Universidad de las Ciencias Informáticas
Facultad 3

Título: Implementación de módulo de reportes de sistema para la gestión económica de los Registros y Notarias de la República Bolivariana de Venezuela

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autores: Daniel Mariano García Fernández

Dalgis Rogelio López Góngora

Tutor: Ing. Marbys Marante Valdivia

Consultante: Dr. Pedro Yobanis Piñero Pérez

mayo de 2007

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a 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 _____.

Dalgis R. López Góngora Daniel M. García Fernández Marbys Marante Valdivia

_____ _____ _____

AGRADECIMIENTOS

Agradecemos a todas las personas que de alguna forma ha contribuido en nuestra formación

Especialmente agradecemos a:

El Dr. Pedro Yobanis Piñero Pérez por habernos guiado hasta aquí.

René Lazo Ochoa, siempre vas delante abriendo los caminos.

Juan Carlos Montané Izaguirre, por las peleas y horas de estudios (explicas muy bien).

Liliana la diseñadora por haber sido un "camaroncito duro".

El Lic. Gustavo Carbonell (Tati)

Elsy La Hoz Guilarte y Alexis La Hoz Sarduy.

Danieluco, siempre ZET.

Ameirys Betancourt Vázquez.

Yoandris Pacheco Jerez y Onel Roselló por formar a DYOS junto a mí.

Yordis Monteserín por las veladas.

Pocholo (NHP).

Al Módulo de Administración Contable y Financiera.

La Revolución y a nuestro invicto Comandante en Jefe Fidel Castro Ruz.

DEDICATORIA

A Tita y Marcia: a ellas porque han sido en todo momento el milagro que apacigua mis angustias, sin ustedes no hubiera sido posible, serán siempre mi inspiración.

A mis amigos: Pacheco, Onel y a mi cuñado Carlitos, porque gracias a ellos se lo que es la amistad verdadera, valor importante en mi vida.

A mis tíos y tías: Nena que ha sido otra madre para mi, a Tata,Yia, Lede, Caquin, Ito, Yeya, Maria y junto a ellos a todos mis primos Pipere, Ale, Bubu, Migue, Mayo, Baby, Mimi, Mamacita que sin todos ellos tampoco hubiera sido posible.

A los que le dedico mi tesis: Por estar conmigo estos años, y por aconsejarme, regañarme, y compartir momentos agradables y momentos tristes, momentos que nos hacen crecer y valorar a las personas que nos rodean, les agradezco mucho esta oportunidad de poder contar con ustedes y su apoyo.

Daniel

A mis padres: Mami y Papi que son mis ojos, sin ellos no hubiera llegado hasta aquí, han sido mi inspiración en todo momento de mi vida. Ustedes se merecen mucho más.

A mis Abuelos: Mima, Papi, Cecilia y a la memoria de mi abuelo Rafael.

A mis Tías: Magays, Balbi, Nancy, Gloria, Raquel.

A mis Tíos: Tito, Cheo y a la memoria de tío Necho y Roly, a quienes he respetado siempre.

A mis Primos: Raiza, Diana, Aixel, Yumi, Papa, May, Liusban, Reni, Yanu, Pablito, en fin a todos sin ponerse celosos lo que se me quedaron.

A ti Monchi, eres lo mejor que me ha sucedido, tu apoyo en la recta final me sirvió mucho, Te AMO.

A mis amigos Juan Carlos y René, todavía recuerdo las palabras de René cuando nos conocimos la primera vez, y así es, somos grandes amigos.

A mis amigos los "perros" Daykel y el Goro.

A mi familia en general.

Dalgis

Resumen

El sistema de reportes contables muestra información confiable de manera dinámica, y es una potente herramienta en la planificación empresarial. En los Registros y Notarias de la República Bolivariana de Venezuela no existe estandarización en el control de los procesos contables en cada una de las entidades, ya que no hay mecanismos de validación de la información económica procesada entre ellas y los entes gubernamentales.

El presente trabajo muestra el desarrollo de un módulo de reportes para la gestión económica de los Registros y Notarías de la República Bolivariana de Venezuela, en su Sistema de Administración Contable y Financiera, la cual se pretende desplegar en todas las oficinas de la República Bolivariana de Venezuela.

Se expone la arquitectura con la que fue desarrollado el sistema de reportes contables, así como algunas de las funcionalidades propuestas para dar la solución informática que hoy necesitan cada una de las oficinas de los Registros y Notarias en la extracción de los mismos. El mayor impacto que se alcanza con este sistema de reportes es la rapidez con que se desarrollan los mismos, sin necesidad de tener mucho conocimiento para crear los módulos correspondientes; así como el valor de una información confiable acerca de cada uno de los movimientos contables y financieros que se realizan en cada una de las oficinas.

Palabras Claves

Sistema de Administración Contable y Financiera, sistema de reportes contables.

TABLA DE CONTENIDOS

INTRODUCCIÓN

Actualidad y necesidad del trabajo

El trabajo de modernización y automatización de Registros y Notarías en la República Bolivariana de Venezuela viene dando sus primeros pasos desde 1993, fecha en la cual se promulgó la Ley de Registro Público, dado que el mismo, autorizó la digitalización y almacenamiento documental de la información contenida en las Oficinas de Registros y Notarías de dicho país.

Atendiendo este mandato constitucional estratégico, el Ministerio de Interior y Justicia decidió continuar y profundizar un conjunto de acciones ya iniciadas, con el fin de dar respuestas al asunto planteado.

Entre las acciones iniciadas y concluidas en su primera etapa, desarrollado por Consultores Nacionales y el Banco Interamericano de Desarrollo, se destacan los esfuerzos realizados por el Registro Primero de Caracas, cuyo esfuerzo se concentró en la automatización del Registro Mercantil y las denominaciones, el cual permitió la automatización de 30 Registros y Notarías en el país.

El sistema contable actual en los Registros y Notarías, deja brecha que afecta de alguna manera al gobierno bolivariano. Dado que no permite una auditoría y control central de los trámites en curso, relativo a los impuestos que se cobran por su ejecución, el tiempo empleado desde la presentación hasta el otorgamiento de los trámites, y que no cuenta con una plataforma tecnológica robusta que permita determinar oportunamente la información, manejos eficientes en cada gestión y rapidez en cada uno de los procesos económicos, teniendo en cuenta que los sistemas de gestión y extracción de reportes contables y financieros actualmente en uso en cada una de estas oficinas, generan inseguridad económica, y es imposible determinar de manera organizada y confiable la información extraída.

La información que obtiene la Dirección General de Registros y Notarías no es oportuna y confiable y por ende impide determinar el control de las transacciones realizadas y los ingresos por los servicios prestados. A partir de aquí surge la necesidad de desarrollar una plataforma

tecnológica sólida que permita determinar de manera organizada y confiable la información extraída.

Situación Problémica

Actualmente los recursos que generan la prestación del servicio de Registros y Notarías en Venezuela están alrededor de 4 billones de bolívares anuales y sólo una pequeña cantidad es tributada al Fisco Nacional, lo que exige el control riguroso del pago de aranceles y tarifas.

En el 90% de las oficinas de Registros y Notarías la infraestructura es inapropiada y la plataforma tecnológica no garantiza el control efectivo y eficiente de la información económica y además no existe interconexión entre ellas, ni con el Ministerio de Interior y Justicia.

No existe ningún sistema generador de reportes, que permita la validación y confiabilidad de la información económica extraída, entre los Registros y Notarías y entes gubernamentales. Una vez mostrada la situación a la que nos enfrentamos estamos en condiciones de plantearnos nuestro problema de la siguiente manera:

Problema Científico

No existencia de un sistema de reportes para la gestión económica de los Registros y Notarías de la República Bolivariana de Venezuela, que permita garantizar la fiabilidad de la información extraída de los mismos.

Novedad práctica

Por primera vez se implementa un Sistema de gestión de reportes contables y financieros, en un ambiente gráfico y dinámico, para las oficinas de los Registros y Notarías de la República Bolivariana de Venezuela.

Objeto

Generador de módulos de reportes.

Objetivo

Desarrollar un módulo generador de reportes para la gestión económica de los Registros y Notarías de la República Bolivariana de Venezuela, en su Sistema de Administración Contable y Financiera.

Objetivos específicos

1. Crear el modelo de implementación que se refiere a la creación de reportes para la gestión contable en la gestión documental de Registros y Notarías.
2. Desarrollar un entorno amigable en la extracción de reportes contables y financieros.
3. Analizar los resultados a partir de la validación del sistema desarrollado.

Campo de acción

Módulo de Administración Contable Financiera.

Hipótesis

Si se implementa el sistema para la gestión de reportes contables y financieros en los Registros y Notarías de la República Bolivariana de Venezuela, entonces se ayuda a resolver el control económico para lograr interpretar, medir y describir la actividad económica de estos registros.

Acciones Concretas

Análisis de la herramienta para diseñar los reportes.

- Definir la puesta en práctica de los estándares de codificación que se van a utilizar en la implementación.
- Lograr mayor celeridad en la creación de reportes.
- Analizar las herramientas según la arquitectura establecida.
- Analizar la plataforma Microsoft Visual Studio .Net 2003.
- Analizar la versión de framework de .Net a utilizar para el desarrollo.
- Diseñar casos de pruebas.
- Realizar pruebas de caja blanca.
- Realizar pruebas de caja negra.

- Analizar la ejecución y resultados de las pruebas.

El presente trabajo cuenta de tres capítulos:

En el capítulo 1 se realiza un estudio del estado del arte a cerca de las principales técnicas de programación existente a nivel mundial, nacional y en la Universidad, además de un análisis sobre las tendencias, tecnologías y metodologías utilizadas en la actualidad a nivel internacional, en el desarrollo de aplicaciones de escritorio y Web, así como las plataformas de desarrollo que las soportan.

En el capítulo 2 se hace una descripción de la solución propuesta se definen y describen las clases y algoritmos utilizados para la solución que se propone, así como los patrones y las estructuras de datos utilizadas.

En el capítulo 3 se hace referencia a la validación de la solución propuesta, a través de los test de unidades, se aplican métricas de código y se realiza una evaluación de los resultados obtenidos.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

1.1. Introducción

En el presente capítulo se realiza un estudio del estado del arte a cerca de las técnicas de programación existentes, además de un análisis de las tendencias, tecnologías y metodologías utilizadas en la actualidad a nivel internacional, en el desarrollo de aplicaciones de escritorio y Web, así como las plataformas de desarrollo que las soportan, fundamentándose las seleccionadas para la solución que se propone.

1.2. Microsoft.Net

.NET es un proyecto de Microsoft 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. Basado en esta plataforma, Microsoft intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el Sistema Operativo hasta las herramientas de mercado.

.NET podría considerarse una respuesta de Microsoft al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de Sun Microsystems (1).

A largo plazo Microsoft pretende reemplazar la API Win32 o Windows API con la plataforma .NET. Esto debido a que la API Win32 o Windows API fue desarrollada sobre la marcha, careciendo de documentación detallada, uniformidad y cohesión entre sus distintos componentes, provocando múltiples problemas en el desarrollo de aplicaciones para el sistema operativo Windows. La plataforma .NET pretende solventar la mayoría de estos problemas proveyendo un conjunto único y expandible con facilidad, de bloques interconectados, diseñados de forma uniforme y bien documentados, que permitan a los desarrolladores tener a mano todo lo que necesitan para producir aplicaciones sólidas.

Debido a las ventajas que la disponibilidad de una plataforma de este tipo puede darle a las empresas de tecnología y al público en general, muchas otras empresas e instituciones se han unido a Microsoft en el desarrollo y fortalecimiento de la plataforma .NET, ya sea por medio de la implementación de la plataforma para otros sistemas operativos aparte de Windows (Proyecto

Mono de Ximian/Novell para Linux/MacOS X/BSD/Solaris), el desarrollo de lenguajes de programación adicionales para la plataforma (ANSI C de la Universidad de Princeton, NetCOBOL de Fujitsu, Delphi de Borland, entre otros) o la creación de bloques adicionales para la plataforma (como controles, componentes y bibliotecas de clases adicionales); siendo algunas de ellas software libre, distribuibles ciertas bajo la licencia GPL.

Con esta plataforma Microsoft incursiona de lleno 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.

.NET intenta ofrecer una manera rápida y económica pero a la vez segura y robusta de desarrollar aplicaciones o como la misma plataforma las denomina, soluciones 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.

1.2.1 .NET Framework

El framework o marco de trabajo, constituye la base de la plataforma .NET y 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 (2).

Bajo el nombre .NET Framework o Marco de trabajo .NET se encuentran reunidas una serie de normas impulsadas por varias compañías además de Microsoft (como Hewlett-Packard , Intel, IBM, Fujitsu Software, Plum Hall, la Universidad de Monash), entre las cuales se encuentran:

La norma que define las reglas que debe seguir un lenguaje de programación para ser considerado compatible con el marco de trabajo .NET (ECMA-335, ISO/IEC 23271).

Por medio de esta norma se garantiza que todos los lenguajes desarrollados para la plataforma ofrezcan al programador un conjunto mínimo de funcionalidad, y compatibilidad con todos los demás lenguajes de la plataforma.

La norma que define el lenguaje C# (ECMA-334, ISO/IEC 23270)

Este es el lenguaje insignia del marco de trabajo .NET, y pretende reunir las ventajas de lenguajes como C/C++ y Visual Basic en un solo lenguaje.

La norma que define el conjunto de funciones que debe implementar la librería de clases base (BCL por sus siglas en inglés) (incluido en ECMA-335, ISO/IEC 23271)

Tal vez el más importante de los componentes de la plataforma, esta norma define un conjunto funcional mínimo que debe implementarse para que el marco de trabajo sea soportado por un sistema operativo. Aunque Microsoft implementó esta norma para su sistema operativo Windows, la publicación de la norma abre la posibilidad de que sea implementada para cualquier otro sistema operativo existente o futuro, permitiendo que las aplicaciones corran sobre la plataforma independientemente del sistema operativo para el cual haya sido implementada. El Proyecto Mono emprendido por Ximian pretende realizar la implementación de la norma para varios sistemas operativos adicionales bajo el marco del software libre o código abierto.

Los principales componentes del marco de trabajo son:

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

Debido a la publicación de la norma para la infraestructura común de lenguajes (CLI por sus siglas en inglés), el desarrollo de lenguajes se facilita, por lo que el marco de trabajo .NET soporta ya más de 20 lenguajes de programación y es posible desarrollar cualquiera de los tipos de aplicaciones soportados en la plataforma con cualquiera de ellos, lo que elimina las diferencias que existían entre lo que era posible hacer con uno u otro lenguaje. Algunos de los lenguajes desarrollados para el marco de trabajo .NET son: C#, Visual Basic, Turbo Delphi for .NET C++, J#, Perl, Python, Fortran y Cobol.NET.

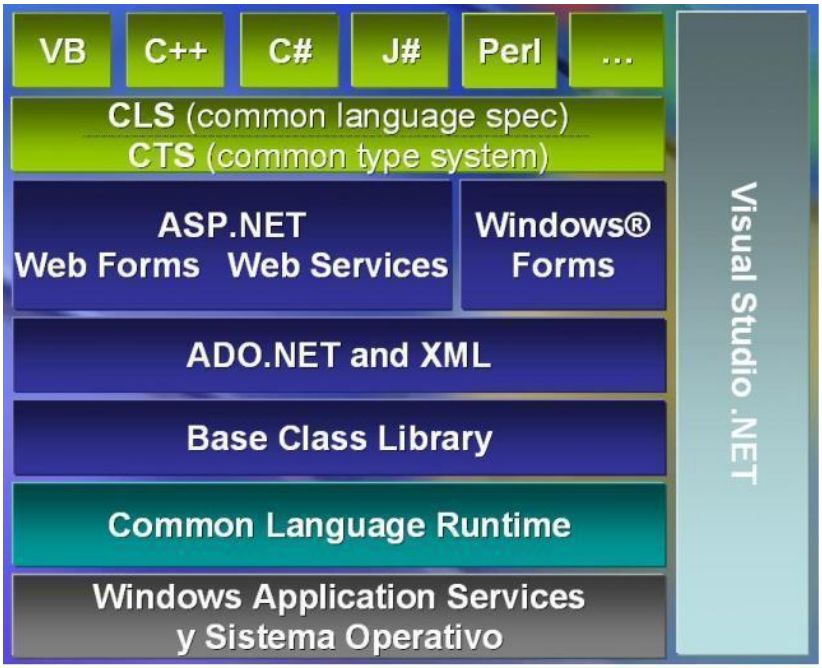

Figura 1 Diagrama detallado del Marco de Trabajo .NET

1.2.2 Visual Studio .NET

Visual Studio .NET es un IDE desarrollado por Microsoft a partir de 2002. Es para el sistema operativo Microsoft Windows y está pensado, principal pero no exclusivamente, para desarrollar plataformas Win32 (3).

La característica más notable del IDE es su soporte de los nuevos lenguajes .NET. Los programas desarrollados en esos lenguajes no se compilan a código máquina ejecutable (como por ejemplo hace C++) sino que son compilados a algo llamado CIL. Cuando los programas ejecutan la aplicación CIL, ésta es compilada en ese momento al código de máquina apropiado para la plataforma en la que se está ejecutando. Mediante este método, Microsoft espera poder soportar varias implementaciones de sus sistemas operativos Windows (como Windows CE). Los programas compilados a CIL pueden ejecutarse sólo en plataformas que tengan una implementación de .NET Framework. Es posible ejecutar programas CIL en Linux o en Mac OS X utilizando algunas implementaciones .NET que no pertenecen a Microsoft, como Mono y DotGNU.

1.2.3 Common Language Runtime (CLR)

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 (4).

La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un código intermedio (MSIL, Microsoft Intermediate Lenguaje), similar al BYTECODE de Java. Para generar dicho código el compilador se basa en el Common Language Specification (CLS) que determina las reglas necesarias para crear ese código MSIL compatible con el CLR.

Para ejecutarse se necesita un segundo paso, un compilador JIT (Just-In-Time) es el que genera el código máquina real que se ejecuta en la plataforma del cliente.

De esta forma se consigue con .NET independencia de la plataforma hardware.

La compilación JIT la realiza el CLR a medida que el programa invoca métodos, el código ejecutable obtenido, se almacena en la memoria caché del ordenador, siendo recompilado de nuevo sólo en el caso de producirse algún cambio en el código fuente.

1.2.4 Biblioteca de Clases Base de .NET

La Biblioteca de Clases Base (BCL por sus siglas en inglés) maneja la mayoría de las operaciones básicas que se encuentran involucradas en el desarrollo de aplicaciones, incluyendo entre otras:

- Interacción con los dispositivos periféricos
- Manejo de datos (ADO.NET)
- Administración de memoria
- Cifrado de datos
- Transmisión y recepción de datos por distintos medios (XML, TCP/IP)
- Administración de componentes Web que corren tanto en el servidor como en el cliente (ASP.NET)
- Manejo y administración de excepciones
- Manejo del sistema de ventanas

- Herramientas de despliegue de gráficos (GDI+)
- Herramientas de seguridad e integración con la seguridad del sistema operativo
- Manejo de tipos de datos unificado
- Interacción con otras aplicaciones
- Manejo de cadenas de caracteres y expresiones regulares
- Operaciones aritméticas
- Manipulación de fechas, zonas horarias y periodos de tiempo
- Manejo de arreglos de datos y colecciones
- Manipulación de archivos de imágenes
- Aleatoriedad
- Generación de código
- Manejo de idiomas
- Auto descripción de código
- Interacción con el API Win32 o Windows API.
- Compilación de código
- Esta funcionalidad se encuentra organizada por medio de espacios de nombres jerárquicos.
- La Biblioteca de Clases Base se clasifica, en tres grupos clave:
  - ASP.NET y Servicios Web XML
  - Windows Forms
  - DO.NET

1.2.5 Ensamblados

Los ensamblados son ficheros con forma de EXE o DLL que contienen toda la funcionalidad de la aplicación de forma encapsulada.

Con los ensamblados ya no es necesario registrar los componentes de la aplicación.

1.3. Java 2 Enterprise Edition

Java Platform, Enterprise Edition o Java EE (anteriormente conocido como Java 2 Platform, Enterprise Edition o J2EE hasta la versión 1.4), es una plataforma de programación parte de la Plataforma Java para desarrollar y ejecutar software de aplicaciones en lenguaje de programación Java con arquitectura de n niveles distribuida, basándose ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones. La plataforma Java EE está definida por una especificación. Similar a otras especificaciones del Java Community Process, Java EE es también considerada informalmente como un estándar debido a que los suministradores deben cumplir ciertos requisitos de conformidad para declarar que sus productos son conformes a Java EE; no obstante sin un estándar de ISO o ECMA (5).

Java EE incluye varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc., y define como coordinarlos. Java EE también configura algunas especificaciones únicas para Java EE para componentes. Estas incluyen Enterprise JavaBeans, servelets, portlests (siguiendo la especificación de Portlets Java), JavaServer Pages y varias tecnologías de servicios web. Esto permite al desarrollador crear una aplicación de empresa que es portable entre plataformas y escalable. Otros beneficios añadidos son, por ejemplo, que el servidor de aplicaciones puede manejar las transacciones, seguridad, escalabilidad, concurrencia y gestión de los componentes que son desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de las tareas de mantenimiento de bajo nivel.

1.4. Crystal Report

Crystal Report permite la elaboración de reportes en .NET. Esta herramienta incluye diversos controles para la graficación, un soporte de visualización múltiple paginado, un panel de tabla de contenidos con una nueva pestaña de miniaturas, posibilidad de búsqueda de informes y personalización completa de su barra de herramientas (6).

Se integra con Visual Studio .NET permitiendo la utilización desde c# o Vb, incluye filtros de exportación a los formatos Adobe PDF, Microsoft Excel, Microsoft Word, RTF, HTML y DHTML.

También se incluye un visor Web Viewer que aprovecha los manejadores de ASP.NET para permitirle visualizar los informes sin necesidad de escribir código personalizado para exportar.
Las versiones disponibles son:
Crystal Reports Server: Novedad entre las diferentes ediciones, al proporcionar una solución completa para diseñar, crear, gestionar y distribuir todos los informes de la empresa. Viene a rellenar el vacío existente para la gestión de informes corporativos en empresas que no podían adquirir Crystal Enterprise por quedar fuera de sus presupuestos.
Crystal Reports Developer Edition: Esta edición permite integrar capacidades de visualización, exportación e impresión de informes en las aplicaciones.
Crystal Reports Professional Edition: Diseñado para crear y mantener informes con los requisitos de diseño más exigentes. Además incorpora la posibilidad de acceder a fuentes de datos corporativas.
La versión actual de Crystal Reports es la versión XI. Crystal Reports está diseñado para conectarse a cualquier base de datos. Los productos pueden ser en idioma inglés o español. Cada paquete incluye una media de instalación y la llave de activación para una licencia.
La forma de licenciamiento que tiene Crystal Reports: Cada usuario que diseñe reportes, modifique reportes debe tener una licencia instalada, las personas que vean la pantalla del reporte no necesitan tener instalada una licencia. Crystal Reports permite exportar el reporte a diferentes formatos.

1.5. ActiveReports for .NET v2.0

La segunda versión de ActiveReports for .NET incorpora diversas y nuevas características para aumentar las capacidades de generación de informes alabadas por los desarrolladores que han utilizado la versión anterior. Entre ellas podemos destacar la incorporación de un nuevo control de gráficos comerciales muy potente, las miniaturas de páginas en el visor de Windows, el soporte HTML y el soporte mejorado para tablas en el control RichTextBox, un editor de guiones mejorado con resaltado de sintaxis, enlace a datos contra clases personalizadas que soporten la interfaz IList y mejoras en la detección de tiempos límites de respuesta en conexiones a bases de datos (7).

ActiveReports for .NET está escrito completamente en C# y ofrece integración con Visual Studio .NET. Esto permite a los desarrolladores de Visual Studio .NET utilizar ActiveReports for .NET desde su lenguaje favorito (C# o Visual Basic .NET). ActiveReports for .NET se licencia por desarrollador y es de libre distribución. El producto incluye un asistente de informes y un asistente de conversión de informes de Microsoft Access para trabajar rápidamente. ActiveReports for .NET incluye filtros de exportación a los formatos Adobe PDF, Microsoft Excel, RTF, HTML, Texto y TIFF, y un visor Windows Viewer con soporte para la visualización simultánea de múltiples páginas, un panel de Tabla de Contenidos con una nueva pestaña de miniaturas, posibilidad de búsqueda de informes y personalización completa de su barra de herramientas.

La edición Professional de ActiveReports for .NET ofrece un Diseñador de Informes para usuario final que le permite embeber el diseñador en sus propias aplicaciones para que sus usuarios puedan crear y modificar sus propios informes. También se incluye un visor Web Viewer que aprovecha los manejadores de ASP.NET para permitirle visualizar los informes sin necesidad de escribir código personalizado para exportar a los formatos más utilizados como HTML y PDF.

Novedades en versión 2.0 (ambas ediciones):

- Control de gráficos comerciales totalmente integrado
- Control RichText con soporte para tablas RTF y etiquetas HTML
- Soporte para el enlace a datos contra clases que implementan IList
- Exportación mejorada a formato PDF

Edición Standard

- Diseñador de informes integrado
- Soporte para fuentes de datos OleDB, SQL Server y XML, además de ADO.NET (lectores de datos, tablas y conjuntos de datos)
- Incluye un visor Windows Viewer personalizable
- Utilidades de importación de informes Crystal y Microsoft Access
- Exportación a HTML, PDF, Excel, RTF, TIFF y Texto

Edición Professional

- Incluye todas las características de la edición Standard
- Incluye un control Diseñador de Informes para usuarios finales
- Incluye un visor Web Viewer ASP.NET

1.6. (XP) Programación extrema

XP es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico (8).

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

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

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración. El ciclo de vida ideal de XP consiste de seis fases: Exploración, Planificación de la Entrega *(release)*, Iteraciones, Producción, Mantenimiento y Muerte del Proyecto.

La metodología se basa en:

- Pruebas Unitarias: se basa en las pruebas realizadas a los principales procesos, de tal manera que adelantándonos en algo hacia el futuro, podamos hacer pruebas de las fallas que pudieran ocurrir. Es como si nos adelantáramos a obtener los posibles errores.
- Refabricación: se basa en la reutilización de código, para lo cual se crean patrones o modelos estándares, siendo más flexible al cambio.
- Programación en pares**:** una particularidad de esta metodología es que propone la programación en pares, la cual consiste en que dos desarrolladores participen en un proyecto en una misma estación de trabajo. Cada miembro lleva a cabo la acción que el otro no está haciendo en ese momento. Es como el chofer y el copiloto: mientras uno conduce, el otro consulta el mapa.

1.7. Proceso Unificado de Desarrollo de Software

El RUP es un proceso de desarrollo de software dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental. Pretende implementar las mejores prácticas en ingeniería de software, con el objetivo de asegurar la producción de software de calidad, dentro de plazos y presupuestos predecibles (9).

Principales características:

1. Desarrollo iterativo: Permite una comprensión creciente de los requerimientos, a la vez que se va haciendo crecer el sistema. RUP sigue un modelo iterativo que aborda las tareas más riesgosas primero. Así se logra reducir los riesgos del proyecto y tener un subsistema ejecutable tempranamente.
2. Administración de requerimientos: RUP describe cómo obtener los requerimientos, cómo organizarlos, cómo documentar los requerimientos de funcionalidad y restricciones, cómo rastrear y documentar las decisiones, y cómo captar y comunicar los requerimientos del negocio.
3. Arquitecturas basadas en componentes: El proceso se basa en diseñar tempranamente una arquitectura base ejecutable. Esta arquitectura debe ser: flexible, fácil de modificar, intuitivamente comprensible, y debe promover la reutilización de componentes.

4. Modelamiento visual: RUP propone un modelamiento visual de la estructura y el comportamiento de la arquitectura y las componentes. En este esquema, los bloques de construcción deben ocultar detalles, permitir la comunicación en el equipo de desarrollo, y permitir analizar la consistencia entre las componentes, entre el diseño y entre la implementación. UML es la base del modelamiento visual de RUP.
5. Verificación de la calidad del software: No sólo la funcionalidad es esencial, también el rendimiento y la confiabilidad. RUP ayuda a planificar, diseñar, implementar, ejecutar y evaluar pruebas que verifiquen estas cualidades.
6. Control de cambios: Los cambios son inevitables, pero es necesario evaluar si éstos son necesarios y también es necesario rastrear su impacto. RUP indica como controlar, rastrear y monitorear los cambios dentro del proceso iterativo de desarrollo.

RUP divide el proceso de desarrollo en ciclos, donde se obtiene un producto al final de cada ciclo. Cada ciclo se divide en cuatro fases: Concepción, Elaboración, Construcción, y Transición. Cada fase concluye con un hito bien definido donde deben tomarse ciertas decisiones**.**

Fase de concepción

En esta fase se establece la oportunidad y alcance el proyecto. Se identifican todas las entidades externas con las que se trata (actores) y se define la interacción en un alto nivel de abstracción: se deben identificar todos los casos de uso, y se deben describir algunos en detalle. La oportunidad del negocio incluye: definir los criterios de éxito, identificación de riesgos, estimación de recursos necesarios, y plan de las fases incluyendo hitos.

Fase de elaboración

Definir y validar una arquitectura estable. Se hace un refinamiento de la Visión del sistema, basándose en nueva información obtenida durante esta fase, se establece una sólida comprensión de los casos de uso más críticos que definen las decisiones arquitectónicas y de planificación. Creación de los planes de desarrollo detallados para las iteraciones de la fase de construcción.

Fase de construcción

Gestión de los recursos, optimización y control de los procesos de construcción del software. Se completa el desarrollo de los componentes y/o subsistemas, probándolos contra un conjunto definido de criterios aprobados al inicio del proyecto.

Fase de transición

Ejecución de los planes de implantación. Se finalizan los manuales de usuario y mantenimiento. Pruebas del sistema en el entorno de explotación. Creación de una *release* del sistema. Validación del sistema por los usuarios. Ajuste fino del sistema según la validación con el usuario. Se facilita la transición del sistema al personal de mantenimiento. Se pone el producto a disposición del usuario.

Metodologías Ágiles	**Metodologías Tradicionales**
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El clientes parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

Figura 2 Diferencias entre metodologías ágiles y no ágiles

1.8. NUnit

NUnit es una herramienta que 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 (9).

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.

NUnit 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 como 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.

NUnit.Forms es una expansión al framework núcleo NUnit y es también open source. Esto busca concretamente ampliar NUnit para que sea capaz de manejar pruebas de elementos de interfaz de usuario en Windows Forms.

NUnit.ASP es una expansión al framework núcleo NUnit y es también open source. Esto busca concretamente ampliar NUnit para que sea capaz de manejar pruebas de elementos de interfaz de usuario en ASP.NET.

1.9. JUnit

JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente (10).

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.
El propio framework incluye formas de ver los resultados (runners) que pueden ser en modo texto, gráfico (AWT o Swing).
En la actualidad las herramientas de desarrollo como NetBeans y Eclipse cuentan con plug-ins que permiten que la generación de las plantillas necesarias para la creación de las pruebas de una clase Java se realice de manera automática, facilitando al programador enfocarse en la prueba y el resultado esperado, y dejando a la herramienta la creación de las clases que permiten coordinar las pruebas.

1.10. Desarrollo basado en pruebas

Desarrollo guiado por pruebas, o Test-driven development (TDD) es una técnica de programación enfatizada en la programación extrema. Esencialmente la técnica implica el escribir primero sus pruebas y luego implementar el código para ejecutarla. La meta del desarrollo conducido por las pruebas es lograr una rápida retroalimentación e implementa el "ilustrar la línea principal" al hacer un programa. Muchos enfatizan que el desarrollo conducido por las pruebas es sobre todo un método de diseño de software, no solo un método de pruebas (11).
Para que funcione el desarrollo guiado por pruebas, el sistema tiene que ser lo suficientemente flexible como para permitir el testeo automático de software, usando casos de prueba que devuelven un simple verdadero o falso en su evaluación. Estas propiedades permiten una rápida retroalimentación en el diseño y la corrección. Frameworks como JUnit y Nunit proveen de un mecanismo para manejar y ejecutar conjuntos de pruebas automatizadas.

1.10.1 Ciclo De Desarrollo Prueba-Conducida

Escribir la prueba: Se comienza escribiendo una prueba. Para escribir la prueba, el desarrollador debe entender claramente las especificaciones y los requisitos. Esto se logra con casos de uso e historias.

Escribir el código: El paso siguiente es escribir el código haciendo que pase la prueba. Este paso fuerza a programador tomar la perspectiva de un cliente considerando el código a través de sus interfaces. Ésta es la parte conducida por el diseño, del TDD.

Ejecutar las pruebas automatizadas: El paso siguiente es ejecutar los casos de prueba automatizados y observar si pasan o fallan. Si pasan, el programador puede garantizar que el código resuelve los casos de prueba escritos. Si hay fallos, el código no resolvió los casos de prueba.

Refactorización: El paso final es la refactorización, aquí está cualquier necesidad de limpieza en el código. Después se vuelven a efectuar los casos de prueba y se observan los resultados.

Repetición: Después se repetirá el ciclo y se comenzará a agregar las funcionalidades adicionales o a arreglar cualquier error.

Hay las varias maneras en que puede usarse el TDD y el más común se basa en los principios de "déjelo simple" ("Keep It Simple, Stupid" (KISS)) y "usted no va a necesitarlo" ("You Ain't Gonna Need It" (YAGNI)). Este estilo se centra en escribir solo el código necesario para pasar las pruebas. Los principios del diseño y de la característica se echan a un lado en nombre de la simplicidad y de la velocidad. Por lo tanto, mientras pasan las pruebas se puede romper cualquier regla. Esto puede ser inquietante para muchos al principio pero permitirá que el programador se centre solamente lo importante. Sin embargo, el programador debe pagar un precio mayor posteriormente en el paso "refactorización" del ciclo, puesto que el código se debe limpiar hasta un nivel razonable antes que el ciclo se pueda repetir. Otra variación del Test Driven Development requiere que el programador primero falle en los casos de prueba. La idea es asegurarse de que los casos de prueba realmente funcionen y puedan recoger (catch) un error. Una vez que se demuestre esto comenzará el ciclo normal. Ésta es una de las variaciones más populares y se ha acuñado el "Test-Driven Development Mantra", conocido como rojo/verde/refactorizar donde el rojo significa falla y el verde es pasa.

Ventajas

A pesar de los requisitos iniciales, el desarrollo guiado por pruebas (TDD) puede proporcionar gran valor para la creación de software mejor y más rápidamente. Ofrece más que una simple

validación de la corrección, también puede guiar el diseño de un programa. Centrándose en primer lugar en los casos de prueba uno debe imaginarse cómo los clientes utilizarán la funcionalidad (en este caso, los casos de prueba). Por lo tanto, al programador solo le importa el interfaz y no la implementación. El poder del TDD radica en la capacidad de avanzar en pequeños pasos cuando se necesita. Permite que un programador se centre en la tarea actual y la primera meta es a menudo hacer que la prueba pase. Inicialmente no se consideran los casos excepcionales y el manejo de errores. Estos, se implementan después de que se haya alcanzado la funcionalidad principal. Otra ventaja es que, cuando está utilizada correctamente, se asegura de que todo el código escrito es cubierto por una prueba. Esto puede dar al programador un mayor nivel de la confianza en el código.

Limitaciones

El desarrollo guiado por pruebas no funciona en un ambiente donde no es factible la prueba automatizada.

Es también importante observar que el desarrollo guiado por pruebas prueba únicamente la corrección del diseño y de la funcionalidad según los casos de prueba escritos. Un caso de prueba incorrecto no evita que las especificaciones produzcan código incorrecto. Por lo tanto, el énfasis en la corrección y el diseño se ha llevado a la escritura de los casos de prueba, puesto que son los conductores. Consecuentemente, el desarrollo guiado por las pruebas es solamente tan bueno como lo son las pruebas.

Algunos practicantes de esta metodología dicen que no se puede utilizar en el desarrollo de aplicaciones multihilos o en aquellas aplicaciones donde la seguridad es una parte esencial aunque hay algunos que no lo creen de esta forma.

1.10.2 Descripción de los test de unidades

La prueba unitaria es un procedimiento utilizado para validar que un módulo de código fuente funciona apropiadamente. Son comprobaciones que se le hacen a las unidades lógicas del software. Se verifica que una unidad funciona correctamente por sí misma, sin tener en cuenta las relaciones que pueda tener con otras partes del sistema. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado. Luego, con las Pruebas de

Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión (12).

El objetivo de realizar las pruebas unitarias es aislar, cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el segmento de código debe satisfacer. Estas pruebas aisladas proporcionan cinco ventajas básicas:

1. **Fomentan el cambio:** Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
2. **Simplifica la integración:** Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las pruebas de integración.
3. **Documenta el código:** Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
4. **Separación de la interfaz y la implementación**.
5. **Los errores están más acotados y son más fáciles de localizar:** dado que tenemos pruebas unitarias que pueden desenmascararlos.

La realización de las pruebas unitarias no descubrirán todos los errores del código. Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, y otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas que puede recibir en realidad la unidad de programa bajo estudio.

1.10.3 Prueba del camino básico

La prueba del camino básico es una técnica de prueba de la Caja Blanca propuesta por Tom McCabe. Esta técnica permite obtener una medida de la complejidad lógica de un diseño y usar esta medida como guía para la definición de un conjunto básico.

La idea es derivar casos de prueba a partir de un conjunto dado de caminos independientes por los cuales puede circular el flujo de control. Para obtener dicho conjunto de caminos independientes se construye el Grafo de Flujo asociado y se calcula su complejidad ciclomática.
La complejidad ciclomática es una métrica de software extremadamente útil pues proporciona una medición cuantitativa de la complejidad lógica de un programa. El valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecute cada sentencia al menos una vez (13).
Un camino independiente es cualquier camino del programa que introduce por lo menos un nuevo conjunto de sentencias de procesamiento o una nueva condición. El camino independiente se debe mover por lo menos por una arista que no haya sido recorrida anteriormente.
Existen tres formas de calcular la complejidad ciclomática:

1. V(G) = Nº de regiones del grafo
2. V(G) = Nº de Aristas - Nº de Nodos + 2
3. V(G) = Nº de Nodos Predicado + 1

Donde:
Nodos (círculos): representan una o más acciones del módulo.
Aristas (flechas): representan el flujo de control entre los distintos nodos.
Nodos predicados: son aquellos que contienen una condición, por lo que de ellos emergen dos o más aristas.

1.11. Estructuras de datos

En programación, una estructura de datos es una forma de organizar un conjunto de datos elementales (un dato elemental es la mínima información que se tiene en el sistema) con el objetivo de facilitar la manipulación de estos datos como un todo o individualmente (14).
Una estructura de datos define la organización e interrelación de estos, y un conjunto de operaciones que se pueden realizar sobre él. Las operaciones básicas son:

- Alta, adicionar un nuevo valor a la estructura.
- Baja, borrar un valor de la estructura.

- Búsqueda, encontrar un determinado valor en la estructura para realizar una operación con este valor, en forma SECUENCIAL o BINARIO (siempre y cuando los datos estén ordenados).

  Otras operaciones que se pueden realizar son:
- Ordenamiento, de los elementos pertenecientes a la estructura.
- Apareo, dadas dos estructuras originar una nueva ordenada y que contenga a las apareadas.

Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada para cada problema depende de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos.

Una de las novedades realmente únicas en .NET es la unificación del sistema de tipos que resuelve de forma brillante muchos de los problemas que siempre se han encontrado en el desarrollo de software. El runtime llamado CLR (Common Lenguaje Runtime) que .NET proporciona, presenta de entre sus muchas características el conjunto de tipos común estandarizado para todos los lenguajes soportados, es el llamado CTS (Common Type System).

Common Type System (CTS): Es el conjunto de reglas que han de seguir las definiciones de tipos de datos para que el CLR las acepte. Es decir, aunque cada lenguaje gestionado disponga de su propia sintaxis para definir tipos de datos, en el MSIL resultante de la compilación de sus códigos fuente se han de cumplir las reglas del CTS. Divide además los tipos de datos en dos categorías:

Tipos por valor: Son aquellos tipos que contienen los datos, el valor en si. Se almacenan directamente en el disco duro de la computadora y cuando trabajamos con ellos, trabajamos directamente con el valor. No requiere de memoria adicional aparte de la estrictamente necesaria para almacenar el valor. Estos tipos son implementados por el runtime y aumentan considerablemente la velocidad.

Tipos por referencia: Son aquellos que hacen referencia a una posición de memoria, están guardados en la pila. Son más lentos que los tipos por valor y ocupan algo más de memoria, a

cambio obtienen varias ventajas como por ejemplo que pueden contener métodos, polimorfismo, entre otros.

En el sistema se utilizó la biblioteca System.Collections la cual contiene interfaces y clases que definen varias colecciones de objetos, como listas, colas, matrices de bits, tablas hash y diccionarios.

La clase CollectionBase es la calse principal de las mayorias de las clases que se encuentran en este namespace. EL ArrayList es una de las clases que implementa CollectionBase, muy importantes en la plataforma .Net pues puede admitir de forma segura varios sistemas de lectura a la vez, siempre y cuando no se modifique la colección. Para garantizar la seguridad para los subprocesos de ArrayList, todas las operaciones deben realizarse a través del contenedor devuelto por el método Synchronized.

La enumeración a través de una colección es un procedimiento sin seguridad intrínseca para la ejecución de subprocesos. Aunque una colección esté sincronizada, otros subprocesos pueden modificarla, lo que hace que el enumerador inicie una excepción. Con el fin de garantizar la seguridad para la ejecución de subprocesos durante la enumeración, se puede bloquear la colección durante la totalidad de la enumeración o detectar las excepciones debidas a cambios efectuados por otros subprocesos.

La capacidad de un objeto ArrayList es el número de elementos que puede contener la lista. Conforme se agregan elementos a un objeto ArrayList, la capacidad aumenta automáticamente según lo requiera la reasignación. La capacidad puede reducirse llamando a TrimToSize o estableciendo la propiedad Capacity de forma explícita. Los índices de esta colección están basados en cero. ArrayList acepta una referencia nula como un valor válido y permite elementos duplicados.

El Hashtable es otra de las estructuras de datos de gran importancia en esta Biblioteca. El Hashtable representa una colección de pares de clave y valor organizados en función del código hash de la clave. Cada elemento es un par de clave y valor almacenado en un objeto DictionaryEntry. Una clave no puede ser una referencia nula, pero un valor sí puede serlo.

Los objetos utilizados como claves en Hashtable deben implementar o heredar los métodos Object.GetHashCode y Object.Equals. Si la igualdad de clave fuera solamente igualdad de referencia, la implementación heredada de estos métodos sería suficiente. Así mismo, estos métodos deben dar los mismos resultados cuando se llamen con los mismos parámetros mientras la clave esté incluida en Hashtable. Los objetos de claves deben permanecer inmutables mientras se utilicen como claves en Hashtable.

Cuando se agrega un elemento a Hashtable, el elemento se coloca en un sector de almacenamiento en función del código hash de la clave. Las búsquedas posteriores de la clave utilizarán su código hash para buscar en un sector de almacenamiento determinado solamente; de este modo, se reducirá considerablemente el número de comparaciones de clave necesarias para encontrar un elemento.

El factor de carga de Hashtable determina la relación máxima de elementos por sectores de almacenamiento. Factores de carga más pequeños causan tiempos de búsqueda más largos a costa de un mayor consumo de memoria. El factor de carga predeterminado de 1,0 suele proporcionar el mejor equilibrio entre velocidad y tamaño. También se puede especificar un factor de carga diferente cuando se cree Hashtable.

Conforme se agregan elementos a Hashtable, va aumentando el factor de carga real de Hashtable. Cuando el factor de carga real alcanza el factor de carga especificado, el número de sectores de almacenamiento en Hashtable aumenta automáticamente al número primo más pequeño que sea superior al doble del número actual de sectores de almacenamiento de Hashtable.

Cada objeto de clave de Hashtable debe proporcionar su propia función hash, a la que se tiene acceso llamando al método GetHash. Sin embargo, se puede pasar cualquier objeto que implemente IHashCodeProvider a un constructor Hashtable y utilizar esa función hash para todos los objetos de la tabla.

La instrucción foreach del lenguaje C# necesita el tipo de cada elemento de la colección. Puesto que cada elemento de Hashtable es un par de clave y valor, el tipo del elemento no se

corresponde con el tipo de la clave o del valor. En su lugar, el tipo del elemento es DictionaryEntry.

Las pilas son otras de las estructuras de datos con gran importancia, en esta biblioteca, la clase Stack es la que contiene todos los métodos de la misma. Como cualquier pila común la clase Stack representa una colección de objetos simple de la clase último en entrar, primero en salir.

Las colas también están incorporadas en este espacio de nombre System.Collections a través de la clase Queue que representa una colección de objetos de tipo primero en entrar, primero en salir. Esta última es de gran utilidad para almacenar mensajes en el orden en el que fueron recibidos para el procesamiento secuencial. Esta clase implementa una cola como una matriz circular. Los objetos almacenados en Queue se insertan en un extremo y se quitan del otro. Si el número de elementos agregados a Queue alcanza la capacidad actual, ésta se ve aumentada automáticamente para albergar más elementos. La capacidad puede disminuir llamando a TrimToSize.

1.12. Eficiencia y complejidad de algoritmos

Aunque las computadoras de hoy en día son muy rápidas comparadas con sus similares de hace algunos años, y son capaces de llevar a cabo millones de operaciones en un segundo, resulta fácil encontrar ejemplos de problemas y soluciones que tardarían años en solucionarse (15).

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva, adecuado al dispositivo. A tal método lo denominamos algoritmo.

Una vez que dispongamos de un algoritmo que funciona correctamente, es necesario definir criterios para medir su rendimiento o comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: el espacio, es decir, memoria que utiliza, y el tiempo, lo que tarda en ejecutarse. Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado entre varios que solucionan un mismo problema.

La complejidad del algoritmo, no tiene que ver con dificultad, sino con rendimiento. Es una función independiente de la implementación; hay que identificar una operación fundamental que realice nuestro algoritmo. Algunos ejemplos de complejidades comunes son:

- O(1) orden constante
- O(log n) orden logarítmico
- O(n) orden lineal O(n log n)
- O(n2) orden cuadrático
- O(na) orden polinomial (a > 2)
- O(an) orden exponencial (a > 2)
- O(n!) orden factorial.

Se puede decir que un algoritmo de complejidad O(n) es más rápido que uno de complejidad O(n2). Otro aspecto a considerar es la diferencia entre el peor y el mejor caso. Cada algoritmo se comporta de modo diferente de acuerdo a cómo se le entregue la información; por eso es conveniente estudiar su comportamiento en casos extremos.

El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los datos de entrada que le suministremos, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo. Hay dos estudios posibles sobre el tiempo (16):

1. Uno que proporciona una medida teórica (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
2. Y otro que ofrece una medida real (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son funciones temporales de los datos de entrada.

Entendemos por tamaño de la entrada el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar.

A menudo se piensa que un algoritmo sencillo no es muy eficiente. Sin embargo, la sencillez es una característica muy interesante a la hora de diseñar un algoritmo, pues facilita su verificación, el estudio de su eficiencia y su mantenimiento. De ahí que muchas veces prime la simplicidad y legibilidad del código frente a alternativas más crípticas y eficientes del algoritmo. Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: el espacio, es decir, memoria que utiliza, y el tiempo, lo que tarda en ejecutarse. Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado de entre varios que solucionan un mismo problema.

1.13. Paradigmas de Programación

En ocasiones se encuentran, a la hora de programar, con problemas que no se pueden resolver de una manera adecuada con las técnicas habituales usadas en la programación imperativa o en la programación orientada a objetos. Con éstas, los desarrolladores se ven forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que los alejan con frecuencia de otras posibilidades.

Por otro lado, la implementación de dichas decisiones a menudo implica escribir líneas de código que están distribuidas por toda, o gran parte, de la aplicación para definir la lógica de cierta propiedad o comportamiento del sistema, con las consecuentes dificultades de mantenimiento y desarrollo que ello implica. En ingles este problema se conoce como código disperso. Otro problema que puede aparecer es que un mismo módulo se implemente de modo que maneje múltiples comportamientos o aspectos del sistema simultáneamente. El hecho es que hay ciertas decisiones de diseño que son difíciles de capturar con las técnicas antes citadas, debiéndose al hecho de que ciertos problemas no se dejan encapsular de igual forma que los que habitualmente se han resuelto con funciones u objetos. La resolución de éstos supone o bien la

utilización de repetidas líneas de código por diferentes componentes del sistema, o bien la superposición dentro de un componente de funcionalidades dispares.
Los tipos o técnicas de programación son bastante variados. En la mayoría de los casos, las técnicas se centran en programación implícita y la programación descriptiva.

1.13.1 Programación descriptiva

La programación descriptiva es una teoría de programación que consiste en construir programas de fácil comprensión, uno de estos tipos de programación es la programación estructurada.
La programación estructurada se basa en una metodología de desarrollo de programas llamada refinamiento sucesivos. Se plantea una operación como un todo y se divide en segmentos más sencillos o de menor complejidad. Una vez terminado todos los segmentos del programa, se procede a unificar las aplicaciones realizadas los programadores. Si se ha utilizado adecuadamente la programación estructurada, esta integración debe ser sencilla y no presentar problemas al integrar la misma, y de presentar algún problema, será rápidamente detectable para su corrección.
La representación gráfica de la programación estructurada se realiza a través de diagramas de flujo o flow chart, el cual representa el programa con sus entradas, procesos y salidas.
Un programa esta estructurado si posee un único punto de entrada y sólo uno de salida, existen de "1 a n" caminos desde el principio hasta el fin del programa y por último, que todas las instrucciones son ejecutables sin que aparezcan bucles infinitos.
La programación estructurada propone segregar los procesos en estructuras lo más simple posibles, las cuales se conocen como secuencia, selección e interacción. Ellas están disponibles en todos los lenguajes modernos de programación imperativa en forma de sentencias. Combinando esquemas sencillos se pueden llegar a construir sistemas amplios y complejos pero de fácil entendimiento.

1.13.2 Programación imperativa

La programación imperativa, en contraposición a la programación declarativa es un paradigma de programación que describe la programación en términos del estado del programa y sentencias

que cambian dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea (17).

La implementación de hardware de la mayoría de computadores es imperativa; prácticamente todo el hardware de los computadores está diseñado para ejecutar código de máquina, que es nativo al computador, escrito en una forma imperativa. Esto se debe a que el hardware de los computadores implementa el paradigma de las Máquinas de Turing. Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias son instrucciones en el lenguaje de máquina nativo del computador (por ejemplo el lenguaje ensamblador).

Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma. Las recetas y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; cada paso es una instrucción, y el mundo físico guarda el estado (Zoom).

Los primeros lenguajes imperativos fueron los lenguajes de máquina de los computadores originales. En estos lenguajes, las instrucciones fueron muy simples, lo cual hizo la implementación de hardware fácil, pero obstruyendo la creación de programas complejos. Fortran, cuyo desarrollo fue iniciado en 1954 por John Backus en IBM, fue el primer gran lenguaje de programación en superar los obstáculos presentados por el código de máquina en la creación de programas complejos. Algunos lenguajes imperativos actuales son BASIC, C, C++, Java, C#, Perl, en los cuales se pueden ver la programación orientada a objetos (POO) que es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

1.13.3 Programación Orientada a Objetos

La Programación Orientada a Objetos (POO u OOP según siglas en inglés) es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar (18).

Las principales diferencias entre la programación estructurada y la orientada a objetos son:

- La programación orientada a objetos es más moderna, es una evolución de la programación estructurada que plasma en el diseño de una familia de lenguajes conceptos que existían previamente con algunos nuevos.
- La programación orientada a objetos se basa en lenguajes que soportan sintáctica y semánticamente la unión entre los tipos abstractos de datos y sus operaciones (a esta unión se la suele llamar clase).
- La programación orientada a objetos incorpora en su entorno de ejecución mecanismos tales como el polimorfismo y el envío de mensajes entre objetos.

1.13.4 La Programación Orientada a Objetos (POO) como solución

La programación orientada a objetos es una nueva forma de programar que trata de encontrar solución a los problemas de la programación estructurada. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos se destacan los siguientes (19):

- **Objeto:** entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos). Corresponden a los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa).
- **Clase:** definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

- **Método:** algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un mensaje. Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un evento con un nuevo mensaje para otro objeto del sistema.
- **Evento**: un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.
- **Mensaje**: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.
- **Propiedad o atributo**: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto, y cuyo valor puede ser alterado por la ejecución de algún método.
- **Estado interno**: es una propiedad invisible de los objetos, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).
- **Componentes de un objeto**: atributos, identidad, relaciones y métodos.
- **Representación de un objeto**: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

En comparación con un lenguaje imperativo, una variable, no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la función es un procedimiento interno del método del objeto.

En nuestra solución se a utilizado la programación imperativa y dentro de esta específicamente la programación orientada a objetos, ya que nos brinda la facilidad de verlo todo como un objeto, ya sea una clase, una interfaz, o un componente. Con el uso de las Clases, Eventos, Métodos y Mensajes entre objetos, se realiza una aplicación con una mayor claridad y un desarrollo poco complejo y muy rápido.

1.13.5 Programación orientada a agentes

El paradigma de programación orientada a agentes es otro emergido en los últimos años como una nueva alternativa para el diseño de sistemas complejos; los objetivos y requerimientos del problema a resolver son realizados por un conjunto de entidades autónomas, capaces incluso de exhibir comportamiento inteligente, permitiendo en forma natural el desacople y cohesión deseables en un sistema, facilitando la construcción y mantenimiento del mismo (20).

El paradigma de agentes permite una abstracción muy alta de los componentes que modelan la solución de un problema, permite la inclusión de conceptos abstractos como pro actividad, cooperación y actos del lenguaje, que no son innatos de la programación orientada a objetos. Las investigaciones en Programación Orientada a Agentes buscan desarrollar metodologías que permitan desarrollar en forma ordenada y esquemática una solución de agentes de buena calidad.

1.13.6 Programación Orientada a Aspectos

La Programación Orientada a Aspectos (POA) 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 POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminando las dependencias inherentes 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 POA 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 (21).

Ventajas de la POA

La programación orientada a aspectos, permite, de una manera comprensible y clara, definir nuestras aplicaciones considerando estos problemas. Por aspectos se entiende dichos

problemas que afectan a la aplicación de manera horizontal y que este paradigma persigue el poder tenerlos de manera aislada de forma adecuada y comprensible, dándonos la posibilidad de construir el sistema componiéndolos junto con el resto de los componentes.

Entre los objetivos que se ha propuesto la POA están, principalmente, (1) el de separar conceptos y (2) el de minimizar las dependencias entre ellos. Con el primer objetivo se persigue que cada decisión se tome en un lugar concreto y no diseminado por la aplicación. Con el segundo, se pretende desacoplar los distintos elementos que intervienen en un programa.

Su consecución implicaría las siguientes ventajas aunque es todavía una metodología en estado de maduración, cada vez atrae a más investigadores e incluso proyectos comerciales en todo el mundo:

- Un código menos enmarañado, más natural y más reducido.
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y las dependencias entre ellos son mínimas.
- Un código más fácil de depurar y más fácil de mantener.
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.

1.14 Conclusiones

En este capítulo se hizo un análisis del estudio del arte de las principales tecnologías y herramientas utilizadas de los procesos que se llevan a cabo en el mismo, así como de los paradigmas de programación. Se definieron las tecnologías y herramientas a utilizar en el desarrollo de aplicaciones de escritorio. A partir de este análisis llegamos a las siguientes conclusiones.

- Utilizar la plataforma Microsoft .NET para el desarrollo de la solución propuesta, especialmente Microsoft Visual Studio .NET 2003 y como lenguaje de programación Microsoft Visual C# .NET, teniendo en cuenta que estos ofrecen una gran productividad componente ineludible por el poco tiempo que se tiene para la entrega de la aplicación.

- Usar el Proceso Unificado de Desarrollo de Software como metodología de desarrollo de software.
- Utilizar el Crystal Reports como herramienta para el diseño y visualización de reportes en Microsoft .NET, ya que se integra a este permitiendo su utilización desde C# y permitirle visualizar los informes sin necesidad de escribir código personalizado para exportar.
- Utilizar NUnit como herramienta para realizar pruebas unitarias en .NET, ya que utiliza atributos personalizados para interpretar las pruebas y provee además métodos para implementarlas, una forma de probar el correcto funcionamiento de un módulo de código.
- Utilizar la Programación Orientada a Objetos, ya que posibilita reutilizar muchas funcionalidades entre objetos y la utilización de muchos patrones que posibilitan la rapidez y mejoramiento funcional del sistema. La posibilidad que nos presta la herencia entre clases y el polimorfismo reduce en gran cantidad el código y el trabajo en la aplicación.

CAPÍTULO 2: DESCRIPCIÓN DE LA SOLUCIÓN PROPUESTA

2.1. Introducción

En el presente capitulo se comienza el diseño de la solución propuesta, especificándose cada una de las clases necesarias, estructuras de datos empleadas y algoritmos no triviales. Se aborda el tema de tratamiento de errores y excepciones, así como el uso de patrones de diseño. Además se plantea las pautas de diseño a seguir para las interfaces graficas, teniéndose en cuenta el tipo de aplicación.

2.2. Principios de diseño

2.2.1. Interfaz de usuario

Para el diseño de las interfaces de usuario fueron aplicadas las siguientes pautas:

Datos generales:

Dimensiones del Área de Trabajo (área gris): (827px x 674px)

- Contiene el icono del identificador en su versión reducida (13px x 14px) separado a 7px del Nombre del producto, 10px del borde gris de la ventana y centrado en la barra.
- Tipografía del nombre del producto en la barra: Verdana negrita, 8 puntos, color blanco.

Menú lateral:

Temáticas:

- Dimensiones de los botones de las temáticas principales: (23px x 166px); color RGB: 12, 28, 140; Tipografía: Verdana negrita, 8 puntos, justificada a la derecha.
- Filete de separación entre temáticas principales: 2px de ancho, color negro.

Subtemáticas:

- Dimensiones de los botones de las subtemáticas: (21px x 166px); color RGB: 212, 208, 200; Tipografía: Verdana regular, 8 puntos, justificada a la derecha.
- Filetes de separación entre subtemáticas: 1px de ancho, color negro.

Dentro del área de trabajo:

Datos generales:

- Tipografía para los títulos primarios y secundarios:
- Verdana 8 puntos en su variante negrita para la descripción o títulos primarios según corresponda (garantizando una adecuada comprensión del contenido y una correspondiente diferenciación visual según el caso) y Verdana regular para los datos específicos o títulos secundarios.

Márgenes externos en el área del trabajo:

- 10px los derechos e izquierdo (desde el borde azul hasta el filete enmarcador).
- 37px hasta el margen inferior, hasta el borde inferior de los botones Aceptar y Cancelar.
- Color de fondo RGB: 212, 208, 200.
- Los contenidos del mismo tipo, se enmarcan en un rectángulo con un filete de dos píxeles de ancho (groupbox = System).

Botones en general:

Los botones serán de 23px de Alto y 10px de aire entre los extremos del texto dentro del botón y los lados del mismo. La tipografía de los botones será Verdana 8 puntos.

Botones de navegación general:

Los botones de navegación general se quedan siempre en la misma ubicación.

Botones de contenido:

Justificados a la derecha de la acción correspondiente y siguiendo el mismo estilo de los botones de navegación.

Botones Aceptar y Cancelar:

- Justificados en la parte inferior derecha del Área de Trabajo, con una separación del borde de 10px por la derecha y 37px por la parte de abajo. Separados entre ellos a 7px.
- En los casos de aumentar el número de botones generales a este nivel se ubicarán a la Izquierda del botón Cancelar y separados a 7px igualmente.

Cuadros de textos:

- Los cuadros de texto grandes irán a todo lo largo del Área de Trabajo, respetando los márgenes de Izquierda y Derecha, 7px. Y su ancho será de 23px al igual que los botones.

- Los cuadros de texto pequeños, un tamaño definido por el programador uniforme para todos.
- Separación de las cajas de texto del recuadro de contenido es de 7px.
- Separación entre las cajas de texto horizontalmente será de 7px (en caso de necesario por el espacio de 5px).
- Separación entre estas cajas de texto por la vertical con título o subtítulo será de 27px.
- Separación entre estas cajas de texto en caso de no tener título encima, será de 7px.
- En el caso de las titulares de los cuadros de textos se utilizará la misma tipografía Verdana, versión negrita para los primarios y normal para los secundarios, justificados a la derecha, color negro.

Área de identificación del contenido:

- Dimensiones: (28px x 827px), color de fondo RGB: 12, 128, 140.
- Tipografía: Verdana negrita, justificada a la derecha, 8 puntos, color blanco, separada a 7px del filete negro y justificada por debajo con la primera línea del menú lateral.

Márgenes en área de trabajo:

- Dentro del Área de Trabajo, los márgenes para los contenidos son de:
- 35px Arriba tomando como referencia la base del texto (Texto equivalente al comentario, Ejemplo: Número de control).
- 0px Izquierda y Derecha.
- 10px Abajo tomando como referencia la base del texto o la parte inferior de los botones o cajas de texto.

Contenido:

Los contenidos se enmarcaran en dentro de un recuadro de 2px de grosor, separado de los bordes del área de trabajo como se explicó anteriormente. Dentro de dicho recuadro el contenido dentro de este irá separado a:

- 18px desde arriba tomando como referencia la base del texto de los titulares de las cajas.
- 7px por debajo tomando como referencia la base del texto o el marco delimitador del contenido de la ventana interna.

- 7px por ambos lados, izquierda y derecha.

Cuando el contenido de las ventanas no alcance a llenar el espacio de la pantalla los marcos delimitadores de contenidos cubrirán el ancho de la página, se picarán únicamente cuando en el área indicada se muestre algún resultado inmediato y se mantendrán los botones de Aceptar, Cancelar y Ayuda en las posiciones pautadas.

Formato de salida de los reportes:

Los reportes en la aplicación se diseñan y generan utilizando Crystal Reports. Cada uno de estos reportes muestran como encabezado el logo que identifica el sistema y el país, el ministerio y el nombre de la oficina. En la parte superior izquierda se coloca el nombre del reporte y en la esquina superior derecha el número de la hoja. En el Anexo I y II puede verse un ejemplo de reporte del sistema.

2.3. Tratamientos de errores

Buscar errores y manejarlos adecuadamente es un principio fundamental para diseñar software correctamente. En teoría, escribimos el código y cada línea funciona como pretendemos y los recursos que empleamos siempre están presentes. Sin embargo, éste no siempre es el caso en el mundo real. Los programadores pueden cometer errores, las conexiones de red pueden interrumpirse, los servidores de bases de datos pueden dejar de funcionar y los archivos de disco pueden no tener los contenidos que las aplicaciones creen que contienen. En pocas palabras, el código que se escribe tiene que ser capaz de detectar errores como éstos y responder adecuadamente. Los mecanismos para informar de errores son tan diversos como los propios errores.

Algunos métodos pueden diseñarse para devolver un valor booleano que indican el éxito o el fracaso de un método. Otros métodos pueden escribir errores en un fichero de registro o una base de datos de algún tipo. La variedad de modelos de presentación de errores nos indica que el código que escribamos para controlar los errores debe ser bastante consistente. Cada método puede informar de un error de un modo distinto, lo que significa que la aplicación estará repleta de gran cantidad de código necesario para detectar los diferentes tipos de errores de las diferentes llamadas al método.

En el sistema las excepciones se trabajan de diferentes formas, se cuenta con un manejador de excepciones que se encarga de darle un tratamiento único a cada una de ellas dependiendo del tipo y el lugar donde se hallan originado.

Las excepciones pueden ser encapsuladas en otras excepciones o pueden ser redefinidas por nuevas excepciones con el objetivo que al usuario llegue un mensaje con una explicación coherente del error surgido. Estos mensajes se encuentran en un fichero XML de configuración donde se guarda el id de la excepción, el mensaje así como la forma en que va a ser tratada.

Existe un gestor de mensaje que controla la forma de mostrar los mensajes y errores de la aplicación (MostrarMensage) así como una interfaz amigable para el usuario la cual se configura dinámicamente dependiendo del tipo de mensaje a mostrar (Error, Confirmación, Mensaje).

En la forma principal de la aplicación (Principal) se controla el hilo que capturará cualquier excepción que se levante en el sistema, buscará en el XML de configuración el id de esta excepción y si es encontrada toma el mensaje correspondiente a esta, se llama al gestor de mensaje y le pedimos que muestre un mensaje de error pasándole el mensaje encontrado en el fichero XML. Este gestor mostrara el error ya tratado y copiará la información de la excepción en un archivo para que pueda ser controlada por el programador. En caso de que la excepción no se halla tratado aun y no se encuentre en el fichero de configuración se mostrará un mensaje al usuario informándole que ha ocurrido un error y que se ponga en contacto con los proveedores del programa para arreglarlo.

Para facilitar el uso del manejador se creó una aplicación que ayudó a configurar los ficheros XML de una forma más sencilla y con una interfaz más amigable. En esta aplicación se define la excepción a tratar y el mensaje que se mostrará así como el tipo de envoltura que se dará a la excepción y en donde se copiará el fichero de error.

De esta manera en el sistema se mantendrá de una forma fácil controlando las excepciones, así como el manejo de mensajes. El usuario verá los errores de una manera coherente y amigable, mientras que el programador podrá ver en los ficheros originados cada una de las excepciones surgidas con todo la información necesaria para evacuar el error.

2.4. Patrones y Arquitectura Base

Los patrones de diseño describen un problema que ocurre repetidas veces en algún contexto determinado de desarrollo de software, y entregan una buena solución ya probada. Esto ayuda a diseñar correctamente en menos tiempo, ayuda a construir problemas reutilizables y extensibles, y facilita la documentación. Los patrones enseñan a aplicar de manera eficaz que hacer con la herencia, el polimorfismo, y todas las ventajas que posee la POO.

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de la solución al problema, de forma que puede utilizarse un millón de veces sin tener que hacer dos veces lo mismo (22).

Por otro lado los patrones de arquitectura: Expresan un paradigma fundamental para estructurar u organizar un sistema software. Proporcionan un conjunto de subsistemas o módulos predefinidos, con reglas y guías para organizar las relaciones entre ellos.

En la propuesta de solución se utilizó como patrón de arquitectura de software, el Modelo Vista Controlador (MVC), ya que este patrón divide la estructura de la aplicación cliente en tres clases distintas.

- Modelo (o Estado): Gestiona el comportamiento y los datos de la aplicación, responde a las peticiones que realizan las vistas sobre su estado y permite su actualización normalmente desde el controlador. En este caso en la propuesta de solución, el modelo es el paquete de acciones.
- Controlador: Interpreta las acciones del usuario, accediendo a las operaciones de negocio de la aplicación y modificando a partir de sus resultados el estado del modelo y la navegación entre vistas. En este caso dentro de la solución propuesta, el controlador es el Gestor MetUtiles, el cual es el encargado de gestionar los tipos de conexiones que se utilizan, así como los métodos auxiliares de gestión de datos.
- Vista: Muestra el estado al usuario de la aplicación, redirigiendo las acciones que realiza sobre la interfaz al controlador. En nuestro caso la vista, está conformada por seis Frm (frmPrincipal.cs, frmOledb.cs, frmFichero.cs, frmEditor.cs, frmDll.cs, frmAdaptador.cs).

Se hizo una adaptación del patrón Modelo Vista Controlador, basándose en las acciones no en las vistas, utilizando el patrón comando para implementar las acciones.

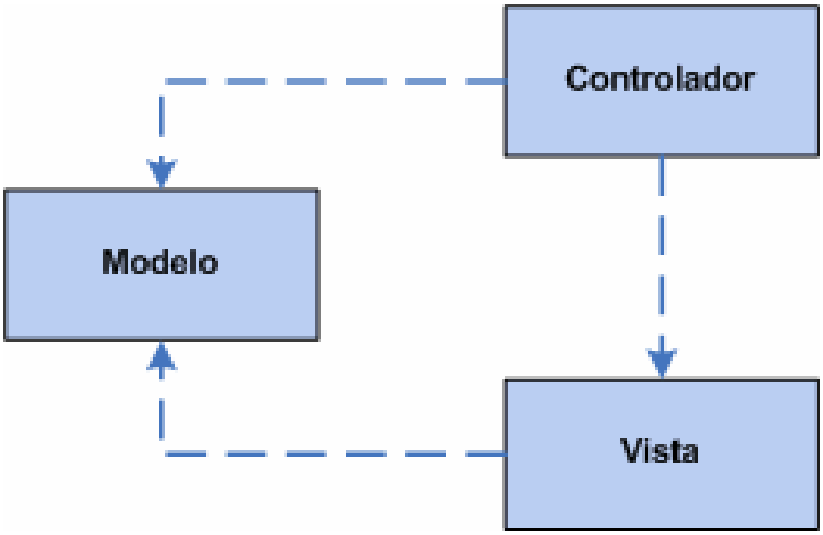

Figura 3 Esquema simplificado del patrón MVC

Los patrones de creación abstraen la forma en la que se crean los objetos, permitiendo tratar las clases a crear de forma genérica dejando para más tarde la decisión de qué clases crear o cómo crearlas (23).

Dentro de los patrones de creación utilizados en la solución, está en primer lugar el *Singleton* (instancia única), el cual está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. El propósito de la aplicación de este patrón está en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

Cada Frm en la solución es un *Singlenton*, ya que solo se construyen una vez, garantizando que exista en la aplicación un solo tipo de conexión y que se actualicen todos los formularios, una vez que ocurran cambios, sin necesidad de volver a reconstruir el mismo.

Los patrones estructurales, tratan de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos, y éstas están determinadas por las interfaces que soportan los objetos.

Estudian como se relacionan los objetos en tiempo de ejecución. Sirven para diseñar las interconexiones entre los objetos.

En este caso dentro de los patrones estructurales utilizados, se encuentra el *Adapter*, encargado de las relaciones entre clases, en este caso por composición. Como su propio nombre hace ver, se usa para convertir el interface de una clase a la de la otra, entendiendo en este caso interface como el conjunto de métodos que ofrece una clase para su manipulación por código, no la herramienta que usa el usuario final de nuestra aplicación. Lo que hacemos es escribir métodos con nuestra interface deseada que deleguen el trabajo en los de la interface ya existente, es decir para cada control que se utiliza en el editor de interfaces se utiliza este patrón, con el objetivo de convertir la interfaz de una clase en otra más compatible con nuestras necesidades. Utilizamos para hacer controles según nuestras necesidades las clases (CtrText.cs, CtrLVew.cs, CtrLabel.cs, CtrGrup.cs, CtrGrid.cs, CtrCombo.cs)

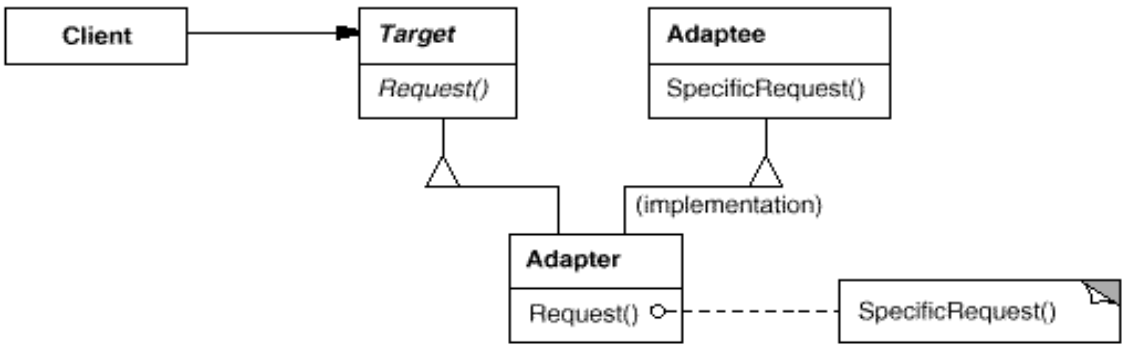

Figura 4 Adaptador de clase

El patrón *Strategy* es otro de los patrones utilizados, siendo muy similar al Adaptador, pero *Strategy* además pretende cambiar la funcionalidad del método llamado. Define una familia de algoritmos, encapsulando cada uno y haciéndolos intercambiables, *Statregy* permite que el algoritmo varíe en dependencia del cliente que lo usa. Ejemplo de su uso es en el Framework de notificaciones, en el que existen varias formas de mostrar las ventanas y mensajes.

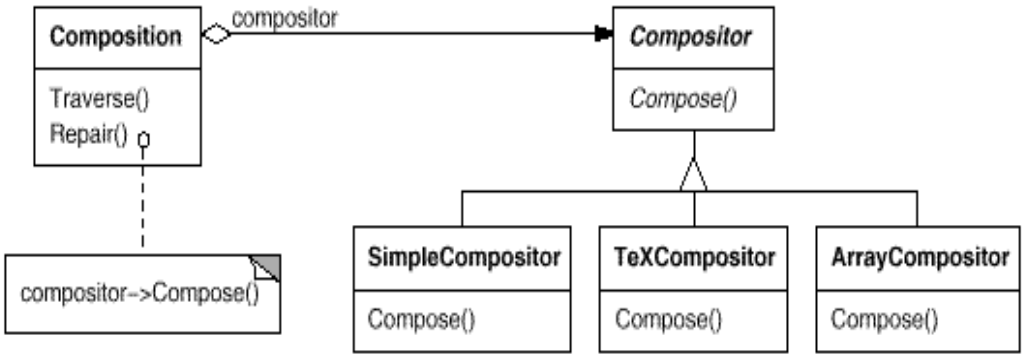

Figura 5 Estructura de clase (Strategy)

El Cliente delega una operación en la superclase, abstrayéndose de los detalles de la misma. La superclase AlgoritmoAbstracto ofrece una interfaz común y oculta la elección del algoritmo empleado. Las clases AlgoritmoConcreto implementan cada una de las diferentes alternativas del algoritmo. Por otro lado está el patrón *Facade*, el cual provee una interface única para cambios en interfaces y subsistemas. Fachada define interfaces de alto nivel que hace a los subsistemas fáciles de usar. Ejemplo del uso de este patrón es en la capa intermedia entre la capa de negocio y la capa de acceso a datos generada por el TierDeveloper.

2.5. Modelo basado en capas

El desarrollo de la solución responde a un modelo multicapas. Un ejemplo de la interacción de estas capas se muestra en la figura que se encuentra debajo.

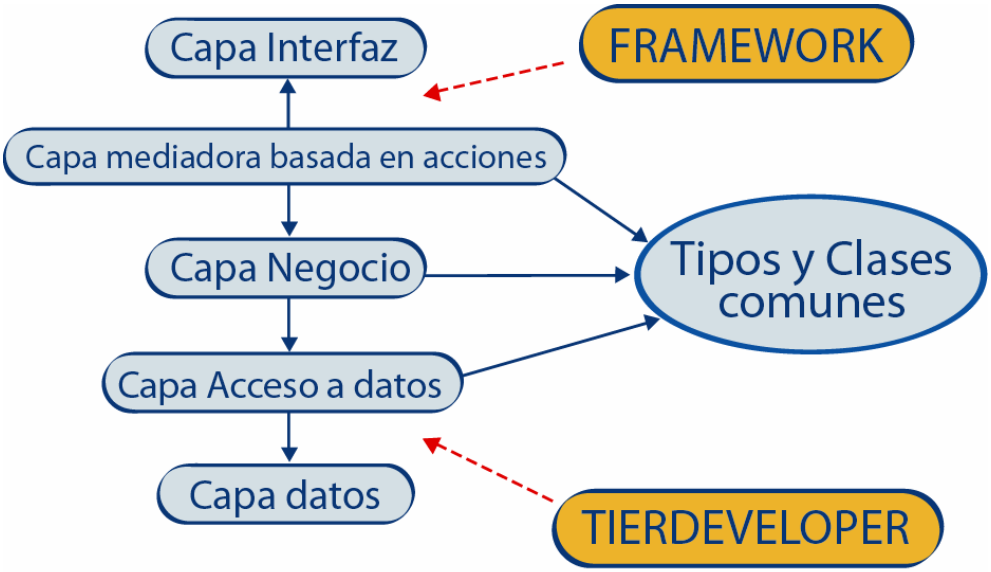

Figura 6 Modelo Multicapas

Los tipos y clases comunes son todos los tipos de datos que deben ser comunes a varias capas por tanto deben ser vistos por varias capas. El sentido de las flechas indica que la capa origen conoce la entidad que está en el destino de la misma.

La capa de interfaz responde a un patrón estándar. Su uso se basa en la explotación de la componente interfaz que realmente provee un framework facilitando el desarrollo del sistema. Este framework se basa en acciones y cada acción que se corresponda con funcionalidades de captura o visualización de datos tiene asociado un formulario cuya forma depende de la funcionalidad específica para la cual fue concebida dicha acción.

La capa mediadora interfaz basada en acciones. Su uso se basa en la explotación de la componente interfaz que realmente provee un framework del sistema basado en acciones. 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 una 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 Accion o AccionSegura.
- Opcionalmente tener una forma visual asociada.
- Propiedades en función del objetivo específico de la misma.
- Se le debe reescribir el método CrearForma con vistas a mostrar lo que se desee.

En el caso de la capa lógica de negocio, el diseño de esta capa depende directamente del negocio específico al que se refiera.

Por ultimo esta la capa de datos que es desarrollada por un equipo en común. Esta capa esta compuesta por dos subcapas. La primera se corresponde con una capa compuesta por clases que constituyen factorías de las entidades del negocio que deben persistir. La segunda subcapa debe ser generada con el paquete TierDeveloper versión 4.0 y debe garantizar todo el manejo de la información que requiera el negocio del problema en cuestión y la conexión con la base de datos.

2.6. Estructuras de Datos

En la plataforma de desarrollo Visual Studio 2003 debido a que es basada bajo el Framework 1.0, y el mismo no permite el desarrollo con genericidad, es un poco engorroso el trabajo con estructuras de datos, ya que la utilización directamente a las mismas, requieren en gran cantidad del uso de casting para acceder a las propiedades de los objetos almacenados en las mismas.

En el sistema se trata dar solución a este problema, principalmente a las listas de una forma tal que nos permita trabajar en ella sin necesidad de castiar ningún objeto.

Se desarrolló un módulo de Colecciones el cual está basado en una clase principal (CColeccion), la cual hereda directamente de (CollectionBase), clase abstracta contenida en el namespace System.Collections del Framework 1.0. y se implementaron todos los métodos abstractos utilizando los métodos comunes de la Intefaz IList que implementa la clase CollectionBase del Framework.

Una vez desarrollada esta clase CColeccion se pasó a implementar cada una de las colecciones del sistema, todos estas colecciones heredan directamente de la clase CColeccion, se desarrollaron por cada una de las colecciones un constructor de copia y se indexaron las colecciones con valores enteros que retornan el objeto en esa posición y en mucha de estas

colecciones como CcolNamesp y CcolCompBase se le crearon sobrecargas a estos indexados con valores string que se utilizan para recorrer las colecciones y comparar con alguna de las propiedades de los objetos contenido en ellos, luego de encontrar una coincidencia en la lista se retorna el objeto que contiene la propiedad con el mismo valor que el string del indexado.

También en colecciones como ColAtributos y MetodoColeccion como era necesario saber el estado de los objetos en las lista si eran adicionados o eliminados fue necesario implementar el patrón *Observe* a través de eventos y delegados, por lo que se crearon dos delegados DEliminar y DAdicionar los cuales como parámetros reciben un objeto del tipo que se quiere escuchar su comportamiento. Ya con los delegados, se crearon dos eventos (OnElinminar y OnAdicionar) del tipo de estos delegados y luego de sobrescribir las funciones OnInsert y OnRemove de la clase CollectionBase se colocaron a la escucha en cada uno de estos métodos los eventos creados anteriormente. De esta forma se sabe que objeto fue eliminado o adicionado a una colección.

Por último se desarrolló el método CopiarDesde con el objetivo de copiar una colección dentro de otra ya existente.

De esta manera se trató de darle una solución lo más cómoda posible con el objetivo de que el trabajo con esta estructura de datos no se convirtiera en algo incómodo y engorroso. Con la solución dada no es necesario castiar ningún objeto porque cada colección ya sabe el tipo de objeto que contiene, podemos a través de los indexadores obtener un objeto con facilidad por distintas vías de búsquedas y mantener un control estricto a la hora de que un objeto haya sido eliminado o adicionado mediante los eventos implementados.

Otra de las estructuras que tienen gran importancia en la aplicación son las conocidas *Hashtable* del mismo namespace que la CollectionBase, esta estructura se utilizó con el objetivo de conocer la información de un objeto.

Dependiendo de la llave que se le pase, son utilizadas principalmente en la capa de acciones del sistema específicamente en la clase accEditor para saber por control creado, que parámetro es el que le pertenece y en la clase accNewClass para tener conocimiento el método, propiedad, constructor o atributo seleccionado en el árbol jerárquico de la clase.

Con estas dos estructuras utilizadas, se ha facilitado en gran por ciento el desarrollo y rapidez de la aplicación y el sistema en particular, y así se ha garantizado la estabilidad de los datos en tiempo de ejecución y la persistencia de los mismos.

2.7. Análisis de los algoritmos no triviales

En el sistema se implementaron algoritmos no triviales que alteran de alguna forma la complejidad en el código de la aplicación. En la búsqueda de soluciones para algunas funcionalidades, se han utilizados algoritmos muy conocidos como Burbuja, Inserción y Selección, principalmente a la hora de efectuar búsquedas y ordenamientos en las listas de objetos.

En el Gestor MetUtiles existe la función estática OrderTobIndex la cual implementa un algoritmo que se encarga de organizar una lista de la de objetos CompBase por el valor TabIndex de la propiedad Comp.

```
PROCEDURE OrderTabIndex (VAR a:CColCompBase;prim,ult:CARDINAL)
VAR i,j:CARDINAL;
BEGIN
FOR i:=prim TO ult-1 DO
FOR j:=ult TO i+1 BY -1 DO
IF (a[j-1].Comp.TabIndex >a[j].Comp.TabIndex) THEN
a[j-1]=a[j]; END
END
END
END;
```

Calculando su complejidad:

- En el mejor caso:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+\sum_{j=i+1}^{n}(3+4)+3\right)\right)+3=\frac{7}{2}n^2+\frac{5}{2}n-3.$$

- En el peor caso:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+\sum_{j=i+1}^{n}(3+4+2+7)+3\right)\right)+3=8n^2-2n-1.$$

- En el mediano caso:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+\sum_{j=i+1}^{n}\left(3+4+\frac{2+7}{2}\right)+3\right)\right)+3=\frac{23}{4}n^2+\frac{1}{4}n-1.$$

Como se observa en las ecuaciones anteriores, en el cálculo este algoritmo que se utilizó, contiene complejidad cuadrática O(n2).

Para el sistema se estudiaron algoritmos como el método de Inserción y Selección para el ordenamiento de listas, los cuales también como el método anterior tienen una complejidad cuadrática. Después de analizar estos algoritmos se llegó a la conclusión de que el método de Selección por el número de operaciones de comparación e intercambio que realiza, es el más adecuado para ordenar pocos registros de gran tamaño. En este sentido, analizando el número de intercambios que realiza el método de Selección vemos que es de orden O(*n*), frente al orden O(*n*2) de intercambios que presentan los métodos de Inserción o Burbuja.

Método de Selección

```
PROCEDURE Seleccion(VAR a:vector;prim,ult:CARDINAL);
VAR i:CARDINAL;
BEGIN
FOR i:=prim TO ult-1 DO
Intercambia(a,i,PosMinimo(a,i,ult))
END
END Seleccion;
```

Método de Inserción

```
PROCEDURE Insercion (VAR a:vector;prim,ult:CARDINAL);
VAR i,j:CARDINAL; x:INTEGER;
BEGIN
FOR i:=prim+1 TO ult DO
x:=a[i]; j:=i-1;
WHILE (j>=prim) AND (x<a[j]) DO
a[j+1]:=a[j]; DEC(j)
END;
a[j+1]:=x
END
END Insercion;
```

Otro algoritmo analizado fue el ordenación por mezcla (Megersort), este utiliza la técnica de Divide y Vencerás para realizar la ordenación del vector. Su estrategia consiste en dividir el vector en dos subvectores, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos subvectores ya ordenados. Esta idea da lugar a la siguiente implementación:

```
PROCEDURE Mezcla(VAR a,b:vector;prim,ult:CARDINAL);
(* utiliza el vector b como auxiliar para realizar la mezcla *)
VAR mitad:CARDINAL;
BEGIN
IF prim<ult THEN
mitad:=(prim+ult)DIV 2;
Mezcla(a,b,prim,mitad);
Mezcla(a,b,mitad+1,ult);
Combinar(a,b,prim,mitad,mitad+1,ult)
END
END Mezcla;
```

En este algoritmo se encuentra una particularidad con respecto a los anteriores en su complejidad y uso de memoria, pues es mucho mayor, Mergesort contiene una complejidad de O(*n*log*n*).

Después de estudiar este algoritmo se pudo llegar a la conclusión de que este algoritmo se puede utilizar no solo en ordenamientos de vectores, sino que se puede implementar de forma tal que el acceso a los datos se realicen de foma secuencial por lo que pudo ser utilizado en nuestras listas principalmente.

En nuestro sistema se utilizaron otros algoritmos de menos complejidad como Cubetas y Residuos que son menos engorrosos a la hora de programarlos y contienen una complejidad O(n).También se desarrollo el conocido método de la Burbuja de complejidad O(n2) y se les realizaron algunas pruebas de tiempo y consumos de memoria.

2.8. Descripción de las Clases

A continuación se detalla la descripción de cada una de las principales clases utilizadas en la solución propuesta.

Nombre: ControlBase	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aContrBase -aTipo	-System.Windows.Forms.Control -string
Responsabilidad:	
Descripción:	Clase abstracta base de donde heredan todos los controles.

Nombre: gExportRpt	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Clase que gestiona la forma de exportar los Rpt.

Nombre: gConectar	
Tipo de clase (Gestor)	
Atributo	**Tipo**

-	-
Responsabilidad:	
Descripción:	Clase que se encarga de gestionar los datos extraídos de la base de datos.

Nombre: gFicheroConecion	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Clase que se encarga de gestionar la conexión a ficheros XML.

Nombre: gRptConeccion	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-aRuta -aStrConeccion	-string -string
Responsabilidad:	
Descripción:	Clase que se encarga de gestionar la conexión a Base de Datos.

Nombre: gRptDataInfo	
Tipo de clase (Gestor)	
Atributo	**Tipo**
Responsabilidad:	
Descripción:	Clase que se encarga de gestionar la información de las tablas y procedimientos almacenados de la Base de Datos.

Nombre: IControlVew	
Tipo de clase (Interfaz)	
Atributo	**Tipo**
Responsabilidad:	
Descripción:	Interfaz de donde heredan los controles visores de información.
Nombre: CGEN	
Tipo de clase (Entidad)	
Atributo	**Tipo**

-aatributo -acomentario -anombre -atipo	-System.CodeDom.MemberAttributes -string -string -string
Responsabilidad:	
Descripción:	Clase de donde heredan todas las entidades del paquete generadora.

Nombre: GGenerador	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-accu -aclas -aclase -anamefile -ans	-System.CodeDom.CodeCompileUnit - Clase -System.CodeDom.CodeTypeDeclaration -string -System.CodeDom.CodeNamespace
Responsabilidad:	
Descripción:	Clase que se encarga de generar el código diseñado de la aplicación.

Nombre: CColeccion	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Clase de donde heredan todas la colecciones de las entidades del paquete generadora.

Nombre: gCompilador	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Se encarga de compilar la clase generada por la aplicación y retornar la lista de errores.

Nombre: gGestorProj	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-aCaminoProj -aDoc	-string -System.Xml.XmlDocument
Responsabilidad:	
Descripción:	Se encarga de gestionar y actualizar información del proyecto en que se trabaja.

Nombre: frmMensajes	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aestilo -agimg -txtmensage	- Estilo -System.Windows.Forms.PictureBox -System.Windows.Forms.Label
Responsabilidad:	
Descripción:	Adapta el formulario al tipo de mensaje que se quiera mostrar.

Nombre: gMostrarMensage	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Se encarga de gestionar el tipo de mensaje que se quiera mostrar, ya sea de alerta o error.

Nombre: gMetUtiles	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-asCout	-int
Responsabilidad:	
Descripción:	Gestor que contiene métodos estáticos útiles en la aplicación.

Nombre: gGenerarFormulario	
Tipo de clase (Gestor)	
Atributo	**Tipo**
-aAns -aCcu -aClase -aNamefile	-System.CodeDom.CodeNamespace -System.CodeDom.CodeCompileUnit -System.CodeDom.CodeTypeDeclaration -string
Responsabilidad:	
Descripción:	Gestor que genera formularios en la aplicación.

Nombre: CtrLVew	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aCampoImagen -aCampos -aDataSource -aEraUnArray -aModo -aVew	-string -string[] -object -bool - tipomodo -System.Windows.Forms.ListView
Responsabilidad:	
Descripción:	Entidad del control ListView.

Nombre: CtrCombo	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aCadena -aComb -aDs -aTabla -aTable	-string -System.Windows.Forms.ComboBox -System.Data.DataSet -string -System.Data.DataTable
Responsabilidad:	
Descripción:	Entidad del control nomenclador.

Nombre: CompAcc

Tipo de clase (Entidad)	
Atributo	**Tipo**
-aFrm -aRpt -aTexto -aVisor	- CompFrm -CompRpt -string -CompVisor
Responsabilidad:	
Descripción:	Componente que gestiona una acción.

Nombre: CompBase	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aCamino -aLineas -aLista -aNombre -aPop -aPosComp -aType	-string -CColLinias -System.Collections.ArrayList -string -string -ePosicion -eType
Responsabilidad:	
Descripción:	Clase base de todos los componentes donde se gestiona la conexión entre ellos.

Nombre: CompClase	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aClas	-Clase
Responsabilidad:	
Descripción:	Componente que gestiona una clase.
Nombre: CompFrm	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aAplc_Formulario -aContolColl	-bool -CColControlBase
Responsabilidad:	
Descripción:	Componente que gestiona una Frm.

Nombre: CompRpt	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-aEsquematizar -aPlantilla -aTitulo	-bool -string - string
Responsabilidad:	
Descripción:	Componente que gestiona un Rpt.

Nombre: CompVisor	
Tipo de clase (Entidad)	
Atributo	**Tipo**
- aEsquema	-bool
Responsabilidad:	
Descripción:	Componente que gestiona el visor.

Nombre: Conector	
Tipo de clase (Entidad)	
Atributo	**Tipo**
- gCompconectado - gNombre	- CompBase - string
Responsabilidad:	
Descripción:	Componente que gestiona la relación entre los componentes.
Nombre: accContComponent	
Tipo de clase (Entidad)	
Atributo	**Tipo**
- aComp - aDocrpt - aLineas - asgCamino - asgCodn01 - asgCompaConect - asgCompBases	- int - CrystalDecisions.CrystalReports.Engine.ReportDocument - CColLinias -string - System.Drawing.Point - CompBase

- asgComponentes - asgCSharpTracker - asgLinea - asgMov - asgProj - aToll	- System.Collections.ArrayList - System.Collections.Hashtable - RectTracker - Linea - bool - GestorProj - int
Responsabilidad:	
Descripción:	Editor de relación entre cada uno de los componentes.

Nombre: CColCompBase	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Gestiona colección de componentes.

Nombre: CColContructores	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Gestiona colección de constructores.

Nombre: CColLineas	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Gestiona colección de líneas.

Nombre: CColName.
Tipo de clase (Entidad)

Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Gestiona colección de namespace.

Nombre: CColPropiedades	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Gestiona colección de propiedades.

Nombre: ColAtributos	
Tipo de clase (Entidad)	
Atributo	**Tipo**
-	-
Responsabilidad:	
Descripción:	Gestiona colección de atributos.

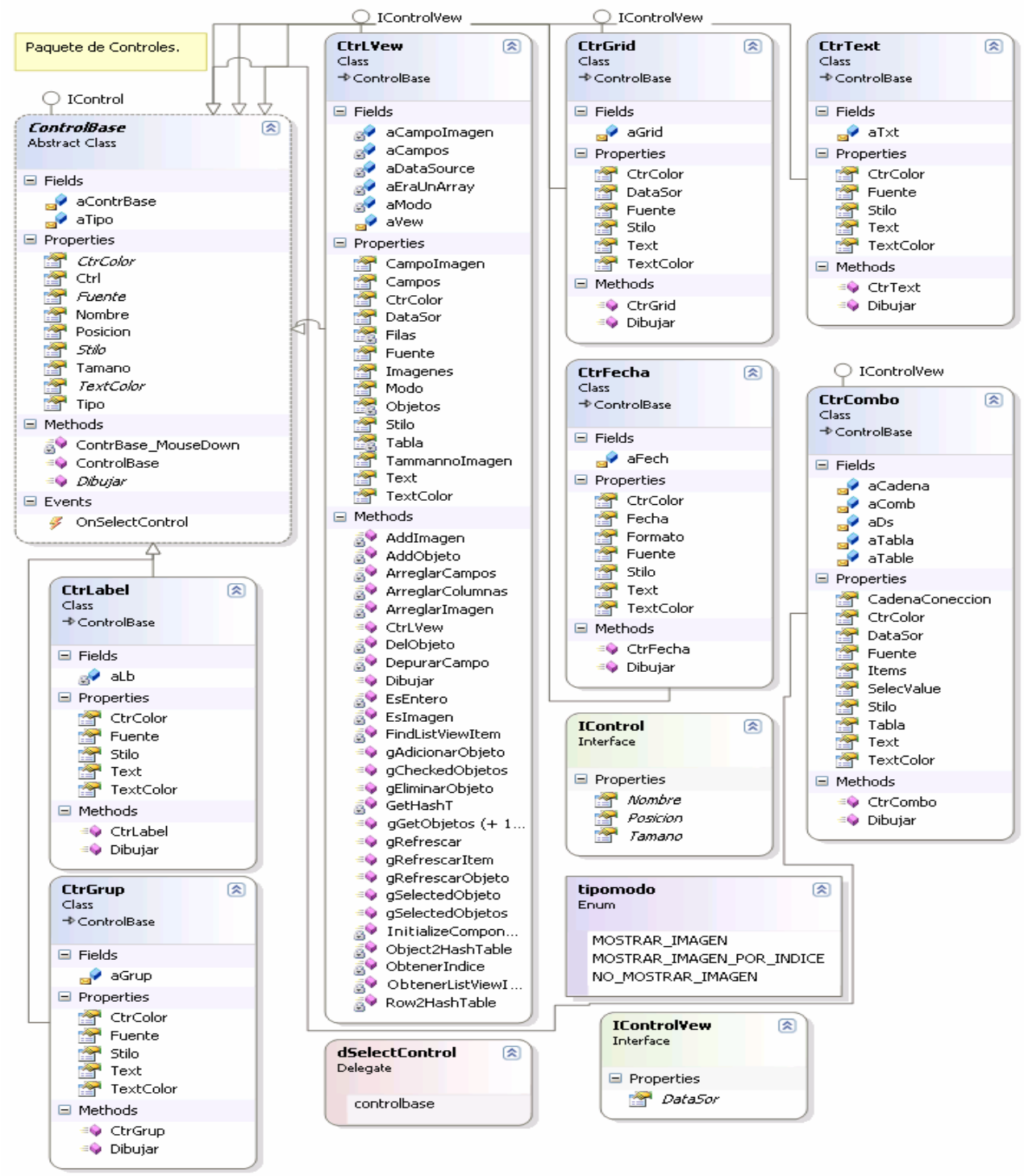

Figura 7 Paquete de controles

Figura 8 Paquetes de entidades generadoras

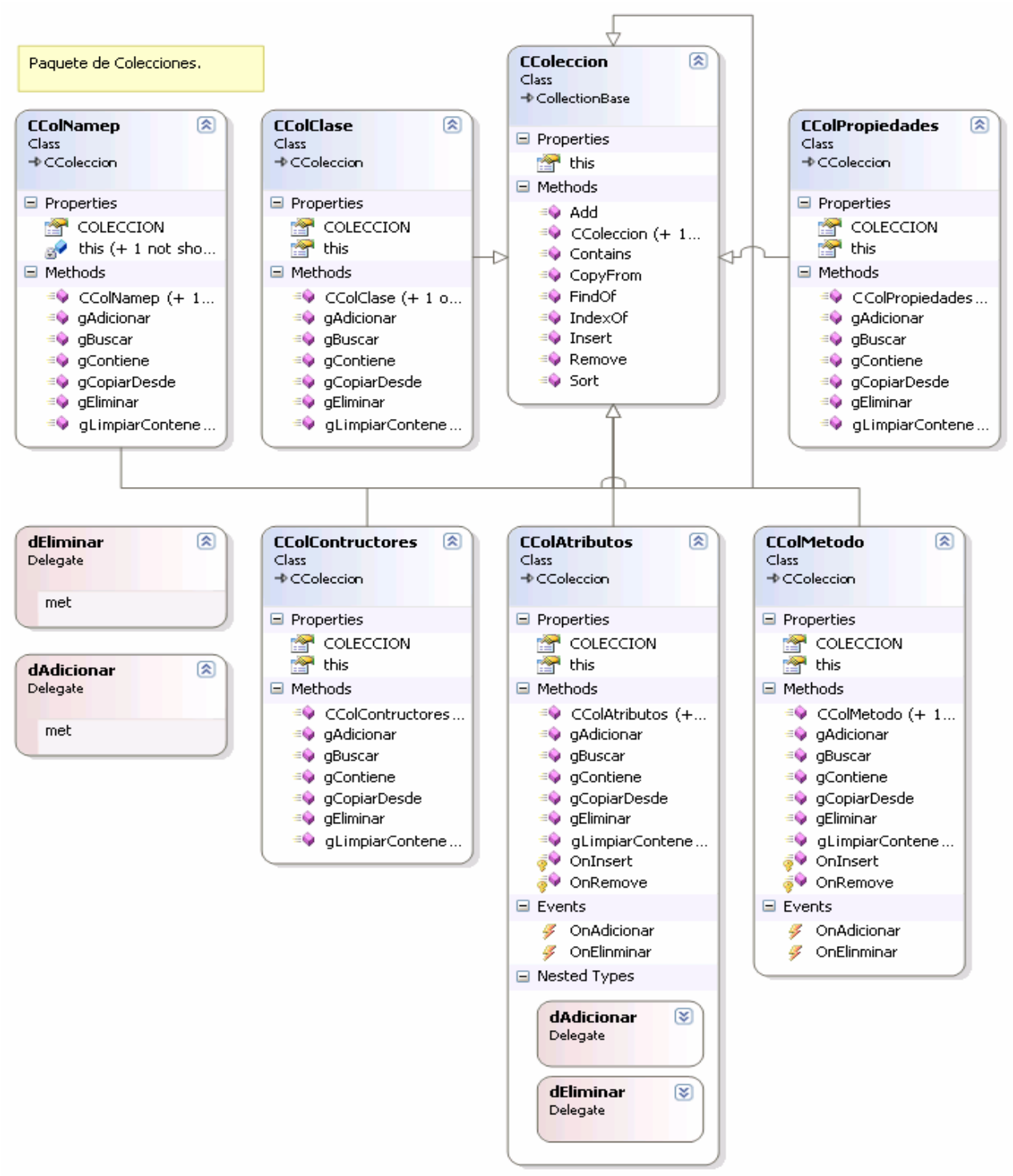

Figura 9 Paquete de colecciones

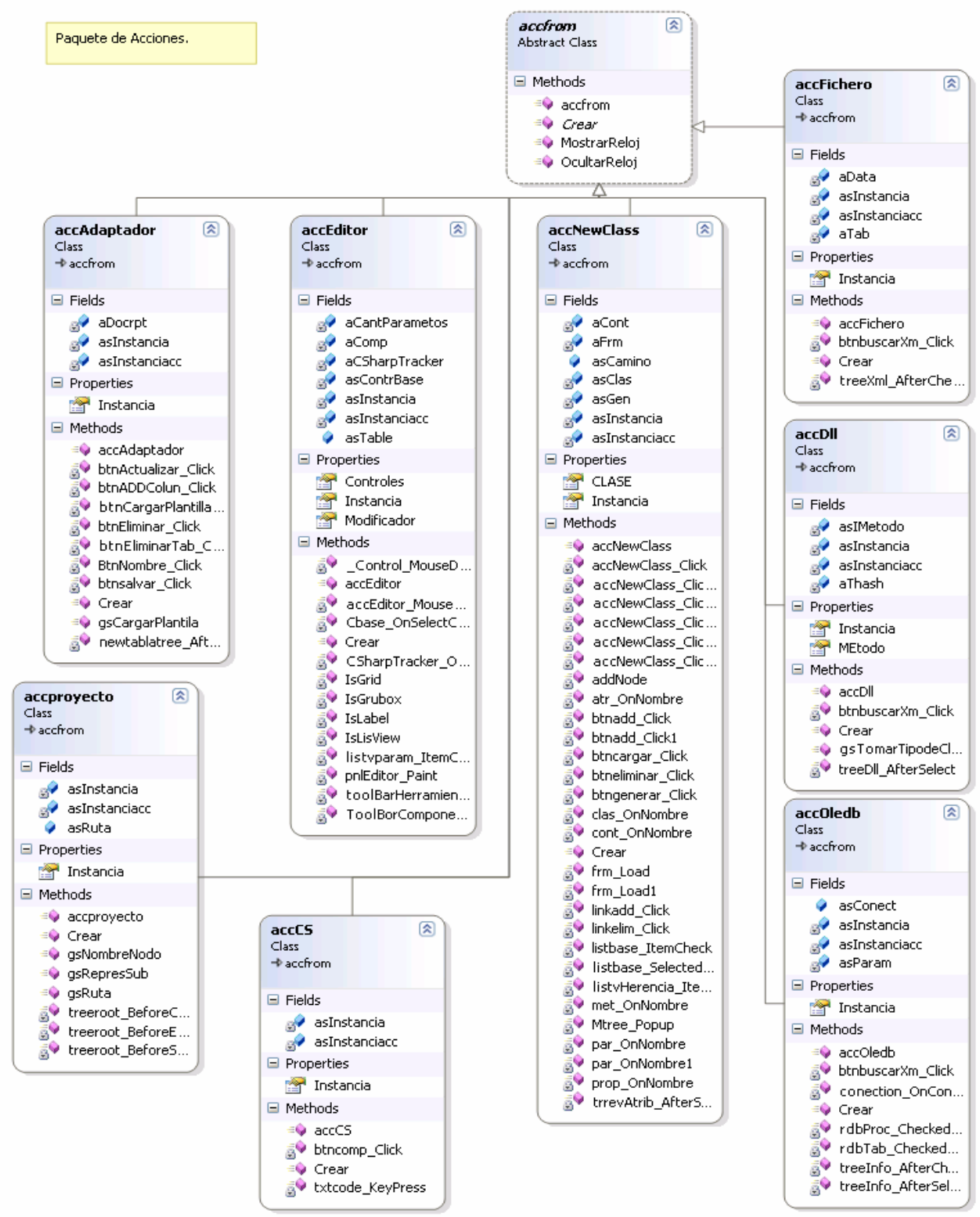

Figura 10 Paquete de acciones

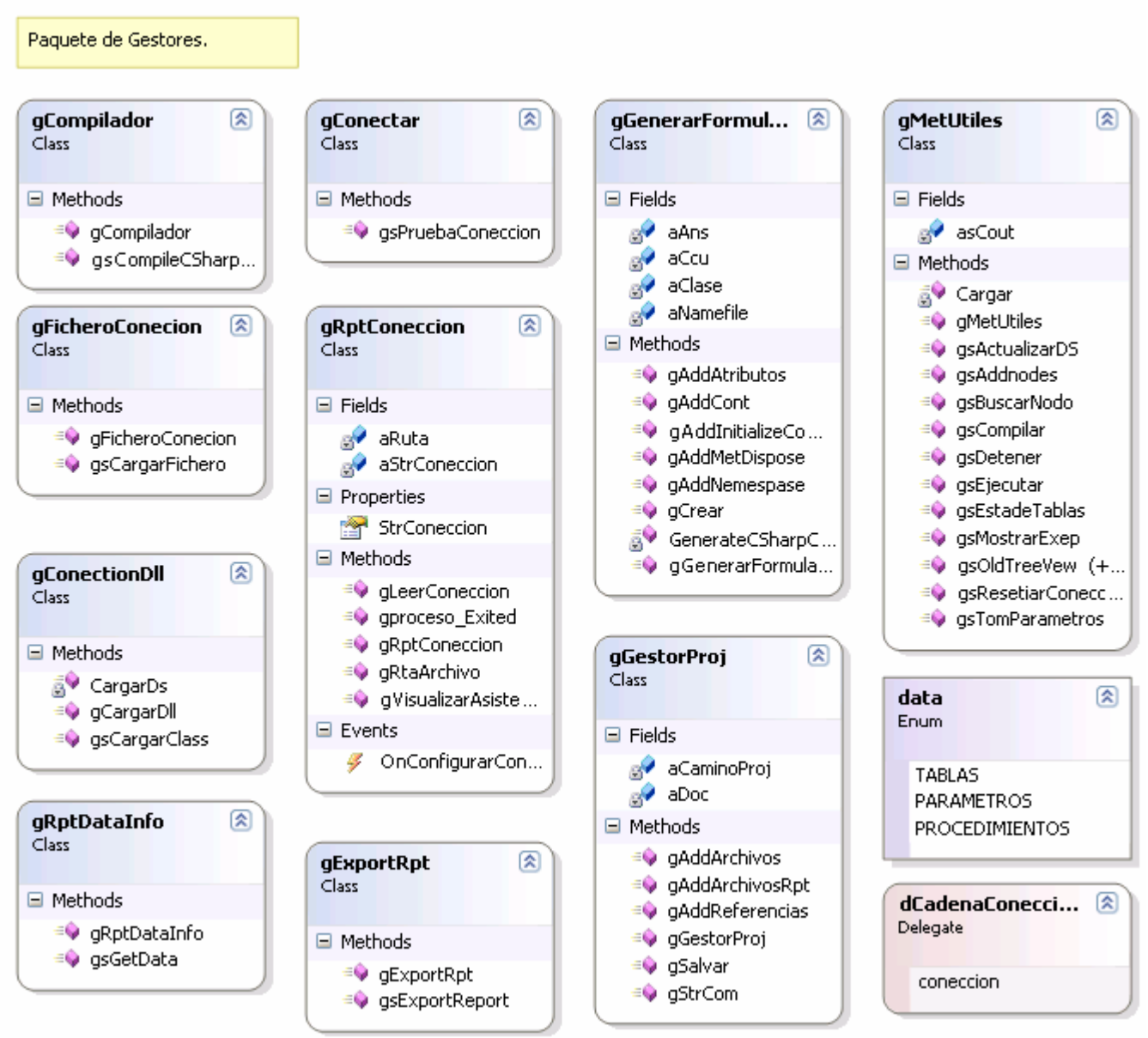

Figura 11 Paquete de gestores

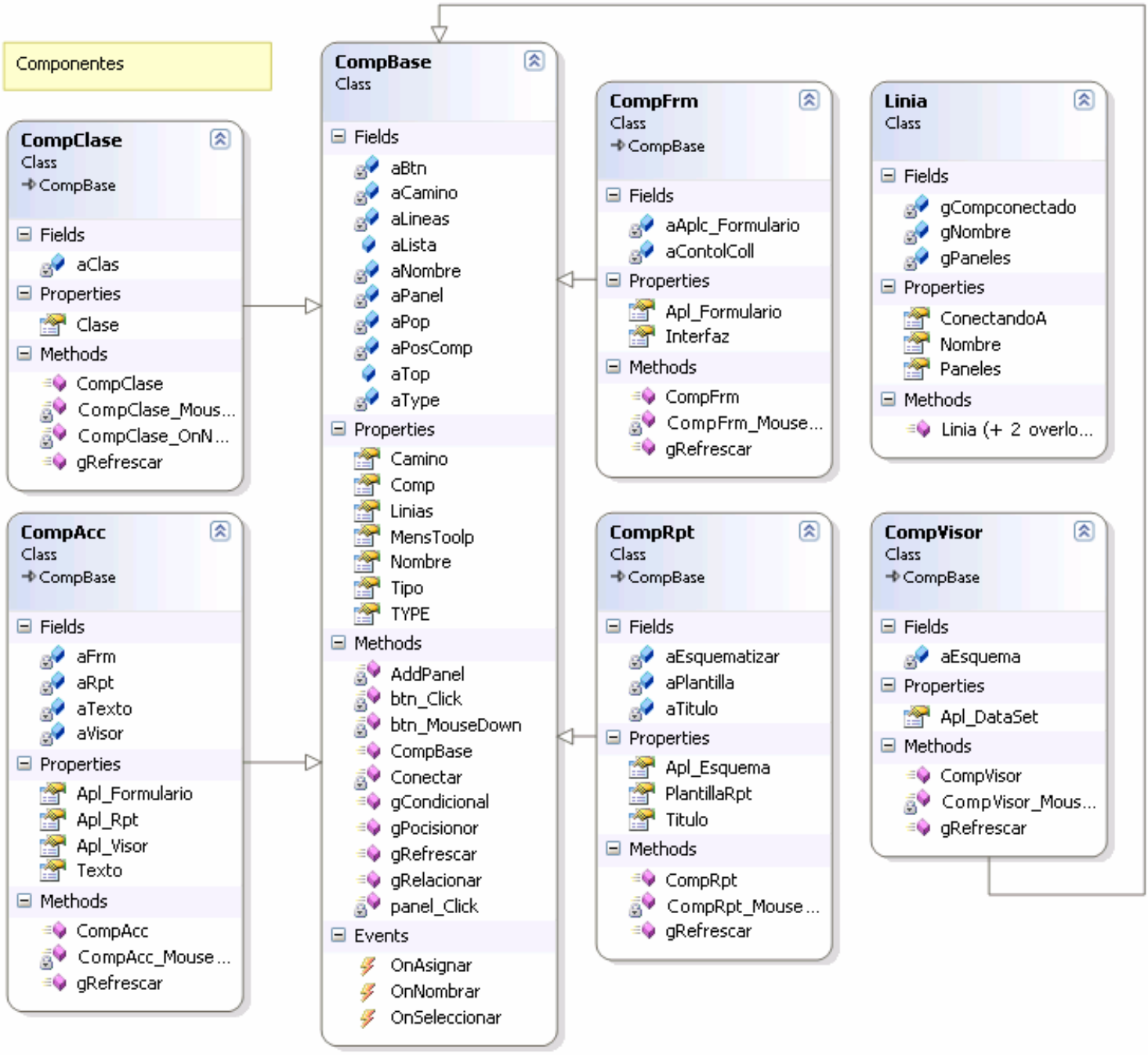

Figura 12 Paquete de componentes

2.8. Estándares codificación

Los estándares de codificación son reglas específicas a una lengua que reducen perceptiblemente el riesgo de que los desarrolladores introduzcan errores. Los estándares de codificación no destapan problemas existentes, evitan más bien que los errores ocurran. Los bugs frecuentes en programas pueden ser detectados mucho anterior o pueden incluso ser evitados totalmente. Durante el desarrollo, estándares de codificación ayudan a los ingenieros a producir un código de alta calidad y a entender y a utilizar el código de sus colegas. Pero también realzan considerablemente la capacidad de mantenimiento y rehúso a largo plazo del producto final. Tal práctica del control de bugs en el proceso del desarrollo mejora la calidad mientras que reduce el tiempo de desarrollo, el coste, y el esfuerzo (23).

A partir de esto se utilizó en la solución estándares de codificación, ya que se reduce la posibilidad de introducir errores, así como precisar algunos errores ocultados o inesperados. Por otro lado se hace el código más uniforme y más fácil de leer, ayudando así al mantenimiento del software y aumentando la robustez y confiabilidad.

2.8.1 Reglas de codificación

- **Constantes**
  - **Regla:** Los nombres de constantes comienzan con las letras cns en minúscula. El nombre de la constante sigue con la primera letra en mayúscula y de ser compuesto cada nueva palabra debe comenzar con Mayúscula.
  - **Justificación:** Tradicionalmente las constantes se nombran con las palabras que la componen en mayúsculas, separadas por underscores ´_´. Esta notación se confunde comúnmente con constantes y macros definidas a nivel de precompilador, por lo que se optó por reservar este formato para estos últimos símbolos. En todo caso, es importante recalcar en el nombre de la variable la naturaleza de ésta.
- **Referencias**

  - **Regla:** Todas las variables referencia deben comenzar con el prefijo "r" en minúscula. El resto del nombre de la variable se escribe igual que las variables miembro de la clase.
  - **Justificación:** Es importante recalcar el tipo de variable con la que se está trabajando.
- **Parámetros (argumentos) de métodos**
  - **Regla:** Los nombres comienzan con el prefijo "a" (de argumento) y continúan siguiendo el esquema de las variables miembro de clases (palabras en minúscula separadas por mayúscula).
  - **Justificación:** De esta forma es posible diferenciar claramente los argumentos de un método/función de las variables locales y miembros de clases (y otros tipos de variables). Cuando se hace referencia al valor de inicialización de una variable miembro de la clase, en caso de que se esté dentro de un constructor, el nombre de la variable debería ser el mismo que el de la variable miembro.
- **Variables globales**
  - **Regla:** Los nombres comienzan con el prefijo "g" y continúan siguiendo el esquema de las variables miembro de clases (palabras en minúscula separadas por mayúscula).
  - **Justificación:** Es importante saber a partir del nombre el alcance de la variable.
- **Variables locales**
  - **Regla:** Los nombres comienzan con el prefijo "l" y continúan siguiendo el esquema de las variables miembro de clases (palabras en minúscula separadas por mayúscula).
  - **Justificación:** Es importante saber a partir del nombre el alcance de la variable.
- **Variables estáticas**
  - **Regla:** Los nombres comienzan con el prefijo "s" y continúan siguiendo el esquema de las variables miembro de clases (palabras en minúscula separadas por mayúscula).

  - **Justificación:** Es importante saber a partir del nombre el alcance de la variable.
- **Variables miembros de clases**
  - **Regla:** Las variables miembros de clase se escriben con minúsculas, las variables con nombres compuestos, comienzan con la primer palabra enteramente en minúscula y el resto comenzando con mayúscula. Otra alternativa es comenzar con una letra "m" minúscula y separar las palabras en la misma forma.
  - **Justificación:** Las variables miembros de las clases suelen ser las más importantes y más usadas en casi todos los programas, por lo tanto es importante que su lectura sea fácil e intuitiva. Además esta es la forma más clásica de nombrar este tipo de variables.
- **Variables estáticas de clases**
  - **Regla:** Las variables estáticas de clase se escriben comenzando con "sm", el resto de la palabra se escribe como una variable miembro ordinaria (es decir, minúsculas separadas por mayúsculas).
  - **Justificación:** Es importante en todo caso explicitar en el nombre de la variable la clase de variable que representa.
- **Clases**
  - **Regla:** Los nombres de clases comienzan con mayúsculas, con las palabras que la forman en minúsculas y separadas por mayúsculas, de la misma forma que las variables miembro.
  - **Justificación:** Al igual que las variables miembro, los nombres de clase abundan en el código por lo que es práctico utilizar nombres intuitivos para ellas. La marca distintiva en este caso es la mayúscula inicial. Este formato es clásico y compatible con estándares de notación para lenguajes como C#.
- **Métodos miembros de clases**
  - **Regla:** Los métodos miembros de clases siguen la notación idéntica que para las variables miembro de clases, es decir, comenzando con minúscula y separando las palabras con mayúsculas.

  - **Justificación:** Al igual que las variables miembros de clase, estos nombres son muy utilizados y es clave una lectura rápida y clara de su significado.

- **Funciones globales**
  - **Regla:** Las funciones que no son miembro de ninguna clase siguen la misma notación que las variables globales, es decir, con una "g" minúscula como prefijo y separando las palabras con mayúsculas.
  - **Justificación:** Al igual que con las variables globales, es importante no confundir este tipo de funciones con métodos de clases, aumentando la claridad del código.
- **Nombres de enumerados**
  - **Regla:** Se agrega un prefijo "enum" al nombre del símbolo.
  - **Justificación:** Con el prefijo "enum", queda claro el tipo de símbolo. Además, es importante prefijar el nombre del enum en cada constante definida en su interior, ya que dichas constantes quedan luego visibles globalmente y pueden colisionar con otros símbolos definidos en el programa.

Objeto	**Prefijo**	**Ejemplo**
Form	Frm_	Frm_principal
Text	Txt_	Txt_nombre
Checkbox	Chk_	Chk_nuevo
Radio Button	Rad_	Rad_opcion
ListView	Lvw_	Lvw_datos
Button	Btn_	Btn_aceptar
Date TimePicker	Dtp_	Dtp_inicio
GroupBox	Gbx_	Gbx_datos
Label	Lbl_	Lbl_numero
DataGrid	Dtg_	Dtg_tablas
TreeView	Tree_	Tree_rpt

Figura 13 Nombres de objetos de formularios

- **Comentarios**

Regla

Se usan los comentarios en línea para facilitar la comprensión del código, sobre todo en procedimientos complejos.

- **Acciones**

Regla

Los nombres de cada una de las acciones comienzan con el prefijo acc, con las palabras que la forman en minúsculas y separadas por mayúsculas.

CAPÍTULO 3: VALIDACIÓN DE LA SOLUCIÓN PROPUESTA

3.1. *Introducción*

En el presente capitulo se comienza la validación de la solución propuesta, a través de diferentes test de unidades. Dentro de estos test, están las pruebas de caja blanca y pruebas de caja negra, definiéndose el objetivo de cada una de estas, así como su alcance y detalles de las mismas. El principal beneficio de realizar estas pruebas unitarias es que permiten aislar segmentos del programa que pueden ser probados para verificar que funcionan correctamente.

3.3. Pruebas de Caja Negra

Objetivo

El objetivo de realizar este tipo de prueba al sistema es para detectar el incorrecto o incompleto funcionamiento de este, así como los errores de interfaces, rendimiento y errores de inicialización y terminación.

Alcance

El proceso de pruebas de caja negra se va a centrar principalmente en los requisitos funcionales del software para verificar el comportamiento de la unidad observable externamente y la calidad funcional.

Descripción

Estas pruebas permiten obtener un conjunto de condiciones de entrada que ejerciten completamente todos los requisitos funcionales del programa. En ellas se ignora la estructura de control, concentrándose en los requisitos funcionales del sistema y ejercitándolos.

La prueba de Caja Negra no es una alternativa a las técnicas de prueba de la Caja Blanca, sino un enfoque complementario que intenta descubrir diferentes tipos de errores a los encontrados en los métodos de la Caja Blanca.

Dentro de las técnicas de Prueba de Caja Negra, se utilizó la Partición de Equivalencia. Esta técnica divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones de software. Mediante la aplicación de esta se examinó los valores válidos e inválidos

de las entradas existentes en el software, descubriendo de forma inmediata clase de errores que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico. La partición equivalente se dirige a la definición de casos de pruebas que descubran clases de errores, reduciendo así en número de clases de prueba que hay que desarrollar.

Para definir las clases de equivalencia se tuvieron en cuenta un conjunto de reglas:

- Si una condición de entrada especifica un rango, entonces se confeccionan una clase de equivalencia válida y dos inválidas.
- Si una condición de entrada especifica la cantidad de valores, se identifica una clase de equivalencia válida y dos inválidas.
- Si una condición de entrada especifica un conjunto de valores de entrada y existen razones para creer que el programa trata en forma diferente a cada uno de ellos, se identifica una clase válida para cada uno de ellos y una clase inválida.
- Si una condición de entrada especifica una situación de tipo “debe ser”, se identifica una clase válida y una inválida.

Luego de tener las clases válidas e inválidas definidas, se procedió a definir los casos de pruebas, para los casos de uso más significativos (Gestionar conexión, Generar clases, Configurar reporte, Crear reporte). Cada uno de estos casos de pruebas se definió teniendo en cuenta lo siguiente:

a. Escribir un nuevo caso de cubra tantas clases de equivalencia válidas no cubiertas como sea posible hasta que todas las clases de equivalencia hayan sido cubiertas por casos de prueba.

b. Escribir un nuevo caso de prueba que cubra una y solo una clase de equivalencia inválida hasta que todas las clases de equivalencias inválidas hayan sido cubiertas por casos de pruebas.

Caso de Pruebas

Caso de Uso	*Conectar Formulario*
Caso de prueba	*Conectar Formulario*

Condiciones				
Clases Válidas	**Clases Inválidas**	**Resultado Esperado**	**Resultado de la Prueba**	**Observ.**
Un Formulario solo puede conectar a otro Formulario o a una Clase.		Se conecta satisfactoriamente el componente.		
	Conectar un formulario a un Rpt		Se muestra un mensaje de Error (Esta relación no es permitida)	
	Conectar un formulario a un visor			
	Conectar un formulario a una acción			

Caso de Uso	*Conectar Clase*			
Caso de prueba	*Conectar Clase*			
Condiciones				
Clases Válidas	**Clases Inválidas**	**Resultado Esperado**	**Resultado de la Prueba**	**Observ.**

Una Clase solo puede conectar a otra clase.		Se conecta satisfactoriamente a la clase.	Se muestra un mensaje de Error (Esta relación no es permitida)	
	Conectar una Clase con un Visor			
	Conectar una Clase con una Acción			
	Conectar una Clase con un Frm			

Caso de Uso	*Conectar Visor*			
Caso de prueba	*Conectar Visor*			
Condiciones				
Clases Válidas	**Clases Inválidas**	**Resultado Esperado**	**Resultado de la Prueba**	**Observ.**
Solamente una acción puede conectar un visor.		Se conecta satisfactoriamente el visor	Se muestra un mensaje de Error (Esta relación no	

	Conectar un visor a un Frm, Clase o Rpt		es permitida)	
	Conectar un visor a una Clase o Rpt			
	Conectar un visor a un Rpt			

Caso de Uso	*Conectar Rpt*			
Caso de prueba	*Conectar Rpt*			
Condiciones				
Clases Válidas	**Clases Inválidas**	**Resultado Esperado**	**Resultado de la Prueba**	**Observ.**
Solamente una acción puede conectar un Rpt.		Se conecta satisfactoriamente el Rpt	Se muestra un mensaje de Error (Esta relación no es permitida)	
	Conectar un Rpt a una Acción			

	Conectar un Rpt a un Visor			

Caso de Uso	*Conectar Acción*			
Caso de prueba	*Conectar Acción*			
Condiciones				
Clases Válidas	**Clases Inválidas**	**Resultado Esperado**	**Resultado de la Prueba**	**Observ.**
Una acción puede estar conectada con el resto de los componentes		Se conecta satisfactoriamente la acción	Se conecta satisfactoriamente la acción	

Caso de Uso	*Salvar Rpt*			
Caso de prueba	*Salvar Rpt*			
Condiciones				
Clases Válidas	**Clases Inválidas**	**Resultado Esperado**	**Resultado de la Prueba**	**Observ.**

La plantilla tiene que estar asociada		Se crea satisfactoriamente	Se conecta satisfactoriamente la acción	
	No incluir plantilla		Se muestra un mensaje de error (Debe asignar una plantilla)	

3.4. Pruebas de Caja Blanca

Objetivo

El objetivo de realizar este tipo de prueba al sistema es que se garantice que se ejerciten por lo menos una vez todos los caminos independientes de cada módulo o método, todos los bucles en sus límites operacionales así como las estructuras internas de datos para asegurar su validez.

Alcance

El proceso de pruebas de caja blanca se va a concentrar principalmente en validar a través del framework de software libre Nunit, que cada uno de los módulos o segmentos de códigos funcione apropiadamente.

Descripción

La prueba de Caja Blanca es considerada como uno de los tipos de pruebas más importantes que se le aplican a los *software*, logrando como resultado que disminuya en un gran por ciento el número de errores existentes en los sistemas y por ende una mayor calidad y confiabilidad (24).

Para el desarrollo de estas pruebas unitarias se utilizó la herramienta Nunit, ya que ofrece las funcionalidades necesarias para implementar pruebas en un proyecto desarrollado en C#. Además provee una interfaz gráfica para ejecutar y administrar las mismas.

Mediante esta herramienta se analizaron ensamblados generados por .NET y se interpretaron las pruebas inmersas en ellos, ya que 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. Para realización de cada una de las pruebas de caja blanca, se cargo en el entorno de Nunit 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 se tiene que volver a cargar el ensamblado al entorno de Nunit, si no que este siempre obtiene la última versión del mismo. NUnit ofrece una interface simple que informa si cada una de las pruebas realizadas o conjunto de pruebas falló, pasó o fue ignorada.

3.4.1 Métrica de la complejidad ciclomática

Se le realizó el cálculo de la complejidad ciclomática en todos los bloques de código dentro del sistema, la cual ayudó a reflejar una medida de la complejidad del código escrito, teniendo en cuenta el número de destinos posibles. Además permitió conocer el esfuerzo a realizar en cada una de las pruebas, el cual es exactamente el valor de la complejidad ciclomática en cada elemento o módulo. A partir de esta medida, se diseñaron pruebas que forzaron el recorrido de estos caminos, lo cual garantiza que se ejecute al menos una vez cada sentencia del programa y que cada condición se ejecute en sus variantes verdadera y falsa.

Ejemplo de segmentos de código analizados:

```
public static void  sgRepresSub(string directo,TreeNode nodo)
{

    System.IO.DirectoryInfo inf=new System.IO.DirectoryInfo(directo);
    TreeNode tree=new TreeNode();
    tree.Text=inf.FullName;

    if(inf.GetDirectories().Length>0 )
    {
        foreach(System.IO.DirectoryInfo indexinf in inf.GetDirectories())
        {
            TreeNode dd=new TreeNode();
            dd.ImageIndex=0;

            dd.Text=sgNombreNodo(indexinf.FullName);
            if(indexinf.GetDirectories().Length>0)
                dd.Nodes.Add("");
            nodo.Nodes.Add(dd);
        }
    }

}
```

Figura 14 Representa en un árbol los subdirectorios de un directorio entrado.

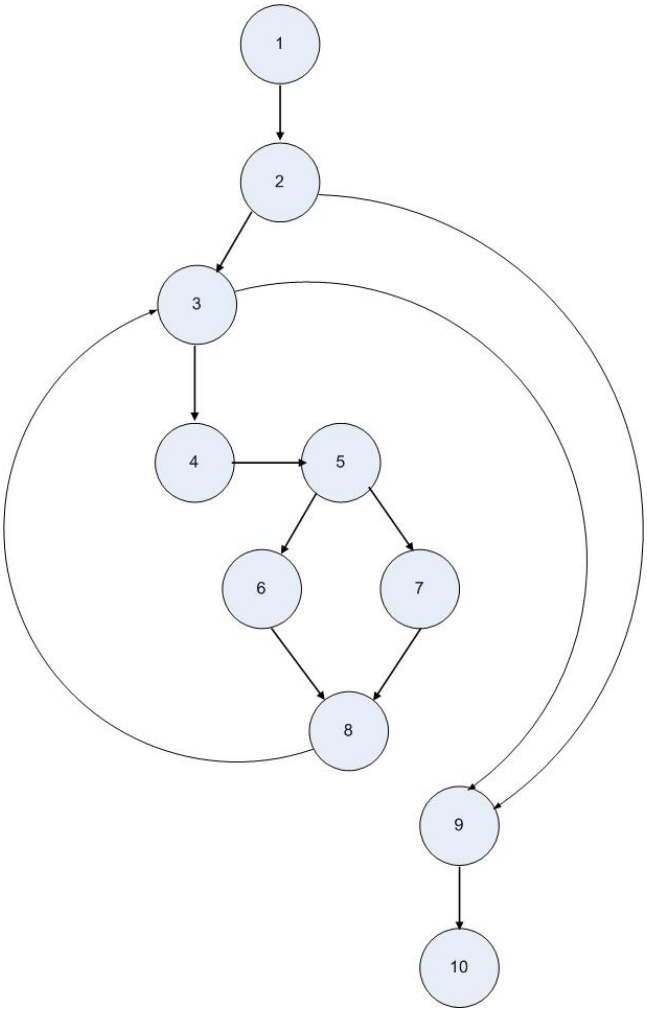

Figura 15 Grafo de flujo

Complejidad Ciclomática *V(G)*

V(G) = Nº de Aristas - Nº de Nodos + 2

V(G) =12-10+2

V(G) =4

Caminos Independientes:

- 1-2-9-10
- 1-2-3-9-10
- 1-2-3-4-5-6-8-3-9-10
- 1-2-3-4-5-7-8-3-9-10

```
public static void sgCompilar()
{
    int Rect=0;
    if(accEditor.Instancia!=null)
    {
        frmRun run=new frmRun();
        run.Closing+=new CancelEventHandler(run_Closing);
         asCout=accEditor.Instancia.pnlEditor.Controls.Count;

        for(int i=0;i<asCout;i++)
        {   if((accEditor.Instancia.pnlEditor.Controls[0] is RectTracker)){
                accEditor.Instancia.pnlEditor.Controls[0].Dispose();
                Rect++;
            }
            else
            run.Controls.Add(accEditor.Instancia.pnlEditor.Controls[0]);

        }
    asCout=asCout-Rect;
    run.Show();
    accEditor.Instancia.pnlEditor.Visible=false;
    sgEjecutar();

    }
}
```

Figura 15 Se pasan los controles del editor de interfaces para el formulario run.

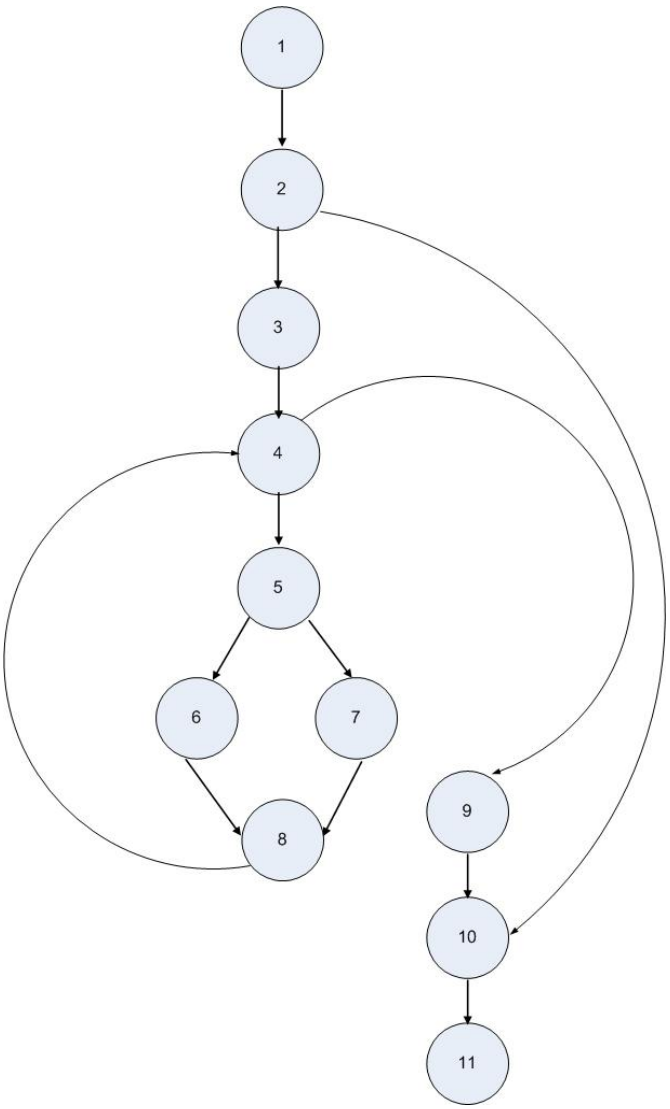

Figura 17 Grafo de flujo

Complejidad Ciclomática *V(G)*

V(G) = Nº de Aristas - Nº de Nodos + 2

V(G) =13-11+2

V(G) =4

Caminos independientes:

- 1-2-10-11
- 1-2-3-4-9-10-11
- 1-2-3-4-5-7-8-4-9-10-11
- 1-2-3-4-5-6-8-4-9-10-11

```
public static void sgEjecutar()
{
    if(accOledb.Instancia.rdbProc.Checked)
    {
        Conectar.sgPruebaConeccion(accEditor.Instanciac.Controles,accOledb.asgConect,
            accOledb.Instancia.treeInfo.SelectedNode.Text);
        sCargar(frmPrincipal.Instancia.DS);
    }
    else
    {
        if(accDll.MEtodo!=null)
        {
            Type Ctipdll =Type.GetType(accDll.sgTomarTipodeClase(accDll.Instancia.treeDll.SelectedNode.Parent.Text)
            object obj=accDll.MEtodo.Invoke(accDll.MEtodo.DeclaringType,sgTomParametros());
            sCargar(obj);
            if(obj is DataSet)
            frmPrincipal.Instancia.DS=(DataSet)obj;

        }
        else sCargar(frmPrincipal.Instancia.DS);
    }
   frmPrincipal.Instancia.DS=frmPrincipal.Instancia.DS;

}
```

Figura 16 Se prueba el tipo de conexión a datos escogido.

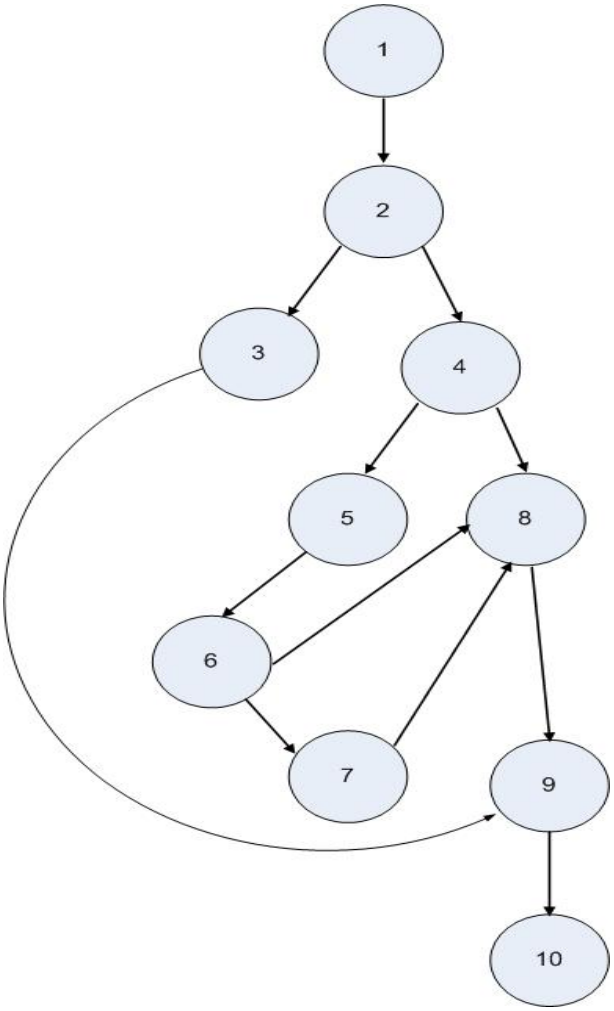

Figura 17 Grafo de flujo

Complejidad Ciclomática V(G)

V(G) = Nº de Aristas - Nº de Nodos + 2

V(G) =12-10+2

V(G) =4

Caminos independientes:

- 1-2-3-9-10
- 1-2-4-8-9-10
- 1-2-4-5-6-8-9-10
- 1-2-4-5-6-7-8-9-10

```
private void Editor_MouseUp(object sender, MouseEventArgs e)
{

    CompBase Cbase;

    if(Instancia.Componetes.Buttons[aComp].Pushed)
    {
        switch(aComp)
        {
            case 0:Cbase=new CompAcc();break;

            case 1: Cbase=new CompClase();break;

            case 2:Cbase=new CompRpt();break;

            case 3:Cbase=new CompVisor();break;

            case 4:Cbase=new CompFrm();break;

            default:Cbase=null; break;


        }
        gADDControl(Cbase,e.X,e.Y);
        Instancia.Componetes.Buttons[aComp].Pushed=false;

    }
    Instancia.Controls.Remove(asgCSharpTracker);
}
```

Figura 18 Se adiciona al Editor de controles de la acción accContComponent, el control escogido en la barra de componentes.

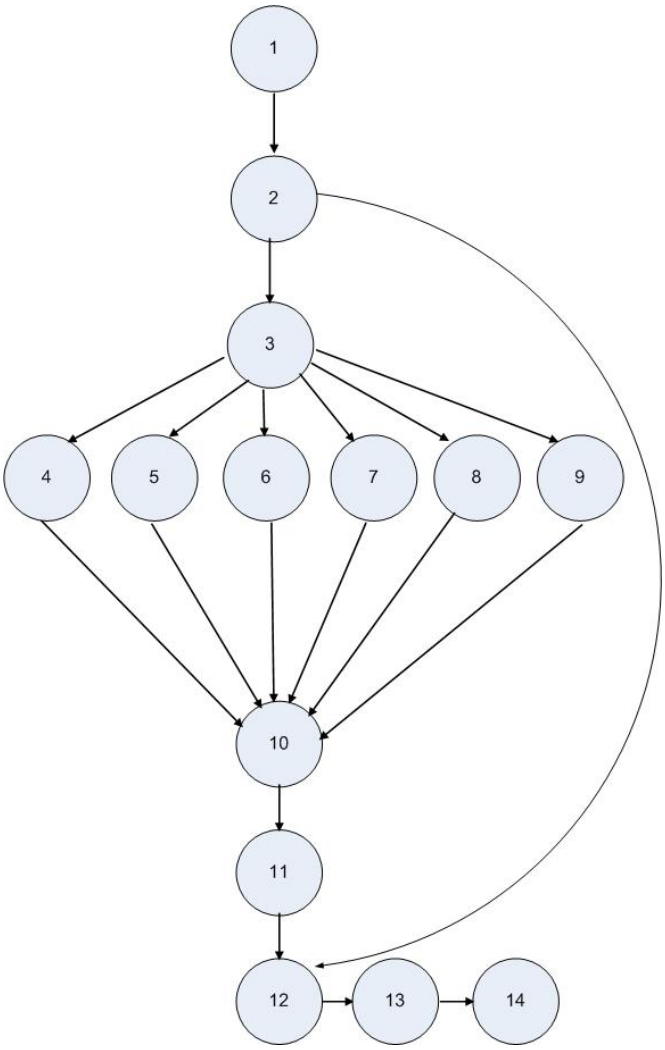

Figura 19 Grafo de flujo

Complejidad Ciclomática *V(G)*

V(G) = Nº de Aristas - Nº de Nodos + 2

V(G) =19-14+2

V(G) =7

Caminos independientes:

- 1-2-12-13-14
- 1-2-3-4-10-11-12-13-14
- 1-2-3-5-10-11-12-13-14
- 1-2-3-6-10-11-12-13-14
- 1-2-3-7-10-11-12-13-14
- 1-2-3-8-10-11-12-13-14
- 1-2-3-9-10-11-12-13-14

```
private void toll_ButtonClick(object sender, ToolBarButtonClickEventArgs e)
{
    aToll=Instancia.toll.Buttons.IndexOf(e.Button);
    if(Instancia.Prop.SelectedObject!=null)
    {
        switch(aToll)
        {
            case 0:asgCSharpTracker.Control.BringToFront();break;

            case 1:asgCSharpTracker.Control.SendToBack();break;

            case 2:CSharpTracker_OnEliminar((Instancia.Prop.SelectedObject as CompBase).Comp);
                   asgCSharpTracker.Control.Dispose();
                   asgCSharpTracker.Dispose();break;
            case 3:gGenerar((Instancia.Prop.SelectedObject as CompBase));
                   MostrarMensage.Mensage("Se ha salvado satisfactoriamente el fichero.");
                break;
            case 4:foreach(CompBase comp in asgCompBases)
                   {
                   gGenerar(comp);
                   }
                   MostrarMensage.Mensage("Se han salvado satisfactoriamente los ficheros.");
                break;

            default: break;
        }
    }
}
```

Figura 20 Se gestionan las operaciones que se les aplicarán a los controles de la clase accContComponent.

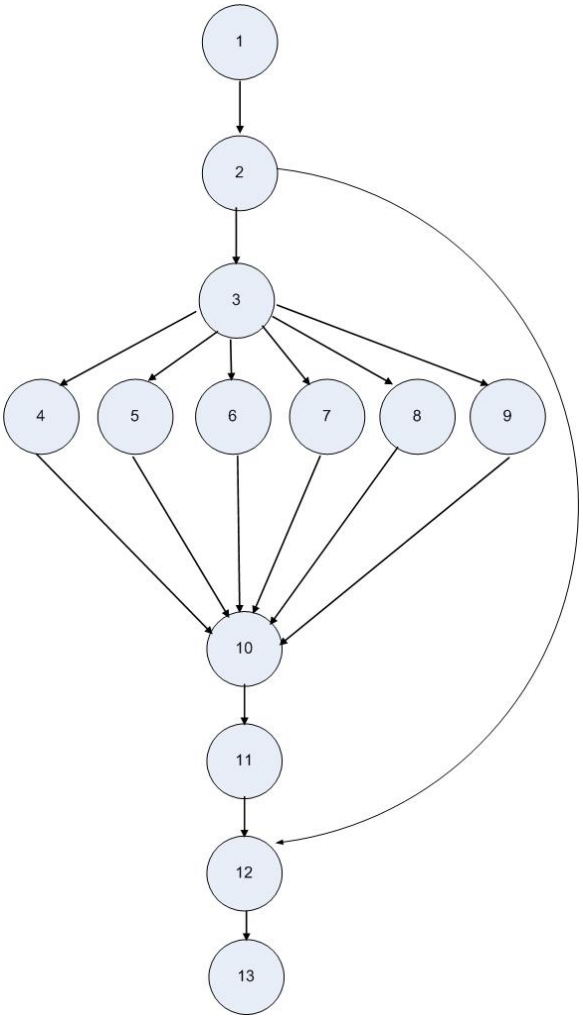

Figura 23 Grafo de flujo

Complejidad Ciclomática *V(G)*

V(G) = Nº de Aristas - Nº de Nodos + 2

V(G) =18-13+2

V(G) =7

Caminos independientes:

- *1-2-12-13*
- *1-2-3-4-10-11-12-13*
- *1-2-3-5-10-11-12-13*
- *1-2-3-6-10-11-12-13*
- *1-2-3-7-10-11-12-13*
- *1-2-3-8-10-11-12-13*
- *1-2-3-9-10-11-12-13*

Muchos actores han definido umbrales respecto al valor de la métrica de la complejidad ciclomática.

Valor de V(G)	**Evaluación**
1-10	Un programa simple sin mucho riesgo
11-20	Medianamente complejo
21-50	Un programa complejo
50+	Programa inestable

Figura 214 Tabla de posibles valores

A partir del análisis del cálculo de la complejidad ciclomática en la propuesta de solución, el cual se encuentra entre los valores 11-20, siendo 11 el valor más alto de *V(G)* en la solución y del resultado de cada una de las pruebas, se llega a la conclusión de que de forma general los resultados obtenidos fueron positivos teniendo en cuenta que el sistema es medianamente complejo, presenta un bajo riesgo de fallos, ya que está demostrado que existe una alta correlación entre código con fallos y alta complejidad ciclomática. Se demostró que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene. La utilización del framework Nunit en las pruebas, permitió encontrar defectos y anomalías no esperadas, lo cual resultó de suma utilidad teniendo en cuenta que el período de tiempo para desarrollar fue relativamente corto y se necesitaba un proyecto que corriera con la mayor calidad y estabilidad posible.

3.5. Conclusiones

Se crearon los diagramas de clases de diseño del sistema, además se hizo una descripción de cada una de las principales clases necesarias en la solución, especificándose su responsabilidad en la misma. Se creó el modelo de implementación que se refiere a la creación de los reportes contables y financieros. Se arribaron a las siguientes conclusiones:

- Se definieron y pusieron en práctica estándares de codificación en la implementación de la solución propuesta, lo que ayudó a tener un código más uniforme, claro y fácil de mantener.
- Se concluye que las pautas de diseño gráfico establecidas para la aplicación, cumplen con el objetivo de desarrollar un entorno amigable en la extracción de reportes contables y financieros.
- Se realizaron pruebas de unidad que permitieron validar la solución, y encontrar errores ocultos, así como la complejidad del software.

Conclusiones Generales

La realización de este Trabajo de Diploma ha aportado importantes conocimientos a los autores, en la ceración de un sistema de reportes. Para llevar a fin el objetivo principal se estudiaron las últimas tendencias y tecnologías actuales, según el rol de programador para desarrollar aplicaciones de este tipo.

Se logró identificar los elementos necesarios para la elaboración del modelo de implementación que se refiere a la creación de reportes para la gestión contable en los Registros y Notarias de la República Bolivariana de Venezuela, basado en la situación problémica actual.

Llegando así a la propuesta de un sistema de reportes para la gestión económica de los Registros y Notarias de la República Bolivariana de Venezuela, lográndose una mayor celeridad en la creación de estos.

Finalmente la investigación ha servido para consolidar conocimientos en esta rama de la ingeniería informática.

Recomendaciones

Se recomienda migrar la solución propuesta hacia alguna plataforma de software libre, preferentemente el Proyecto Mono lo cual permitirá la reutilización de todo el código ya desarrollado. Además de migrarla hacia la plataforma Java para así estar en consecuencia con el objetivo que persigue el Polo de la Facultad (Sistemas Legales), y pueda ser utilizado en proyectos semejantes.

Bibliografía

1. **Ceballos, Geykel Raúl Moreno.** SERVICIO AUTÓNOMO DE REGISTROS Y DEL NOTARIADO: MÓDULO INMOBILIARIO. *.Net framework.*

2. —. SERVICIO AUTÓNOMO DE REGISTROS Y DEL NOTARIADO: MÓDULO INMOBILIARIO. *Microsoft.Net .* 2006.

3. —. SERVICIO AUTÓNOMO DE REGISTROS Y DEL NOTARIADO: MÓDULO INMOBILIARIO. *Visual Studio .NET.* 2006.

4. **Cebellos, Geykel Raul Moreno.** SERVICIO AUTÓNOMO DE REGISTROS Y DEL NOTARIADO: MÓDULO INMOBILIARIO. *Common Language Runtime (CLR).*

5. Sun. *Java 2 Enterprise Edition.* [En línea] [Citado el: 06 de 12 de 2006.] http://es.sun.com.

6. Crystalintelligence. *Crystal Report.* [En línea] [Citado el: 15 de 04 de 2006.] http://www.crystalintelligence.com.

7. xtras.net. *ActiveReport.* [En línea] [Citado el: 12 de 04 de 2006.] http://www.xtras.net.

8. informatizate.net. *Metodologías De Desarrollo De Software XP.* [En línea] [Citado el: 20 de 11 de 2006.] http://www.informatizate.net.

9. abcdatos.com. *Nunit.* [En línea] [Citado el: 09 de 03 de 2007.] http://www.abcdatos.com.

10. **Mañas, Jose A.** Prueba de Programas. *Prueba de Programas.* [En línea] 16 de Marzo de 1994. [Citado el: 25 de Enero de 2007.] Prueba de Programas.htm.

11. **Ceballos, Geyquel Raul Moreno.** SERVICIO AUTÓNOMO DE REGISTROS Y DEL NOTARIADO: MÓDULO INMOBILIARIO. *Desarrollo basado en pruebas.*

12. **Guilarte, Alianis La Hoz.** Gestión Documental de Registros y Notarias, Rol Ingeniero de Pruebas. La Habana : s.n., 2007.

13. *Conferencia Pruebas- 1.* **UCI.** La Habana : s.n., 2005.

14. microsoft.com. *Estructuras de datos.* [En línea] http://www.microsoft.com/spanish/MSDN/estudiantes/algoritmica/estructuras/default.asp.

15. **Hidalgo, Julián.** conclase.net. *Algoritmos de ordenamiento.* [En línea] http://www.conclase.net.

16. **Vallecillo, Rosa Guerequeta y Antonio.** *Técnica de Diseño de Algoritmos.* Málaga : s.n., 2000.
17. foldoc. *Programación imperativa.* [En línea] http://www.foldoc.org.
18. desarrolloweb. *Qué es la programacion orientada a objetos.* [En línea] http://www.desarrolloweb.com.
19. webcindario. *POO como solución.* [En línea] http://programarenc.webcindario.com .
20. *Programación orientada a agentes.* [En línea] http://listserv.rediris.es .
21. *Programación orientada a aspectos.* [En línea] http://www.angelfire.com.
22. **Ramírez, Elizabeth.** microsoft. *Patrones de diseño.* [En línea] http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art163.asp.
23. elrincondelprogramador. *Patrones de creación.* [En línea] http://www.elrincondelprogramador.com .
24. **Pressman, Roger S.** *Ingenieria del Software .* La Habana : Félix Varela, 2005.

GLOSARIO DE TÉRMINOS

A

API

Del inglés (Application Programming Interface - Interfaz de Programación de Aplicaciones) es un conjunto de especificaciones de comunicación entre componentes software. Se trata del conjunto de llamadas al sistema que ofrecen acceso a los servicios del sistema desde los procesos y representa un método para conseguir abstracción en la programación, generalmente entre los niveles o capas inferiores y los superiores del software.

D

DLL

Es el acrónimo de Dynamic Linking Library (Bibliotecas de Enlace Dinámico), término con el que se refiere a los archivos con código ejecutable que se cargan bajo demanda del programa por parte del sistema operativo. Esta denominación se refiere a los sistemas operativos Windows siendo la extensión con la que se identifican los ficheros, aunque el concepto existe en prácticamente todos los sistemas operativos modernos

G

GDI+

Es la tecnología contenida en .NET Framework con la que se pueden generar salidas gráficas, textuales y trabajo con mapas de bits e imágenes.

GPL

Del ingles (General Public License o licencia pública general) es una licencia creada por la Free Software Foundation a mediados de los 80, y está orientada principalmente a proteger la libre

distribución, modificación y uso de software. Su propósito es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios.

I

IDE

Un entorno de desarrollo integrado o en inglés Integrated Development Environment ('IDE') es un programa compuesto por un conjunto de herramientas para un programador. Puede dedicarse en exclusiva a un sólo lenguaje de programación o bien, poder utilizarse para varios. Es un entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica GUI.

J

JDBC

Es el acrónimo de Java Database Connectivity, un API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java independientemente del sistema de operación donde se ejecute o de la base de datos a la cual se accede utilizando el dialecto SQL del modelo de base de datos que se utilice.

JIT

Es una técnica para mejorar el rendimiento de sistemas de programación que compilan a bytecode, consistente en traducir el bytecode a código máquina nativo en tiempo de ejecución. La compilación en tiempo de ejecución se construye a partir de dos ideas anteriores relacionadas con los entornos de ejecución: la compilación a bytecode y la compilación dinámica.

JMS

Es un estándar de mensajería que permite a los componentes de aplicaciones basados en la plataforma de Java 2 crear, enviar, recibir y leer mensajes.

John Backus

Fue un informático estadounidense, ganador del Premio Turing en 1977 por sus trabajos en sistemas de programación de alto nivel, en especial por su trabajo con FORTRAN.

P

Polimorfismo

En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen objetos de diferentes clases de responder al mismo mensaje.

R

RMI

Java Remote Method Invocation, es un mecanismo ofrecido en Java para invocar un método remotamente. Al ser RMI parte estándar del entorno de ejecución Java usarlo provee un mecanismo simple en una aplicación distribuida que solamente necesita comunicar servidores codificados para Java.

S

Servicio Web

En inglés (Web Service) es una colección de protocolos y estándares que sirven para intercambiar datos entre aplicaciones.

W

Win32

Significa "Windows 32 bits". Hace referencia a todas las plataformas de 32 bits del sistema operativo Windows: Windows NT, Windows 95, Windows 98, Windows CE.