Universidad de las Ciencias Informáticas

Facultad 3

Título: *Diseño e Implementación de la Solución Informática del proceso de Recaudación en la Administración Financiera de los Registros y Notarías de la República Bolivariana de Venezuela.*

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor(es): Alejandro Abiague Napoles

Maikel Yonelis García Batista

Tutores: Ing. Yaumarys Pino Cueto

Ing. Lourdes J. Perojo Martínez

La Habana, Cuba.

Mayo, 2008.

DECLARACIÓN DE AUTORÍA

Declaramos ser los únicos autores de este trabajo y autorizamos a la Facultad 3 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.
Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Autores:

_____ _____

Alejandro Abiague Napoles Maikel Yonelis García Batista

Tutores:

_____ _____

Ing. Lourdes Julia Perojo Martínez Ing. Yaumarys Pino Cueto

AGRADECIMIENTOS

A Lulú (Lourdes) y a Yaumi (Yaumarys), nuestras amiguísimas y tutoras por su gran dedicación, ayuda, apoyo incondicional, su esmero constante y preocupación, pues hicieron posible que este trabajo fuera confeccionado, guiándonos en todo momento hacia el objetivo final.

A nuestros amigos del proyecto Registros y Notarías, Julito, Jony, María, Rene, Yunieski, Henry, Yosvany, Anita e Iosmel por su apoyo incondicional y amistad, por acompañarnos en todos los momentos duros de producción.

A Yunier Perez amigo, hermano, profesor, compañero de estudio durante los 5 años, por su ayuda, apoyo, y preocupación constante sobre este trabajo, por su compañía en todo momento, por las cosas que nos ha enseñado y hemos aprendido juntos.

Al por siempre nuestro profesor Oscar Camacho, el amigo que nos encaminó desde temprano en la carrera, el que siempre nos aconsejó para tomar el buen camino, el que siempre cargó con nosotros, y del que aprendimos muchas cosas importantes para la vida.

A nuestra Universidad (Universidad de las Ciencias Informáticas) por permitirnos crecer profesionalmente, por todo lo que hemos aprendido en su estancia, a nuestro comandante en Jefe por sus brillantes ideas que permitieron su creación, y por darnos la oportunidad de cursar nuestros estudios superiores con condiciones de estudio inigualables en la Universidad del Futuro.

Agradecemos inmensamente a todas las personas, conocidos, amigos y profesores, que de alguna manera se preocuparon por este trabajo y aportaron su granito de arena a la confección del mismo.

A mi familia por darme en todo momento su apoyo incondicional durante mi vida, mi carrera, y la confección de este trabajo.

DEDICATORIA

A mis padres y hermanito del alma que son mi mayor inspiración, lo que más quiero en esta vida, sin su apoyo no hubiese podido llegar hasta aquí.

A mi abuelita Esther que siempre está conmigo, y a mi tío Wicho les dedico este trabajo con todo el amor del mundo.

A mi familia enterita de Cueto, mi primaso Luis Alberto, mi tía Yoyi, mi tío Badía, mi tía Tita, mi tío Chogi, mi primita Leticia, y mi primo Sergito.

A las dos cositas que no tienen noción de este trabajo, mis sobrinitos, Akilito y Jose, que los adoro de todo corazón.

En especial a mis mejores amigos Yunier, Maikel, Yami, Yuli, Susy, Dania, Mari, Enriquito de Cuento y Lilibet Fonseca mi eterna amiga.

A mi angelito del alma, Blanquita, le dedico este trabajo tan importante para mí.

Alejandro Abiague Napoles

DEDICATORIA

Especialmente a mi abuelita (Mami Fimo) que aunque no pudo llegar a verme graduado se la dedico con todo el corazón, ya que gracias a su ayuda y apoyo pude llegar a ser quien soy en estos momentos.

A Nene e Irma mis padres queridos que me apoyaron en todo desde el primer momento, quienes supieron guiarme por un buen camino en todo instante de la vida.

A mi hermanito Luis Ángel que lo quiero muchísimo y le deseo lo mejor del mundo.

A mis abuelos Papi Lupe, Mami Irene, Papa Angelio por portarse tan bien conmigo y siempre dar lo mejor de sí para que yo siguiera adelante.

A mi tía Lise, tío Kiri, tío Fernando, tía Paula, tío Fermín que los adoro muchísimo.

A mis primos y primas del alma Yoan, Yumi, Yuni, Yurimita y a los pequeñines que también quiero cantidad.

A Yunier y Alejandro a quienes quiero y los considero como mis hermanos.

A mis amigos y amigas Yosvani, Ricardo, Aldo, Yoly, Yari, Dainerys, Yumi, Yaimara, Maite, Yare, Lili a quienes aprecio muchísimo.

Maikel Yonelis García Batista.

RESUMEN

La Dirección General de Registros y Notarías de la República Bolivariana de Venezuela, es la entidad encargada de ordenar los procedimientos registrales y notariales en cuanto a la celeridad jurídica.

Uno de los propósitos fundamentales de la Dirección General de Registros y Notarías con su nueva estructura administrativa financiera es la centralización y control de dicha actividad a nivel nacional. Las Unidades Organizativas que en el sector público integran esta estructura, son responsables de programar y evaluar el presupuesto, administrar el sistema de recaudación tributaria y administrar el tesoro.

Como actividad clave en la Administración Financiera el proceso de Recaudación se encarga de la ejecución, supervisión, registro y control de los ingresos que se originan por conceptos de tasa por servicios registrales y notariales establecidos en la Ley de Registro Público y del Notariado, generados a escala nacional. Actualmente en el proceso de Recaudación ocurren algunas deficiencias.

Basado en el estudio de algunos estilos arquitectónicos, patrones de arquitectura y diseño, el presente trabajo pretende proponer una solución de diseño de software e implementación del módulo de Recaudación comprendido en el sistema Administración Financiera del Producto SAREN[1]

La solución en cuestión se integra a los demás módulos comprendidos en el sistema de Administración Financiera, de esta manera se permitirá centralizar la recaudación a nivel nacional, así como facilitar además a la Dirección General de Registros y Notarías la obtención de las metas trazadas con la automatización de la actividad Administrativa Financiera a nivel nacional.

[1] SAREN: Servicio Autónomo de Registros y Notarías.

ÍNDICE

INTRODUCCIÓN

La Dirección General de Registros y Notarías de la República Bolivariana de Venezuela, es la entidad encargada de ordenar los procedimientos registrales y notariales en cuanto a la celeridad jurídica, así como de efectuar inspecciones ordinarias y extraordinarias en los Registros y Notarías.

Actualmente, funcionan en todo el territorio venezolano 483 oficinas, adscritas a la Dirección General de Registros y Notarías, laboran en las mismas aproximadamente 590 personas en calidad de contrata o supernumeraria y 6160 Funcionarios pertenecientes al Ministerio del Poder Popular para Relaciones Interiores y Justicia. Este conjunto de oficinas se encuentran clasificadas según sus funciones en diferentes grupos: 21 Registros Civiles, 47 Mercantiles, 207 Públicos y 208 Notarías, llegando a abarcar el 98% de las transacciones por la gran demanda de sus servicios, los Registros Mercantiles y los Registros Públicos.

Una de las actividades fundamentales de esta dirección es la Administración Financiera de estas oficinas. Se entiende por Sistema de Administración Financiera Integrada el conjunto de leyes, normas y procedimientos destinados a la obtención, asignación, uso, registro y evaluación de los recursos financieros del Estado, que tiene como propósito la eficiencia de la gestión de los mismos para satisfacer necesidades colectivas.

Un modelo de Administración Financiera está integrado cuando las Unidades Organizativas y Entes que lo conforman actúan en forma absolutamente interrelacionada bajo la dirección de un único órgano coordinador que debe tener la suficiente competencia para reglamentar el funcionamiento de ésta, y cuando el conjunto de principios, normas y procedimientos que están vigentes en el sistema son coherentes entre sí y permiten interrelacionar automáticamente sus actividades.

Las Unidades Organizativas que en el sector público integran la Administración Financiera, son los responsables de programar y evaluar el presupuesto, administrar el sistema de recaudación tributaria y aduanera, gestionar las operaciones de crédito público, programar la ejecución del presupuesto de gasto, administrar el tesoro y contabilizar, tanto física como financieramente, las transacciones relacionadas con la captación y colocación de los fondos públicos. Los recursos humanos, materiales y financieros que demanden el funcionamiento de estas unidades, forman parte de la Administración Financiera.

El Ministerio del Poder Popular para la Relaciones Interiores y Justicia de la República Bolivariana de Venezuela no posee un control centralizado de la Administración Financiera de los Registros y Notarías, los cuales funcionan con autonomía en los servicios que brindan, es decir, funcionan independientemente cada uno. El organismo constitucional ha determinado crear un conjunto de acciones encaminadas a garantizar un Sistema de Administración Financiera integrada para todos los órganos que conforman el gobierno general, el cual contempla los siguientes módulos:

- Presupuesto.
- Requisiciones.
- Tesorería.
- Contabilidad.
- Retenciones.
- Compras y Servicios.
- Recaudación.
- Fondos en Anticipo.
- Fondos de Caja Chica.
- Ejecución y Cierre de Presupuesto.

Los procesos administrativos que ejecuten los organismos públicos afectan a uno o varios de estos módulos simultáneamente debido a la fuerte integración existente entre ellos.

En la actualidad, el Sistema Registral y Notarial vigente, presenta algunas deficiencias relacionadas en la Administración Financiera de los Registros y Notarías, las cuales se mencionan a continuación:

- No cuenta con una plataforma tecnológica que permita determinar oportuna y rápidamente la información.
- Existencia de malversación y corrupción.
- Necesidad de una solución administrativa centralizada, que permita mantener un monitoreo global de cada subsistema.
- Necesidad de controlar y estandarizar los procesos de gestión financiera en los Registros y Notarías.

Como actividad clave en la Administración Financiera se encuentra el proceso de Recaudación; el cual se encarga de la ejecución, supervisión, registro y control de los ingresos que se originan por

conceptos de tasa por servicios regístrales y notariales establecidos en la Ley de Registro Público y del Notariado, generados a escala nacional.

Actualmente este proceso ocurre de manera descentralizada, cada registro lo lleva a cabo de manera independiente, originando algunas dificultades en el mismo, como son:

- Falsificación de los pagos de las planillas originadas por los trámites.
- Existencia de problemas con la conciliación de las cuentas recaudadoras.
- Desconocimiento de cuánto se recauda a nivel nacional por cada registro, sucursal bancaria y banco.
- No existe una integración del Sistema Registral con el Sistema de Recaudación.

Lo antes expuesto resalta la situación actual de la Recaudación en la Administración Financiera en los Registros y Notarías de la República Bolivariana de Venezuela, y el por qué de la automatización y estandarización de este proceso, permitiendo llevar a cabo un control centralizado y eficiente de la información. En estos momentos existe una modelación del proceso de Recaudación, donde se definen cada uno de los procesos que lo identifican, pero aun no existe un sistema automatizado que de respuesta a dicha modelación. La situación problemática abordada da origen a la siguiente **interrogante:**

¿Cómo obtener el diseño y la implementación que permita el desarrollo de un sistema informático para el proceso de Recaudación del Sistema de Administración Financiera del proyecto Registros y Notarías a partir de los requerimientos identificados que garantice recaudar los ingresos a nivel central?

Teniendo en cuenta la interrogante antes planteada, constituye entonces una novedad, la introducción de un sistema informático integrado que automatice el proceso de Recaudación del Sistema de Administración Financiera perteneciente al producto SAREN, que permita dar cumplimiento a lo estipulado en la ley, acorde a la nueva estructura administrativa financiera y a los objetivos trazados por la Dirección Nacional de los Registros y Notarías de la República Bolivariana de Venezuela.

Definiendo el proceso de desarrollo de software para la actividad Administrativa Financiera como objeto de estudio, centrándose el desarrollo de este trabajo en el diseño e implementación del proceso de Recaudación en el Sistema de Administración Financiera del proyecto Registros y Notarías identificándose así como campo de acción.

Dada la interrogante planteada anteriormente se traza el siguiente **Objetivo General**:

Obtener el modelo de diseño e implementación de la solución informática para el proceso de Recaudación en el Sistema de Administración Financiera de los Registros y Notarías de la República Bolivariana de Venezuela.

Para alcanzar la meta propuesta orientada fundamentalmente a proveer las especificaciones del diseño e implementación, se han trazado varios **objetivos específicos**, los cuales se listan a continuación:

- Estudiar las reglas del negocio y requerimientos identificados por los analistas.
- Obtener el modelo de diseño de la solución informática que permita la Recaudación de Tasas por servicios regístrales y notariales en la República Bolivariana de Venezuela.
- Obtener el modelo de implementación de la solución informática que permita la Recaudación de Tasas por servicios regístrales y notariales en la República Bolivariana de Venezuela.

Luego:

Si se realiza el diseño e implementación del proceso de Recaudación, se podrá obtener un sistema informático que permita la recaudación de los ingresos a nivel central.

Para poder desarrollar el diseño e implementación de este sistema, es preciso realizar tareas de investigación que guíen los caminos a seguir para dar cumplimiento a los objetivos definidos y obtener los resultados esperados.

Entre las **tareas identificadas** se encuentran las siguientes:

- Estudiar la Ley Orgánica de Administración Financiera del Sector Público. Gaceta Oficial No. 38.198 del 31 de Mayo del 2005.
- Estudiar los artefactos generados por el equipo de analistas para facilitar la comprensión del proceso.
- Estudiar los tipos de software existentes, con funcionalidades o características similares en otros países.

- Realizar el diseño de clase, el modelo lógico y los diagramas de interacción para la solución informática que permita la Recaudación de Tasas por servicios regístrales y notariales de la República Bolivariana de Venezuela.
- Implementar la solución informática que permita la Recaudación de Tasas por servicios regístrales y notariales en la República Bolivariana de Venezuela.
- Valorar las experiencias adquiridas en el trabajo.

El presente documento se estructura en tres capítulos y varios anexos:

Capítulo 1: Fundamentación Teórica

Aborda una breve caracterización sobre los Sistemas Financieros informáticos más usados en el mundo y en Cuba, enfatizando en los procesos de conciliación bancaria. Se realiza un breve recorrido por las distintas técnicas de programación existentes, patrones de arquitectura y diseño utilizados, herramientas case, frameworks, así como las plataformas de desarrollo de software.

Capítulo 2: Tendencias y Tecnologías

Aborda las principales tendencias y tecnologías que son empleadas en la actualidad en el mundo del desarrollo de software, herramientas, patrones arquitectónicos, de diseño, plataformas de desarrollo, etc.

Capítulo 3: Diseño e Implementación

Se realiza el diseño de la solución informática para el proceso de Recaudación en el Sistema de Administración Financiera, obteniéndose los artefactos correspondientes al flujo de trabajo de Análisis y Diseño, específicamente los correspondientes a la etapa de diseño, mostrándose los diagramas más representativos de las actividades del proceso de Recaudación. Se describe además la solución correspondiente al proceso objeto de este trabajo, mostrándose la arquitectura del mismo, así como el modelo de implementación.

Capítulo 4: Análisis de los resultados

Se estudian algunas métricas destinadas a la medición de la calidad del diseño de software, se realizan pruebas de unidad al código implementado. Se analizan y evalúan finalmente los resultados obtenidos.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA.

En este capítulo se realiza una breve explicación sobre el Proceso de Recaudación en la Administración Financiera de los Registros y Notarías de la República Bolivariana de Venezuela, inicialmente se aborda en qué consiste el proceso y luego se hace una exposición de los pasos que se realizan para llevarlo a cabo. Se hace una breve caracterización sobre los Sistemas Financieros Informáticos más usados en el mundo y en Cuba, enmarcados en los procesos de conciliación bancaria, pues es el tema que ocupa el presente trabajo. Se hace un breve recorrido por las distintas técnicas de programación existentes, patrones de arquitectura y diseño utilizados, herramientas case, así como el surgimiento de las plataformas de desarrollo de software empresarial, se seleccionan las más utilizadas y se analizan sus características, semejanzas y diferencias; al finalizar el capítulo se detalla la plataforma empleada en el desarrollo de este trabajo.

1.1. Sistema de Recaudación.

Según el artículo 2 de la Ley Orgánica de la Administración Financiera del Sector Público la Administración Financiera comprende el conjunto de sistemas, órganos, normas y procedimientos que intervienen en la captación de ingresos públicos y en su aplicación para el cumplimiento de los fines del Estado, y estará regida por los principios constitucionales de legalidad, eficiencia, solvencia, transparencia, responsabilidad, equilibrio fiscal y coordinación macroeconómica. El proceso de Recaudación dentro de la Administración Financiera, consiste en la recepción de los pagos que deben realizar los clientes de los Registros y Notarías por los trámites solicitados, en las oficinas de los bancos recaudadores correspondientes, mediante una Planilla Única Bancaria, diseñada de acuerdo a las necesidades de las oficinas, para ello se permite registrar el estatus de las Planillas Únicas Bancarias desde que son emitidas en las oficinas hasta que le dan continuidad al trámite después de pagada, así como el proceso de Reintegros y Recaudación por abandono de las mismas. También se permite evaluar la cantidad de Planillas Únicas Bancarias exoneradas, exentas y caducadas, también se registran y controlan todos los Ingresos que se obtienen por Derechos de Registro Público, Mercantil, Principal y Notariales, Habilitación, Traslado, Inserción Anticipada del documento y Transporte. (Marquez, Proyecto de Modernización de los Registros y Notarías Administración Financiera Módulo de Recaudación Documento de Prototipo de Interfaz de Usuario, 2007). A continuación se muestra un esquema con la estructura financiera actual de la dirección nacional de Registros y Notarías.

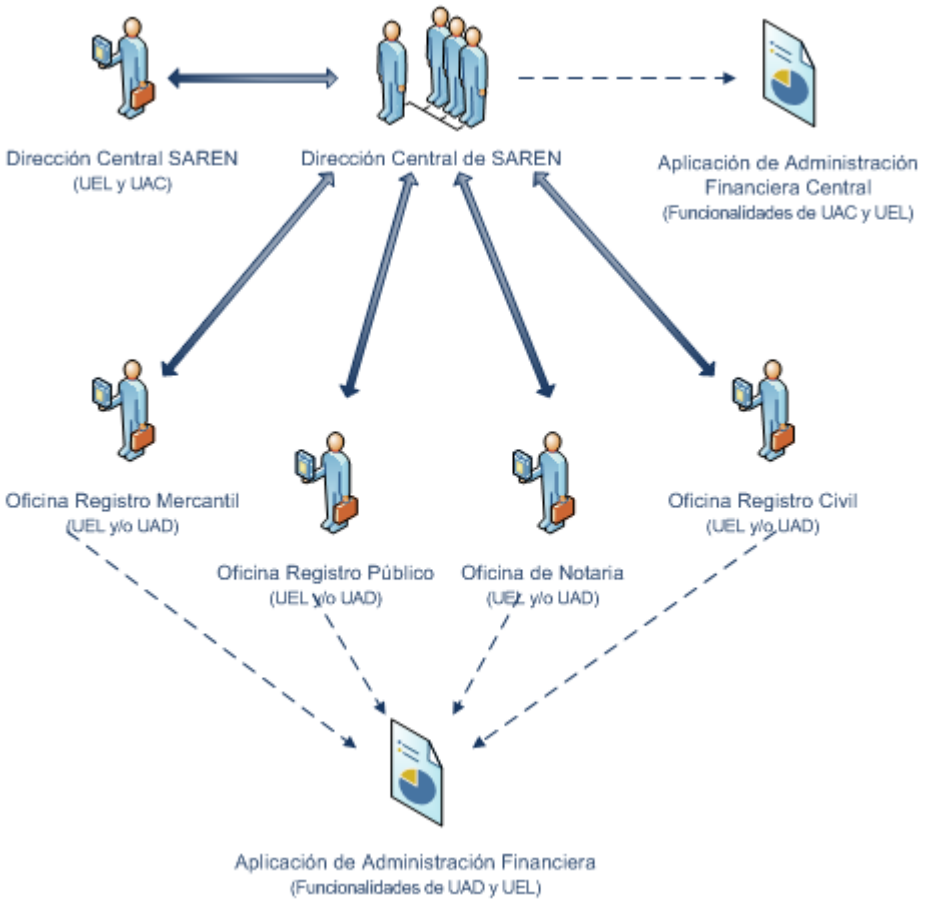


Figura 1. Estructura financiera de los Registros y Notarías de la República de Venezuela.

1.2. Pasos en el Proceso de Recaudación.

A continuación se explica brevemente como ocurre este proceso:

Inicialmente el usuario se presenta ante un funcionario del Registro o Notaría para realizar un trámite legal, el funcionario de taquilla del Registro atiende la solicitud del cliente registrando los datos necesarios para efectuar dicho trámite legal, emite la Planilla Única Bancaria (PUB) y entrega al cliente 5 copias de la misma, el usuario se retira del registro para efectuar el pago de la PUB y se dirige al banco a realizar la cancelación de la planilla, una vez allí, el funcionario de taquilla del banco solicita al cliente la PUB para procesar el pago según requerimientos de SAREN, luego registra la información necesaria y valida los datos de la PUB. El usuario efectúa el pago de la PUB, el funcionario de taquilla

del Banco imprime en la PUB la ráfaga de seguridad por el pago efectuado, luego entrega al cliente 3 copias de la PUB debidamente selladas para que se dirija al Registro para su presentación, el usuario recibe las 3 copias de la PUB y se retira del banco, en este momento regresa al registro y presenta 2 copias de la PUB pagada para darle continuidad a su solicitud. El funcionario de taquilla del registro captura y valida la información correspondiente al pago emitida por el banco dándole continuidad al trámite, archiva en el expediente una copia de la PUB y la otra es archivada para uso. De esta manera ocurre una notificación Periódica de la Recaudación por parte del Registro, para ello envía periódicamente una relación de todas las PUB presentadas a la UR para su control, la UR[2] recibe la relación de las Planillas Únicas Bancarias y envía copia de la Relación de PUB a la UC[3], la UC recibe la Relación de las PUB recaudadas y entonces registra el asiento contable de dichos recaudos. Por parte del banco ocurren también notificaciones del depósito de las Planillas Bancarias de la siguiente manera; el banco recaudador notifica y envía a la UR, cada PUB que fue recaudada por sus sucursales, la UR concilia cada PUB reportada por el banco recaudador con cada PUB reportada por los Registros y Notarías, la UR verifica la planilla para ver si se encuentra registrada en los libros de recaudación como presentada, la UR la señala como trámite formal chequeado, en caso contrario la anota en una Relación de Depósitos en Tránsito. La UR envía a la UC la Relación de Depósitos en Tránsito y realiza los asientos contables de los depósitos en tránsito una vez realizada la recepción de los estados de cuenta del período contable (Marquez, Proyecto de Modernización de los Registros y Notarías Administración Financiera Módulo de Recaudación Documento de requerimientos, 2007).

A continuación se muestran varias figuras con dos de los procesos que se llevan a cabo para la Recaudación de Tasas en el Administración Financiera.

[2] **UR:** Unidad Recaudadora de fondos.

[3] **UC:** Unidad Central.

Figura 2. Proceso Notificación en línea con la Institución Financiera.

Figura 3. Proceso Conciliación Bancaria.

1.3. Sistemas financieros computarizados en el mundo

El surgimiento de una nueva tecnología experimentada en las últimas cuatro décadas se presenta como la antesala del desarrollo de lo que hoy se conoce como Sistemas Automatizados.

La aparición de las computadoras, la generalización de las corporaciones, el surgimiento de grandes empresas multinacionales y la globalización de los comercios internacionales, dio un nuevo giro a la orientación de la información financiera, surgiendo la necesidad de crear Sistemas Financieros que sean más útiles en el desenvolvimiento de la gestión administrativa, y que dichas informaciones sean efectivas, confiables y oportunas; esta necesidad fue lo que impulsó hacia la creación de los Sistemas de Finanzas Computarizados.

Existe una amplia gama de Softwares que han formado parte de las herramientas de trabajo de muchas empresas desde hace más de 40 años, hasta tal punto que hoy en día son el motor de las operaciones de muchas de ellas. Lo que ha permitido a los Ingenieros en Sistemas buscar la forma de satisfacer de una manera más completa las necesidades en las que se ve envuelta la empresa, de

acuerdo al volumen de las operaciones que esta maneja; por lo que han tratado crear Sistemas Computarizados de Finanzas que brinden los mismos beneficios que ofrecen los Sistemas Manuales.

En su afán de crear este tipo de Software los expertos en la materia han diseñado diversidad de programas que tratan de suplir la necesidad de determinadas empresas de acuerdo a las actividades que realiza. Dentro de estos Software, los más conocidos son:

- Dac Easy Accounting System
- PeachTree.

Existen softwares ideados para determinadas empresas, los cuales han sido creados para uso exclusivo de cada una de ellas y no son producidos con fines comerciales. Ahora, existen softwares comerciales donde el más usado y reconocido por todos es el DacEasy Accounting System, el cual desde los años 70 es conocido en el mercado como uno de los Software de Contabilidad más completo.

1.3.1. DacEasy Accounting System

DacEasy Accounting System es un software que combina información financiera para un eficaz control de los negocios y una oportuna y acertada toma de decisiones.

El DacEasy está compuesto por varios módulos que pueden o no trabajar como un sistema totalmente integrado, ofreciendo así una ventaja más sobre los sistemas manuales de contabilidad que se caracterizan por su lentitud en el procesamiento de la información.

Este software ha sido diseñado para su utilización en diversos tipos de negocios, como lo son los orientados a la prestación de servicios, a la fabricación y venta de productos, y a cualquier tipo de negocio que combine estas áreas.

El DacEasy es uno de los softwares de contabilidad de mayor venta en toda América, desde su lanzamiento en los años ’70 ha demostrado ser uno de los sistemas computarizados de contabilidad más completo, obteniendo varios galardones en el área de la programación informática.

DacEasy Accounting System desde sus inicios ha revolucionado la forma de hacer contabilidad computarizada, presentando un programa capaz de trabajar en módulos con la cualidad de

relacionarse por interfaz, permitiendo así un mejor desenvolvimiento para el usuario en el desarrollo de sus actividades cotidianas.

En su afán de ir mejorando cada día los beneficios de este sistema, las personas encargadas de trabajar en el desarrollo de dicho programa han ido creando nuevas formas para que el programa DacEasy sea cada vez más eficaz. Esta es la razón por la que a través de los años se han creado nuevas versiones de este programa.

Contabilidad DacEasy ofrece un paquete de Contabilidad, que a pesar de tener múltiples características, es muy fácil de usar. Este programa de contabilidad contempla siete (7) módulos que son:

- Contabilidad General.
- Cuentas por Cobrar.
- Cuentas por Pagar.
- Efectivo (Caja y Banco).
- Facturación.
- Compras.
- Inventarios.

Estos módulos trabajan en conjunto como un sistema completamente integrado o pueden ser usados individualmente sin importar cual se escoja. Contabilidad DacEasy ofrece una interfaz rápida y fácil, reduciendo sus tareas contables a la simple entrada de datos. Las complejas características de los reportes permiten mirar y analizar la información facilitando una completa auditoria a medida que esta se procesa.

Dada la importancia que adquiere el módulo de Efectivo (Caja y Banco) dentro de este trabajo, será abordado a continuación.

El módulo de Efectivo de Contabilidad DacEasy demostrará ser uno de los módulos más útil y más frecuentemente usado en el sistema. Se utiliza para entrar y rastrear todos los depósitos y cheques. DacEasy también ofrece una característica de reconciliación de un ilimitado número de cuentas.

Este módulo permite consolidar todo el movimiento de efectivo que incluye tanto los pagos como las devoluciones de clientes y proveedores.

Dependiendo de su naturaleza, los depósitos y los cheques pueden ser empleados en un saldo específico de un cliente o directamente en una cuenta de Contabilidad General. Se puede entrar un depósito grande y distribuir su valor en un número ilimitado de cuentas. Esto hace más fácil su proceso de reconciliación pues el valor del depósito cuadrará con los movimientos de su estado bancario.

Los cheques y las conciliaciones pueden aplicarse al saldo de un proveedor o directamente a cualquier cuenta de Contabilidad General; además su proceso de distribución es sencillo.

Las ventanas de programa muestran una lista completa de las cuentas de contabilidad general. Cuando se seleccionan cuentas para distribución, se puede usar Clientes y Proveedores. Si se está aplicando un depósito o un pago a una factura en particular, las facturas abiertas para el cliente o el proveedor se mostrarán en pantalla para su selección (Smith, 1991).

1.3.2. PeachTree.

Peachtree es un sistema contable que ha tenido gran acogida en los Estados Unidos, y en Panamá, por sus grandes bondades. La facilidad de uso ha hecho bastante fácil la implantación de Peachtree en empresas pequeñas y medianas sin que sea necesario tener un gerente de cómputo dedicado a su mantenimiento. Y su amplitud le ha dado cabida en una diversa gama de empresas, desde restaurantes, arquitectos, corredores de seguros, abogados, hasta contadores y auditores (Easy, 2000).

La facilidad en el uso de Peachtree se desprende de un diseño basado en facsímiles de los documentos que normalmente se utilizan en los negocios.

Peachtree incluye, en un solo programa, los módulos de mayor general, planilla, cuentas por pagar, cuentas por cobrar, inventario, conciliación bancaria, costo de obras, y análisis financiero. También está incluido un módulo separado para el control de activos fijos.

Peachtree es un programa totalmente integrado. Bajo un solo programa se encuentran todas las capacidades que en otros sistemas se considerarían módulos discretos. La organización de los módulos dentro del antes mencionado está más bien relacionada con la funcionalidad del paradigma de los documentos que usa el sistema. Lo que en otros sistemas se consideraría cuentas por pagar, en este software sería compras.

Peachtree está compuesto por los siguientes módulos:

- Ventas.
- Compras.
- Inventario.
- Mayor General.

Dada la importancia que adquiere el modulo de Mayor General dentro de este trabajo, será abordado a continuación.

El mayor general es posiblemente la única parte de Peachtree que se puede considerar estándar en relación a otros programas. Esta parte comprende el diario general, la conciliación y la impresión de estados financieros.

En esta parte se cotejan los estados del banco contra los movimientos de las cuentas en el mayor general. El resultado de este registro es la conciliación bancaria, que toma en cuenta los cheques y depósitos en tránsito para sustentar las diferencias entre el saldo de una cuenta a fin de mes y el estado del banco. La conciliación es muy importante para verificar que todos los cheques que llegaron al banco estén registrados en la contabilidad.

1.3.3. Sistemas financieros computarizados en Cuba.

Sistema Económico Integrado Versat – Sarasola.

El **VERSAT - Sarasola** es un Sistema integrado de gestión económica, diseñado para ser utilizado por el sector empresarial Cubano, que se adecua a las características de cada entidad, es configurable por cada una de ellas en el momento de su instalación y tiene como objetivo fundamental ofrecerle a los usuarios la posibilidad de contar con un instrumento seguro, rápido, eficaz y de fácil manejo para la Planificación, Control y el Análisis de la Gestión Económica (Cabrera González, Obregón Rodríguez, Cárdenas Negrin, & Carralero Silva, 2004).

Permite llevar el control y el registro contable individual de todos los hechos económicos que se originan en las estructuras internas de las entidades y obtener los Estados Financieros y Análisis Económicos y Financieros en estos niveles.

Se estructura en un grupo de Subsistemas, donde se procesan y contabilizan documentos primarios y se anotan los movimientos de los recursos materiales, financieros y laborales que se utilizan en una entidad a partir de una configuración previa de los comprobantes que se originan.

A continuación se mencionan los principales subsistemas, detallando el más importante para nuestro trabajo.

- Configuración.
- Contabilidad General.
- Costos y Procesos.
- Inventarios.
- Activos Fijos.
- Contratación y Facturación.
- Finanzas.

Finanzas.

Este Subsistema está concebido para que las administraciones dispongan de una información actualizada sobre el movimiento y la disponibilidad de efectivo a partir del procesamiento de todos los documentos, tanto de obligaciones como de pagos.

Otro de los objetivos principales es dar la posibilidad a los usuarios de configurar, a través de conceptos, todos los movimientos que se procesan en el Subsistema, con lo que se logra que los comprobantes se asienten en la contabilidad bajo los criterios predeterminados, facilitando la obtención de la información deseada y agrupada según las necesidades de cada empresa.

Junto a los aspectos anteriores, el Subsistema controla la existencia, uso y disponibilidad de los instrumentos de pagos, vinculados a las cuentas bancarias que la empresa posee, las cuales puede conciliar y procesar de ellas, los estados de cuentas correspondientes.

1.3.4. Subsistema de AF en el sistema SAREN.

Los dirección de los Registros y Notarías de Venezuela en su afán de automatizar y modernizar las actividades registrales, han dado varios pasos de avances, de los cuales no ha estado exento la actividad Administrativa Financiera, para lo cual ha reformado la estructura que responde a esta

actividad, dándose a la tarea de desarrollar el subsistema de Administración Financiera en el sistema SAREN.

La Administración Financiera comprende la automatización de los siguientes módulos:

- Presupuesto.
- Requisiciones.
- Tesorería.
- Contabilidad.
- Retenciones.
- Compras y Servicios.
- Recaudación.
- Fondos en Anticipo.
- Fondos en Caja Chica.
- Ejecución y Cierre de Presupuesto.

Partiendo del estudio realizado de los sistemas antes mencionados y de las particularidades de los módulos que estos implementan muy relacionados con el módulo de Recaudación, se desarrolla el presente trabajo.

CAPÍTULO 2: TENDENCIAS Y TECNOLOGÍAS

Con el surgimiento de los sistemas de computo se hizo necesario comenzar a darle órdenes para que cumplieran cierta función, de esta forma comenzaron a surgir lenguajes de programaciones y técnicas de programación las cuales han estado evolucionando incluso en la actualidad. Las técnicas de programación son aplicables cuando se desea trasladar a la computadora el funcionamiento de un proceso, o la solución a un problema determinado, para ello se realiza una abstracción, o sea, un modelo simplificado de la realidad tomando los elementos más significativos y transformándolos en variables, de esta forma la computadora podrá entenderlo y se habrá obtenido el resultado que se estaba buscando.

Lo que permite especificar, a modo de instrucciones, cuáles son los pasos que tendrá que seguir la computadora para resolver el problema se denomina lenguaje de programación, que no es otra cosa que una herramienta. La forma en que se especifique y se elabore la solución es a lo que se le llama técnica de programación.

La computadora automática debe su derecho a existir, su utilidad, precisamente a su capacidad de efectuar muchos cálculos que no pueden realizar los seres humanos. Se desea que la computadora efectúe lo que nunca podrían hacer los seres humanos, y la potencia de las máquinas actuales es tal, que inclusive los cálculos pequeños, por su tamaño, escapan al poder de la imaginación limitada.

2.1. Tendencias y Tecnologías

2.1.1 Programación Estructurada.

Los programas computarizados pueden ser escritos con un alto grado de estructuración, lo cual les permite ser más comprensibles en actividades tales como pruebas, mantenimiento y modificación de los mismos. Mediante la Programación Estructurada todas las divisiones de control de un programa se encuentran estandarizadas, de forma tal que es posible leer la codificación del mismo desde su inicio hasta su terminación en forma continua, sin tener que saltar de un lugar a otro del programa siguiendo el rastro de la lógica establecida por el programador, como es la situación habitual con codificaciones desarrolladas bajo otras técnicas.

Programación Estructurada: Es una técnica en la cual la estructura de un programa, la escritura de sus partes se realiza tan claramente como es posible mediante el uso de tres estructuras lógicas de control:

- **Secuencia:** Sucesión simple de dos o más operaciones.
- **Selección:** División condicional de una o más operaciones.
- **Interacción:** Repetición de una operación mientras se cumple una condición.

Estos tres tipos de estructuras lógicas de control pueden ser combinados para producir programas que manejen cualquier tarea de procesamiento de información.

2.1.2 Programación Procedimental.

La Programación Procedimental es un paradigma de programación basado en el concepto de "llamado de procedimientos", los procedimientos, también conocidos como rutinas, subrutinas, métodos o funciones, simplemente consisten en una series de pasos computacionales. La mayor parte de los lenguajes de alto nivel la soportan, permiten la creación de procedimientos, que son pequeños fragmentos de código que realizan una tarea determinada.

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, se puede dividir el 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.

2.1.3 Programación Orientada a Objetos.

El término de Programación Orientada a Objetos indica más una forma de diseño y una metodología de desarrollo de software que un lenguaje de programación, en realidad se puede aplicar el Diseño Orientado a Objetos (en inglés abreviado OOD[4]), a cualquier tipo de lenguaje de programación.

Básicamente la Programación Orientada a Objeto (OOP[5]) permite a los programadores escribir software, de forma que esté organizado en la misma manera que el problema que trata de modelar.

[4] OOD: Object Oriented Design

Los lenguajes de programación convencionales son poco más que una lista de acciones a realizar sobre un conjunto de datos en una determinada secuencia. Si en algún punto del programa se modifica la estructura de los datos o la acción realizada sobre ellos, el programa cambia.

La OOP aporta un enfoque nuevo, convirtiendo la estructura de datos en el centro sobre el que pivotan las operaciones. De esta forma, cualquier modificación de la estructura de datos tiene efecto inmediato sobre las acciones a realizar sobre ella, siendo esta una de las diferencias radicales respecto a la programación estructurada.

La OOP proporciona las siguientes ventajas sobre otros lenguajes de programación:

- **Uniformidad:** La representación de los objetos lleva implícita tanto el análisis como el diseño y la codificación de los mismos.
- **Comprensión:** Tanto los datos que componen los objetos, como los procedimientos que los manipulan, están agrupados en clases, que se corresponden con las estructuras de información que el programa trata.
- **Flexibilidad:** Al tener relacionados los procedimientos que manipulan los datos con los datos a tratar, cualquier cambio que se realice sobre ellos quedará reflejado automáticamente en cualquier lugar donde estos datos aparezcan.
- **Estabilidad:** Dado que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo sobre aquellos que cambian con frecuencia permite aislar las partes del programa que permanecen inalterables en el tiempo.
- **Reusabilidad:** La noción de objeto permite que programas que traten las mismas estructuras de información reutilicen las definiciones de objetos empleadas en otros programas e incluso los procedimientos que los manipulan. De esta forma, el desarrollo de un programa puede llegar a ser una simple combinación de objetos ya definidos donde estos están relacionados de una manera particular. (Tejerina, 2003).

2.1.4 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

[5] OOP: Programación Orientada a Objeto

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.

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 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.

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.

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.

2.1.5 Programación Orientada a Aspectos

La Programación Orientada a Aspectos (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. Es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la AOP se pueden encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término AOP el cual es usado para referirse a varias tecnologías relacionadas como los métodos adaptivos, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

2.2. Patrones de Arquitectura y diseño.

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. (Alexander, 1997)

Un patrón de arquitectura de software describe un problema particular y recurrente del diseño, que surge en un contexto específico, y presenta un esquema genérico y probado de su solución. (Alexander, 1997)

2.2.1. Fachada

El patrón de diseño fachada sirve para proveer de una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas.

Fachada puede:

- Hacer una biblioteca de software más fácil de usar y entender, ya que implementa métodos convenientes para tareas comunes.
- Hacer el código que usa la librería más legible, por la misma razón.
- Reducir la dependencia de código externo en los trabajos internos de una librería, pues la mayoría del código lo usa Fachada, permitiendo así más flexibilidad en el desarrollo de sistemas.
- Envolver una colección mal diseñada de API[6] con un solo API bien diseñado.

Problemas que soluciona:

Problema: Un cliente necesita acceder a parte de la funcionalidad de un sistema más complejo.

Solución: Definir una interfaz que permita acceder solamente a esa funcionalidad.

Problema: Existen grupos de tareas muy frecuentes para las que se puede crear código más sencillo y legible.

Solución: Definir una funcionalidad que agrupe estas tareas en funciones o métodos sencillos y claros.

[6] API: Interfaz de Programación de Aplicaciones

Problema: Una biblioteca es difícilmente legible.

Solución: Crear un intermediario más legible.

Problema: Dependencia entre el código del cliente y la parte interna de una biblioteca.

Solución: Crear un intermediario y realizar llamadas a la biblioteca sólo o, sobre todo, a través de él.

Problema: Necesidad de acceder a un conjunto de APIs, que pueden, además, tener un diseño no muy bueno.

Solución: Crear una API intermedia, bien diseñada, que permita acceder a la funcionalidad de las demás (Ramirez, 2003).

2.2.2. Proxy

El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

Este es un patrón estructural fundamental, muy general que ocurre frecuentemente en muchos otros patrones. El patrón obliga a que las llamadas a métodos de un objeto ocurran indirectamente a través de un objeto proxy, que actúa como sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.

Un objeto proxy recibe llamados a métodos que pertenecen a otros. Los objetos clientes llaman a los métodos de los objetos proxy, este último no provee directamente el servicio que el cliente espera, sino que invoca los métodos en el objeto específico que provee cada servicio.

Un objeto proxy provee alguna administración requerida para los servicios. Por lo general un objeto proxy comparte una interfaz o superclase común con el objeto que realmente provee el servicio. De esta manera, los objetos cliente no se dan cuenta que están invocando métodos en el proxy, y piensan que los están invocando directamente en los objetos respectivos. Esta transparencia en la administración de los servicios de otros objetos, es la razón fundamental para usar un proxy (Ramirez, 2003).

Este patrón sugiere que cuando se necesite tener una administración transparente de ciertos servicios provistos por otros objetos, se fuerce a que todos los accesos a estos servicios pasen a través de un objeto proxy.

Problema que soluciona:

Se necesita crear objetos que consuman muchos recursos, pero no se quiere que los mismos sean instanciados a no ser que el cliente lo solicite o se cumplan otras condiciones determinadas.

2.2.3. Abstract Factory

Problema:

El problema que trata de resolver es proporcionar una interfaz para la creación de familias de objetos interdependientes o interrelacionados, sin especificar sus clases concretas (Unidad Docente de Ingeniería del Software,Facultad de informática - Universidad Politécnica de Madrid , 2003).

Cuándo usarlo:

- Cuando el sistema debe ser independiente de como sus productos se crean, componen y representan.
- Cuando el sistema debe configurarse con una familia de productos de entre múltiples posibles.
- Cuando se quiere dar énfasis a la restricción de que una familia de productos ha sido diseñada para que actúen todos juntos.
- Cuando se quiere proporcionar una librería de clases de productos, de los que sólo se desea revelar su interfaz, no su implementación.

Ventajas:

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

Inconvenientes:

- Se dificulta la extensibilidad, puesto que no es fácil añadir nuevos tipos de productos.

2.2.4. Modelo Vista Controlador

El patrón Modelo Vista Controlador (MVC) es un patrón de diseño de software en el cual todo el proceso está dividido en 3 capas, típicamente estas capas son el Modelo, la Vista y el Controlador (Reynoso, Kiccillof, & AIRES, 2004).

El Modelo incorpora la capa del dominio y persistencia, la cual es la encargada de guardar los datos en un medio persistente (ya sea una base de datos, un archivo de texto, XML, registro, etc.). En el modelo es donde se hace el levantamiento de todos los objetos que el sistema debe utilizar, es el proveedor de los recursos.

La Vista se encarga de presentar la interfaz al usuario, en sistemas web, esto es típicamente HTML, aunque pueden existir otro tipo de vistas. En la vista solo se deben de hacer operaciones simples, como if, ciclos, formateo, etc.

El Controlador es el que escucha los cambios en la vista y se los envía al modelo, el cual le regresa los datos a la vista, es un ciclo donde cada acción del usuario causa que se inicie de nuevo un nuevo ciclo.

La forma más sencilla de implementar este patrón es pensando en capas, como regla, los accesos a la base de datos se hacen en el modelo, la vista y el controlador no deben de saber si se usa o no una base de datos. El controlador es el que decide qué vista se debe de imprimir y qué información es la que se envía.

2.2.5. Layers

Descripción:

Descompone una aplicación en un conjunto de capas independientes y ordenadas jerárquicamente según el nivel de abstracción que tenga cada una de ellas. Cada nivel o capa usa los servicios del nivel inmediatamente inferior y ofrece servicios a la capa inmediatamente superior (Reynoso, Kiccillof, & AIRES, 2004).

Problema:

Imaginando que se está diseñando un sistema cuya característica dominante es una mezcla de problemas de distintos niveles de abstracción, esto quiere decir que se presentarán problemas de un nivel alto de abstracción y problemas de un nivel bajo de abstracción, donde los niveles de abstracción altos confían en los niveles bajos de abstracción para su funcionamiento.

En ocasiones se puede presentar un sistema cuyos niveles de abstracción son pocos, pero en uno de los niveles bajos de abstracción se tienen distintos tipos de abstracciones, esto hace que en vez de tener una estructura vertical, se tiene en el nivel varias abstracciones diferentes e independientes las cuales se van a estructurar de forma horizontal.

Solución:

Se deberá estructurar el sistema en un número apropiado de capas, que esto se realiza a priori. Comenzando por la capa que posea el nivel de abstracción más baja, y continuando hasta la capa de nivel de abstracción más alta. Puede ocurrir que cuando se está diseñando el sistema se quiten capas, o agreguen capas por un mal diseño supuesto desde un principio.

Ventajas:

- Reutilización de un mismo nivel en varias aplicaciones. Si una capa individual posee una abstracción y una interfaz bien definida, con una buena documentación, la capa puede rehusarse en distintos contextos múltiples. Sin embargo, a pesar de los costos más altos de no rehusar la tal capa, los diseñadores prefieren a menudo volver a escribir esta funcionalidad. Ellos concluyen que la capa existente no encaja con sus propósitos exactamente, puesto que si la usaran tardarían más tiempo en adaptar la capa que rehaciéndola.
- Permite la estandarización.
- El cambio de un nivel no afecta al resto. Pero un mal diseño o un cambio importante de funcionalidad pueden forzar cambios que se transmitan en cascada de un nivel a otro.
- División clara de trabajo entre los miembros de un equipo.
- Dará soporte a la arquitectura MVC.

Desventajas:

- Si el número de niveles es excesivo, puede ser muy ineficiente. Debido a que la transacción de datos entre los niveles de capas superiores e inferiores puede demandar mucho tiempo.
- Trabajo innecesario de paso de argumentos entre niveles.
- Si hay pocos niveles el diseño se encuentra poco organizado. Si hay excesivos niveles el sistema es muy complejo e ineficiente.

2.3 Arquitectura de Software.

La Arquitectura del Software es el diseño de más alto nivel de la estructura de un sistema, programa o aplicación y tiene la responsabilidad de:

- Definir los módulos principales.
- Definir las responsabilidades que tendrá cada uno de estos módulos.
- Definir la interacción que existirá entre dichos módulos:
    - Control y flujo de datos.
    - Secuenciación de la información.
    - Protocolos de interacción y comunicación.
    - Ubicación en el hardware.

Esta aporta una visión abstracta de alto nivel, posponiendo el detalle de cada uno de los módulos definidos a pasos posteriores del diseño.

La definición oficial de Arquitectura del Software es la IEEE Std 1471-2000 que reza así: “La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución”.

2.3.1 Estilos Arquitectónicos.

Un estilo afecta a toda la arquitectura de software y puede combinarse de acuerdo a la propuesta de solución (Reynoso, Kiccillof, & AIRES, 2004) establecen que la imposición de ciertos estilos arquitectónicos mejora o disminuye las posibilidades de satisfacción de ciertos atributos de calidad del sistema. Afirman con esto que cada estilo propicia atributos de calidad, y la decisión de implementar alguno de los existentes depende de los requerimientos de calidad del sistema.

En la caracterización de David Garlan, Andrew Kompanek, Ralph Melton y Robert Monroe, también de Carnegie Mellon, se define el estilo como una entidad consistente en cuatro elementos:

1. Un vocabulario de elementos de diseño: componentes y conectores tales como tuberías, filtros, clientes, servidores, parsers, bases de datos, etcétera.

2. Reglas de diseño o restricciones que determinan las composiciones permitidas de esos elementos.
3. Una interpretación semántica que proporciona significados precisos a las composiciones.
4. Análisis susceptibles de practicarse sobre los sistemas construidos en un estilo, por ejemplo análisis de disponibilidad para estilos basados en procesamiento en tiempo real, o detección de abrazos mortales para modelos cliente-servidor.

Los estilos son entidades que ocurren en un nivel sumamente abstracto, puramente arquitectónico, que no coincide ni con la fase de análisis propuesta por la temprana metodología de modelado orientada a objetos (aunque sí un poco con la de diseño), ni con lo que más tarde se definirían como paradigmas de arquitectura, ni con los patrones arquitectónicos.

Los patrones arquitectónicos, por su parte, se han materializado con referencia a lenguajes y paradigmas también específicos de desarrollo, mientras que ningún estilo presupone o establece preceptivas al respecto. Si hay algún código en las inmediaciones de un estilo, será código del lenguaje de descripción arquitectónica o del lenguaje de modelado; de ninguna manera será código de lenguaje de programación.

Lo mismo en cuanto a las representaciones visuales: los estilos se describen mediante simples cajas y líneas, mientras que los patrones suelen representarse en UML.

- **Estilos de Llamada y Retorno.**

Los estilos de Llamada y Retorno son los más generalizados en sistemas de gran escala que enfatizan la modificabilidad y la escalabilidad. Entre los miembros de la familia se encuentran las arquitecturas de programa principal y subrutina, los sistemas basados en llamadas a procedimientos remotos, los sistemas orientados a objeto y los sistemas jerárquicos en capas.

Model-View-Controller (MVC): En ocasiones se le define más bien como un patrón de diseño o como práctica recurrente, y en estos términos es referido en el marco de la estrategia arquitectónica de Microsoft.

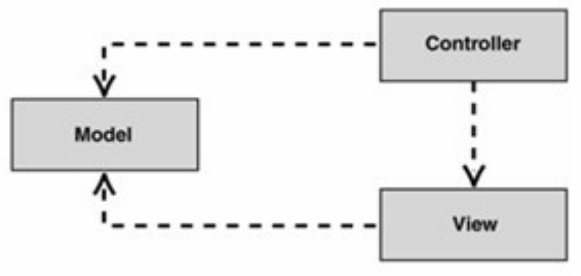


Figura 4. Model-View-Controller.

Lo que se persigue con este estilo de arquitectura es separar sus tres componentes, el modelo, la vista y el controlador de manera que puedan ser reutilizados y desarrollados por diferentes personas de forma paralela. Para el desarrollo del sistema se escoge un Framework existente en la Universidad el cual se basa en el MVC, de esta forma se aprovechan las ventajas de este estilo o patrón.

Arquitecturas Orientadas a Objetos: Nombres alternativos para este estilo han sido Arquitecturas Basadas en Objetos, Abstracción de Datos y Organización Orientada a Objetos. Los componentes de este estilo son los objetos, o más bien instancias de los tipos de dato abstractos. Los objetos representan una clase de componentes que en algunas ocasiones son denominados managers, debido a que son responsables de preservar la integridad de su propia representación.

El principal problema de esta arquitectura es que es necesario tener un nivel de abstracción superior que agrupe estos objetos de modo tal que sean más fáciles de entender y desarrollar en paralelos por diferentes grupos de desarrolladores; en sistemas de gran tamaño, las interacciones entre los objetos y su volumen se pueden volver inmanejables.

Arquitectura Basada en Componentes: es una tendencia que surgió después de la Programación Orientada a objetos, debido a la necesidad creciente de la reutilización de código, pues aunque los objetos se puedan reutilizar, el componente podría entregarse o venderse como un paquete, exponiendo una interfaz y manteniendo su integridad. El desarrollo de una Arquitectura Basada en Componentes lleva un esfuerzo por parte del equipo de desarrollo, si se tiene en cuenta que se está en presencia de un negocio que tiene propias reglas, no sería apropiado emplear este tipo de arquitectura; lo cual no excluye la posibilidad de desarrollar componentes para aquellas funcionalidades que podrían emplearse en el desarrollo de otros sistemas.

La Arquitectura en Capas: podría decirse que su finalidad es abstraer las funcionalidades de una capa, de manera tal que esta pueda ser totalmente reemplazada. La Arquitectura de Capas más común es la que está compuesta por tres, Presentación, Modelo o Reglas del Negocio de la Empresa y Acceso a Datos. De esta forma se podrá cambiar cualquiera de estas sin afectar a las restantes. Aunque tres capas es lo más común, a medida que aumenta la complejidad de los sistemas, las capas crecen. A su vez cada capa puede estar compuesta por subcapas y una capa o subcapa puede estar compuesta por una o más clases del diseño.

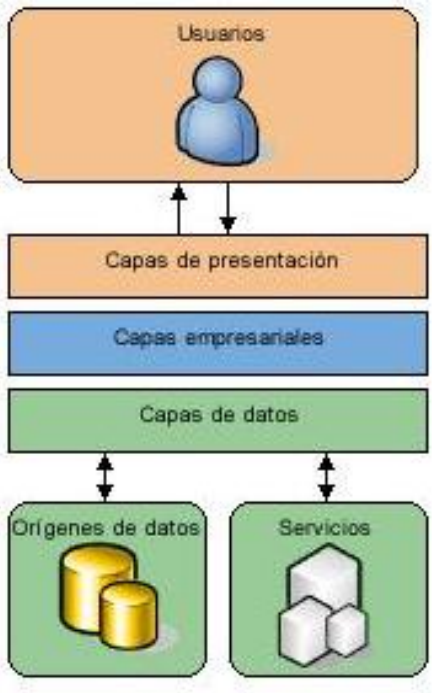


Figura 5. Arquitectura en tres capas.

2.4. Herramientas Case.

Las herramientas CASE[7] representan una forma que permite Modelar los Procesos de Negocios de las empresas y desarrollar los Sistemas de Información Gerenciales.

2.4.1. Enterprise Architect

Enterprise Architect (EA) 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. Con un gran conjunto de características, EA puede equipar al equipo entero, incluyendo analistas, evaluadores, administradores de proyectos, personal del control de calidad, equipo de desarrollo y más.

EA 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.

Trazabilidad de extremo a extremo:

EA 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 EA están construidas sobre la especificación de UML 2.0 - pero no se detiene ahí, usa perfiles UML para extender el dominio de modelado, mientras que la validación del modelo asegura integridad. Combina Procesos de Negocio, Información y Flujos de trabajo en un modelo (Sparx Systems, 2007).

[7] CASE: Ingeniería de Software Asistida por Computación

2.4.2. Embarcadero ER/Studio 7.0

Embarcadero ER/Studio 7.0 posee capacidades colaborativas de modelado y administración, así como mayor soporte en la integración de almacenamiento de datos diseñados para visualizar, documentar y compartir el conocimiento. Para el diseño y modelado de bases de datos físicas, Embarcadero ER/Studio 7.0 también incluye capacidades de planeación de modelado seguros.

Incorpora nuevas capacidades de modelado, que permiten administrar grandes modelos en forma colaborativa, al facilitar y reforzar los estándares en estos modelos, además de la creación de diagramas en su documentación. La solución captura los metadatos clave en un formato de datos lineales.

Las nuevas características de ER/Studio incluyen:

Administración del modelado empresarial, permite la creación y consolidación de modelos de proyecto en un modelo global, para la administración holística de la información. Ayuda en la administración de grandes modelos mediante la colaboración central en equipo, utilidades para la automatización de tareas clave, promoción de elementos compartidos por niveles, y permite el reforzamiento de estándares con la finalidad de promover la utilización de información consistente en la organización.

Capacidades avanzadas en la administración de metadatos, incluyen la consolidación de los mismos en un repositorio y en un formato visual navegable. Otras nuevas capacidades incluyen la administración avanzada de sub-modelos y versiones mejoradas de control.

Integración de datos y soporte para el almacenamiento de información mejorada. Trae nuevas capacidades para la documentación lineal de la información, de tal forma que los usuarios pueden rastrear los múltiples niveles de la información lineal mediante Sistemas de Procesamiento de Transacciones Online (OLTP), almacenamiento de información empresarial y mercados de datos.

Presenta un Modelado Físico Mejorado, para soportar la transición de lógico a físico, ER/Studio 7.0 provee a los administradores de base de datos dos nuevas características: capacidad de planificación y modelado seguro. La capacidad de planificación permite a los diseñadores comunicar la forma en que esperan que la base de datos crezca, y la funcionalidad de modelado seguro permite a los administradores de la información desarrollar roles de acceso en el modelo lógico, que pueden ser transferidos al modelo físico (FinancialTech, 2005)

2.5. Plataformas de desarrollo.

Una plataforma de desarrollo es el entorno común en el cual se desenvuelve la programación de un grupo definido de aplicaciones. Comúnmente se encuentra relacionada directamente a un sistema operativo, sin embargo, también es posible encontrarlas ligadas a una familia de lenguajes de programación o a una Interfaz de programación de aplicaciones (API).
Las mismas han de ofrecer un conjunto de servicios a los arquitectos y desarrolladores que ayuden a facilitar el desarrollo de aplicaciones, al tiempo que deben ofrecer la mayor cantidad posible de funcionalidades a los usuarios.

2.5.1. Plataforma de desarrollo .NET

La Plataforma de Desarrollo .NET, es un proyecto de Microsoft diseñado para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma y que permita un rápido desarrollo de aplicaciones.

Con esta plataforma Microsoft incursiona en el campo de los Servicios Web y establece el XML como norma en el transporte de información en sus productos y lo promociona como tal en los sistemas desarrollados utilizando sus herramientas, la plataforma .NET intenta ofrecer una manera rápida y económica pero a la vez segura y robusta de desarrollar aplicaciones permitiendo a su vez una integración más rápida y ágil entre empresas y un acceso más simple y universal a todo tipo de información desde cualquier tipo de dispositivo.

La base de la plataforma.NET la constituye el "framework" o marco de trabajo, y este 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.

Los principales componentes del marco de trabajo son:

- El conjunto de lenguajes de programación.
- El Entorno Común de Ejecución para Lenguajes o CLR por sus siglas en inglés.
- La Biblioteca de Clases Base o BCL.

El CLR es el verdadero núcleo del Framework de .NET, entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios del sistema operativo (González, Marzo 2007)

Es responsable de los servicios en tiempo de ejecución como la integración de lenguajes, la aplicación de seguridad y la administración de la memoria, los procesos y los subprocesos. Además, juega un importante papel en tiempo de desarrollo, pues contiene un conjunto de características como la administración de la duración, la aplicación de nombres de tipos seguros, el control de excepciones entre lenguajes, la creación de enlaces dinámicos, etc.

La Librería de Clase Base (BCL) es una librería incluida en el .NET Framework formada por cientos de tipos de datos que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente usadas a la hora de escribir programas (González, Marzo 2007).

2.5.2. Plataforma de desarrollo Java.

La plataforma Java es el nombre de un entorno o plataforma de computación originaria de Sun Microsystems, capaz de ejecutar aplicaciones desarrolladas usando el lenguaje de programación Java y un conjunto de herramientas de desarrollo. En este caso, la plataforma no es un hardware específico o un sistema operativo, sino más bien una máquina virtual encargada de la ejecución, y un conjunto de librerías estándar que ofrecen funcionalidad común.

La plataforma es así llamada la Plataforma Java (antes conocida como Plataforma Java 2), e incluye:

- Plataforma Java, Edición Estándar (Java Platform, Standard Edition), o Java SE (antes J2SE).
- Plataforma Java, Edición Empresa (Java Platform, Enterprise Edition), o Java EE (antes J2EE).
- Plataforma Java, Edición Micro (Java Platform, Micro Edition), o Java ME (antes J2ME).

Un programa destinado a la Plataforma Java necesita dos componentes en el sistema donde se va a ejecutar: una máquina virtual de Java (JVM), y un conjunto de librerías para proporcionar los servicios que pueda necesitar la aplicación. La JVM, junto con su implementación de las librerías estándar, se conocen como Java Runtime Environment (JRE). El JRE es lo mínimo que debe contener un sistema para poder ejecutar una aplicación Java sobre el mismo. Para el desarrollo de programas se ofrece un paquete de utilidades y herramientas conocido como JSDK (Java Software Development Kit).

El corazón de la Plataforma Java es el concepto común de un procesador “virtual” que ejecuta programas escritos en el lenguaje de programación Java. En concreto, ejecuta el código resultante de la compilación del código fuente, conocido como bytecode. Este “procesador” es la máquina virtual de Java o JVM, que se encarga de traducir (interpretar o compilar al vuelo) el bytecode en instrucciones nativas de la plataforma destino. Esto permite que una misma aplicación Java pueda ser ejecutada en una gran variedad de sistemas con arquitecturas distintas.

La Plataforma Java está pensada para ser independiente del sistema operativo subyacente, por lo que las aplicaciones no pueden apoyarse en funciones dependientes de cada sistema en concreto. Lo que hace la Plataforma Java, es ofrecer un conjunto de librerías estándar, que contiene mucha de las funciones reutilizables disponibles en los sistemas operativos actuales (Hernández, 2004).

2.6. ¿Qué se conoce como Framework?

Un Framework no es más que una estructura de software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la cual se le puede añadir las últimas piezas para construir una aplicación concreta.
Los objetivos principales que persigue un framework son: acelerar el proceso de desarrollo, reutilizar código ya existente y promover buenas prácticas de desarrollo como el uso de patrones. (Seco, 2001)

2.6.1. Framework NHibernate.

NHibernate es un framework de Object-Relational-Mapping open-source que resuelve en forma automática la persistencia de objetos de dominio .NET. NHibernate está basado en el popular framework open-source Hibernate surgido en la comunidad Java en el año 2002 (Red Hat, 2002).

Ventajas del Framework NHibernate:

- Código abierto.
- Soporta DataBinding.
- Puede aceptar consultas SQL directas.
- Se puede integrar con frameworks de MVC.
- Soporta las relaciones entre objetos.
- Soporta agrupamiento (GROUP BY).

- Soporta agregación (COUNT, AVG, ETC.).
- Soporta llaves primarias compuestas.
- Soporta asociaciones muchos a muchos y uno a muchos.
- Soporta persistencia de propiedades a través de los campos de propiedades.
- Soporta persistencia de propiedades a través de accessors (get/set methods or properties y pueden ser privados).
- Soporta trabajo offline y luego aplicar los cambios a la base de datos.
- Soporta WebServices (la tecnología más moderna para la Integración de aplicaciones web, y el paradigma de programación moderno de programación orientada a servicios, publicación de servicios).
- Soporta tipos nulos.
- No se requiere generar código pre compilado.

Desventajas:

- No soporta carga no transaccional lazy de relaciones.
- No soporta Aggregate Mappings - Single Un campo a muchos campos en la base de datos.
- No sorpota querying transparente a múltiples recursos de data.

2.6.2. Framework NUnit

NUnit es una herramienta utilizada para escribir y ejecutar pruebas en .NET, NUnit es un framework desarrollado en C# que ofrece las funcionalidades necesarias para implementar pruebas en un proyecto. Además provee una interfaz gráfica para ejecutar y administrar las mismas.

Se encarga de analizar ensamblados generados por .NET, interpretar las pruebas inmersas en ellos y ejecutarlas. Utiliza atributos personalizados para interpretar las pruebas y provee además métodos para implementarlas. En general, NUnit compara valores esperados y valores generados, si estos son diferentes la prueba no pasa, caso contrario la prueba es exitosa.

NUnit carga en su entorno un ensamblado y cada vez que lo ejecuta, o mejor, ejecuta las pruebas que contiene, lo recarga. Esto es útil porque se pueden tener ciclos de codificación y ejecución de pruebas simultáneamente, así cada vez que se compile no tiene que volver a cargar el ensamblado al entorno de NUnit si no que este siempre obtiene la última versión del mismo. Ofrece una interfaz simple que informa si una prueba o un conjunto de pruebas, falló, pasó o fue ignorada.

Basa su funcionamiento en dos aspectos, el primero es la utilización de atributos personalizados. Estos atributos le indican al framework de NUnit que debe hacer con determinado método o clase, es decir, estos atributos le indican a NUnit cómo interpretar y ejecutar las pruebas implementadas en el método o clase.

El segundo son las aserciones, que no son más que métodos del framework de NUnit utilizados para comprobar y comparar valores (Hunt & Thomas, 2004).

2.6.3. Framework Spring.NET

Spring.NET es un framework de código abierto (liberado bajo la Apache License 2.0) para el desarrollo de aplicaciones que facilita la construcción de aplicaciones empresariales en .NET. Suministrando componentes basados en patrones de diseño ya probados y que pueden ser integrados en todas las capas de la arquitectura de una aplicación, Spring.NET ayuda a incrementar la productividad y mejora la calidad y el desempeño de aplicaciones.

Surge como una variante del proyecto Spring Framework, orientado para aplicaciones Java/J2EE Spring Framework hizo que aquellas técnicas que resultaban desconocidas para la mayoría de los programadores se volvieran populares en un período muy corto de tiempo, el ejemplo más notable es la inyección de dependencia. Spring disfruta de unas altísimas tasas de adopción y al ofrecer su propio framework de AOP, ha conseguido hacer más popular su paradigma de programación en la comunidad Java.

La estructura arquitectónica de Spring.NET difiere de su predecesor, Spring para Java, y se concentra en ofrecer funcionalidades agrupadas en seis librerías:

- **Spring.Core:** Es la parte fundamental del framework, y provee la funcionalidad de la Inyección de Dependencia. Muchas de las librerías de Spring.NET dependen de este módulo y extienden sus funcionalidades.

- **Spring.Aop:** Provee soporte de Programación Orientada a Aspectos a objetos de negocio. Esta librería complementa el contenedor de Inyección de Dependencia de la librería Spring.Core para proveer de una base sólida para la construcción de aplicaciones empresariales y aplicar servicios a objetos de negocio.

- **Spring.Web:** Extiende a ASP.NET añadiendo una variedad de características como la Inyección de Dependencia para páginas ASPX, enlace de datos bidireccional, páginas principales para ASP.NET 1.1 y un mejorado soporte de localización (generación de contenido en base a la ubicación geográfica del usuario).

- **Spring.Services:** Deja exponer cualquier objeto "normal" (que no herede de una clase de servicio base) como si se tratase de un servicio empresarial (COM+) u objeto remoto. Los servicios Web de .NET consiguen una flexibilidad adicional al momento de configurarse con soporte para inyección de dependencia y sobre escritura de metadata de sus atributos. También se provee integración con Windows Service.

- **Spring.Data:** Provee una capa de abstracción de acceso a datos que puede ser usada con una variedad de proveedores de acceso a datos, desde ADO.NET a otros proveedores de Mapeo de Objetos a partir de bases de datos Relacionales. También contiene una capa de abstracción que elimina la necesidad de escribir código y declarar el manejo de transacciones para ADO.NET.

- **Spring.ORM:** Provee capas de integración con librerías populares de mapeo relacional de objetos (Object-Relational Mapping, ORM), como NHibernate e IBatis for .NET. Esto provee funcionalidades como el soporte para la gestión declarativa de transacciones. (The Spring Java Team, 2006).

2.7. Herramientas utilizadas para el desarrollo de este trabajo

A partir del estudio previo realizado se decidió emplear para el Sistema de Administración Financiera y por ende para el módulo de Recaudación la herramienta de modelado Enterprise Architect 7.0, pues proporciona ventajas para la gestión de cambios de los requisitos de software, facilidad de uso, acceso de varios usuarios a la misma información en un espacio de tiempo.

Como plataforma de desarrollo se emplea Microsoft Visual Studio .NET 2003 pues el Sistema de Administración Financiera forma parte de la Solución Informática SAREN, la cual en sus inicios se pactó con la parte cliente su implementación en software propietario.

NUnit 2.4.3 posibilitó llevar a cabo las pruebas de unidad al trabajo realizado, permitiendo comprobar la fiabilidad del mismo.

CAPÍTULO 3: DISEÑO E IMPLEMENTACIÓN.

El diseño de sistemas se ocupa de desarrollar las directrices propuestas durante el análisis con la intención de satisfacer los requisitos planteados tanto funcionales como no funcionales. El objetivo del presente capítulo es el diseño del sistema para la Recaudación de Tasas por Servicios Registrales y Notariales de la República de Venezuela, partiendo del estudio del arte realizado en el capitulo anterior de las herramientas case, plataformas y frameworks, se facilita la obtención de los artefactos generados por este flujo de trabajo. Se utiliza la metodología *Rational Unified Process(RUP)* y Enterprise Arquitect como herramienta de modelado, el cual se basa en el uso del UML como lenguaje de modelado. 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.

En el desarrollo de este capítulo se exponen una serie de diagramas correspondientes al diseño del sistema para la Recaudación de Tasas.

3.1. Modelo del diseño.

El modelo de diseño es un modelo de objetos que describe la realización física de los casos de uso centrándose en cómo los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar. Como parte del modelo de diseño del proceso de Recaudación de Tasas, se realizaron los casos de usos esenciales según su alto valor para la arquitectura, representados a través de los diagramas de interacción y diagramas de clases, estos diagramas fueron agrupados en pequeños grupos equivalentes a los casos de usos significativos arquitectónicamente identificados, pues constituyen las funciones fundamentales del sistema:

- Diferencias por sucursal bancaria.
- Diferencias por oficina.
- Gestionar Reintegros.
- Recaudar Fondos por Abandono.

Para dar soporte al diseño de este sistema, existe una arquitectura definida, de la cual se expondrán algunos elementos importantes a tener en cuenta para la realización del diseño. A continuación una breve explicación de cómo estarán separadas las capas de diseño.

- **Capa de diseño.**

En una arquitectura de n niveles como la presente, las acciones de la aplicación están lógicamente divididas por funciones. Lo anterior se basa en el patrón multicapa como se puede ver en la figura:

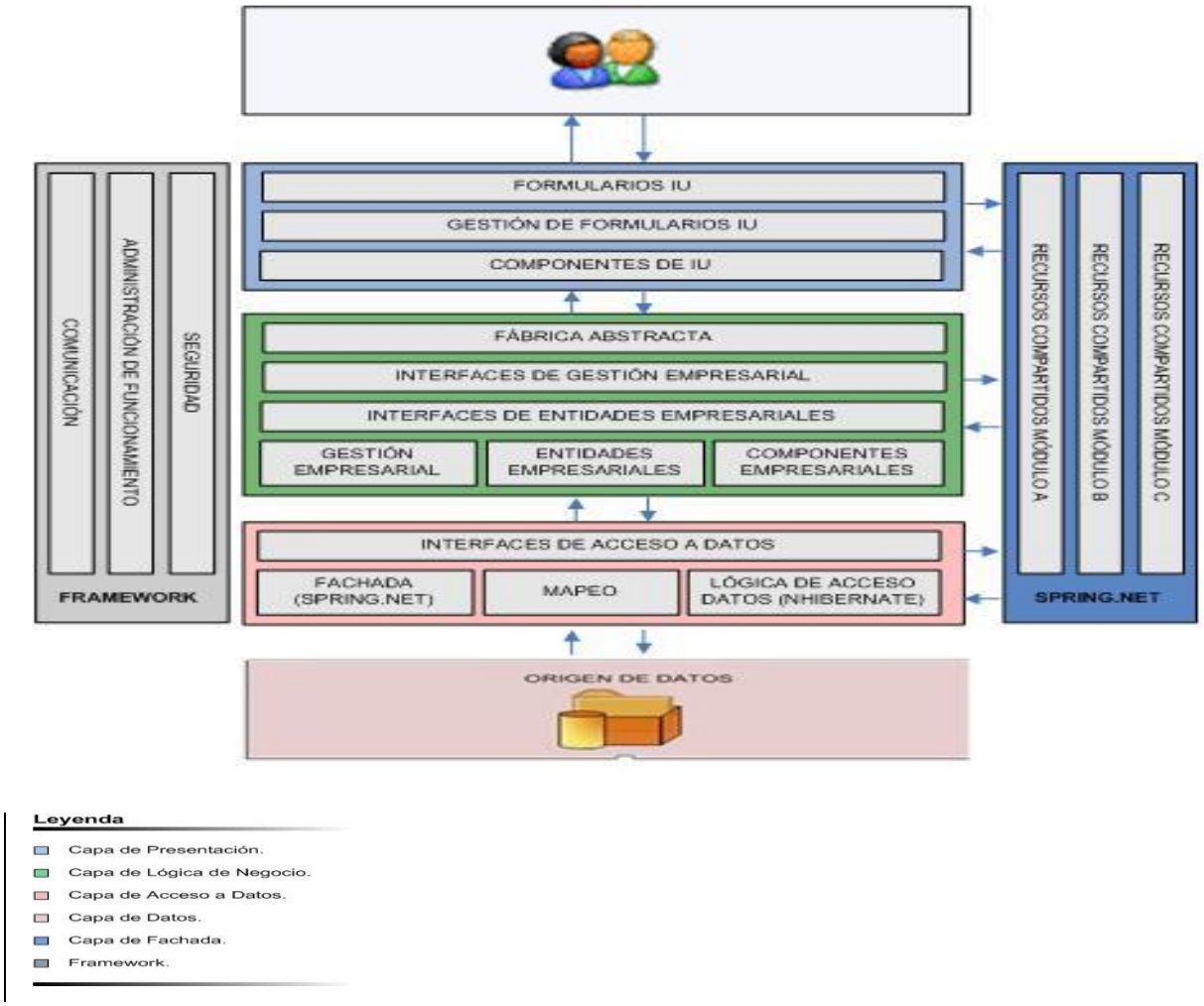


Figura 6 Representación del enfoque de la Arquitectura.

De manera general lo más representativo de esta distribución es la fuerte presencia que tienen los componentes y más aún los conectores, estos últimos resueltos en su mayoría con la implementación del Patrón Fachada, Proxy y la utilización del framework Spring.net. A continuación se detallan cada una de las capas y los componentes utilizados:

- **Capa de Presentación:**

Formularios de Interfaz de Usuario.

Los formularios responden a un comportamiento y diseño estándar, todos los módulos tienen la misma forma con vistas a facilitar el trabajo con ellos y la estandarización del sistema completo.

Su uso se basa en la explotación de los Formularios de Interfaz de Usuario que realmente provee un framework (Framework Común) facilitando el desarrollo del sistema. Este framework se basa en acciones contempladas en la Gestión de Formularios de Interfaz de Usuario y cada acción se corresponde con funcionalidades de captura o visualización de los datos que tiene asociado un formulario cuya forma depende de la funcionalidad específica para la cual fue concebida dicha acción.

Gestión de Formularios de Interfaz de Usuario.

Su uso se basa en la explotación de la componente gestión de interfaz que realmente provee el framework antes mencionado facilitando el desarrollo del sistema basado en acciones.

¿Qué es una acción?

Cada acción debe tener sentido semántico propio y completo. Deben ser atómicas.

Las acciones básicamente definen un bloque de código reusable por varias operaciones sobre el sistema.

Las acciones pueden estar asociadas a vistas del sistema.

Cada acción puede representar un estado del sistema que puede o no persistir.

Una acción es una clase que representa precisamente la ejecución de alguna tarea concreta. Estas tareas no deben ser excesivamente complejas y la clase debe: Heredar de la clase **“Acción”** o “**Acción Segura”** (Framework Común).

Opcionalmente tener una forma visual asociada.

Propiedades en función del objetivo específico de la misma.

En caso de que la acción tenga asignada una forma visual, se deben programar dentro de la misma todos los eventos que puedan ocurrir y que se deseen utilizar a partir de la interacción con la forma visual.

Componentes de Interfaz de Usuarios.

Los componentes de interfaz de usuarios son recursos utilizados para encapsular una función determinada y serán utilizados por más de un proceso en el módulo. El hecho de que estén en la Capa de Presentación se debe a que trabajan con los formularios directamente, ya sea de forma visual o con los datos que se encuentran relacionados en el mismo a través de otros componentes o controles. Aquí pueden encontrarse clases de negocio e inclusive elementos de acceso a datos.

Como se puede ver esta capa implementa el patrón Modelo Vista Controlador (MVC) puesto que se separa la vista (Formularios de Interfaz de Usuario) de sus funcionalidades, las cuales radican en las Acciones (Gestión de Formularios de interfaz de Usuario) que juegan el papel de los controladores, el caso del modelo se puede ver en el trabajo que realiza el framework (Framework Común) con los datos que captura la vista y procesan los controladores para pasarlos al negocio (Capa Lógica de Negocio).

- **Capa Lógica de Negocio**

El Negocio recibe datos y/o información capturada en las interfaces de usuario, gestiona o procesa la misma, de ser necesario solicitándola a la capa de Acceso a Datos y finalmente enviársela a la Presentación nuevamente para que ésta la presente al usuario en el punto donde se inició la petición.

Entidades del Negocio.

Las entidades empresariales son clases objeto - valor que representan los datos con los que se van a trabajar en cada uno de los procesos que se están automatizando.

Cada entidad es un elemento auto sustentado, es decir, se encarga de procesar sus propios datos o valores sin interactuar con los demás elementos del negocio, con esto se garantiza la independencia y el encapsulamiento de la información según la competencia que se tenga sobre la misma.

Gestión Empresarial.

Tiene como objetivo agrupar una serie de entidades con un fin común, lo anterior significa resolver determinadas funcionalidades donde intervienen una serie de datos que se encuentran en dichas clases objeto - valor. Las clases correspondientes a las funcionalidades se denominan Gestores.

Para la implementación de los Gestores se propone la implementación del patrón Singleton, pues se van a manejar clases que solamente presentan funcionalidades, por tanto teniendo una única instancia de las mismas, que inicialice las propiedades que necesita el Gestor para trabajar, se resuelve la ejecución efectiva de cada una de ellas.

Componentes Empresariales.

Los componentes de negocio son recursos utilizados para encapsular una función determinada que será utilizada por más de un proceso de negocio en el módulo. Lógicamente estas funcionalidades son netamente de este nivel, es decir no tienen presentación pero si pueden utilizar recursos del acceso a datos.

Interfaces de Gestión y Entidades.

Como se ha observando, el pilar fundamental es el uso de las interfaces como recurso que se utiliza para brindar funcionalidades que representan un nivel de abstracción. Este nivel es precisamente el que asegura el patrón Bajo Acoplamiento conjuntamente con otros elementos, algunos de los cuales ya se han visto.

La mayoría de las interfaces de la aplicación van a estar precisamente en el Negocio, pues aquí se encuentran en gran medida la implementación de estas funcionalidades, ya sea en las Entidades, en los Gestores o en los Componentes de Gestión.

- **Capa Acceso a Datos**

El Acceso a Datos es la capa más crítica y sensible a cambios, pues controla todo lo concerniente a la información que se encuentra en la fuente de almacenamiento (Capa de Datos).

Al ser la capa inferior no conoce los niveles superiores, únicamente se limita al manejo de la información, ya sea para persistirla o proporcionarla para su procesamiento y propagación por la

aplicación. Todo este manejo es responsabilidad de los Objetos de Acceso a Datos (DAOs por sus siglas en ingles).

Lógica de Acceso a Datos.

Aquí recaen todas las funcionalidades del Acceso a Datos en los DAOs, éste ensamblado presenta la totalidad de las operaciones de persistencia y obtención de datos explotando los recursos que brinda NHibernate framework que cumple perfectamente con el objetivo de este nivel, dígase trabajo con procedimientos almacenados y métodos de persistencia o consultas.

Fachada.

La fachada brinda una interfaz de alto nivel con la cual se va a interactuar cada vez que se necesite comunicarse, en este caso con el Acceso a Datos logrando una completa enajenación ante cualquier modificación que pueda ocurrir en esta.

Esta estructura responde a la implementación de patrón Fachada que está constituida por una clase y una estructura que enumera (*enum*) los DAOs a los cuales se puede acceder desde el negocio. Para su implementación se decidió el uso de Spring.Net versión 1.1, framework.

Mapeo.

Este elemento es de uso exclusivo de NHibernate, se decidió aislarlo en un ensamblado ya que básicamente son ficheros XML de configuración que son independientes de cualquier implementación, por tanto resulta muy útil tenerlos separados de todo código y así se evita recompilar toda una capa ante cualquier modificación en una de estas configuraciones.

Interfaces de Acceso a Datos.

Representan las funcionalidades que brinda este nivel, es decir las referidas a los DAOs. Su uso e importancia es la misma que las de la capa Lógica de Negocio.

- **Capa de Datos**

Esta capa corresponde a los almacenes de datos. A ella pertenecen las bases de datos disponibles en los servidores de bases de datos.

- **Capa Fachada**

La Capa de Fachada es una representación del patrón Proxy y Fachada a gran escala, este nivel es un recurso utilizado para mantener una representación de los demás módulos en el que se está implementando, siempre y cuando lo necesite. Es decir, cada módulo que se vaya a comunicar con el otro debe dar las funcionalidades que necesita (ver figura 4), las cuales se van a agrupar aquí. Esta distribución tiene su utilidad en el enfoque horizontal de la Arquitectura. Al emplear esta forma de comunicación entre los módulos se garantiza la abstracción y enajenación gracias a Spring.net, el cual ya se ha mencionado con anterioridad.

3.2 Diagramas de Clases.

Otro artefacto generado dentro del flujo de trabajo diseño es el diagrama de clase, que describe gráficamente las especificaciones de las clases de software y de las interfaces en una aplicación. Normalmente contiene clases, asociaciones y atributos, interfaces, con sus operaciones y constantes, métodos, información sobre los tipos de los atributos, navegabilidad y dependencias. Un diagrama de este tipo contiene las definiciones de las entidades del software. (Larman, 1999)

3.2.1 Diferencias por Sucursal Bancaria.

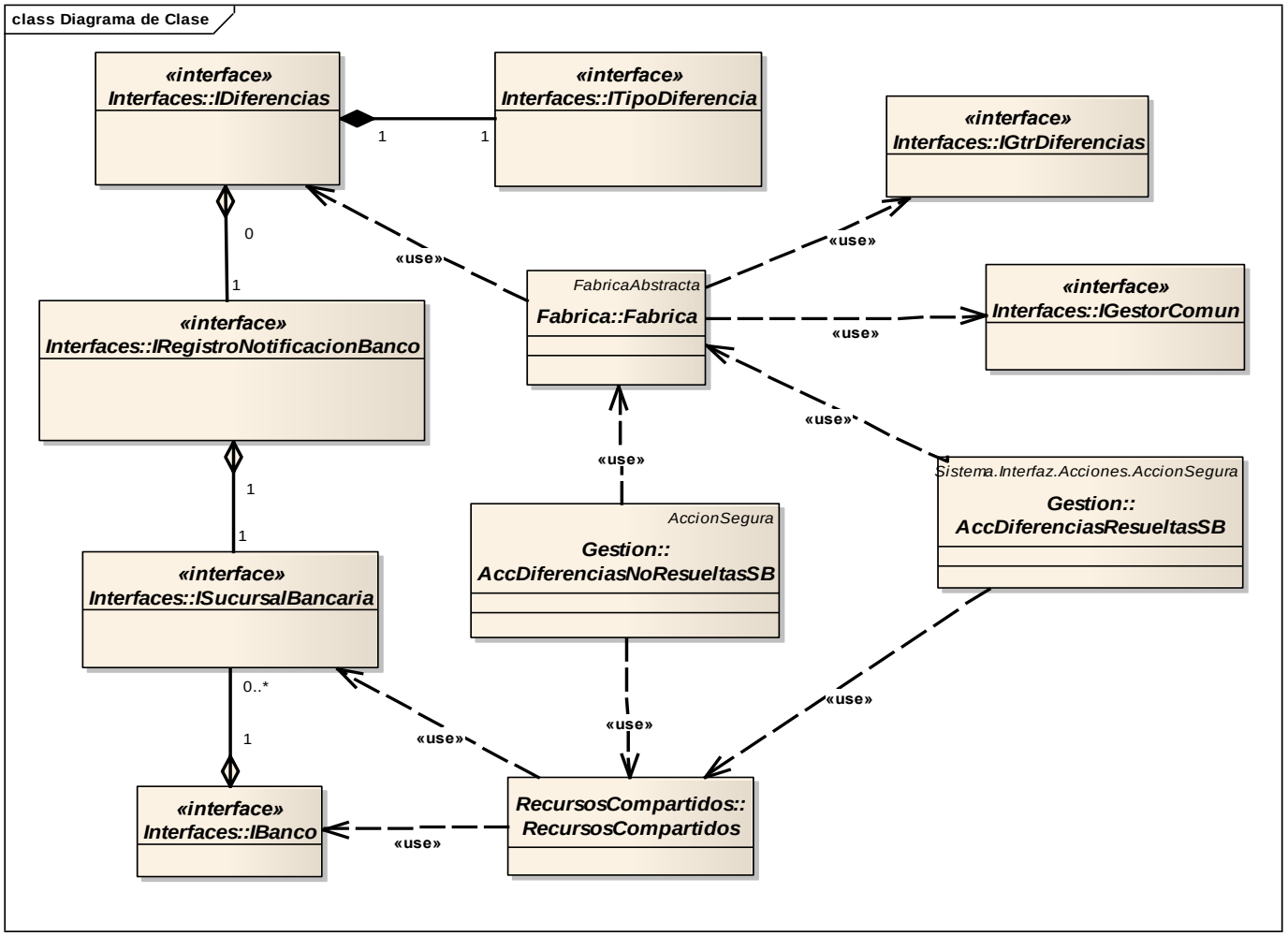


Figura 7. Diagrama de clases caso de uso Diferencias por Sucursal Bancaria.

IDiferencia: Posee las propiedades referentes a los datos asociados a las diferencias detectadas por medio de las notificaciones que son enviadas desde la entidad bancaria hacia SAREN.

IGtrDiferencias: Posee las operaciones necesarias para la gestión de las diferencias existentes tanto por sucursal bancaria como por oficinas, dígase búsquedas, cambio de estado a Resueltas o No Resueltas.

IGestorComun: Posee todas las operaciones comunes para la gestión de todas las entidades del negocio, dígase registro, actualización, eliminación y búsquedas genéricas.

Fábrica: Implementa el patrón de creación abstract factory, esta entidad es la encargada de la creación de las instancias tanto de entidades del negocio como de los gestores.

Recursos compartidos: Tiene la responsabilidad de instanciar los objetos pertenecientes a otros subsistemas que interactúan con el módulo de Recaudación.

AccDiferenciasResueltasSB: Es la encargada de manejar el flujo de eventos del formulario relacionado con las diferencias por sucursal bancaria.

3.2.2 Diferencias por oficina.

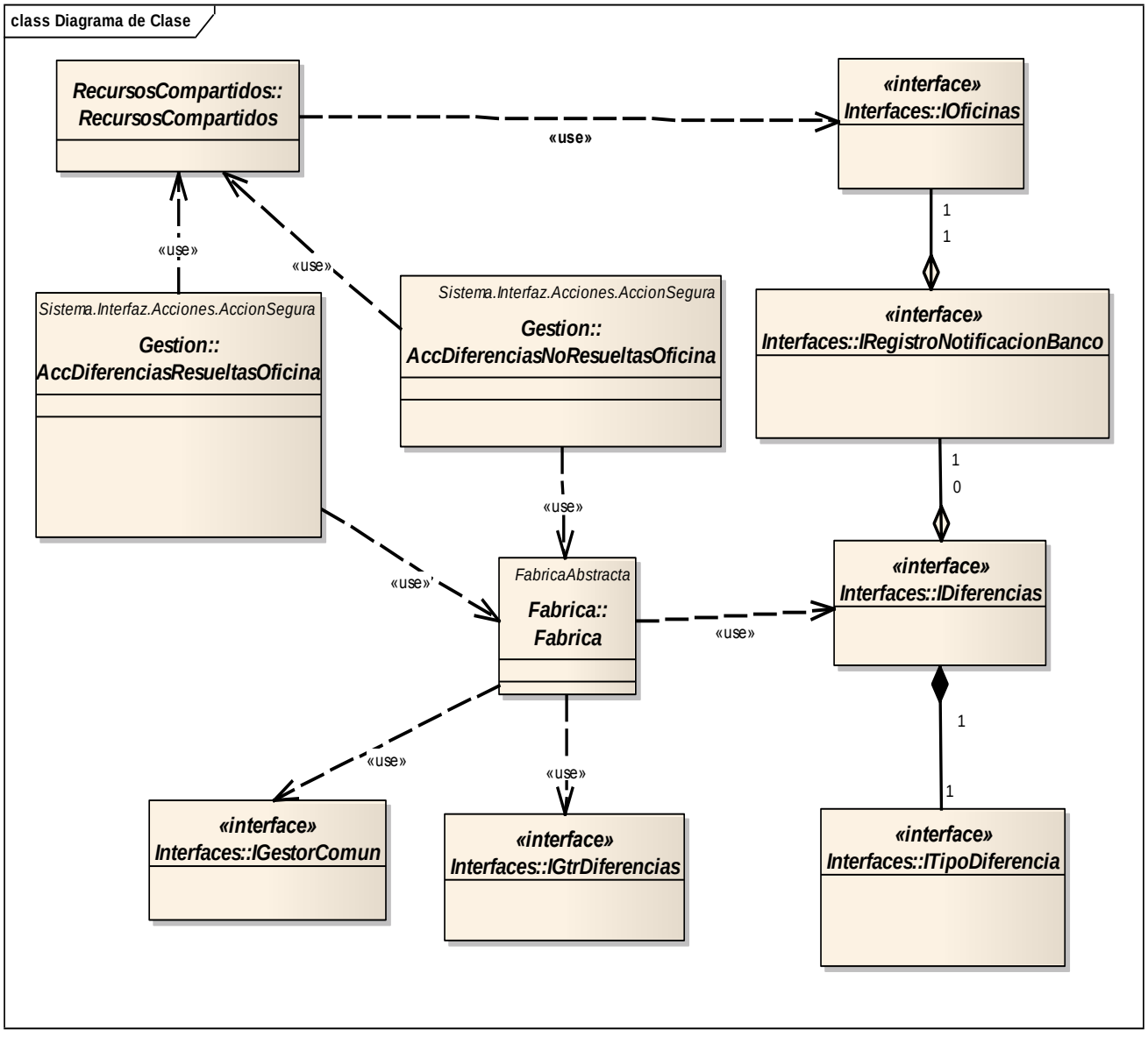


Figura 8. Diagrama de clases caso de uso Diferencias por Oficina.

AccDiferenciasResueltasOficina: Es la encargada de manejar el flujo de eventos del formulario relacionado con las diferencias por oficinas.

3.2.3 Gestionar Reintegros.

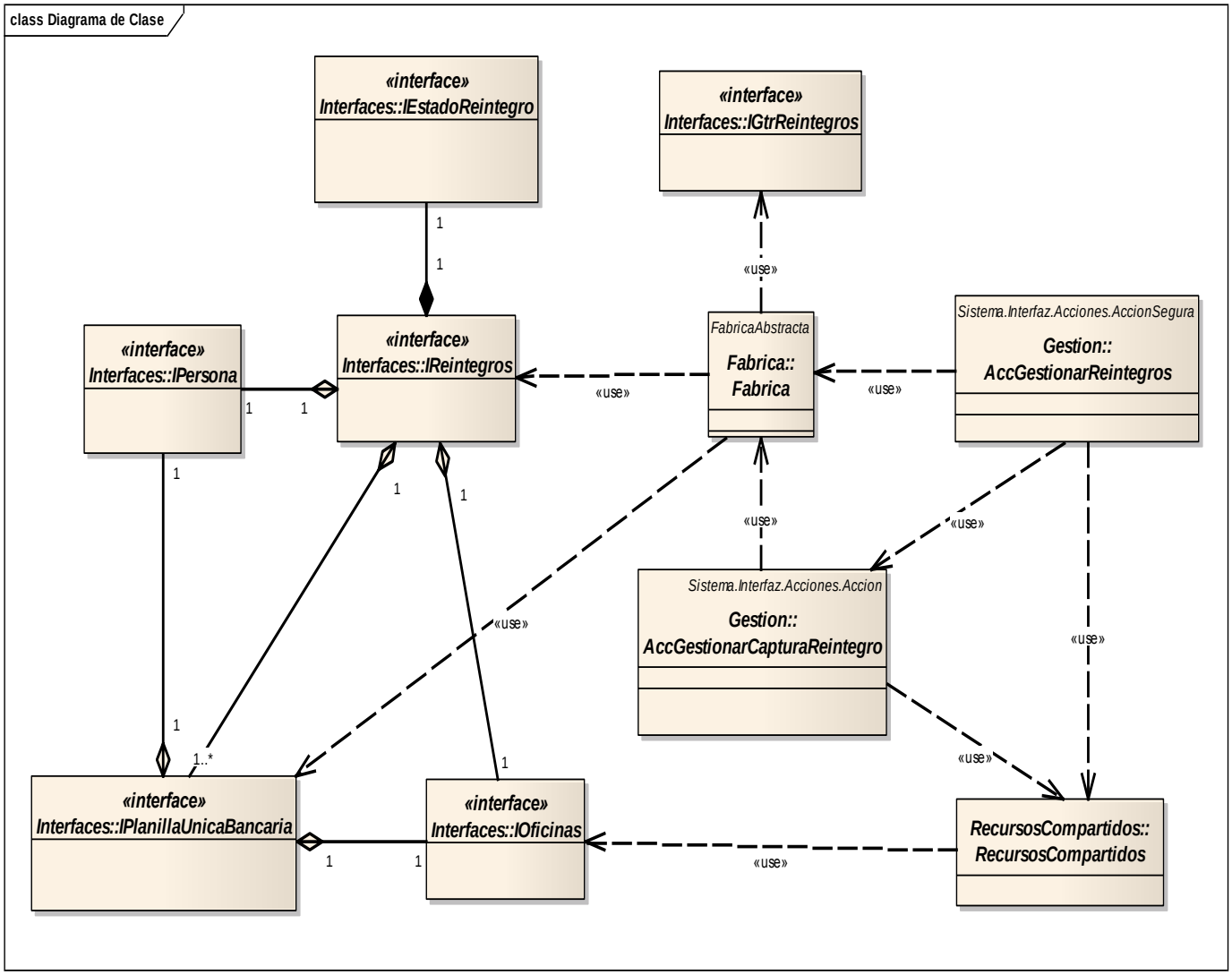


Figura 9. Diagrama de clases caso de uso Gestionar Reintegros.

IPlanillaUnicaBancaria: Posee las propiedades necesarias para la manipulación de los datos referentes a las Planillas Únicas Bancarias asociadas al reintegro.

IReintegro: Posee las propiedades necesarias para la manipulación de los datos referentes a los reintegros de las planillas.

AccGestionarRintegros: Acción encargada de la gestión de los reintegros, dígase búsqueda, registro, actualización y eliminación.

AccCapturaReintegro: Acción encargada de la captura y actualización de los reintegros.

3.2.4 Recaudar Fondos por Abandono.

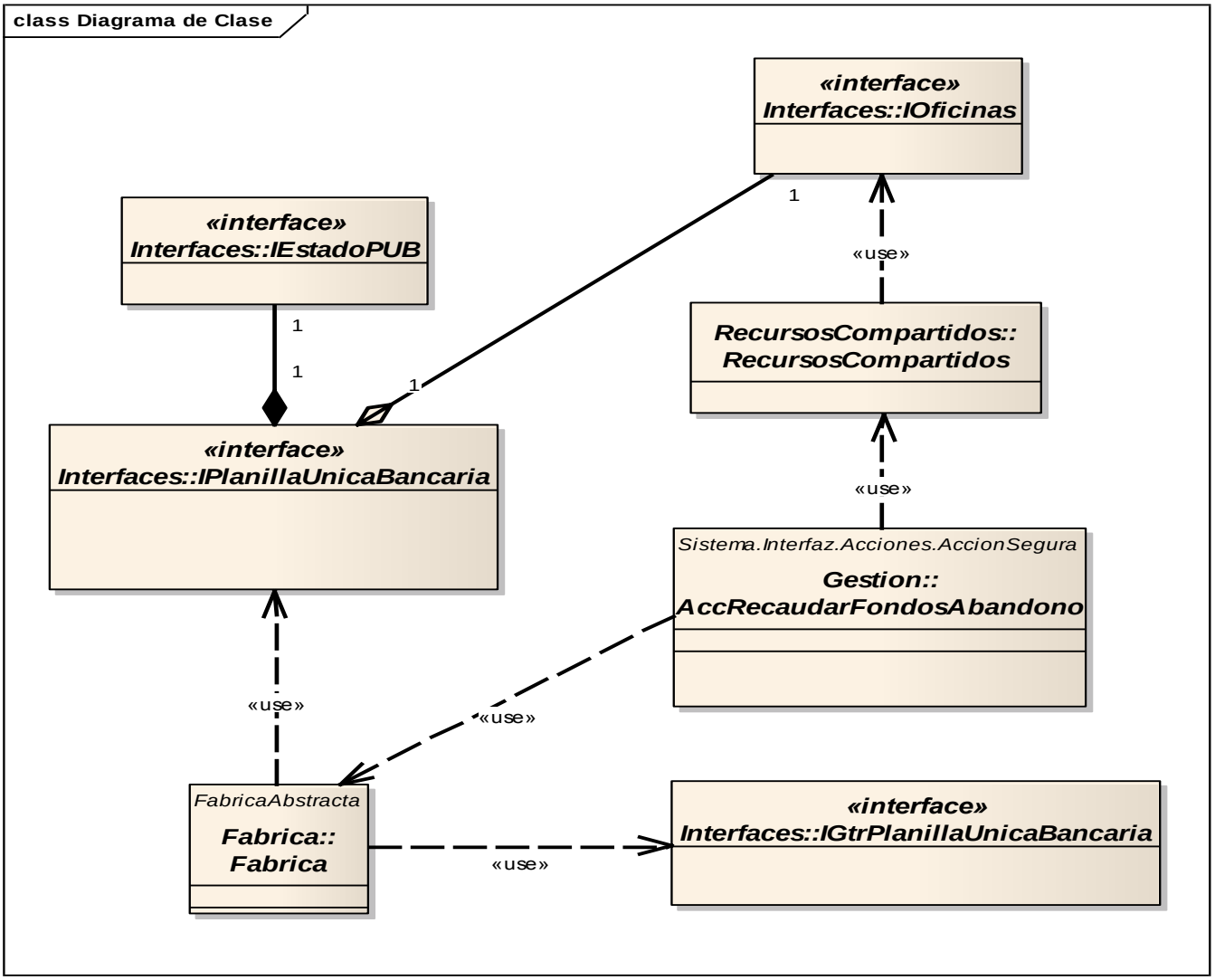


Figura 10. Diagrama de clases caso de uso Recaudar Fondos por Abandono.

IGtrPlanillaUnicaBancaria: Posee las operaciones necesarias para la gestión de las planillas únicas bancarias, por ejemplo cambio de estado de la planilla.

AccRecaudarFondosAbandono: Acción encargada de la recaudación de fondos por abandono de la planilla.

3.3. Diagramas de Interacción.

Como artefacto generado dentro del flujo de trabajo diseño existen los diagramas de interacción que explican gráficamente las interacciones existentes entre las instancias (y las clases) del modelo de estas. El UML define dos tipos de estos diagramas; ambos sirven para expresar interacciones semejantes o idénticas de mensaje:

- Diagramas de colaboración.
- Diagramas de secuencia.

Los Diagramas de colaboración describen las interacciones entre los objetos en un formato de grafo o red y los diagramas de secuencia describen las interacciones en una especie de formato de cerca o muro, que a su vez muestra en determinado escenario de un caso de uso[8], los eventos generados por actores externos, su orden y los eventos internos del sistema. A todos los sistemas se les trata como una caja negra; los diagramas se centran en los eventos que trascienden las fronteras del sistema y que fluyen de los actores a los sistemas. (Larman, 1999)

De esta forma se muestran a continuación los diagramas de secuencia correspondientes a los CU antes mencionados como arquitectónicamente significativos, para ello solo se muestran los principales escenarios que los integran.

[8] **Caso de Uso:** El escenario de un caso de uso es una instancia o trayectoria realizada por medio del uso: un ejemplo real de su ejecución.

3.3.1 Diferencias por Sucursal Bancaria.

Consiste en gestionar las diferencias existentes por Sucursal Bancaria dándole solución a las mismas. A continuación se muestra el escenario donde se realiza la búsqueda de las diferencias teniendo en cuenta varios parámetros, como Banco, Sucursal y fecha. Ver Anexo 1 para el resto de los diagramas correspondientes al caso de uso.

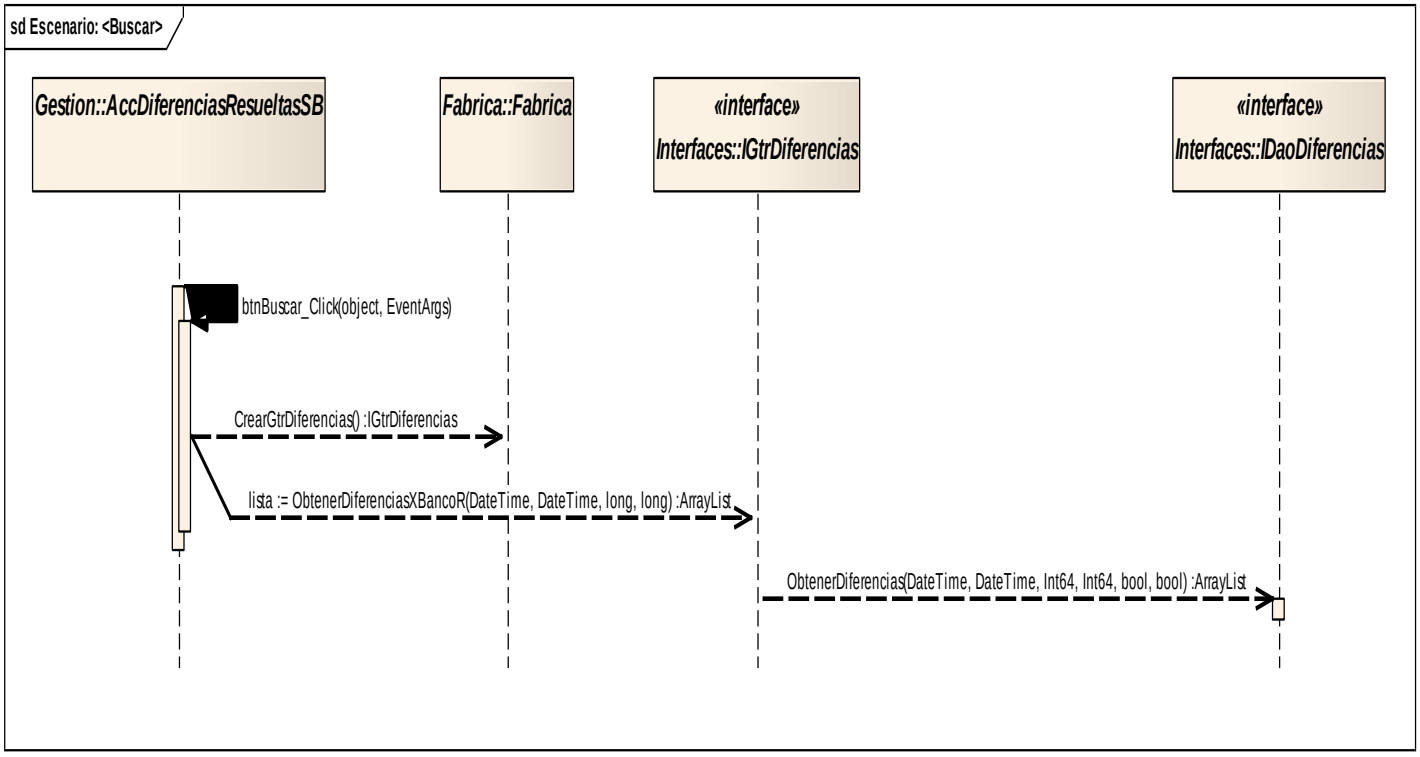


Figura 11 Escenario Buscar diferencias por Sucursal Bancaria.

3.3.2 Diferencias por oficina.

Consiste en gestionar las diferencias existentes por Oficinas dándole solución a las mismas. A continuación se muestra el escenario donde se realiza la búsqueda de las diferencias teniendo en cuenta varios parámetros, como Oficina y fecha. Ver Anexo 2 para el resto de los diagramas correspondientes al caso de uso.

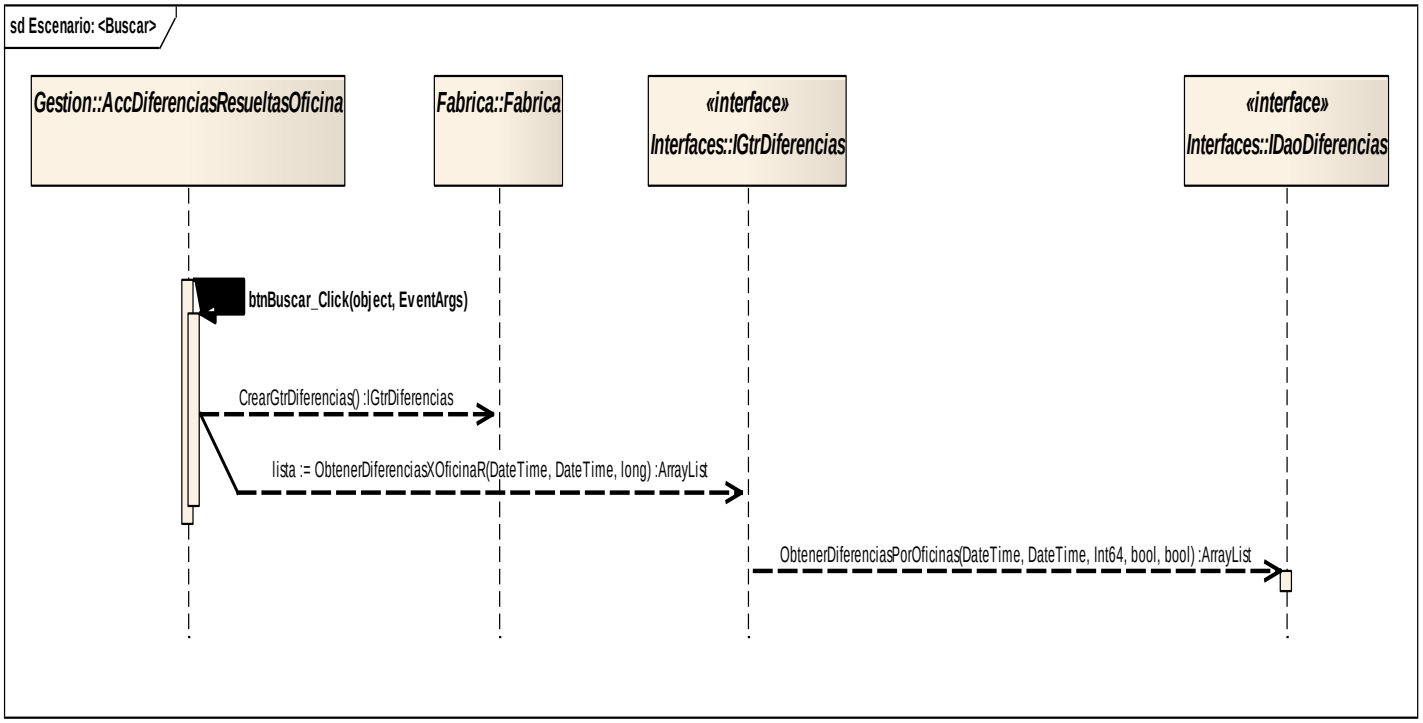


Figura 12 Escenario Buscar Diferencias por Oficina.

3.3.3 *Gestionar Reintegros.*

Consiste en gestionar los reintegros presentados, una vez realizado el pago de la Planilla Única Bancaria. A continuación se muestra el escenario para la creación y modificación de los mismos. Ver Anexo 3 para el resto de los diagramas correspondientes al caso de uso.

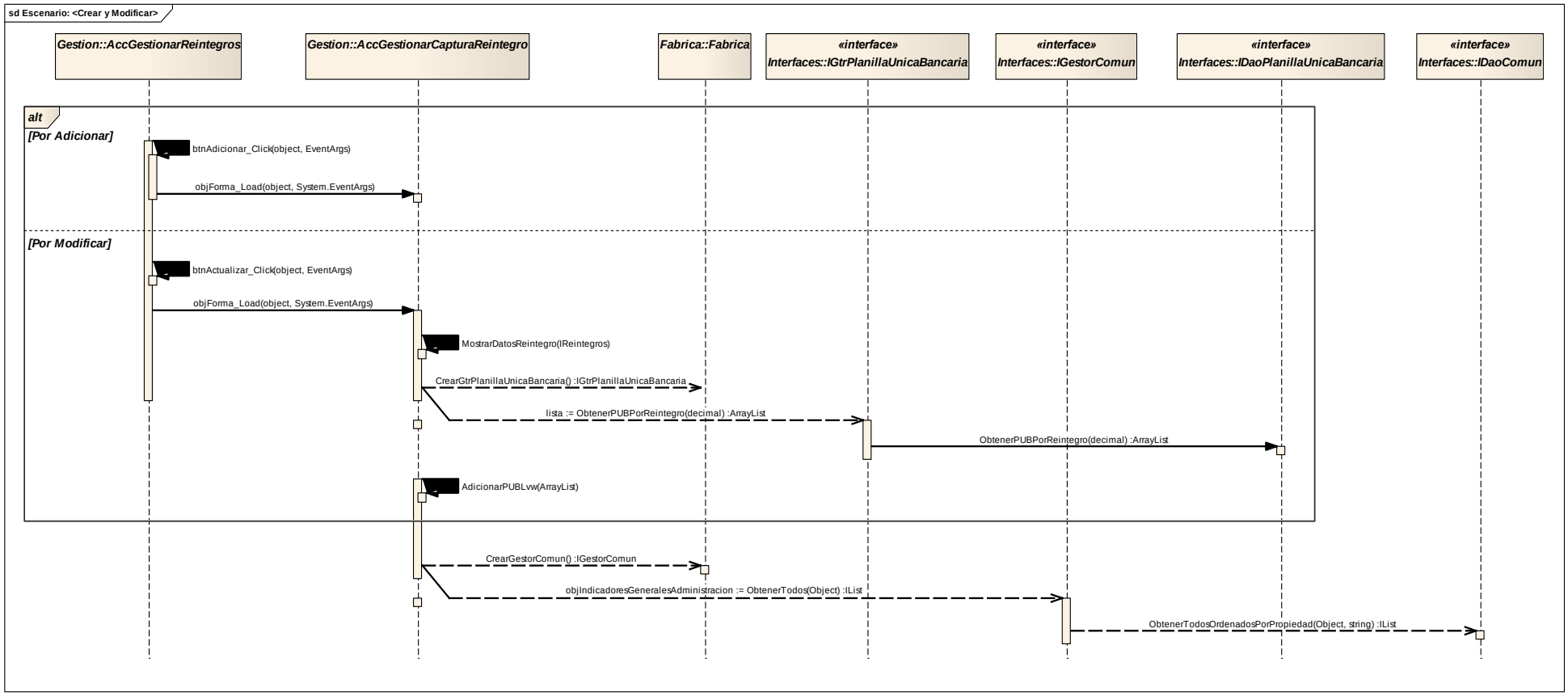


Figura 13 Escenario Gestión Crear y Modificar.

3.3.4 Recaudar Fondos por Abandono.

Consiste en mostrar las Planillas Únicas Bancarias por oficina que están pagadas, ha culminado el tiempo de realizar el trámite y de reclamar el monto depositado. A continuación se muestra el escenario para la recaudación de las Planillas Únicas Bancarias seleccionadas. Ver Anexo 4 para el resto de los diagramas correspondientes al caso de uso.

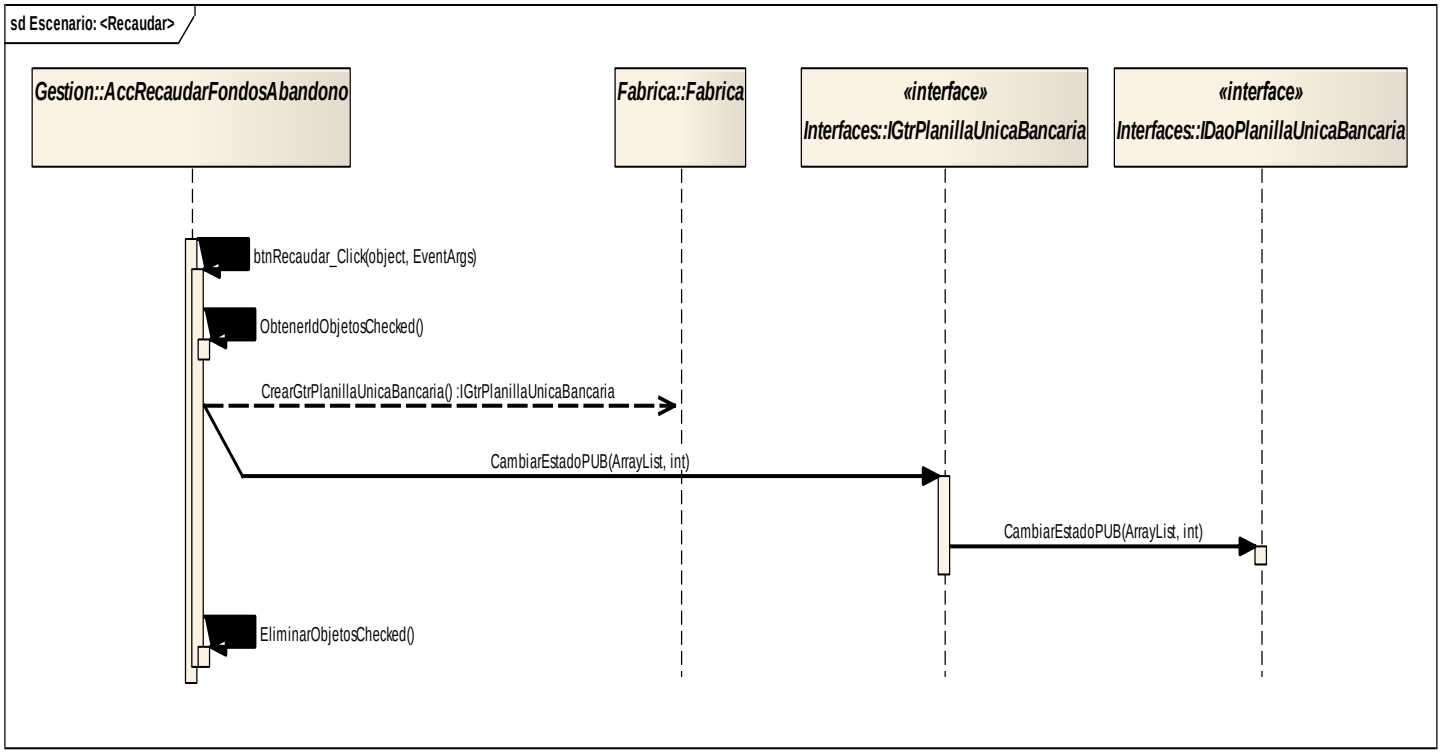


Figura 14. Escenario Recaudar.

Para ver los diagramas de clases e interacción de los restantes Casos de Uso del módulo de Recaudación consultar los siguientes anexos:

- Caso de Uso Reporte Consolidado de las Sucursales Bancarias ver Anexo 5.
- Caso de Uso Reporte Consolidado de los Bancos ver Anexo 6.
- Caso de Uso Reporte Consolidado por Estatus ver Anexo 7.
- Caso de Uso Reporte de Planilla Única Bancaria por Estatus ver Anexo 8.
- Caso de Uso Reporte de Planilla Única Bancaria por Tipo de Ingreso ver Anexo 9.

3.4 Modelo de Datos.

Un modelo de datos no es más que la representación de un fenómeno de la realidad objetiva a través de los objetos, sus propiedades y las relaciones que se establecen entre ello (Mato Garcia, 1999). El modelo de datos creado para este sistema es una traza directa del diseño de clases del negocio, este describe de una forma abstracta como están representadas físicamente las entidades de negocio en una base de datos relacional. Consultar este modelo en el Anexo 10.

3.5 Modelo de Implementación.

Un diagrama de implementación muestra las dependencias entre las partes de código del sistema (diagrama de componentes) o la estructura del sistema en ejecución (diagrama de despliegue): (Larman, 1999).

3.5.1 Diagrama de Componentes.

Los diagramas de componentes muestran las dependencias del compilador y del "runtime" entre los componentes del software; por ejemplo, los archivos del código fuente y las DLL. (Larman, 1999) A continuación se muestra el diagrama de componentes generado para el sistema desarrollado.

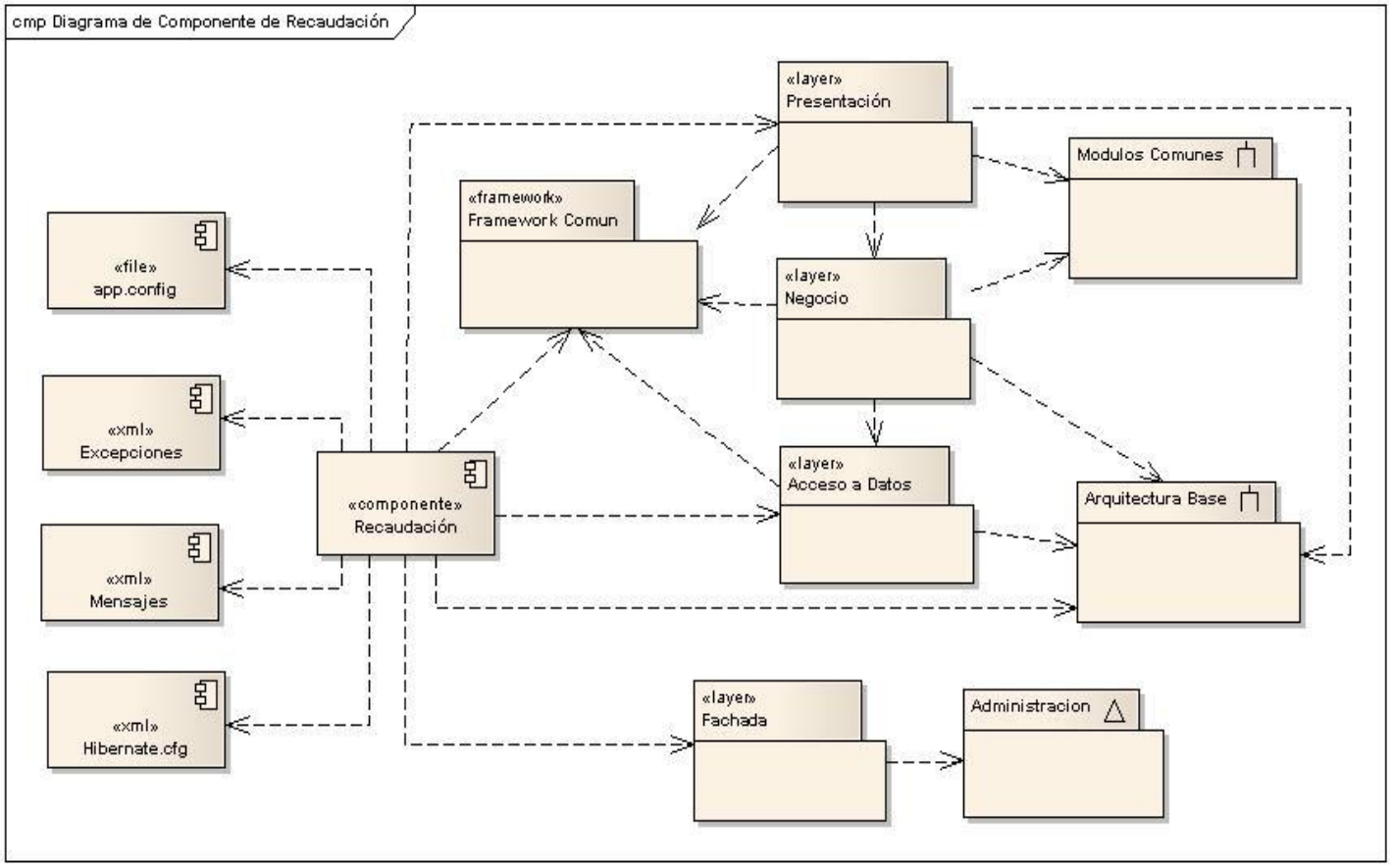


Figura 15. Diagrama de Componentes subsistema Recaudación.

3.5.2 Diagrama de Despliegue.

Los diagramas de despliegue muestran a los nodos procesadores, la distribución de los procesos y de los componentes (Larman, 1999). A continuación se muestra el diagrama de despliegue generado para la implantación de este subsistema en los Registros y Notarías de la República Bolivariana de Venezuela.

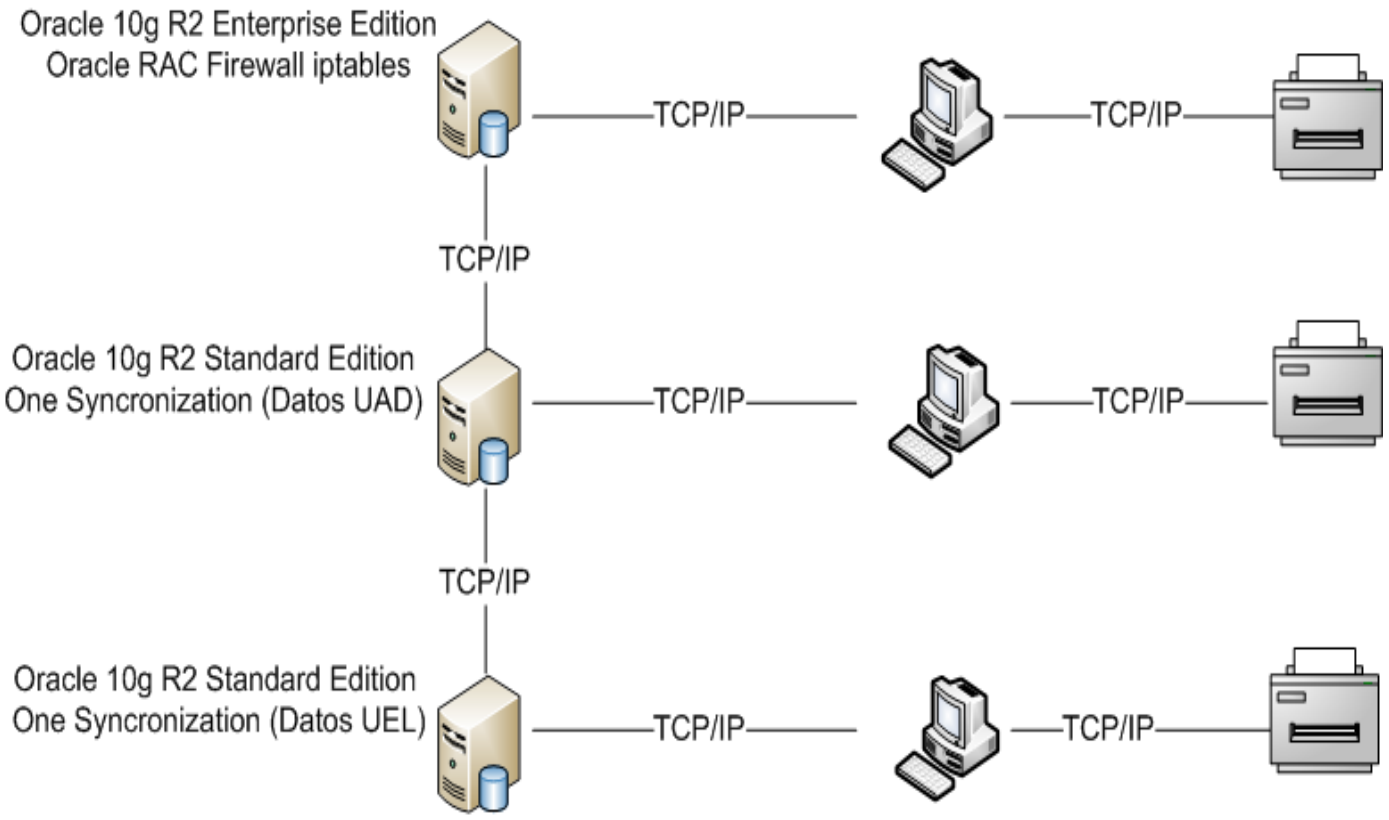


Figura 16 Diagrama de Despliegue subsistema Recaudación.

CAPÍTULO 4: ANÁLISIS DE LOS RESULTADOS

En este capítulo se evalúan los resultados obtenidos con el desarrollo de este trabajo, teniendo por objeto la aplicación de algunas de las métricas empleadas internacionalmente para tal fin, específicamente las asociadas a las de la medición de la calidad del diseño de software. Es preciso señalar que las métricas de diseño para el software, como otras métricas del software, no son perfectas. En la actualidad continúa el debate relacionado de cómo deberían aplicarse y la eficacia de estas; muchos expertos plantean que se necesita más experimentación.

4.1. Métricas del diseño arquitectónico.

Las métricas de diseño de alto nivel se concentran en las características de la arquitectura del programa, con especial énfasis en la estructura arquitectónica y en la eficiencia de los módulos. Estas métricas son de caja negra en el sentido que no requieren ningún conocimiento del trabajo interno de un módulo en particular del sistema.

Los expertos definen tres medidas de la complejidad del diseño del software: complejidad estructural, complejidad de datos y complejidad del sistema.

La complejidad estructural, *S*(*i*) , de un módulo i se define de la siguiente manera: $S(i) = f^2_{out}(i)$ donde *f* (*i*) *out* es la expansión del módulo i.

La complejidad de datos, *D*(*i*) , proporciona una indicación de la complejidad de la interfaz interna de un módulo i y se define como: $D(i) = V(i) / \left\lfloor f_{out}(i) + 1 \right\rfloor$ donde *V* (*i*) es el número de variable de entrada y de salida que entran y salen del módulo i.

La complejidad del sistema, *C* (*i*), se define como las sumas de las complejidades estructurales y de datos, y se define como: *C*(*i*) = *S*(*i*) + *D*(*i*).

A medida que crecen los valores de complejidad, la complejidad arquitectónica o global del sistema también aumenta. Esto lleva a una mayor probabilidad de que aumente el esfuerzo necesario para la integración y las pruebas. (Pressman, Vol I, 1998)

Métricas de diseño a nivel de componentes.

Las métricas de diseño a nivel de componentes se concentran en las características internas de los componentes del software e incluyen medidas de las “3Cs” (la cohesión, acoplamiento y complejidad) del módulo. Estas tres medidas pueden ayudar al desarrollador de software a juzgar la calidad de un diseño a nivel de los componentes. Estas métricas son de caja blanca en el sentido de que requieren conocimiento del trabajo interno del módulo en cuestión. Las métricas de diseño de los componentes se pueden aplicar una vez que se ha desarrollado un diseño procedimental. También se pueden retrasar hasta tener disponible el código fuente.

4.1.1. Métricas orientadas a clase.

Las medidas y métricas para una clase individual, la jerarquía de clases y las colaboraciones de clases poseen un valor incalculable, para el ingeniero del software que ha de evaluar la calidad del diseño. Las clases encapsulan operaciones (procesamiento) y atributos (datos).

Así mismo, la clase <<padre>> es de la que heredan las subclases (algunas veces llamadas hijas), sus atributos y operaciones. La clase, normalmente, colabora con otras clases. Cada una de estas características puede emplearse como base de la medición. Los expertos indican que es razonable declarar que todas las métricas proporcionan una visión útil para el ingeniero de software (Pressman, Vol I, 1998).

Algunas de las métricas que se han definido a nivel de clases por algunos expertos son:

La serie de métricas CK propuestas por Chidamber y Kemerer siendo estas uno de los conjuntos de métricas OO más ampliamente referenciadas, los autores han propuesto seis métricas basadas en clases para sistemas OO:

1. Métodos ponderados por clase (MPC).
2. Árbol de profundidad de herencia (APH).
3. Número de descendiente (NDD).
4. Acoplamiento entre clases objeto (ACO).
5. Respuesta para una clase (RPC).
6. Carencia de cohesión en los métodos (CCM).

Lorenz y Kidd en su libro sobre métricas OO, separan las métricas basadas en clases en cuatro amplias categorías:

- Tamaño.
- Herencia.
- Valores internos.
- Valores externos.

Las métricas orientadas al tamaño para las clases OO se centran en el recuento de atributos y operaciones para cada clase individual, y los valores promedio para el sistema OO como un todo. Las métricas basadas en la herencia se centran en la forma en que las operaciones se reutilizan en la jerarquía de clases. Las métricas para valores internos de clase examinan la cohesión y los aspectos orientados al código; las métricas orientadas a valores externos, examinan el acoplamiento y la reutilización.

1. Tamaño de clase **(TC).**
2. Número de operaciones redefinidas por una subclase **(NOR).**
3. Número de operaciones añadidas por una subclase **(NOA).**

4.2. Aplicación de las métricas en el subsistema Recaudación.

A continuación se aplican algunas de las métricas antes mencionadas, para determinar el grado de calidad y fiabilidad del diseño propuesto en este trabajo.

4.2.1. Métrica Tamaño de clase (TC) propuesta por Lorenz y Kidd.

Esta métrica consiste en medir el tamaño de una clase a partir de las siguientes medidas:

- 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 las dos métricas anteriores para el sistema en general.

Si existen valores grandes de TC, estos mostrarán que una clase puede tener demasiada responsabilidad, lo cual reducirá la reutilizabilidad de la clase y complicará la implementación y la comprobación, por otra

parte cuanto menor sea el valor medio para el tamaño, más probable es que las clases existentes dentro del sistema se puedan reutilizar ampliamente (Lorenz & Kidd, 1994).

La métrica **TC** fue aplicada solamente a la capa de negocio por ser la más compleja dentro de las Capas; debido a que es donde se encuentra más acumulación de clases, conteniendo además la mayor cantidad de procesos pues es la capa conectora entre Presentación y Acceso a Datos donde de una forma u otra está involucrada entre ambas.

Un **TC** grande afecta los indicadores de calidad definidos para esta métrica por los especialistas, ver la **Tabla 1**.

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
Responsalbilidad	La clase debe tener bastante responsabilidad

Tabla 1. Parámetros de calidad para valores grandes de TC (Lorenz & Kidd, 1994)

La siguiente tabla refleja las medidas o umbrales aplicados en el diseño del sistema.

No de Operaciones y/o Atributos	
TC	**Umbral**
Pequeño	<=20
Medio	>20 y <=30
Grande	>30

Tabla 2. Umbrales para TC (Lorenz & Kidd, 1994)

A continuación se muestran las clases de la capa de negocio

No	Clase	Total de atributos	Total de operaciones	Tamaño
1	EBancoRecaudacion	6	0	Pequeño
2	EConceptoRecIngresosConc	3	0	Pequeño
3	EConceptosRecaudacion	3	0	Pequeño
4	EDesglosePUB	4	0	Pequeño
5	EDiferencias	5	0	Pequeño
6	EEstadoPUB	2	0	Pequeño
7	EEstadoReintegro	2	0	Pequeño
8	EIdReintegroPUB	2	0	Pequeño
9	EIngresosConceptosPrest	3	0	Pequeño
10	EPersona	15	0	Pequeño
11	EPlanillaUnicaBancaria	16	0	Pequeño
12	ERegistroDiarioBanco	6	0	Pequeño
13	ERegistroNotificacionBanco	9	0	Pequeño
14	ERegistroResumenDiarioBanco	5	0	Pequeño
15	EReintegroPUB	3	0	Pequeño
16	EReintegros	10	0	Pequeño
17	ESucursalRecaudacion	8	0	Pequeño
18	ETipoDiferencia	3	0	Pequeño
19	GtrDiferencias	3	3	Pequeño
20	GtrPlanillaUnicaBancaria	3	2	Pequeño
21	GtrReintegros	3	2	Pequeño
22	GtrReportes	4	14	Pequeño

Tabla 3. Clases de la capa de negocio.

Resultados obtenidos.

La capa de negocio cuenta con 22 clases, para un promedio de cantidad de atributos de 5.36 y un promedio de cantidad de operaciones de 0.95 como se muestra en la tabla 4.

Total de Clases	Promedio de Atributos	Promedio de Operaciones
22	5.36	0.95

Tabla 4. Total de clases del negocio y los promedios de atributos y operaciones.

La capa de negocio presenta 22 clases pequeñas, ninguna de tamaño medio y ninguna de tamaño grande, en la **tabla 5** se puede ver la distribución por tamaño.

Umbral	Tamaño	Cantidad de Clases
<=20	Pequeño	22
>20 y <=30	Medio	0
>30	Grande	0

Tabla 5. Cantidad de clase por tamaño.

A partir de los umbrales definidos en la **tabla 5** se puede observar que existe en la capa de negocio un 100% de clases pequeñas, un 0% de clases medianas y un 0% de clases grandes.

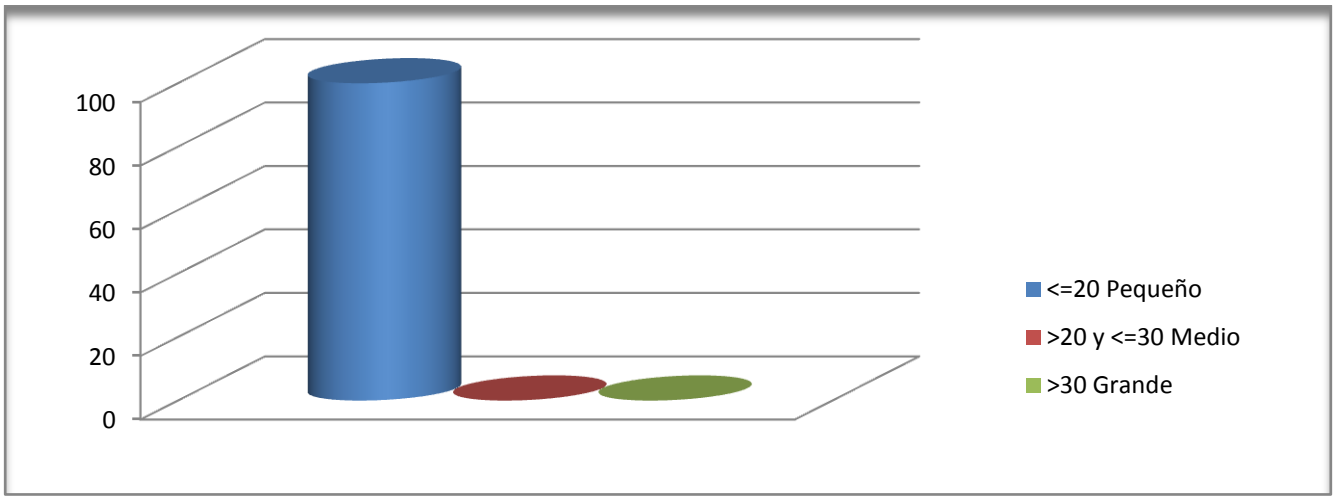


Figura 17. Por ciento de clases por tamaño.

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

4.2.2. 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 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, Vol I, 1998)

Resultados Obtenidos: Una vez aplicado al sistema la métrica **APH,** se observa que la profundidad de jerarquía de clases es baja, debido que el nivel más alto de herencia es **2**; esto conlleva a que no exista un alto acoplamiento y no sea difícil de dar mantenimiento.

4.2.3. Número de descendiente (NDD)

Las subclases que son inmediatamente subordinadas a una clase son denominadas descendientes.

A medida que crece el número de descendientes, se incrementa la reutilización, pero también es cierto, que a medida que crece NDD, la abstracción representada por la clase predecesora puede verse diluida. Esto es, existe la posibilidad de que algunos de los descendientes no sean realmente miembros propios de la clase predecesora. A medida que NDD va creciendo, la cantidad de pruebas crecerá también. (Chidamber & Kemne, 1994)

Resultados Obtenidos

El sistema presenta como máximo nivel de descendiente 1, obteniéndose un nivel de reutilización bajo pero sin verse afectada la abstracción representada por la clase predecesora.

4.3. Resultados obtenidos en pruebas de caja blanca.

La prueba de caja blanca, denominada a veces *prueba de caja de cristal* es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que garanticen que se ejercite por lo menos una vez todos los caminos independientes de cada módulo; ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa; ejecuten todos los bucles en sus límites y con sus límites operacionales; y ejerciten las estructuras internas de datos para asegurar su validez (Pressman, Vol I, 1998).

Para la evaluación de la calidad en la implementación del subsistema de Recaudación se realizaron pruebas de caja blanca a algunos escenarios fundamentales de los Casos de Uso explicados en el

Capítulo 2, así como a algunos de los componentes de software utilizados, para ello se tomó como herramienta de validación NUnit, anteriormente especificada en el Capítulo 1.

4.3.1. Resultados obtenidos de las pruebas aplicadas a los componentes.

Método que declara el punto de inicio de la prueba.

```
[SetUp]
public void Init()
{
    Assembly assembly = Assembly.GetExecutingAssembly();
    System.IO.Stream stream = assembly.GetManifestResourceStream(
        assembly.FullName.Substring(0, assembly.FullName.IndexOf(',')) + ".hibernate.cfg.xml");
    NHibernateUtil.Iniciar(stream);
    ArrayList lista = GtrTest.CrearClase();
    lvwTest.Objetos = lista;
}
```

Este método valida la veracidad de la inserción en el componente ListView. Después de la inserción en el metodo Init(), se verifica que la cantidad de elementos contenidos en el ListView no sea igual a 0, en caso de serlo se mostraría el mensaje “Valor no esperado”.

```
[Test]
public void AdicionarListView()
{
    Assert.AreNotEqual(0, lvwTest.Objetos.Count, "Valor no esperado");
}
```

El componente está diseñado para que a cada columna se le asigne una propiedad del objeto a mostrar, en caso de que no se configure de esta manera se lanzaría una excepción; para esta prueba en específico se espera una excepción de tipo “Exception”, ya que se dejó una columna sin configurar.

```
[Test, ExpectedException(typeof(Exception))]
public void ExpectAnException()
{
    ArrayList lista = GtrTest.CrearClase();
    lvwTestProperty.Objetos = lista;
}
```

Este método valida que el objeto previamente insertado en el componente mantiene su integridad. En este caso se verifica que el valor de la fecha no haya sido alterado.

```
[Test]
public void ValorEsperado()
{
    Assert.AreEqual(new DateTime(2008, 5, 25),
        (lvwTest.Objetos[0] as ETest).Fecha, "Valor no esperado");
}
```

Este método verifica que el formato del Rif sea correcto.

```
[Test]
public void ValidacionRif()
{
    Assert.AreEqual(true, rifTest.ValidarRif("J-31405641-7"), "Valor no esperado");
}
```

4.3.2. Resultados obtenidos de las pruebas aplicadas a los gestores del negocio.

Con los parámetros especificados se esperan resultados, este método valida la existencia de los mismos.

```
[Test]
public void DiferenciasResueltaSBBuscar()
{
    ArrayList obj = Fabrica.Instancia.CrearGtrDiferencias().
        ObtenerDiferenciasXBancoR(new DateTime(2008, 3, 25), new DateTime(2008, 3, 25), -1, -1);
    Assert.AreNotEqual(0, obj.Count, "Valor no esperado");
}
```

Con los parámetros especificados se esperan resultados, este método valida la existencia de los mismos.

```
[Test]
public void DiferenciasResueltaOficinaBuscar()
{
    ArrayList obj = Fabrica.Instancia.CrearGtrDiferencias()
        .ObtenerDiferenciasXOficinaR(new DateTime(2008, 3, 25), new DateTime(2008, 3, 25), 488);
    Assert.AreNotEqual(0, obj.Count, "Valor no esperado");
}
```

Este método verifica que el estado de la Planilla Única Bancaria haya sido modificado satisfactoriamente en la base de datos.

```
[Test]
public void RecaudarFodo()
{
    decimal id = 1;
    ArrayList array = new ArrayList();
    array.Add(id);
    Fabrica.Instancia.CrearGtrPlanillaUnicaBancaria().CambiarEstadoPUB( array, 4);
    Assert.AreEqual(4, (Fabrica.Instancia.CrearGestorComun()
        .BuscarPorId(Fabrica.Instancia.CrearPlanillaUnicaBancaria(), id) as
        IPlanillaUnicaBancaria).ObjEstadoPUB.IdEstado, "Valor no esperado");
}
```

Como resultado de las pruebas realizadas a los componentes y gestores especificados, a continuación se puede observar la interfaz de NUnit, donde muestra que las pruebas fueron realizadas satisfactoriamente.

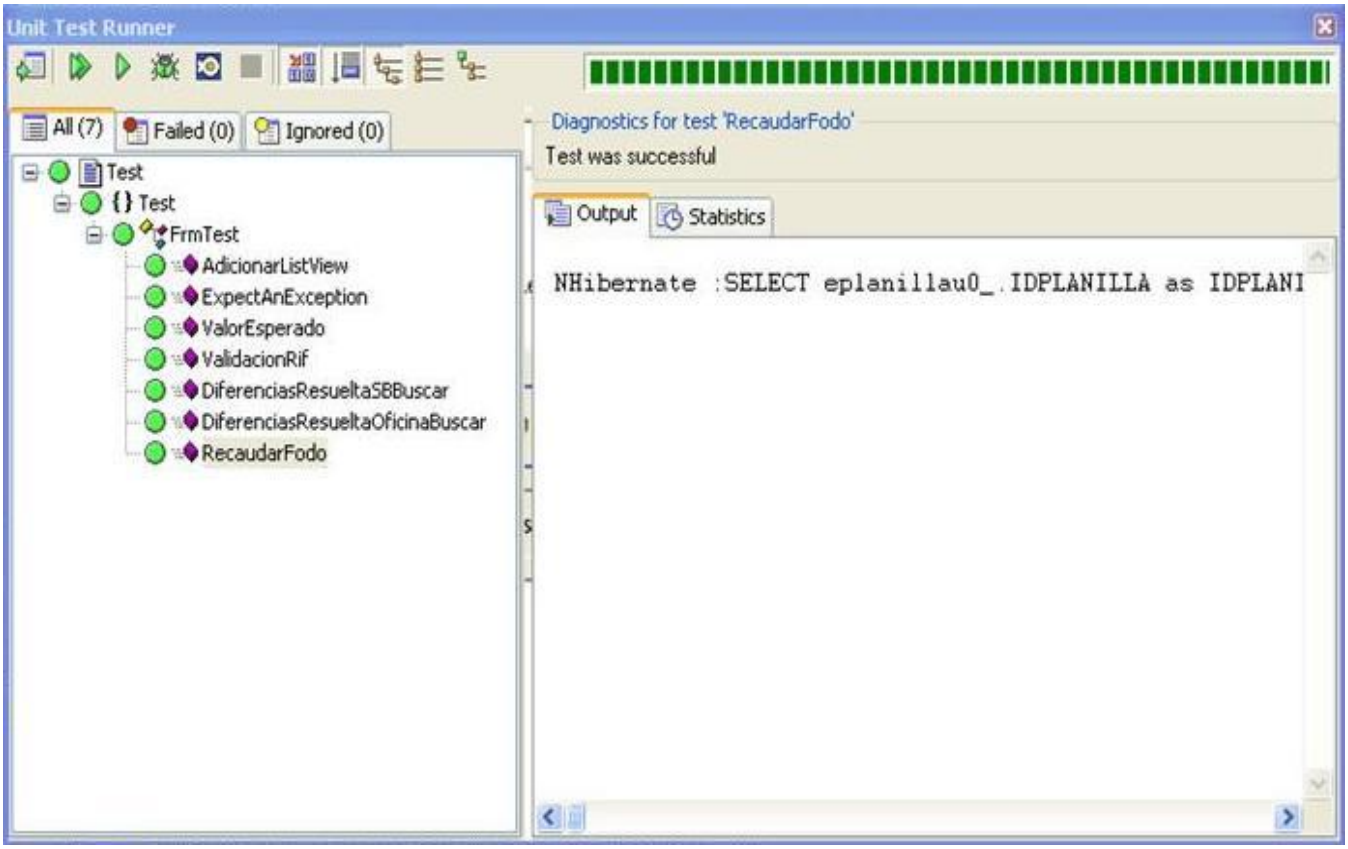


Figura 18 Interfaz NUnit.

Leyenda:

Resultados Satisfactorios

Resultados fallidos.

Resultados ignorados.

CONCLUSIONES

Con la realización del presente trabajo se arriba a las siguientes conclusiones.

- Fueron analizados algunos sistemas financieros usados en el mundo y en Cuba que sirvieron de ejemplo en la confección de este trabajo.
- Se definió la estrategia de diseño basada en el uso de patrones de arquitectura y diseño previamente abordados, en conjunto con el análisis de algunas tendencias de la programación, en este punto las herramientas CASE lograron un papel decisivo en la automatización de todos los procesos llevados a cabo para obtener las metas propuestas para el diseño del subsistema desarrollado.
- De acuerdo a las condiciones en que se desarrolló este subsistema, el cual sigue algunos requerimientos establecidos por los sistemas ya implementados en la solución SAREN, se seleccionó la plataforma de desarrollo, apoyándose además en un estudio breve de todas las plataformas existentes.
- Con el estudio de los casos de uso del sistema, requisitos funcionales y no funcionales, así como las reglas del negocio, identificadas por los analistas se construyó el modelo del diseño, que sirvió como traza directa para la implementación del subsistema, satisfaciendo de esta forma los requerimientos del cliente.
- Con la creación del modelo de diseño del subsistema de Recaudación, se obtuvieron los artefactos necesarios para más tarde realizar la implementación, flujo de trabajo que le sigue al diseño.
- El uso de métricas asociadas a la medición de la calidad de un diseño de software y sus resultados una vez aplicadas al diseño realizado, sirven como argumento para expresar la calidad del mismo.

RECOMENDACIONES

- Continuar en la investigación de sistemas con la misma finalidad que el presente, que aporten soluciones novedosas a la dada en este trabajo.
- Documentar el framework empleado para facilitar el uso de otros desarrolladores.
- Realizar pruebas de las antes mencionadas a todos los componentes y código en sentido general, ya que se garantiza un alto porciento de calidad del mismo.
- Integrar finalmente el módulo de Recaudación a los restantes módulos del Sistema de Administración Financiera para corroborar su buen funcionamiento e integración total.

BIBLIOGRAFÍA

1. Alexander, S. I.-K. (1997). *A Pattern Language.* New York.
2. Cabrera González, M., Obregón Rodríguez, G., Cárdenas Negrin, M., & Carralero Silva, L. M. (2004). *SISTEMA ECONÓMICO INTEGRADO.*
3. Chidamber, & Kemne. (1994). *Métricas CK.*
4. Easy, P. M. (2000). *John V. Hedtke.*
5. FinancialTech. (2005). Embarcadero lanza ER/Studio 7.0 con soporte para el modelado y análisis de datos. (056).
6. González, B. (Marzo 2007). *Introducción al entorno de desarrollo de Microsoft .NET. Desarrolloweb.*
7. Hernández, J. M. (2004). *Mucho más que una implementación libre de .Net. 2004.*
8. Hunt, A., & Thomas, D. (2004). *Pragmatic Unit Testing in C# with NUnit The Pragmatic Bookshelf.* Raleigh.
9. Larman, C. (1999). *UML y Patrones Introducción al Análisis y Diseño orientados a objetos.*
10. Lorenz, & Kidd. (1994). *Métricas de Lorenz y Kidd.*
11. Marquez, Y. (2007). *Proyecto de Modernización de los Registros y Notarías Administración Financiera Módulo de Recaudación Documento de Prototipo de Interfaz de Usuario.* Caracas.
12. Marquez, Y. (2007). *Proyecto de Modernización de los Registros y Notarías Administración Financiera Módulo de Recaudación Documento de requerimientos.* Caracas.
13. Mato Garcia, R. M. (1999). *Diseno de Bases de Datos.*
14. Pressman, R. (Vol I, 1998). *Ingenería de Software. Un enfoque práctico.*
15. Ramirez, A. O. (2003). *Patrones Estructurales.*
16. Red Hat. (2002). *Hibernate*. (Red Hat Americas;Red Hat EMEA;Red Hat APAC) Recuperado el 15 de febrero de 2008, de Hibernate: http://www.hibernate.org/343.html

17. Reynoso, C., Kiccillof, N., & AIRES, U. D. (2004). *Estilos y Patrones en la Estrategia de Arquitectura de Microsoft.*

18. Seco, J. A. (2001). *El lenguaje de programación C#. Programación en Castellano.*

19. Smith, B. E. (1991). *Dac Easy Made Easy.*

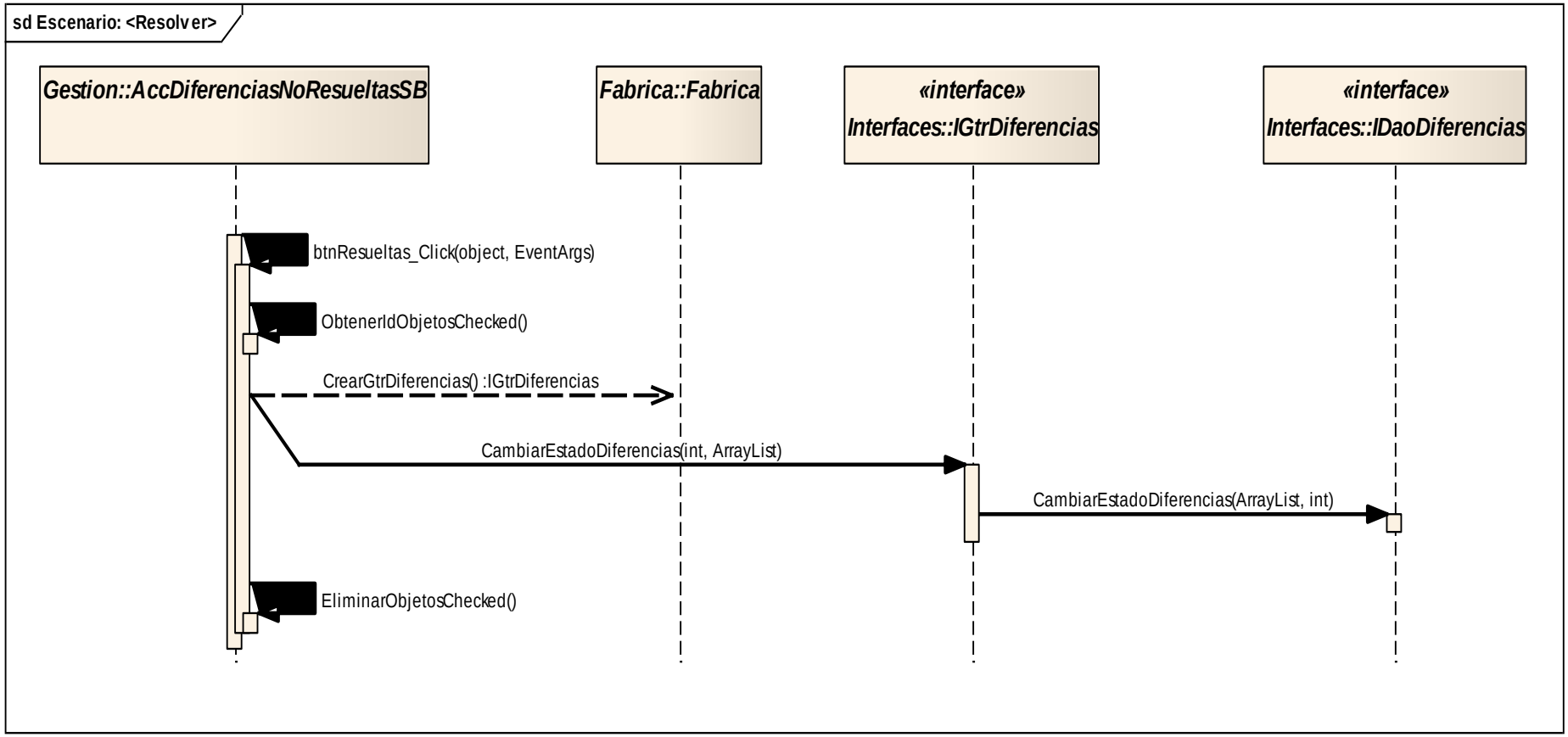
20. Sparx Systems. (2007). *Enterprise Architect*. (Sparx Systems) Recuperado el 29 de 3 de 2007, de Enterprise Architect: http://www.sparxsystems.com.ar/products/ea.html

21. Unidad Docente de Ingeniería del Software,Facultad de informática - Universidad Politécnica de Madrid . (2003). *Patrones del "Gang of Four".*

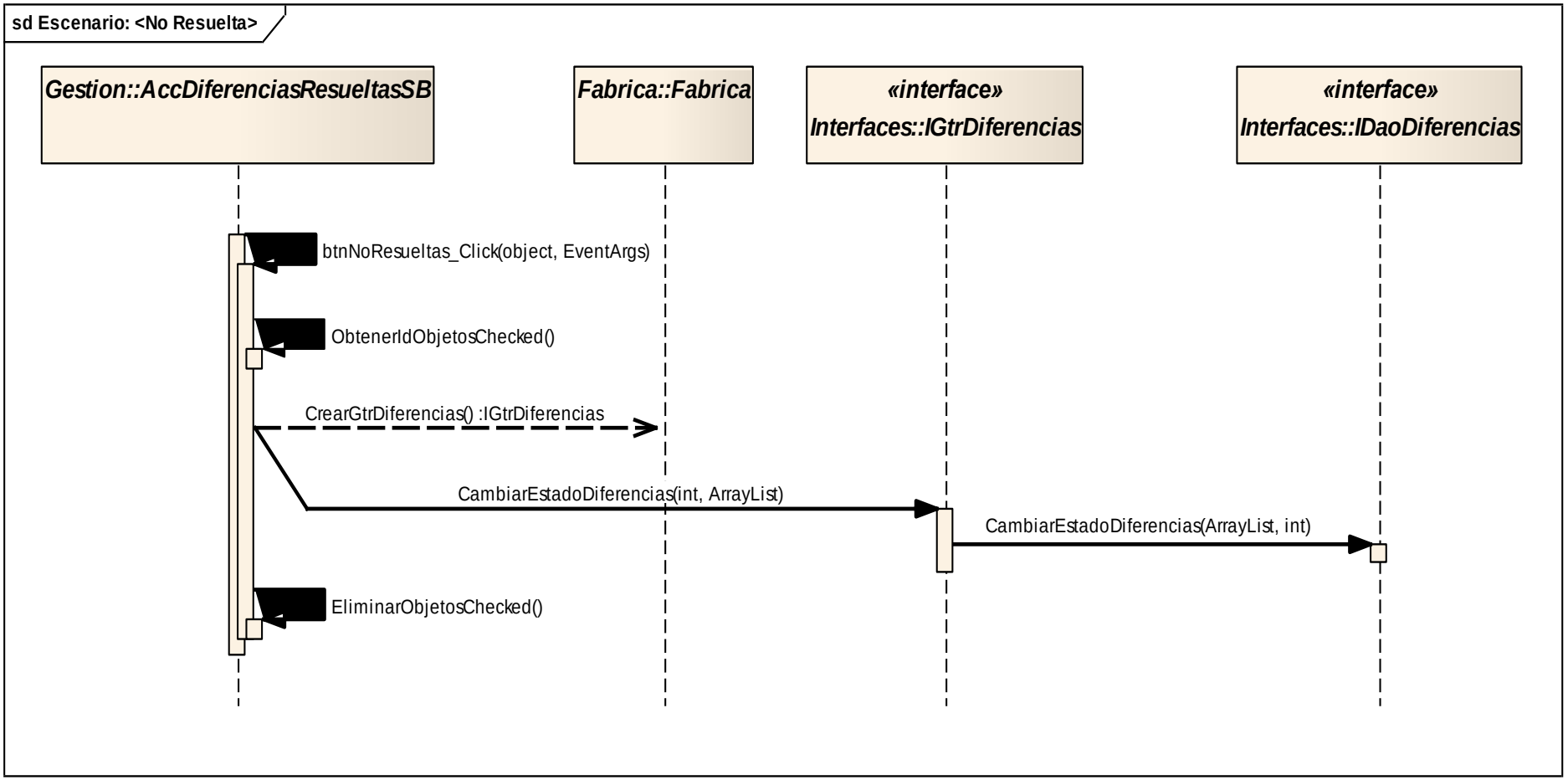
ANEXOS

Anexo 1 Diagrama de interacción: Caso de Uso Diferencias por Sucursal Bancaria.

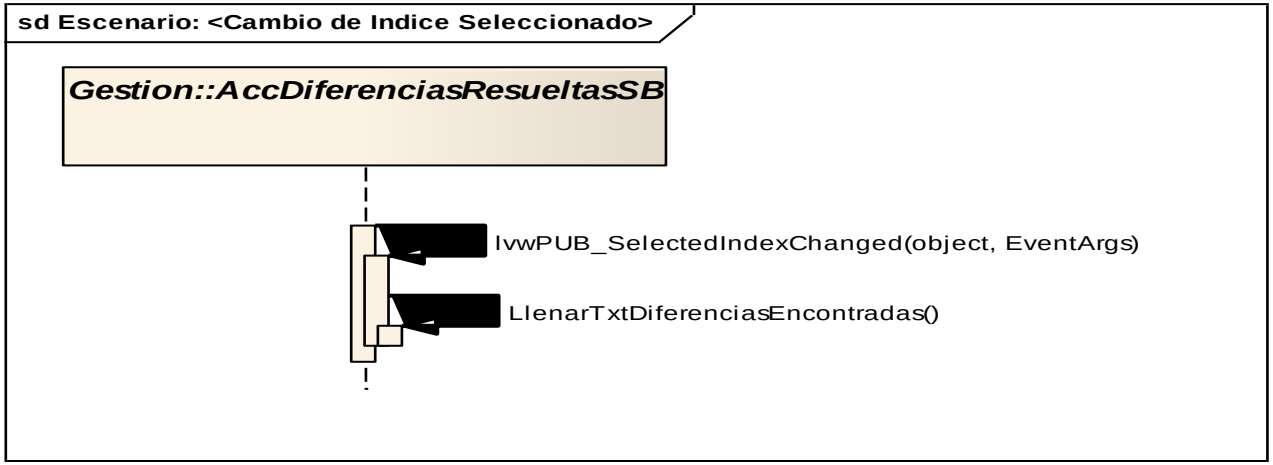
Escenario Resolver.



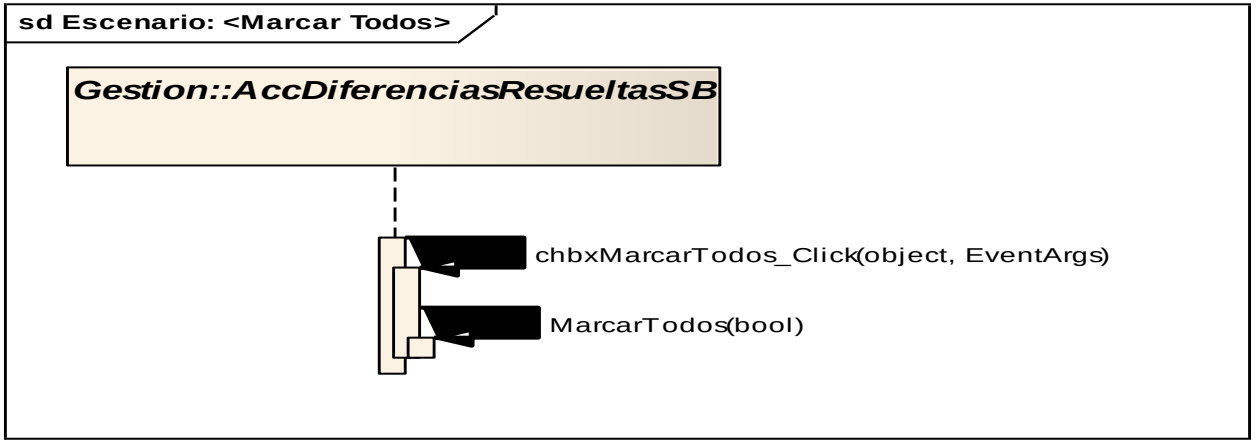
Escenario No Resuelta.



Escenario Cambio de Índice Seleccionado.

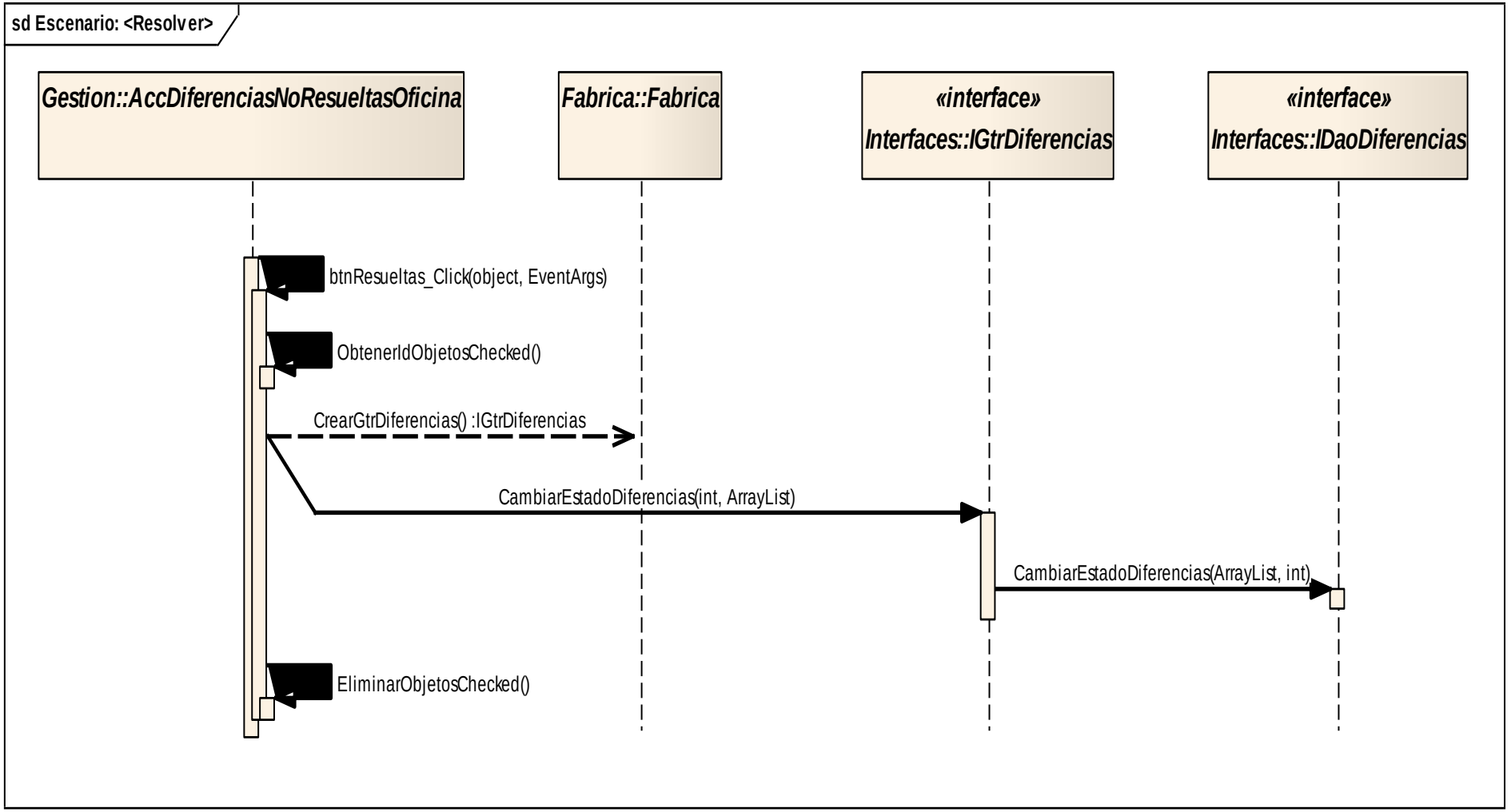


Escenario Marcar Todos.

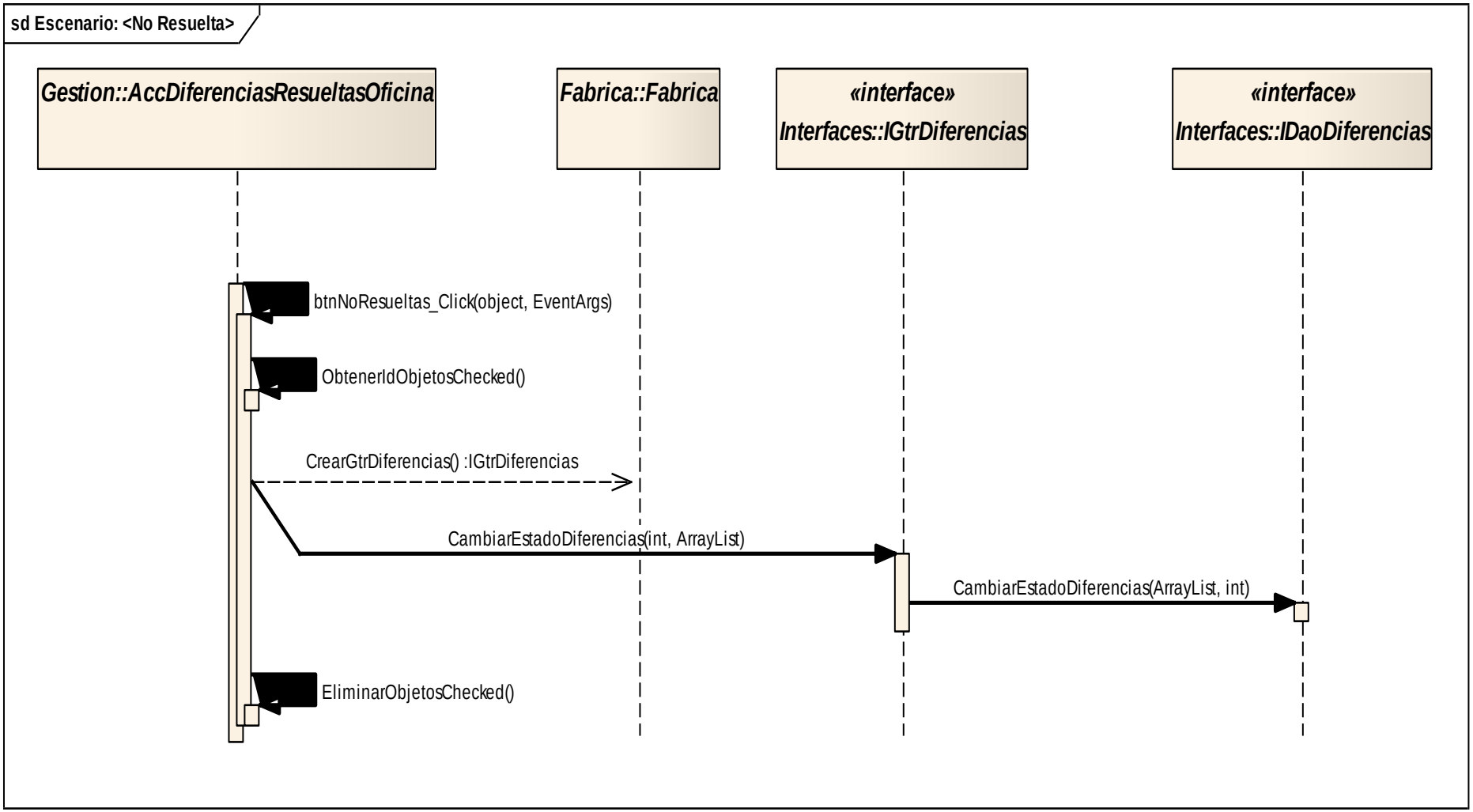


Anexo 2 Diagramas de interacción: Caso de Uso Diferencias por Oficina.

Escenario Resolver.

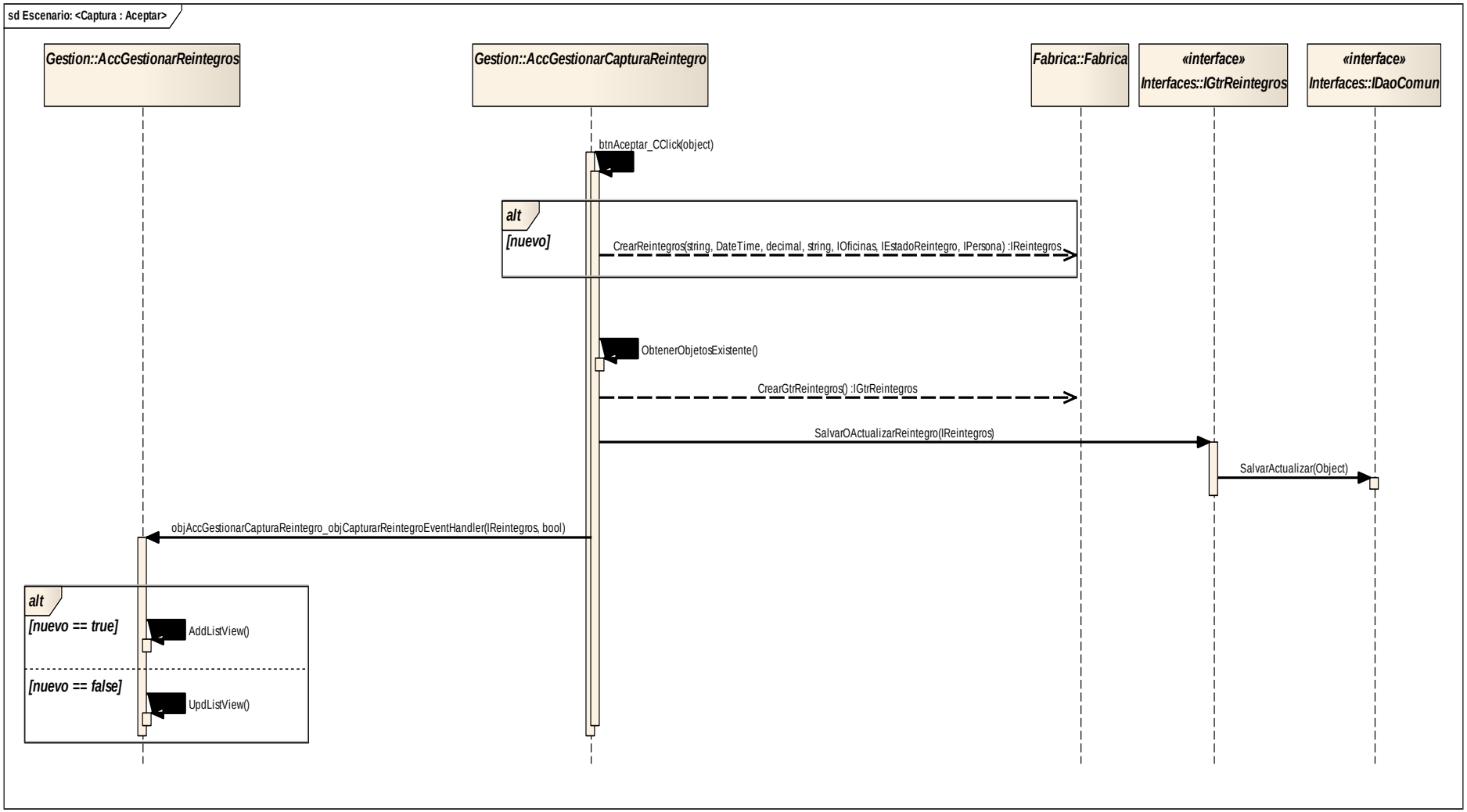


Escenario No Resuelta.

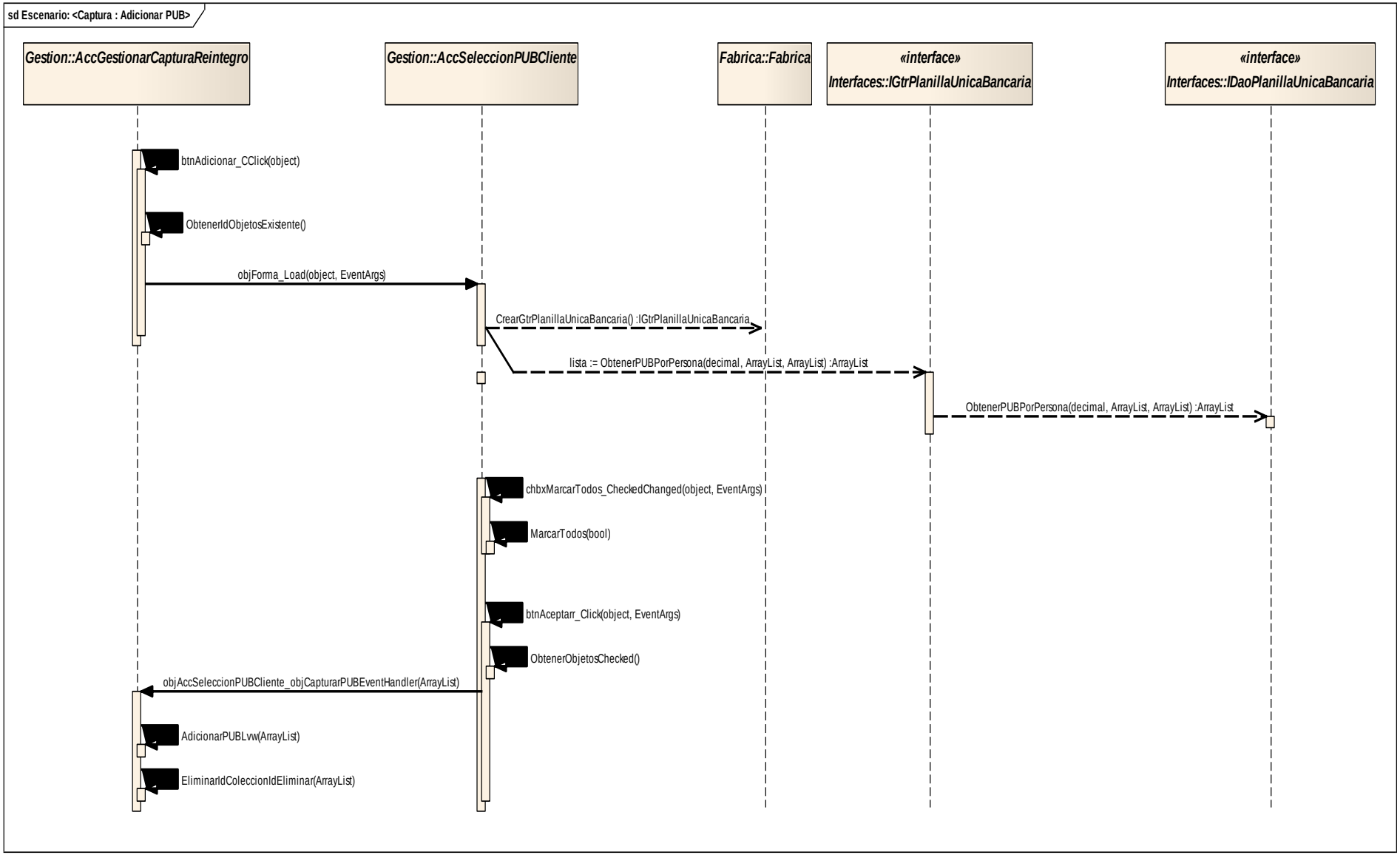


Anexo 3 Diagramas de interacción: Caso de Uso Gestionar Reintegros.

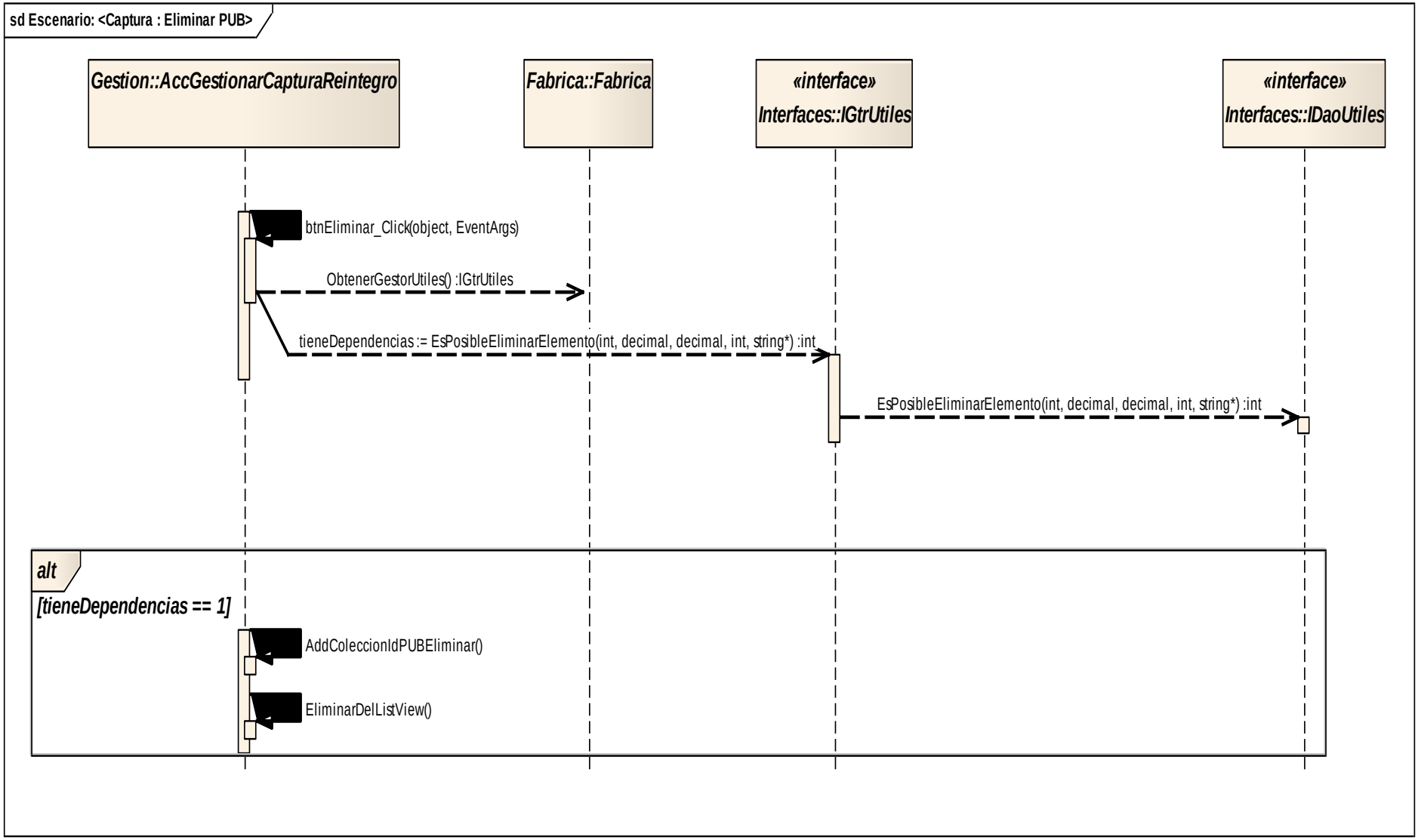
Escenario Captura Aceptar.



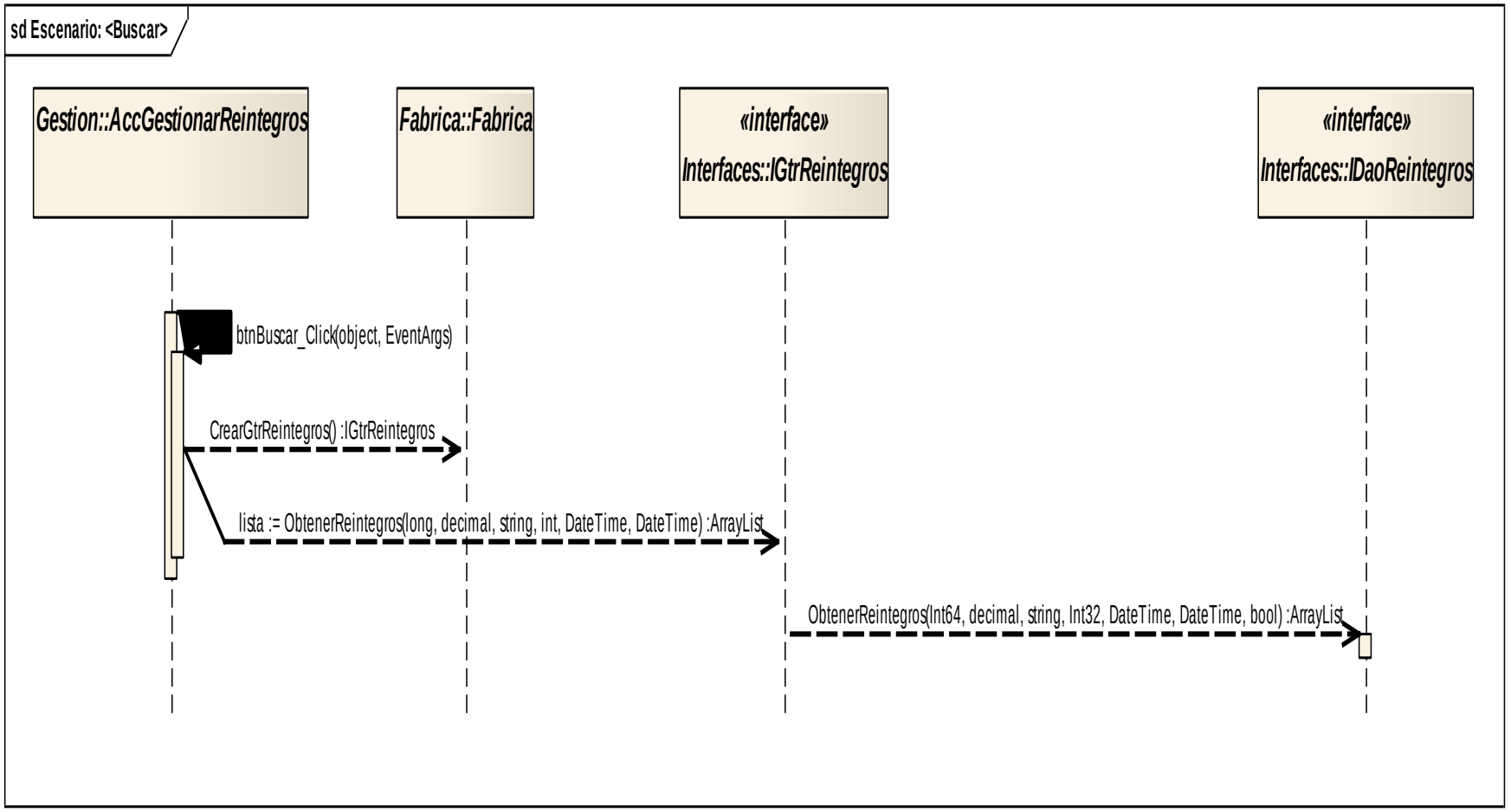
Escenario Captura Adicionar PUB.



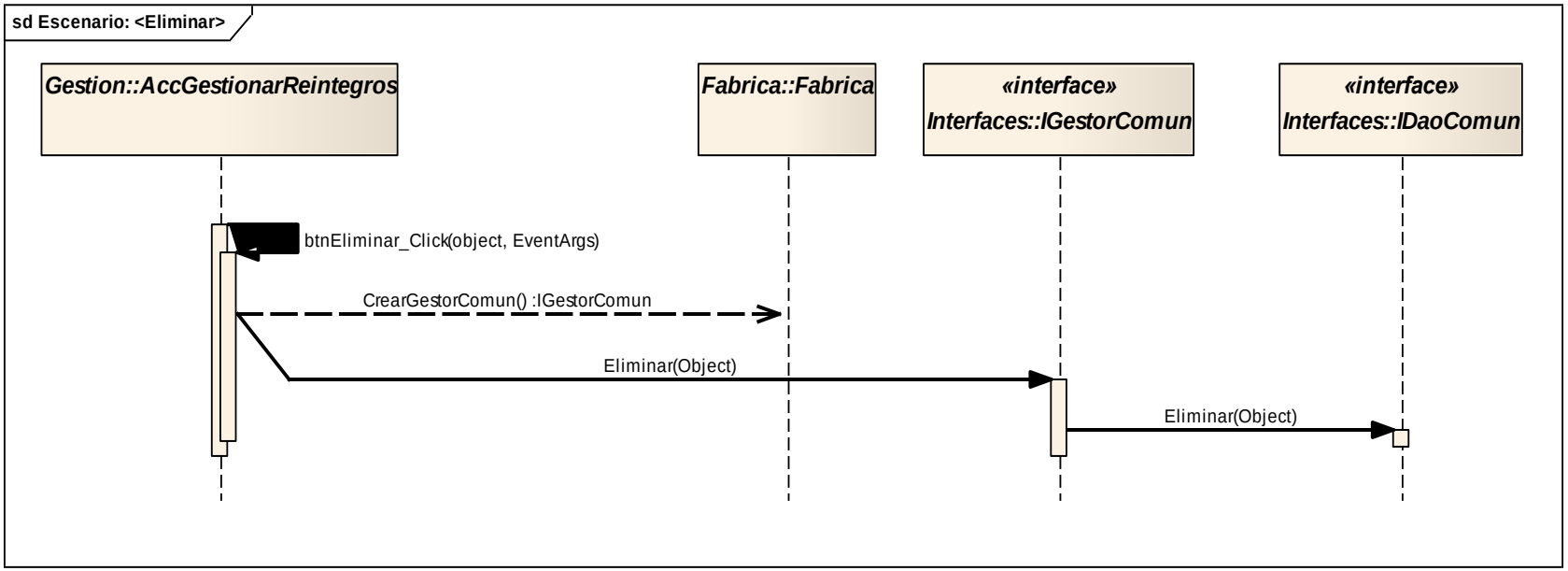
Escenario Captura Eliminar PUB.



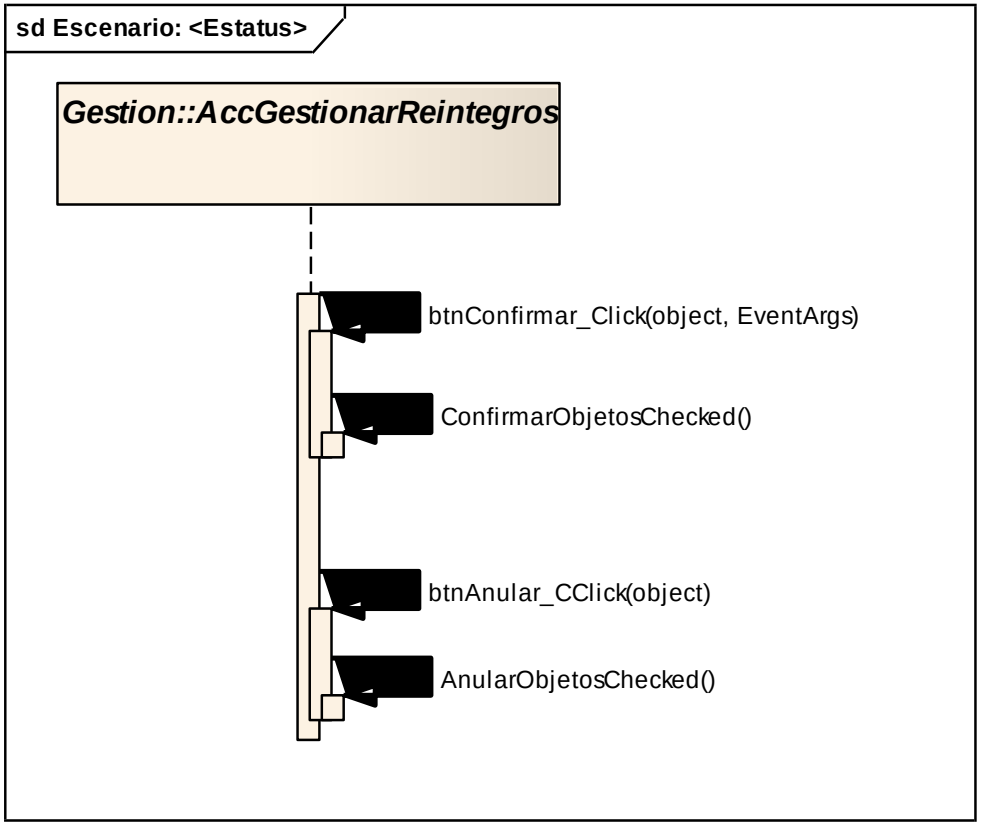
Escenario Gestión Buscar.



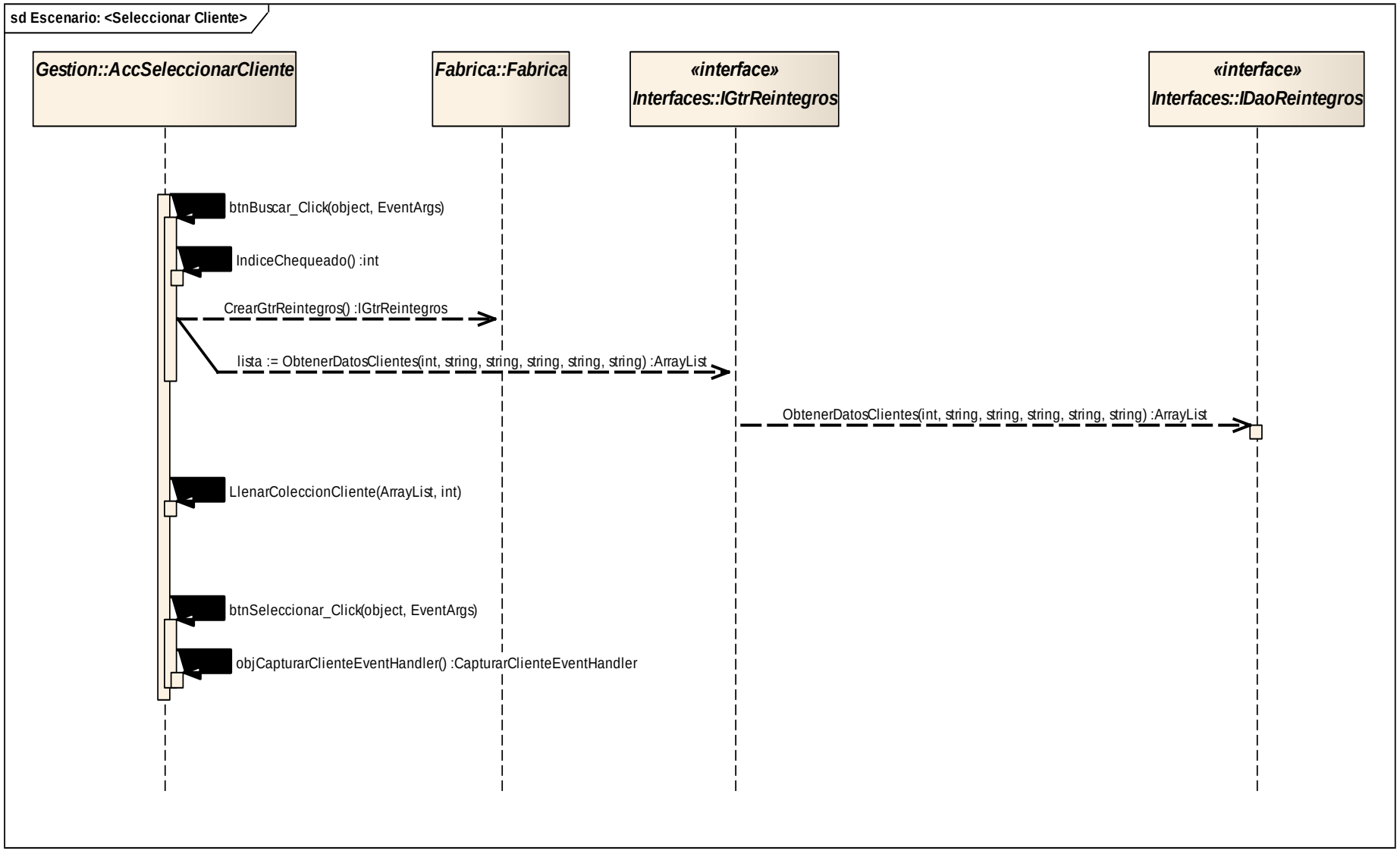
Escenario Gestión Eliminar.



Escenario Gestión Estatus.

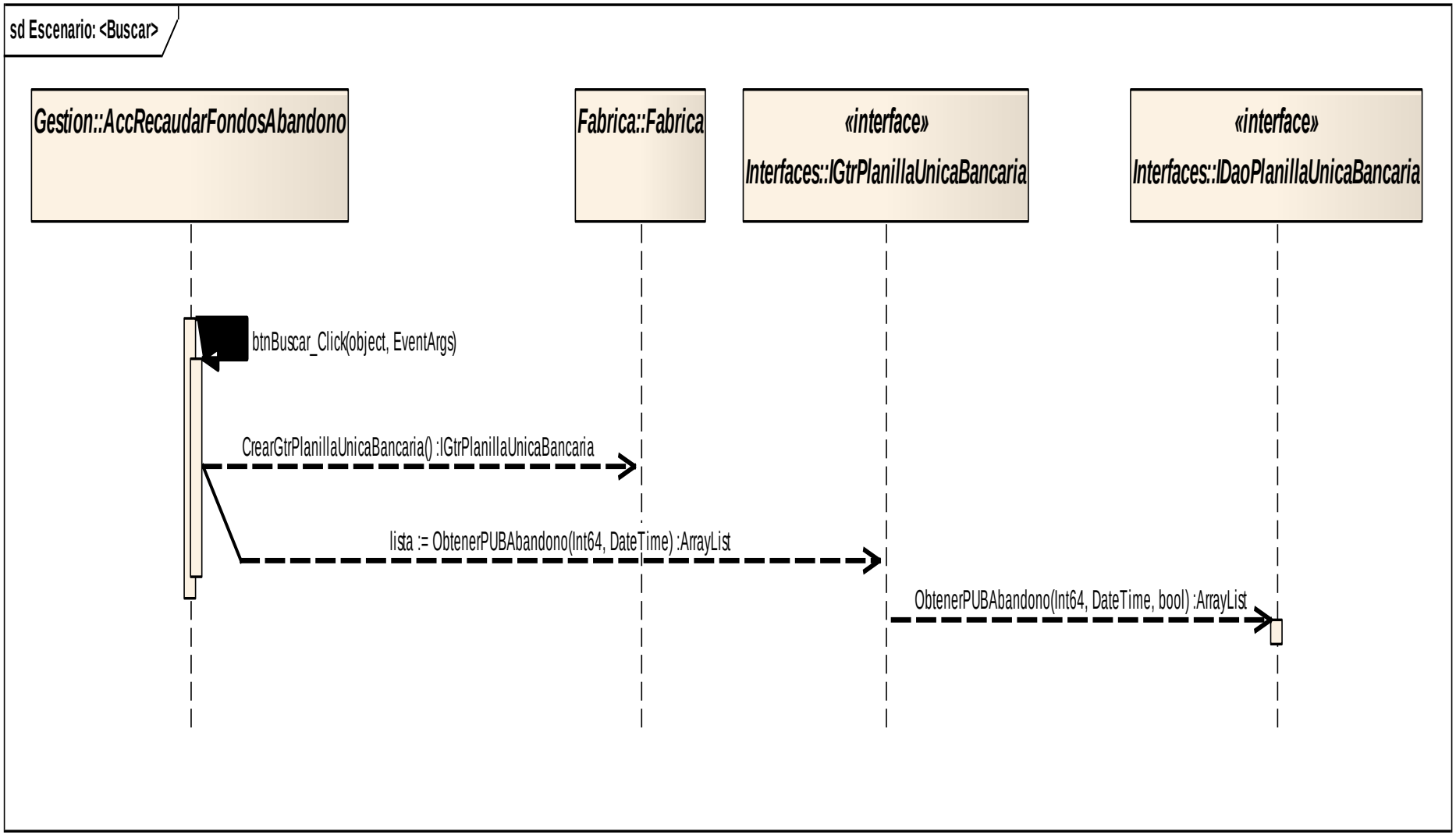


Escenario Seleccionar Cliente.

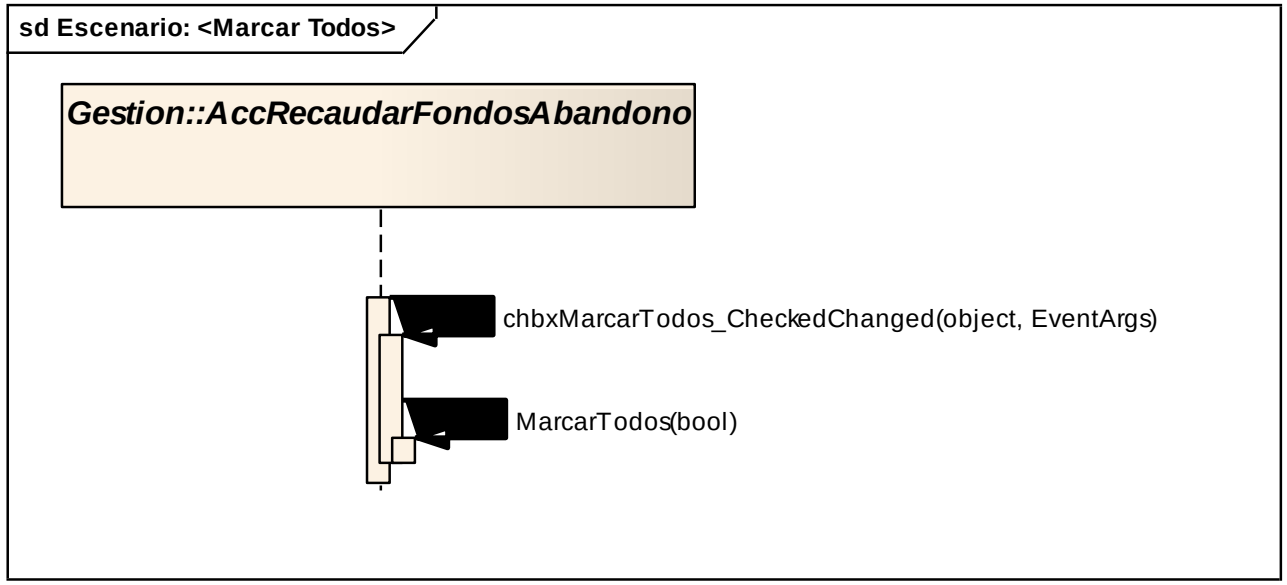


Anexo 4 Diagrama de Interacción: Caso de Uso Recaudar Fondos por Abandono.

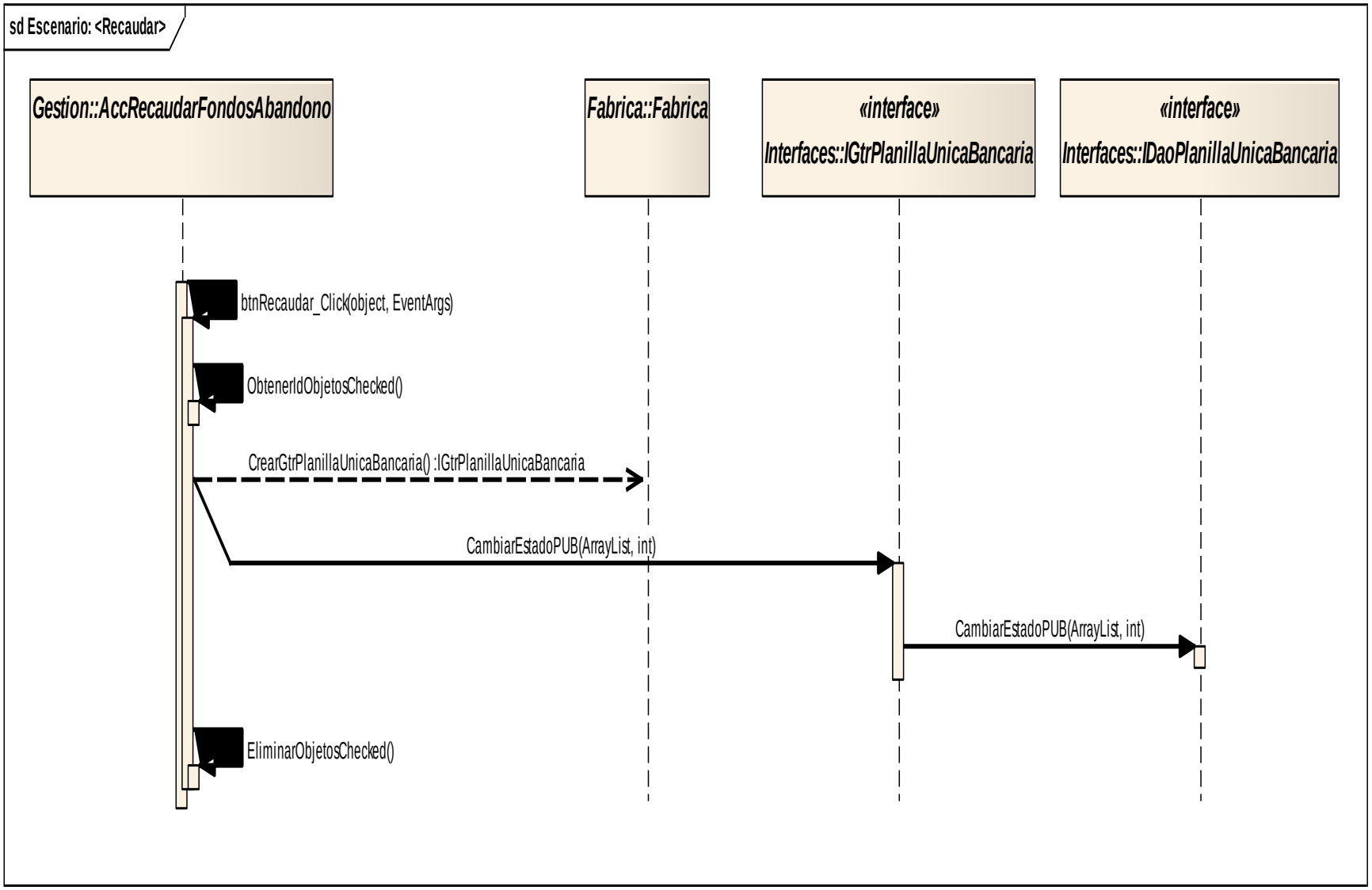
Escenario Buscar.



Escenario Marcar Todos.



Escenario Recaudar.



Anexo 5 Caso de Uso Reporte Consolidado de las Sucursales Bancarias.

Diagrama de Clases

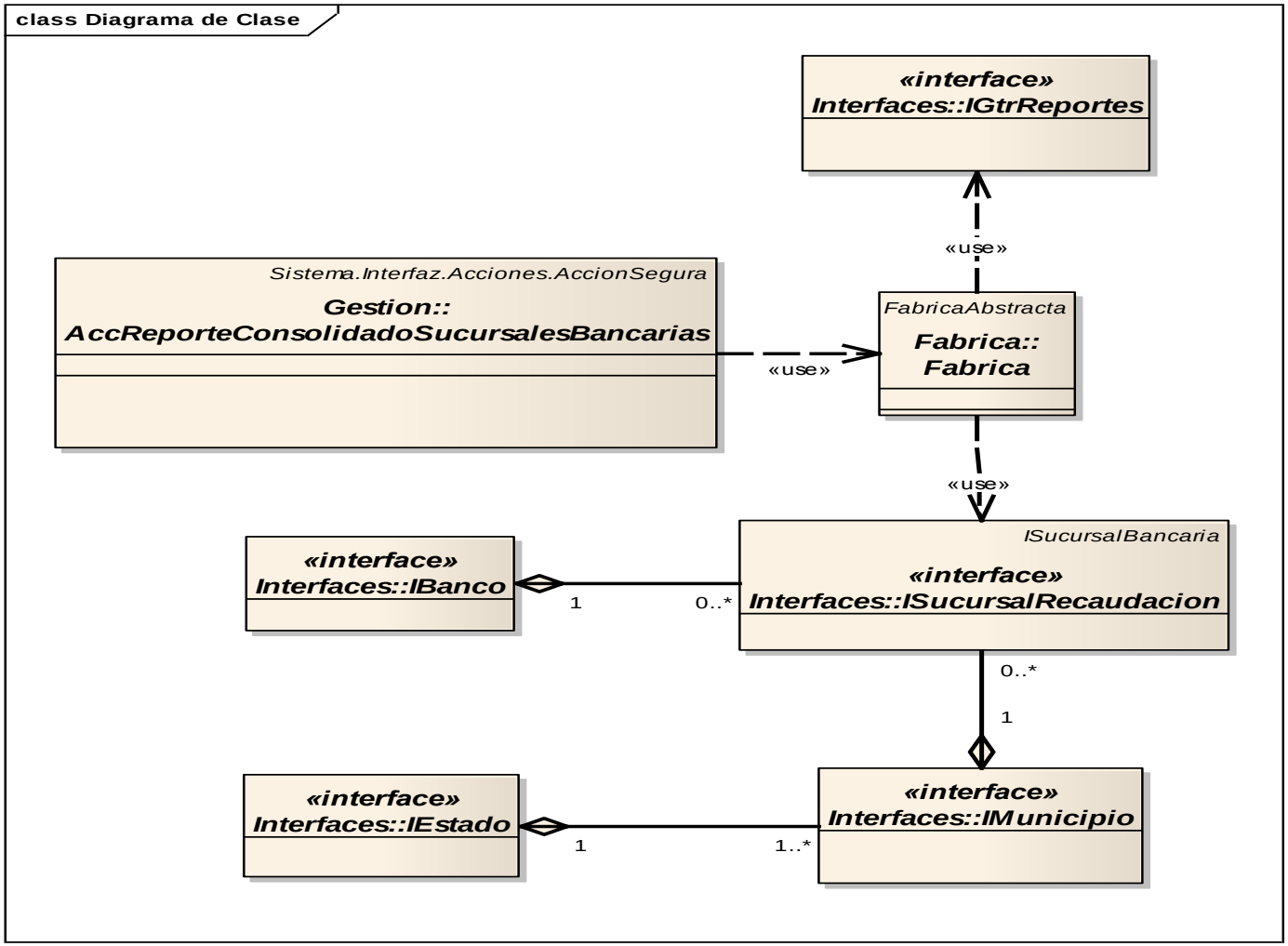
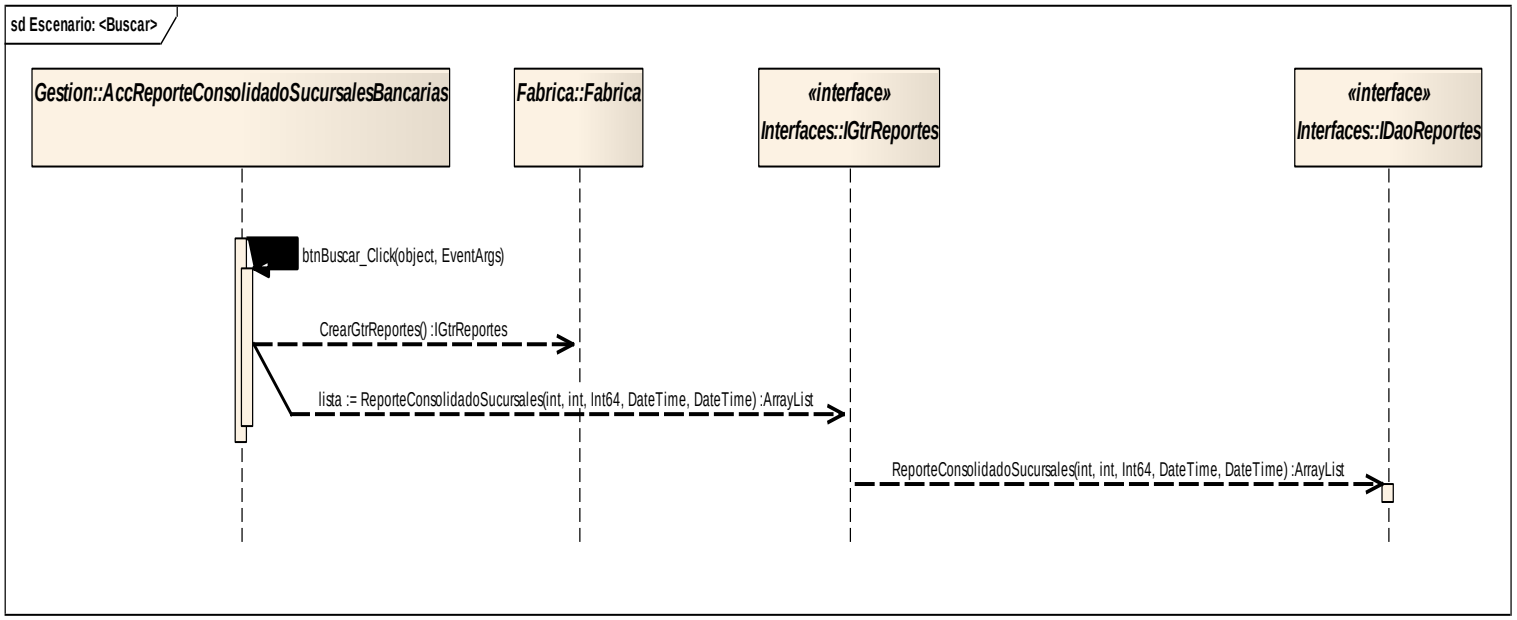


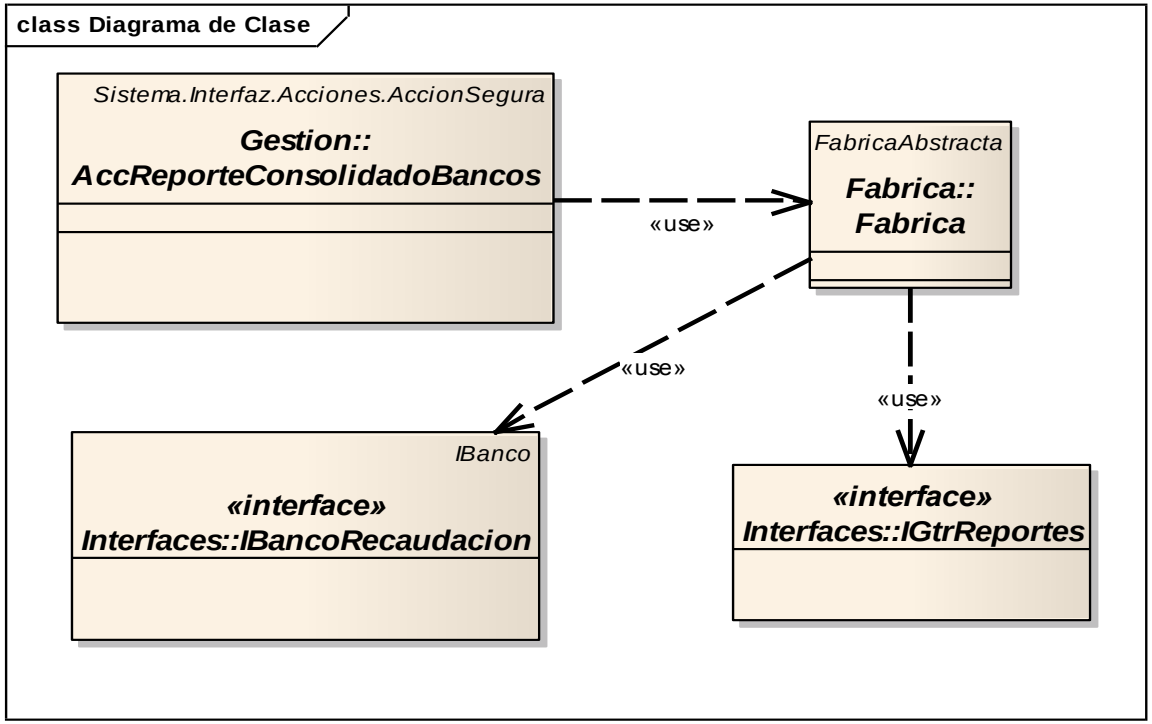
Diagrama de Interacción

Escenario Buscar.



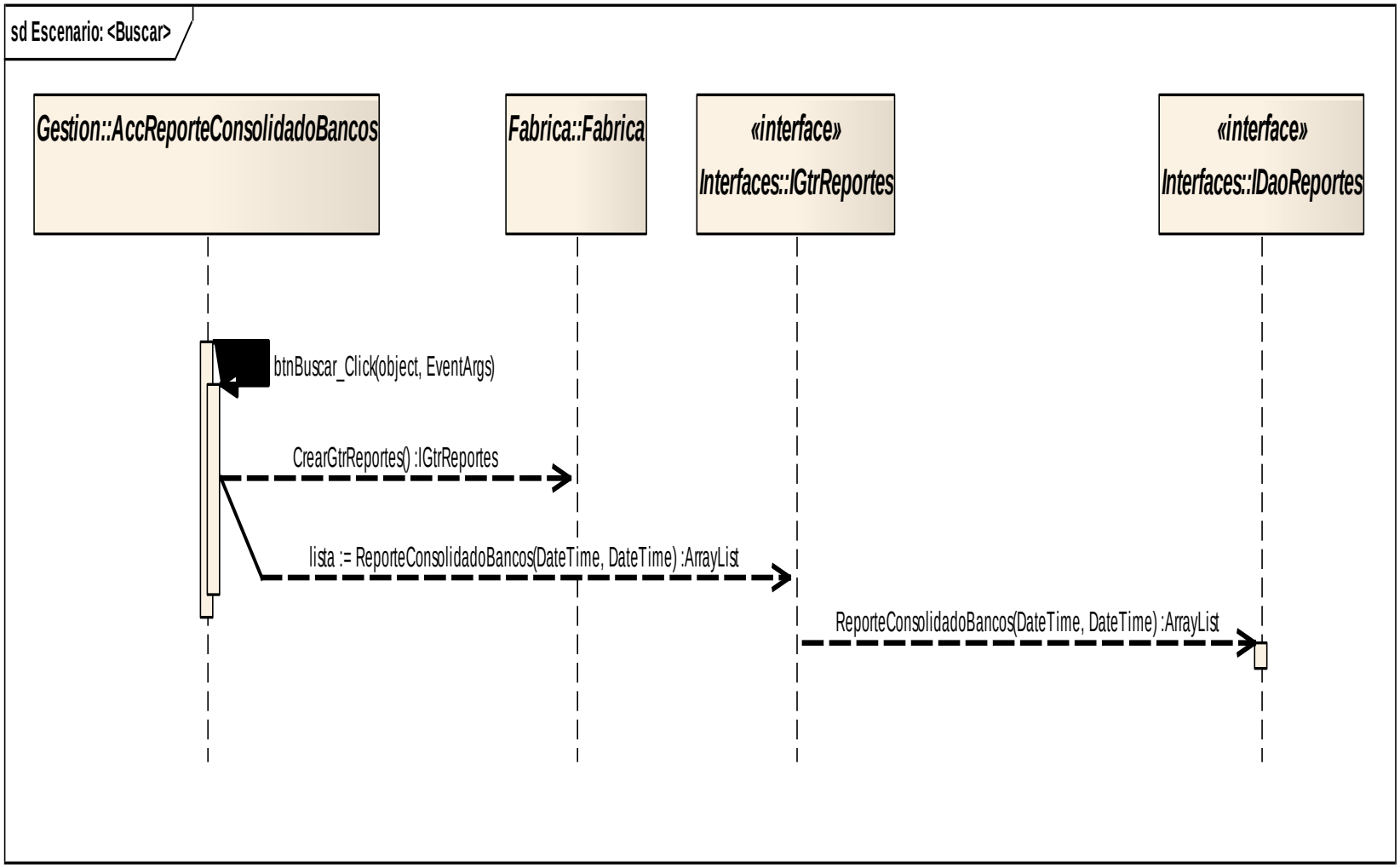
Anexo 6 Caso de Uso Reporte Consolidado de los Bancos.

Diagramas de Clases



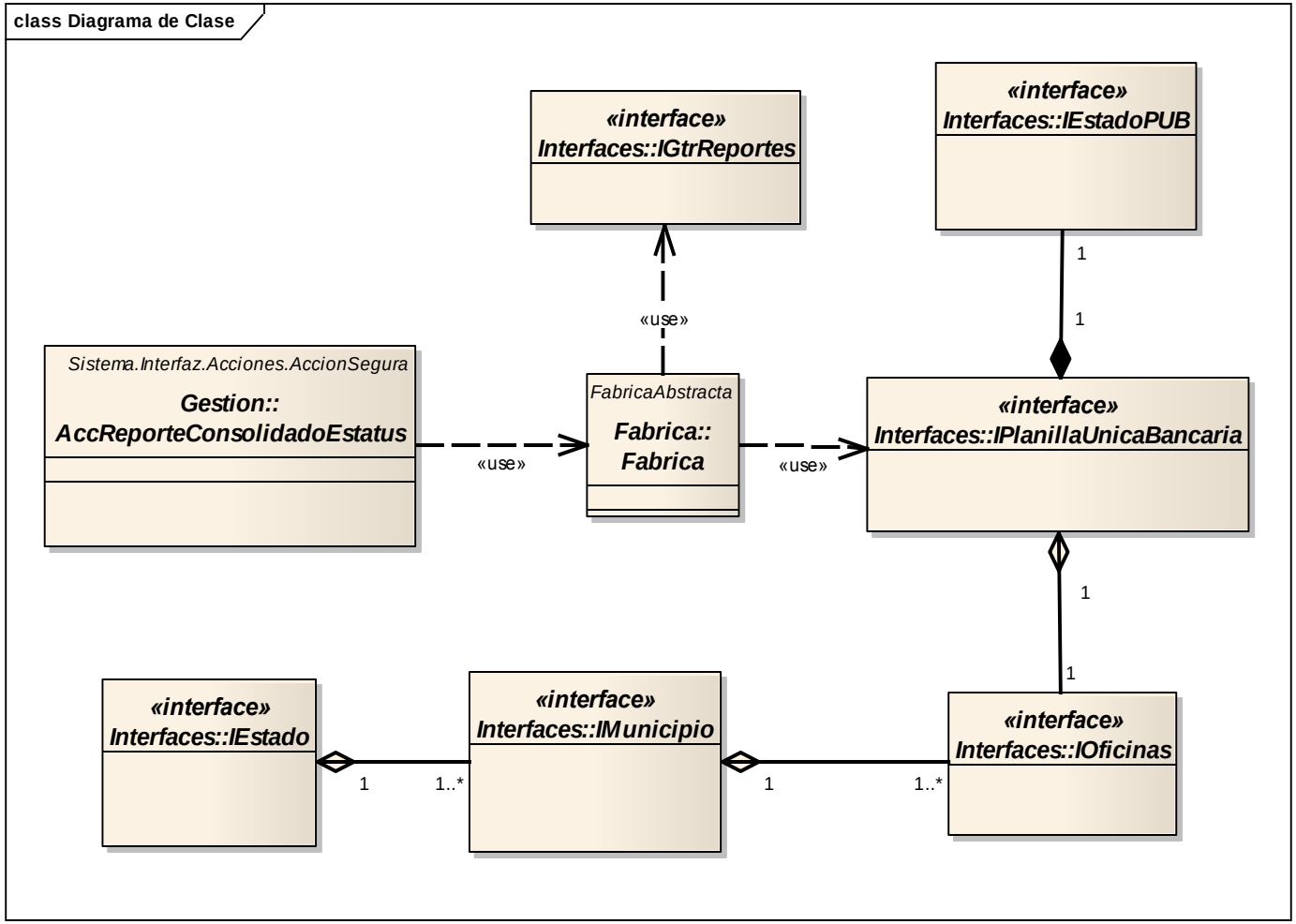
Diagramas de Interacción

Escenario Buscar.



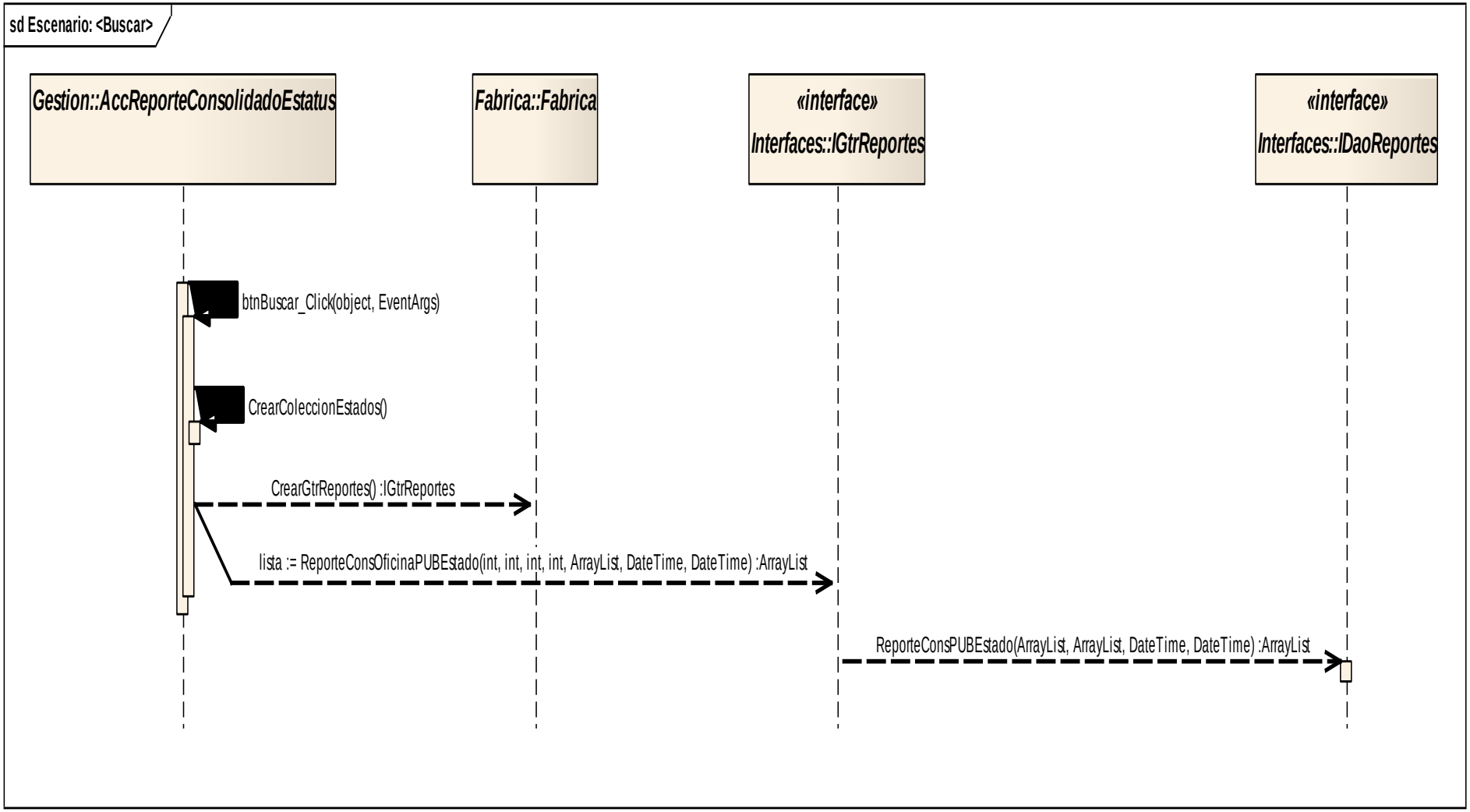
Anexo 7 Caso de Uso Reporte Consolidado por Estatus.

Diagramas de Clases



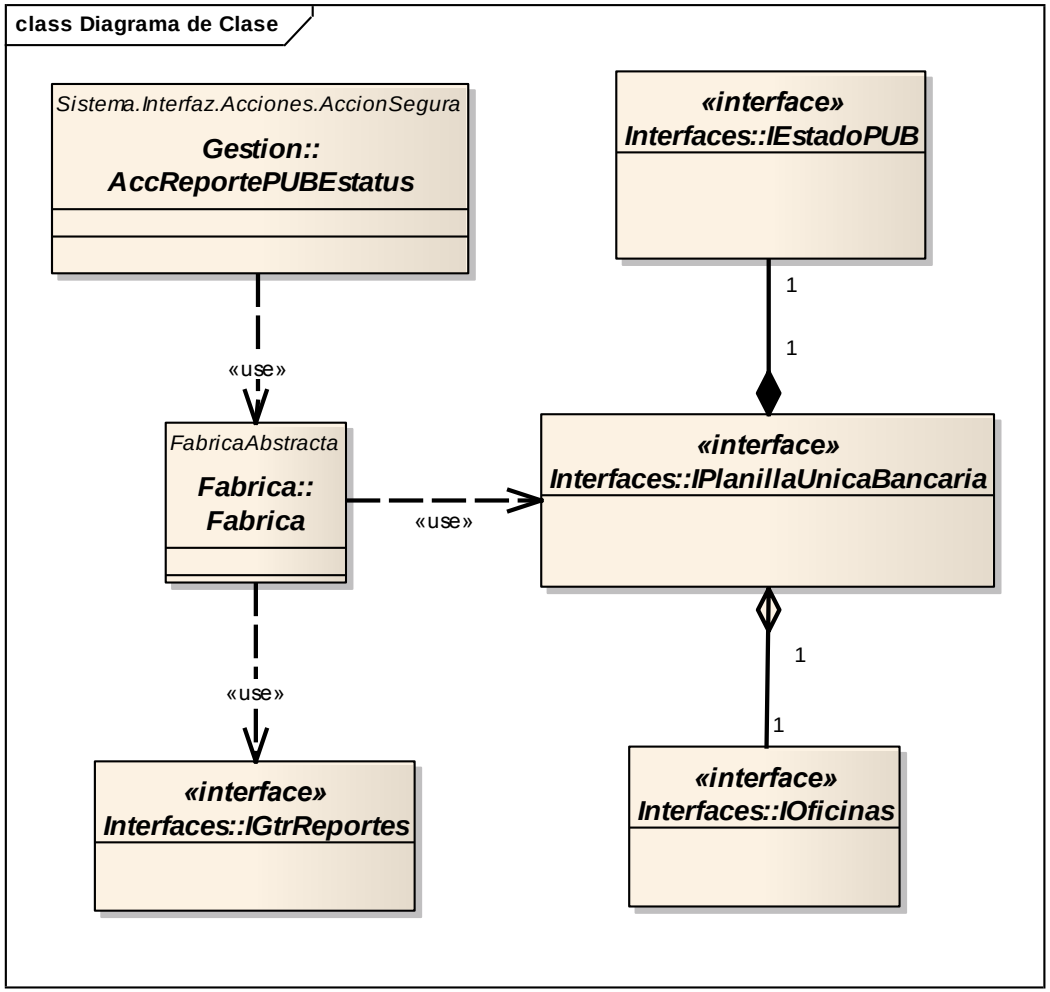
Diagramas de Interacción

Escenario Buscar.



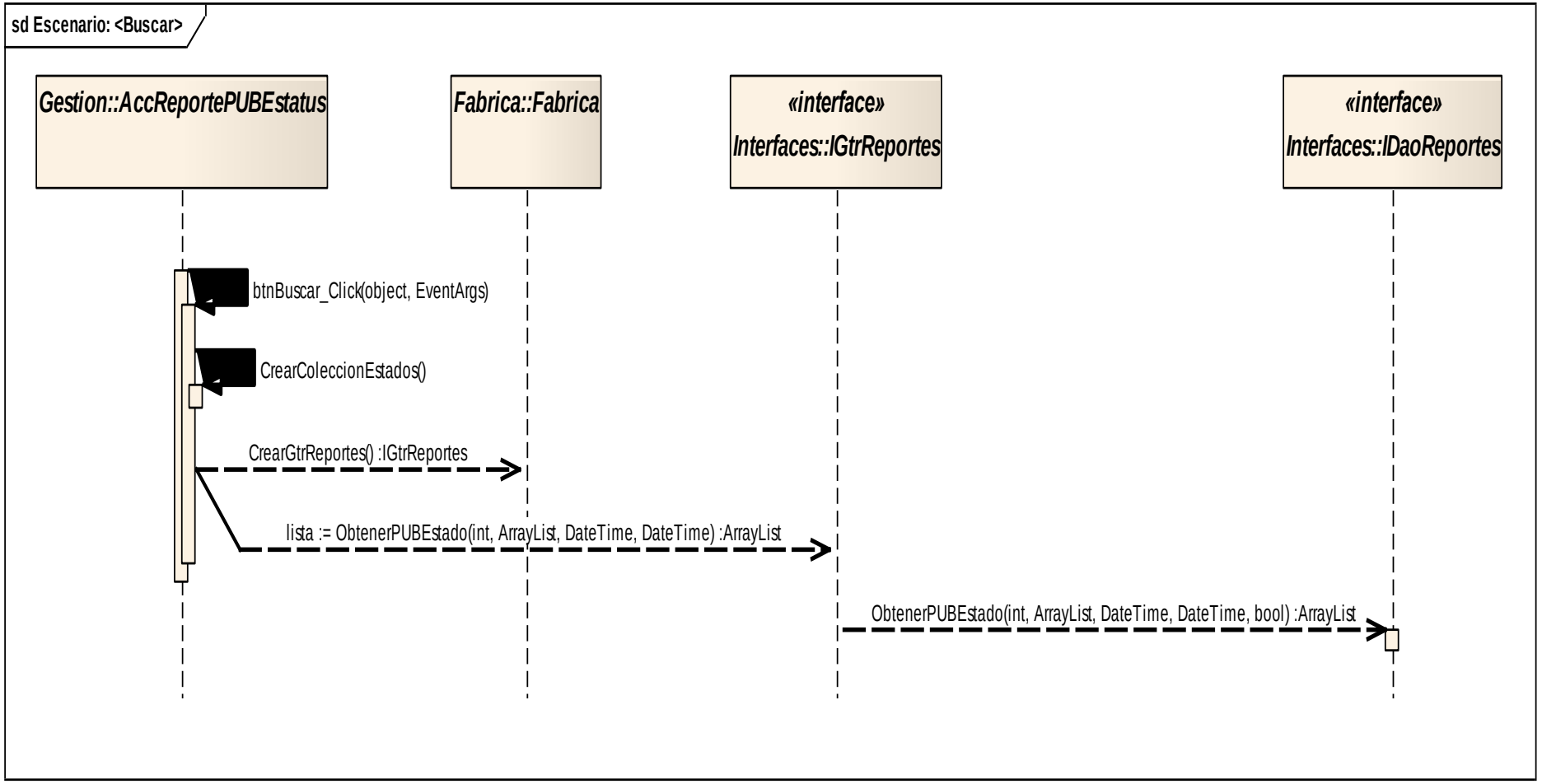
Anexo 8 Caso de Uso Reporte de Planilla Única Bancaria por Estatus.

Diagramas de Clases



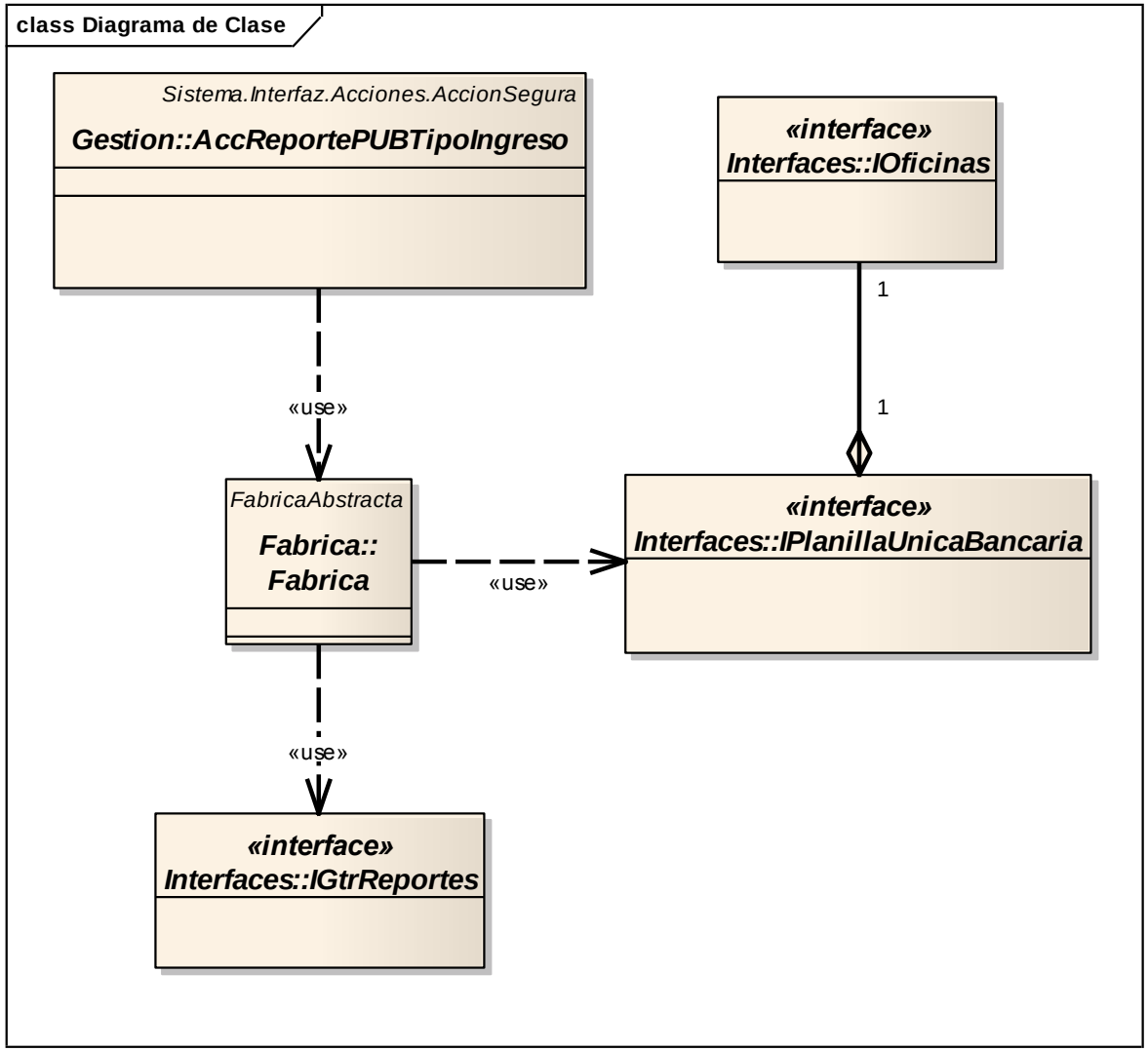
Diagramas de Interacción

Escenario Buscar.



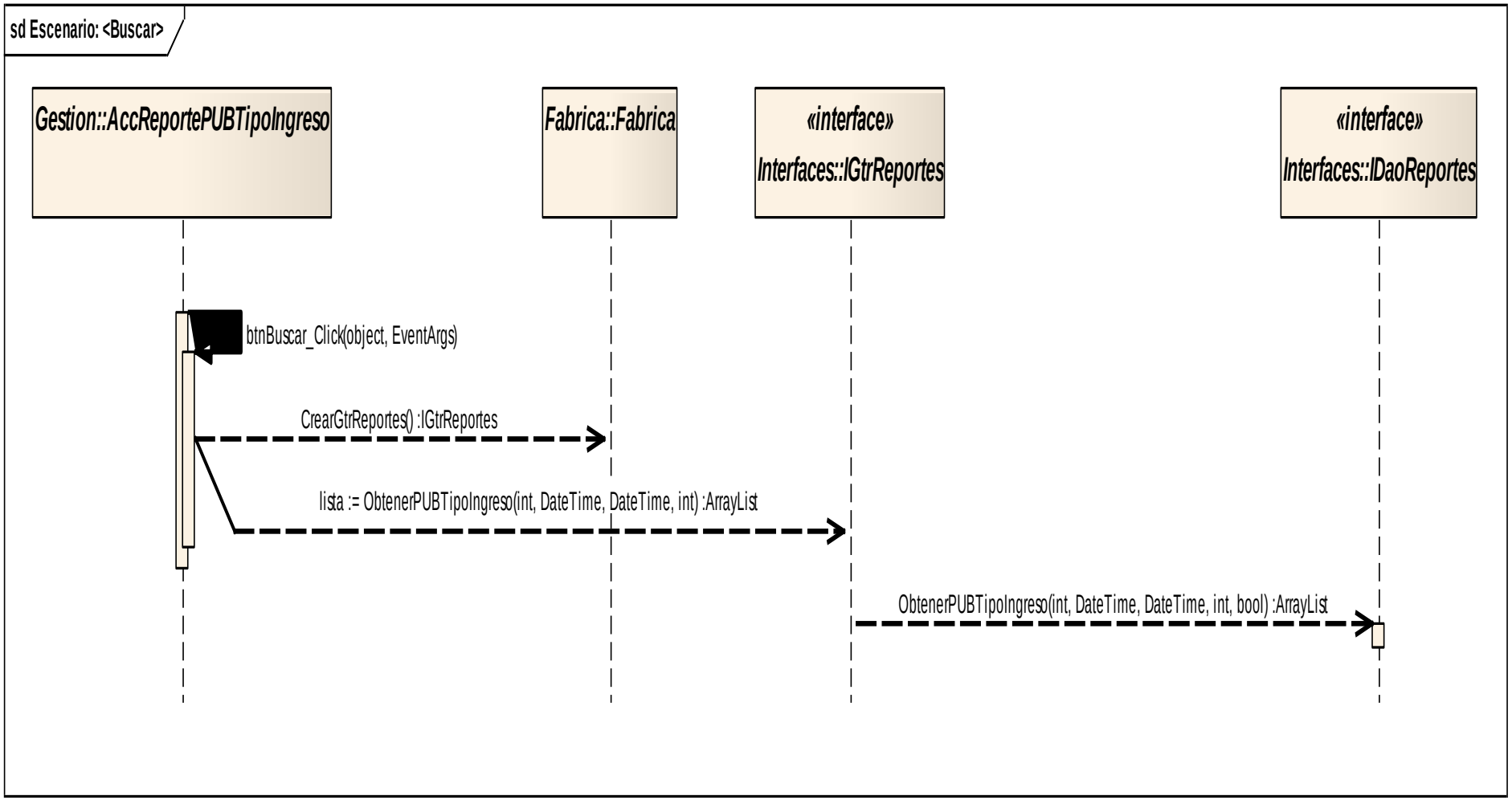
Anexo 9 Caso de Uso Reporte de Planilla Única Bancaria por Tipo de Ingreso.

Diagramas de Clases



Diagramas de Interacción

Escenario Buscar.



Anexo 10 Modelo de Datos