContable = objComprobanteContable()
Buscar_Click()
BuscarComprobante(int, DateTime, DateTime)
IvwComprobantes = Coleccion<IComprobanteContable>()
Cerrar_Click()
Culminar()
Eliminar_Click()
IvwComprbanteContable -= objComprobanteContable()
Visualizar_Click()
Mostrar Reporte
Ayuda_Click()
Mostrar Ayuda
(from Comun)

(Parte 2)

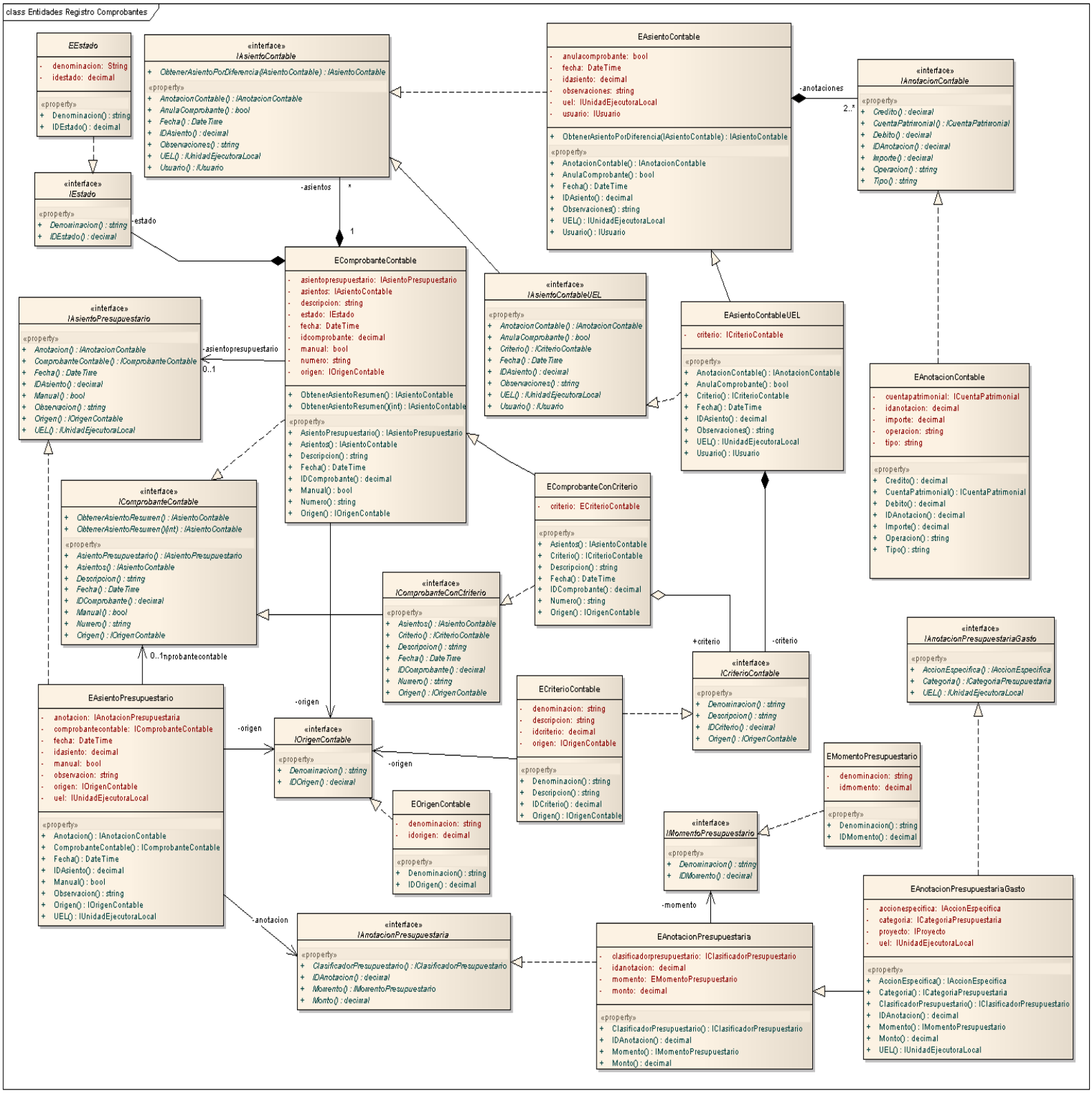

sd Registro Comprobantes II
Usuario
Diagrama de Clases::AccRegistroComprobantes
Diagrama de Clases::AccComprobanteContable
Diagrama de Clases::AccAsignacionPresupuestariaIngreso
Diagrama de Clases::AccAsignacionPresupuestariaGasto
«interface» Comun::IGtrContabilidad
«interface» Comun::IGtrComun
Load()
TabIndexChange()
I Ayuda_Click()
Mostrar Ayuda
Visualizar_Click()
Visualiza el reporte
Cancelar_Click()
Culminar()
Aceptar_Click()
[nuevo]:objComprobante = new EComprobante()
[else]:objComprobante.Asientos.Add(ObtenerAsientoDiferencia)
CapturarComprobanteContableEventHandler(IComprobanteContable) :IList
Salvar(objComprobante)
IwwComprobantes = objComprobantes()
Culminar()
[TabIndex == 0]:Eliminar_Click()
IwwAsientoPatrimonial -= objAsientoContable()
[TabIndex == 1]:Eliminar_Click()
IwwAsientoPresupuestario -= objAsientoPresupuestario()
TabIndexChange()
[TabIndex == 0]:ObtenerAsientosContables(objComprobanteContable)
IwwAsientoPatrimonial = Coleccion<IAsientoContable>()
[TabIndex == 1]:ObtenerAsientosPresupuestario(objComprobanteContable)
IwwAsientoPresupuestario = Coleccion<IAsientoPresupuestario>()
Modificar_Click()
[!objAnotPres is AnotacionPresupuestar]:Load()
[else]:Load()
(from Comun)

(Parte 3).


sd Registro Comprobantes III
Usuario
Diagrama de Clases::AccRegistroComprobantes
Diagrama de Clases::AccComprobanteContable
Diagrama de Clases::AccAsignacionPresupuestariaIngreso
Diagrama de Clases::AccAsignacionPresupuestariaGasto
Load()
Visualizar Datos
Ayuda_Click()
Mostrar Ayuda
Cancelar()
Culminar()
Aceptar()
CapturarAnotacionPresupuestariaEventHandler(iAsientoPresupuestario)
lwwAsientosPresupuestarios = objAsientoPresupuestario()
Load()
Mostrar datos
Ayuda_Click()
Mostrar Ayuda
Cancelar_Click()
Eliminar_Click()
Aceptar_Click()
Culminar()
CapturarAnotacionPresupuestariaGastoEventHandler(IAsientoPresupuestario)
lwwAsientosPresupuestarios += objAsientoPresupuestario()
lwwAnotacionPresupuestariaGasto -= objAnotacionPresupuestariaGasto()
Culminar()
(from Comun)

Anexo9:

- **Caso de uso registro de comprobantes.(Entidades)**



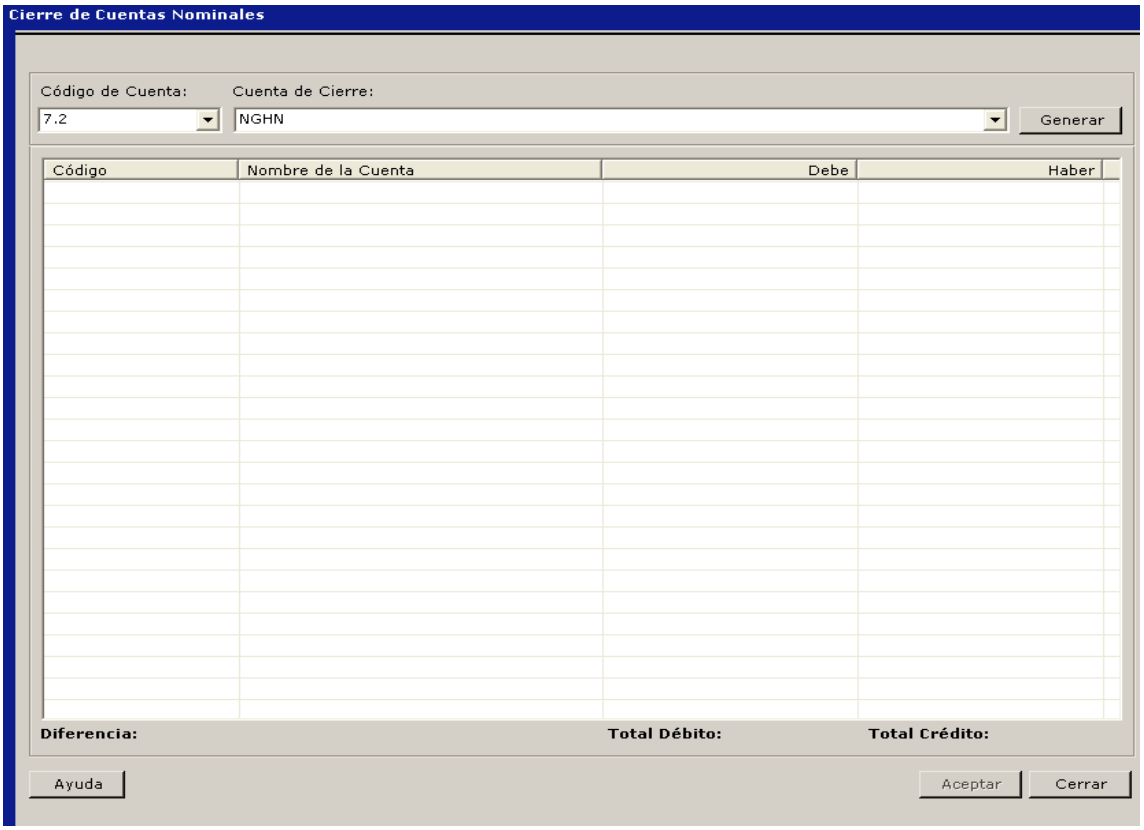
Anexo10:

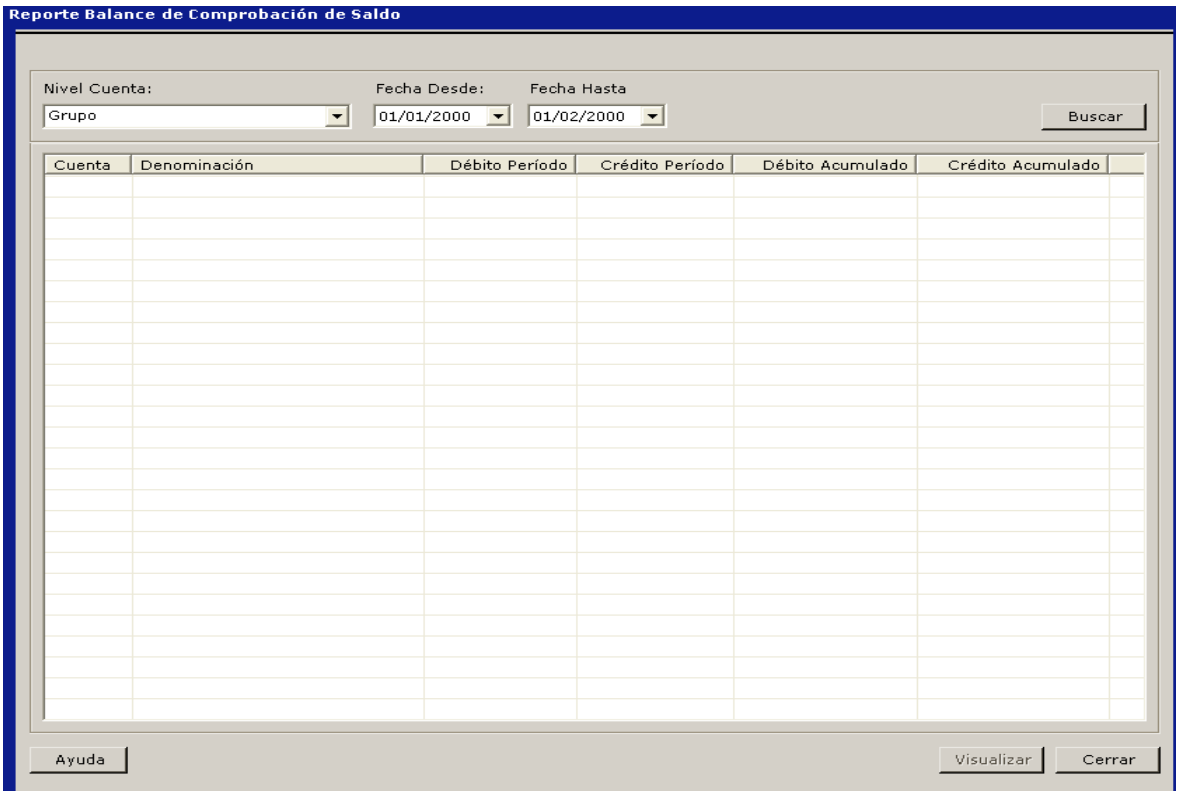
No	Clase	Total de atributos	Total de operaciones	Tamaño
1	EAnotacionContable	6	1	Pequeño
2	EAnotacionPresupuestaria	5	0	Pequeño
3	EAnotacionPresupuestariaGasto	3	0	Pequeño
4	EAsientoContable	1	0	Pequeño
5	EAsientoContableUEL	1	0	Pequeño
6	EAsientoPatrimonial	7	0	Pequeño
7	EAsientoPresupuestario	8	0	Pequeño
8	EComprobanteConCtriterio	1	2	Pequeño
9	EComprobanteContable	11	0	Pequeño
10	ECriterioContable	4	0	Pequeño
11	ECuentaNominal	5	0	Pequeño
12	EEstadoComprobanteContable	2	0	Pequeño
13	EMomentoPresupuestario	2	0	Pequeño
14	EOrigenContable	2	0	Pequeño
15	Fabrica	1	40	Medio
16	GtrContabilidad	3	21	Pequeño
17	ERArbolCuenta	2	7	Pequeño
18	ERBalanceComprobacionSaldos	14	1	Pequeño
19	ERSaldo	8	1	Pequeño
20	ERBalanceGeneral	12	1	Pequeño
21	ERCuentaBalance	3	0	Pequeño
22	ERComportamiento	11	0	Pequeño
23	ERComportamientoIngreso	9	1	Pequeño
24	ERComprobanteContable	13	1	Pequeño
25	ERCuentaPatrimonial	4	0	Pequeño
26	ERCuentaPresupuestaria	3	0	Pequeño

No	Clase	Total de atributos	Total de operaciones	Tamaño
27	ERCuentaEstado	4	0	Pequeño
28	EREstadoResultado	11	1	Pequeño
29	ERCuenta	6	0	Pequeño
30	ERLibroMayor	10	1	Pequeño
31	EREstatus	5	0	Pequeño
32	ERRegistroComprobantes	6	1	Pequeño
33	ERComprobante	8	0	Pequeño
34	ERSubmayorAnalitico	10	1	Pequeño
35	GtrReportes	1	9	Pequeño
36	AccCierreCuentaNominales	2	9	Pequeño
37	AccAsignaccionPresupuestariaIngreso	3	5	Pequeño
38	AccAsignacionPresupuestariaGasto	8	21	Pequeño
39	AccComprobanteContable	12	40	Medio
40	AccRegistroComprobante	1	17	Pequeño
41	AccBalanceComprobacionSaldo	1	8	Pequeño
42	AccReporteEstadoResultado	2	8	Pequeño
43	AccLibroMayor	1	8	Pequeño
44	AccReporteSubmayorAnalitico	1	11	Pequeño
45	AccReporteComportamientoIngreso	0	1	Pequeño
46	AccBalanceGeneral	2	6	Pequeño
47	Total	225	222	

Tabla 3. Clases de la aplicación.

Anexo11:


Cierre de Cuentas Nominales
Código de Cuenta:
Cuenta de Cierre:
7.2
NGHN
Generar
Código
Nombre de la Cuenta
Debe
Haber
Diferencia:
Total Débito:
Total Crédito:
Ayuda
Aceptar
Cerrar


Reporte Balance de Comprobación de Saldo
Nivel Cuenta:
Fecha Desde:
Fecha Hasta
Grupo
01/01/2000
01/02/2000
Buscar
Cuenta
Denominación
Débito Período
Crédito Período
Débito Acumulado
Crédito Acumulado
Ayuda
Visualizar
Cerrar