Universidad de las Ciencias Informáticas

Facultad 3



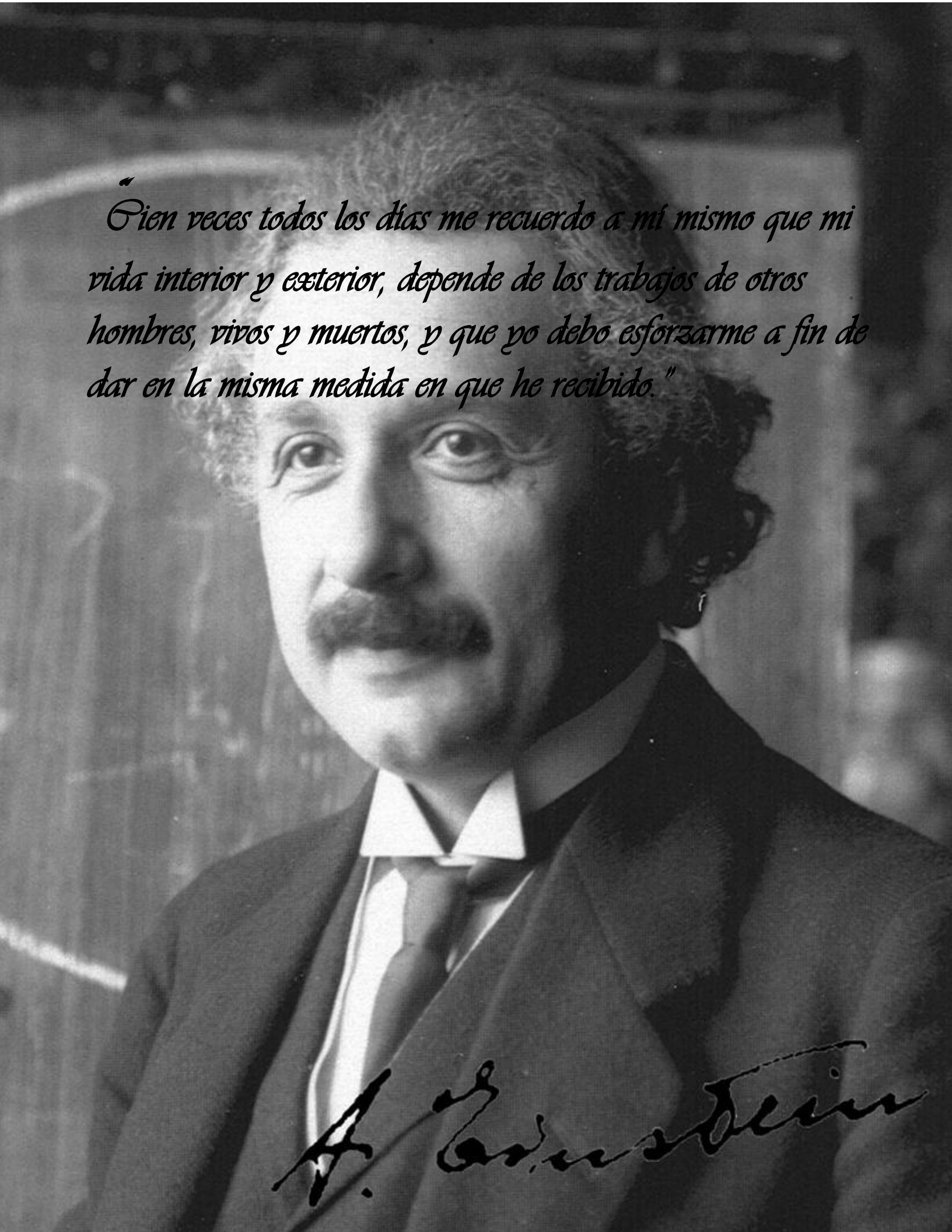
“Diseño e Implementación del módulo Interfaces del plugins Génesis”.

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

Autor(es): Jorge Antonio Fariñas Acosta

Tutor(es): Ing. Yulier Matías León

La Habana, Cuba
junio , 2011


"Cien veces todos los días me recuerdo a mí mismo que mi vida interior y exterior, depende de los trabajos de otros hombres, vivos y muertos, y que yo debo esforzarme a fin de dar en la misma medida en que he recibido."
A. Einstein

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor del trabajo titulado: *Diseño e Implementación del módulo Interfaces del plugins Génesis,* y autorizo a la Universidad de las Ciencias Informáticas los derechos patrimoniales del mismo, con carácter exclusivo.

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

Jorge Antonio Fariñas Acosta

_____

Firma del Autor

Yulier Matías León

_____

Firma del Tutor

DATOS DE CONTACTO

Ing. Yulier Matías León.

Graduado en la Universidad de las Ciencias Informáticas. Título de oro. 2 años de experiencia. Analista del sistema Quarxo. Categoría docente Instructor. Profesor de la asignatura P3.

RESUMEN

En la actualidad la complejidad de los sistemas a desarrollar se incrementa constantemente debido a los constantes cambios y avances en el mundo de la computación. El éxito de los departamentos y compañías de Tecnología de Información (TI) está dado en gran medida por la habilidad de construir sistemas de acuerdo a las cada vez más estrictas restricciones de tiempo y calidad que impone el mercado, exigiendo además que estos sistemas sean los suficientemente flexibles para responder de forma ágil a los cambios de requerimientos tecnológicos o del negocio mismo. Es fácil suponer entonces, que aquello que buscan todos los métodos, tecnologías, arquitecturas y lenguajes existentes, es brindar a las personas que trabajan en TI, herramientas y/o procesos capaces de acelerar la construcción de sistemas de software con calidad.

Por tales motivos y después de una previa investigación, el Departamento de Soluciones Financieras del centro CEIGE concibe la necesidad de la creación de una solución informática personalizada que permita la gestión de un modelo de interfaz en el que se almacenen los datos de una interfaz, independiente de la plataforma o arquitectura que se utilice. Esta solución es denominada Génesis y tiene como objetivo llevar a cabo la creación de modelos que recojan toda la información necesaria sobre el sistema. Para ello se hace un estudio sobre el Desarrollo Dirigido por Modelos (MDD), paradigma que establece a los modelos como principal elemento del proceso de desarrollo de software, es decir, sólo el modelo de una aplicación se ha de desarrollar y a partir de este se generará el resto de la aplicación.

PALABRAS CLAVE

Proyectos productivos, interfaz, modelo, entidad, metodología, MDD, desarrollo de software.

TABLA DE CONTENIDO

RESUMEN II

INTRODUCCIÓN 6

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA 10

1.1 Introducción 10

1.2 Marco Conceptual 10
1.2.1 Proceso de desarrollo de software 10
1.2.2 Modelo y Metamodelo 12
1.2.3 Desarrollo de software dirigido por modelos 13

1.3 Sistemas de desarrollo de software dirigido por modelos 17

1.4 Tecnologías y Herramientas 20
1.4.1 eXtensible Markup Language (XML) 20
1.4.2 Lenguaje de modelado 20
1.4.3 Metodologías 21
1.4.4 Herramienta CASE 22
1.4.5 Herramienta Ecore 23
1.4.6 Lenguaje de programación 23
1.4.7 Entorno de Desarrollo Integrado (IDE) 24
1.4.8 Subversion (SVN) 27
1.4.9 Patrones 27

1.5 Conclusiones parciales 32

CAPÍTULO 2: PROPUESTA DE SOLUCIÓN 33

2.1 Introducción 33

2.2 Importancia del diseño 33

2.3 Modelado del sistema 33
2.3.1 Especificación de los Casos de Uso (CU) del sistema 33

2.4 Arquitectura de la solución.....35
2.4.1 Vista lógica del módulo Interfaces del plugin Génesis.....36
2.4.2 Modelo Vista Controlador (MVC).....38

2.5 Patrones de diseño utilizados.....40

2.6 Modelo de diseño.....40
2.6.1 Diagrama de paquetes.....41
2.6.2 Diagrama de clases visuales del diseño.....43
2.6.3 Diagrama de interacción: diagrama de secuencia.....45
2.6.4 Diagrama de clases persistentes (metamodelo).....45

2.7 Conclusiones parciales.....47

CAPÍTULO 3: IMPLEMENTACIÓN Y PRUEBA.....48

3.1 Introducción.....48

3.2 Estándares de codificación.....48

3.3 Modelo de implementación.....50
3.3.1 Diagrama de componentes.....51
3.3.2 Diagrama de despliegue.....51

3.4 Métricas de software.....52
3.4.1 Resultados obtenidos de la aplicación de la métrica TOC.....55
3.4.2 Resultados obtenidos de la aplicación de la métrica RC.....58
3.4.3 Matriz de inferencia de indicadores de calidad.....61

3.5 Pruebas de software.....63
3.5.1 Pruebas estructurales o de caja blanca.....63
3.5.2 Pruebas funcionales o de caja negra.....74

3.7 Conclusiones parciales.....76

CONCLUSIONES.....77

RECOMENDACIONES.....78

REFERENCIAS BIBLIOGRAFICAS 79

INDICE DE FIGURAS

Figura 1: Estructura de la arquitectura del módulo por capas 37
Figura 2: Estructura del patrón arquitectónico MVC 39
Figura 3: Estructura de paquetes del módulo Vistas 41
Figura 4: Diagrama de paquetes del sistema 42
Figura 5: Diagrama de clases del diseño 44
Figura 6: Diagrama de secuencia (Crear vista) 45
Figura 7: Diagrama de clases persistentes (metamodelo) 46
Figura 8: Diagrama de componentes 51
Figura 9: Diagrama de despliegue 52
Figura 10: Resultados de la evaluación de la métrica TOC para los procedimientos 56
Figura 11: Resultados de la evaluación de la métrica TOC para el atributo Responsabilidad 57
Figura 12: Resultados de la evaluación de la métrica TOC para el atributo Reutilización 57
Figura 13: Resultados de la evaluación de la métrica TOC para el atributo Complejidad 58
Figura 14: Resultados de la evaluación de la métrica RC para las dependencias 59
Figura 15: Resultados de la evaluación de la métrica RC para el atributo Acoplamiento 59
Figura 16: Resultados de la evaluación de la métrica RC para el atributo Complejidad de Mantenimiento 60
Figura 17: Resultados de la evaluación de la métrica RC para el atributo Reutilización 60
Figura 18: Resultados de la evaluación de la métrica RC para el atributo Cantidad de Pruebas 61
Figura 19: Resultados obtenidos de la evaluación de los atributos de calidad 62
Figura 20: Código fuente de la funcionalidad Eliminar Fila 65
Figura 21: Grafo de flujo asociado a la funcionalidad Eliminar Fila 66
Figura 22: Entrada para el camino básico #1 68
Figura 23: Entrada para el camino básico #2 69
Figura 24: Resultados esperados para el camino básico #2 70
Figura 25: Entrada para el camino básico #3 71
Figura 26: Resultados esperados para el camino básico #3 72
Figura 27: Entrada para el camino básico #4 73
Figura 28: Resultados esperados para el camino básico #4 74

INDICE DE TABLAS

Tabla 1: Atributos de calidad evaluados por la métrica TOC.....53
Tabla 2: Criterios de evaluación para la métrica TOC.....53
Tabla 3: Atributos de calidad evaluados por la métrica RC.....54
Tabla 4: Criterios de evaluación para la métrica RC.....55
Tabla 5: Instrumento de evaluación de la métrica TOC.....55
Tabla 6: Instrumento de evaluación de la métrica RC.....58
Tabla 7: Resultados de la evaluación de la relación atributo/métrica.....61
Tabla 8: Rango de valores para la evaluación de la relación atributo/métrica.....62

INTRODUCCIÓN

Según la *Real Academia Española* (RAE), la informática no es más que el conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores.

Este campo de estudio, investigación y trabajo comprende el uso de la computación para resolver problemas mediante programas, diseño, fundamentos teóricos científicos y diversas técnicas. En la actualidad, la informática se usa en todos los trabajos y en casi todos los ambientes por los que nos movemos. Sin duda la mayor ventaja que nos ha ofrecido es la facilidad de comunicación y el poder compartir toda la información que queramos con todo el mundo. Es considerada como el soporte de los conocimientos y las comunicaciones.

La interfaz de usuario es uno de los medios por el cual el usuario puede comunicarse con una máquina, un equipo o una computadora y comprende todos los puntos de contacto entre el usuario y el equipo, normalmente suelen ser fáciles de entender y fáciles de accionar. Una interfaz de software es la parte de una aplicación que el usuario ve y con la cual interactúa. Está relacionada con la subyacente estructura, la arquitectura y el código que hace el trabajo del software, pero no se confunde con ellos. (1)

Los programas son utilizados por usuarios con distintos niveles de conocimientos, desde principiantes hasta expertos, por lo que no existe una interfaz válida para todos los usuarios y todas las tareas. Existen tres puntos de vista distintos en la creación de una Interfaz: el del *usuario*, el del *programador* y el del *diseñador*. Cada uno tiene un modelo mental propio de la interfaz, que contiene los conceptos y expectativas acerca de la misma, desarrollados a través de su experiencia:

- **Modelo del usuario**: el usuario tiene su visión personal del sistema, y espera que éste se comporte de una cierta forma.
- **Modelo del diseñador**: el diseñador mezcla las necesidades, ideas, deseos del usuario y los materiales de que dispone el programador para diseñar un producto de software. Es un intermediario entre ambos.

- **Modelo del programador**: es el más fácil de visualizar, al poderse especificar formalmente. Está constituido por los objetos que manipula el programador, distintos de los que trata el usuario (ejemplo: el programador llama base de datos a lo que el usuario podría llamar agenda). Los conocimientos del programador incluyen la plataforma de desarrollo, el sistema operativo, las herramientas de desarrollo y especificaciones. Sin embargo, esto no significa necesariamente que tenga la habilidad de proporcionar al usuario los modelos más adecuados. (2)

Estos modelos deben estar claros para los participantes en el desarrollo de un producto, de forma que se consiga una interfaz atractiva y a la vez efectiva para el trabajo con el programa. Una interfaz no es simplemente una cara bonita; esto puede impresionar a primera vista pero decepcionar a la larga. Lo importante es que el programa se adapte bien al modelo del usuario, cosa que se puede comprobar utilizando el programa más allá de la primera impresión.

Las interfaces permiten que el usuario/cliente interactúe con el software. Estas posibilitan enviar datos al servidor mediante la utilización de los formularios y mostrar los datos de cualquier entidad consultada en el sistema. Actualmente estas se crean manualmente ya que las tecnologías de generación existentes no producen una interfaz que cumpla totalmente con los requisitos deseados por los desarrolladores y los cambios manuales realizados en una interfaz son difíciles de conservar en la regeneración de la misma.

Teniendo en cuenta lo anterior, en el Departamento de Soluciones Financieras del centro CEIGE se decidió realizar un sistema, de nombre Génesis, que permita llevar a cabo un desarrollo dirigido por modelos (MDD[1]) que permita crear un modelo único y flexible, donde se gestione el modelo de interfaz de las entidades del sistema. El modelo resultante sería independiente de la plataforma o arquitectura que se utilice para el desarrollo de la aplicación final.

Dada la situación problemática anterior, podemos identificar el siguiente **problema a resolver**: ¿cómo lograr la gestión de modelos de interfaces teniendo en cuenta los requisitos propuestos en Génesis de forma que contribuya a agilizar el desarrollo de software?

[1] *Model Driven Development (Desarrollo Dirigido por Modelos)*

El **objeto de estudio** del presente trabajo se enfocará hacia las herramientas de gestión de modelos de interfaces y el **campo de acción** hacia la gestión de modelos de interfaces en el plugin[2] Génesis.

Para responder al problema anteriormente planteado, el autor se trazó como **objetivo general:** desarrollar el módulo Interfaces del plugin Génesis para el entorno de desarrollo Eclipse.

Para cumplir lo planteado, se persiguen los siguientes **objetivos específicos**:

1. Caracterizar algunos sistemas que implementan un desarrollo dirigido por modelos.
2. Realizar el modelo de diseño y el modelo de componentes del módulo Interfaces del plugins Génesis.
3. Implementar el módulo Interfaces del plugin Génesis.
4. Realizar las pruebas unitarias a la implementación del módulo Interfaces del plugin Génesis.

Como **posibles resultados:** obtener el módulo Interfaces del plugin Génesis para el entorno de desarrollo Eclipse.

La **idea a defender** queda planteada de la siguiente manera: si se logra la gestión de los modelos de interfaces teniendo en cuenta los requisitos propuestos en Génesis, esto reducirá el tiempo invertido por los desarrolladores actualmente en la implementación y acabado de las interfaces de los diferentes módulos de una aplicación web, contribuyendo a agilizar el desarrollo de software.

Como **métodos científicos de investigación** utilizados para la obtención del diseño e implementación del módulo, están:

Métodos teóricos:

1. Análisis-Síntesis: extracción de elementos relacionados con el objeto de estudio, a partir del análisis de documentos y diferentes herramientas para la gestión de modelos de interfaces.

[2] *Programa que interactúa con una aplicación principal para proveerle de cierta función normalmente muy específica.*

2. Histórico-Lógico: para conocer los antecedentes y tendencias actuales en lo relacionado a la gestión de modelos de interfaces.

3. Modelación: se propone la solución del Sistema.

Métodos empíricos:

1. Observación: utilizada en el proceso de diagnóstico del estado actual del objeto de estudio.

Para lograr la comprensión y claridad de los contenidos de la investigación realizada se ha estructurado el documento en tres capítulos. En el Capítulo 1 se realiza la fundamentación teórica de la investigación, la cual incluye un estudio de la metodología que guiará el proceso de desarrollo del software, las herramientas a utilizar y el estado del arte del tema relacionado a la creación de modelos de interfaces durante la implementación de los diferentes módulos de una aplicación web. El Capítulo 2 constituye la propuesta de solución, en el mismo se exponen las características del sistema a desarrollar además de los artefactos generados durante el diseño del mismo, así como se construirá el metamodelo de las interfaces. Mientras que en el Capítulo 3 se exponen los artefactos generados durante la implementación de la solución así como las métricas y pruebas utilizadas para la validación de la misma.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

1.1 Introducción

El presente capítulo define el conjunto de actividades a llevar a cabo para la realización del diseño y la posterior implementación del módulo Interfaces del plugin Génesis. Se incluye además, un estudio relacionado con los diferentes sistemas existentes en la actualidad para la gestión de modelos de interfaces, así como las herramientas a utilizar con el fin de lograr de forma productiva los objetivos propuestos.

1.2 Marco Conceptual

En este epígrafe se abordan los conceptos básicos relacionados con el dominio del problema.

1.2.1 Proceso de desarrollo de software

Un proceso de desarrollo de software consiste en organizar las actividades relacionadas con la creación, presentación y mantenimiento de los sistemas de software. Depende de las habilidades del personal, de la naturaleza del problema y de las herramientas, entre otros factores.

Los pasos principales de la presentación de una aplicación son los siguientes:

- ✓ *Planificación y elaboración*: planificar, definir los requerimientos, construir prototipos, entre otras actividades.
- ✓ *Construcción*: la creación del sistema.
- ✓ *Aplicación*: la transición de la implementación del sistema a su uso.

Tras una fase preliminar de planificación y especificación, el desarrollo pasa a la fase de construcción a través de una serie de ciclos de desarrollo. En cada ciclo se aborda un conjunto relativamente pequeño de requerimientos, pasando por el análisis, el diseño, la construcción y las pruebas. Una estrategia muy útil consiste en limitar el ciclo de desarrollo a un marco temporal, un lapso rígidamente fijo. Para tener

éxito con un programa de duración fija es necesario escoger los requerimientos con mucho cuidado y asignarle la selección al equipo de desarrollo. (3)

Ingeniería de software (IS)

La creación de software es un proceso intrínsecamente creativo y la Ingeniería del Software trata de sistematizar este proceso con el fin de minimizar el riesgo del fracaso en la consecución del objetivo creativo por medio de diversas técnicas. Esta ingeniería trata con áreas muy diversas de la informática y de las ciencias de la computación, abordando todas las fases del ciclo de vida del desarrollo de cualquier tipo de sistemas de información y siendo aplicable a infinidad de áreas: negocios, investigación científica, medicina, producción, logística, banca, control de tráfico, meteorología, Internet, etc. Una de las metas de IS es reducir costes del desarrollo de software y de su mantenimiento.

En la actualidad en la industria de software hay tendencia al crecimiento del volumen y complejidad de los productos, los proyectos están excesivamente tardes, se exige mayor calidad y productividad en menos tiempo y hay insuficiente personal calificado (4); por lo que se puede decir que las fallas de los proyectos de software se deben a:

- Planificación irreal.
- Mala calidad del trabajo.
- Personal inadecuado.
- Cambios no controlados.

Para enfrentar esta situación las empresas requieren desarrollar o adquirir una disciplina en el desarrollo del software y controlar que los ingenieros usen de forma consistente los nuevos métodos. La industria del software debe asentar cada vez más sus bases en la ingeniería de software, con el objetivo de que sus productos se realicen con calidad y que respondan a los requerimientos definidos.

1.2.2 Modelo y Metamodelo

Un modelo es una representación de un objeto, sistema o idea, de forma diferente al de la entidad misma. El propósito de los modelos es ayudarnos a explicar, entender o mejorar un sistema. Un requerimiento básico para cualquier modelo, es que debe describir al sistema con suficiente detalle para hacer predicciones válidas sobre el comportamiento del sistema. Al realizar un análisis de sistemas, se crea un modelo del sistema que muestre las entidades e interrelaciones.

Un modelo de un sistema es una descripción o una especificación de ese sistema y su entorno para desempeñar un determinado objetivo. Los modelos se presentan normalmente como una combinación de texto y dibujos. El texto se puede presentar en lenguaje de modelado, o en lenguaje natural.

Un modelo es la representación concisa de una situación, por eso representa un medio de comunicación más eficiente y efectivo.

Un metamodelo de un lenguaje es una definición precisa de sus elementos mediante conceptos y reglas de cierto metalenguaje, necesaria para crear modelos en ese lenguaje. Se trata de usar modelos para describir otros modelos.

Los metamodelos, en los sistemas de información, son modelos de datos de referencia que establecen parámetros del comportamiento de otros modelos de datos -modelos de aplicación. Son complementados por procesos y/o restricciones que validan que estos parámetros no sean violados cuando se creen, modifiquen o eliminen datos del modelo de aplicación. (5)

El metamodelado es un mecanismo que permite definir formalmente lenguajes de modelado (6), de forma que su definición no sea ambigua. Esta no ambigüedad permite que una herramienta pueda leer, escribir y entender modelos como los construidos con UML[3].

[3] *Unified Modeling Language (Lenguaje Unificado de Modelado)*

1.2.3 Desarrollo de software dirigido por modelos

En las aplicaciones de gestión de entidades, un alto por ciento de los módulos presenta el patrón CRUD[4] en parte o en su totalidad, el cual se refiere a operaciones que permiten crear, consultar, actualizar y eliminar una entidad. Esto implica la existencia de similitudes entre los distintos módulos, donde solo cambia la entidad que se gestiona. Para desarrollar estos módulos de una manera más rápida y sencilla, se propone crear modelos que recojan toda la información necesaria sobre el sistema. Esta propuesta contribuiría a acortar el tiempo de desarrollo y a aumentar las ganancias.

MDE

La Ingeniería Dirigida por Modelos, Model Driven Engineers (MDE), es una metodología de desarrollo de software, que se centra en la creación de modelos o abstracciones. El uso de modelos permite aumentar el nivel de abstracción con que se realizan los sistemas.

MDA

La Arquitectura Dirigida por Modelos, Model Driven Architecture (MDA), se ha concebido para dar soporte a la ingeniería dirigida por modelos de los sistemas de software. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos. Esta considera diferentes tipos de modelos, en función del nivel de abstracción de los mismos: en el Modelo Independiente de Computación (Computation Independent Model, CIM) se detallan los requisitos del sistema; en los Modelos Independientes de Plataforma (Platform Independent Model, PIM) se representa la funcionalidad del sistema sin considerar la plataforma final; y los Modelos Específicos de Plataforma (Platform Specific Model, PSM) se obtienen al combinar las especificaciones contenidas en el PIM con los detalles de la plataforma seleccionada. A partir de los diferentes PSMs se pueden generar automáticamente distintas implementaciones del sistema. (7)

[4] *Acrónimo en inglés de Create, Retireve, Update y Delete*

MDA es un acercamiento al desarrollo de sistemas, que potencia el uso de modelos en el desarrollo. Se dice que MDA es dirigido por modelos porque usa los modelos para dirigir el ámbito del desarrollo, el diseño, la construcción, el despliegue, la operación, el mantenimiento y la modificación de los sistemas.

En MDA es crucial analizar, automatizar y transformar modelos, es decir, mantener trazas y relaciones entre diferentes modelos y lograr interoperabilidad en diferentes niveles, haciendo descripciones precisas de la semántica de los modelos y metamodelos.

Los estándares principales de MDA (UML, MOF, XMI y CWM)[5] son la base para construir esquemas coherentes para crear, publicar y administrar modelos en una arquitectura dirigida por modelos, sin importar el tipo de sistema que se va a construir.

Estándares funcionales MDA

- UML: empleado para la definición de los modelos independientes de la plataforma y los modelos específicos de las plataformas de destino. Es un estándar para el modelado introducido por el OMG[6].

- MOF: establece un marco común de trabajo para las especificaciones del OMG, a la vez que provee de un repositorio de modelos y metamodelos.

- XMI: define una traza que permite transformar modelos UML en XML[7] para poder ser tratados automáticamente por otras aplicaciones.

- CWM: define la transformación de los modelos de datos en el modelo de negocio a los esquemas de base de datos.

---

[5] *UML* (*Unified Modeling Language)*
MOF (Meta Object Facility)
XMI (XML Metadata Interchange (XML de Intercambio de Metadatos))
CWM (Common Warehouse Metamodel)

[6] *Object Management Group (Grupo de Gestión de Objetos)*

[7] *Siglas en inglés de eXtensible Markup Language (Lenguaje de marcas Extensible)*

MDD

El Desarrollo Dirigido por Modelo, Model Driven Development (MDD), considera como principales elementos del proceso de desarrollo a los modelos, es decir establece que solo el modelo de una aplicación se ha de desarrollar, y que el resto de la aplicación se generará desde este modelo. (8)

Para poder integrar sistemas de software con tecnologías diversas y que puedan adaptarse rápidamente a los negocios cambiantes, la tendencia es apostar al desarrollo dirigido por modelos donde los artefactos principales del desarrollo son los modelos (no los programas) y la transformación de modelos. MDD implica una generación semiautomática de la implementación a partir de los modelos.

Bajo este enfoque, un experto del negocio puede expresar su conocimiento en un lenguaje formal de modelado y el equipo de tecnologías de la información define cómo se implementará y si se desea, el mismo modelo puede implementarse en diversas plataformas. Un cambio en los modelos independientes de plataforma (representan el conocimiento del negocio) puede generar diversos modelos dependientes de plataforma sin tanto esfuerzo.

El principio básico detrás del desarrollo basado en modelos es que si sabemos que algo va a cambiar en una forma predecible, modelemos la descripción del cambio de tal forma que sea fácil modificarlo. En otras palabras: si algo va a cambiar mucho, hazlo fácil de cambiar.

El Desarrollo Dirigido por Modelos (MDD) se ha convertido en un nuevo paradigma de desarrollo software que promete una mejora de la productividad y de la calidad del software a través de un proceso guiado por modelos y soportado por potentes herramientas que generan código a partir de modelos.

La idea fundamental de MDD es sustituir al código de lenguajes de programación específicos por modelos. De este modo y en el contexto de este paradigma, los modelos son considerados como entidades de primera clase, permitiendo nuevas posibilidades de crear, analizar y manipular sistemas a través de diversos tipos de herramientas y de lenguajes. Cada modelo trata generalmente un aspecto, y las transformaciones entre modelos proporcionan un vínculo que permite la implementación automatizada de un sistema a partir de sus correspondientes modelos, generando también modelos, por lo tanto constituyen una parte integral de esta propuesta basada en modelos. Estas transformaciones requieren

soporte especializado en varios aspectos para que modeladores de sistemas, desarrolladores de transformaciones y desarrolladores de herramientas puedan aplicarla en su máximo potencial.

En general el término Desarrollo de Software Dirigido por Modelos es utilizado para englobar a los diferentes paradigmas de desarrollo de software que tienen en común el uso de modelos para representar diferentes aspectos de un sistema de software con el fin de elevar el nivel de abstracción y de automatización. Los conceptos teóricos que sustentan esta nueva visión son: metamodelado, transformaciones de modelos y Lenguajes Específicos del Dominio (DSL). Un DSL consta de tres elementos: la sintaxis abstracta que define los conceptos del lenguaje y las relaciones entre ellos, la sintaxis concreta que provee de una notación textual o gráfica y la semántica del lenguaje que normalmente es definida a través de la traducción a conceptos de otro lenguaje destino cuya semántica se conoce (por ejemplo, Java) (9). El término DSL se está imponiendo al de lenguaje de modelado, aunque éste último se usa con frecuencia cuando se trata de DSL gráficos o visuales. En MDD, la sintaxis abstracta de un DSL se define mediante un metamodelo.

A través de la creación de modelos y metamodelos es posible expresar las soluciones de software a un nivel de abstracción más alto. Los modelos han sido usados desde hace décadas como forma de documentación o de razonar sobre el código, pero en MDD se han convertido en el principal artefacto del desarrollo de software, de modo que ahora son parte de la solución, de la misma forma que las clases o los ficheros de configuración de una aplicación.

La otra gran ventaja de MDD es el aumento del grado de automatización mediante la generación automática de código a partir de los modelos creados, ya sea directamente a través de transformaciones modelo a texto (m2t) o de forma indirecta mediante la definición de modelos intermedios y con transformaciones modelo a modelo (m2m) que facilitan el paso de los modelos de alto nivel de abstracción al código final. Un metamodelo también puede representar la estructura de cierta información (metadatos) relacionada con una aplicación. En este caso, un metamodelo no llevaría asociado una notación sino que los modelos se crearían programáticamente, por ejemplo, mediante una transformación de modelos. (Ver Anexo # 1)

Durante las dos últimas décadas, se ha venido desarrollando una intensa actividad investigadora e industrial orientada al objetivo de generar código desde especificaciones de alto nivel de un sistema. El

paradigma MDA/MDD ha dado un nuevo empuje a esta estrategia. Nuevos métodos plantean transformaciones de modelos que, desde una perspectiva ingenieril, cubren las diferentes etapas de un proceso de producción de software. Dichas transformaciones recogen conocimientos de múltiples áreas, como Modelado Organizacional, Ingeniería de Requisitos, Modelado Conceptual y técnicas de Generación de Código basadas en Modelos. La definición de las primitivas conceptuales utilizadas, junto con las reglas de conversión entre modelos y su correspondiente software, debe ser realizada formalmente, posibilitando incluso la completa automatización del proceso de conversión a través del uso de Compiladores de Modelos. (10)

1.3 Sistemas de desarrollo de software dirigido por modelos

La existencia de una metodología y de un marco de trabajo que guíen el proceso de creación de modelos, ha contribuido al surgimiento de un conjunto de herramientas que crean aplicaciones a partir de la definición de modelos.

AndroMDA

Sigue el paradigma de arquitectura MDA. AndroMDA es de código abierto, pero los modelos utilizados son los obtenidos utilizando UML (11), no posee herramienta de modelado, por lo que no es posible registrar toda la información de un sistema, como los aspectos visuales, las reglas de validación, las expresiones booleanas, los orígenes de datos, entre otras necesidades, por lo que esta herramienta no soportaría todos los requisitos necesarios. Admite cualquier lenguaje de programación como salida, y admite código propio para la generación de código. Posee como principal virtud un esquema de transformación basado en cartuchos; sin embargo, los cartuchos disponibles obligan a estereotipar los modelos de forma demasiado dependiente de los componentes a generar, con lo que limita la reutilización de dichos modelos.

AndroMDA elimina la posibilidad de tener código redundante, el modelo de la aplicación va a reflejar al código, el proyecto es documentado/diagramado de una forma en una plataforma estándar independiente lo que hace más fácil adaptar la aplicación al cambio acelerado de tecnologías, además produce artefactos (modelos) que permitirán de una forma más fácil la comunicación con el cliente.

Genexus

GeneXus es una herramienta inteligente cuyo objetivo es asistir al analista y a los usuarios en todo el ciclo de vida de las aplicaciones. El repositorio de GeneXus mantiene las especificaciones de diseño en forma abstracta, o sea que no depende del ambiente objeto, lo que permite que, a partir del mismo repositorio, se puedan generar aplicaciones funcionalmente equivalentes, para ser ejecutadas en diferentes plataformas.

GeneXus está pensado para escribir menos y declarar más, que junto a la posibilidad de reutilizar código generado, la mayor automatización, usabilidad e integración, hacen que el desarrollo de aplicaciones sea más simple, efectivo y productivo. Reduce considerablemente la complejidad de construir aplicaciones de tipo web 2.0, logrando una mayor calidad de producto en menos tiempo de desarrollo. Está diseñado para facilitar el trabajo en equipo. Permite un mayor grado de colaboración en la documentación, integrar el conocimiento de cada desarrollador y la posibilidad de que convivan diferentes visiones de desarrollo.

El desarrollo con GeneXus permite que una aplicación pueda ser dividida para ser ejecutada en diferentes plataformas y generada, para cada una, en el lenguaje más adecuado, obteniendo así arquitecturas de múltiples partes y que hagan un mejor uso de los recursos disponibles.

GeneXus ha avanzado mucho en el tema de la generación de aplicaciones, pero utiliza otro enfoque para sus modelos, aunque siguiendo el paradigma MDD, además es una aplicación comercial que no ofrece las posibilidades del software libre (12).

OptimalJ

Implementa completamente la especificación MDA. Está desarrollado en Java, lo que le hace portable a cualquier plataforma para su ejecución. Dispone de plugins para los entornos de desarrollo integrado Eclipse y NetBeans. Admite XMI tanto para la importación de ficheros como para su salida. La principal desventaja de esta herramienta es que está pensada específicamente para el desarrollo de aplicaciones J2EE[8]. (13)

[8] *Java 2 Platform, Enterprise Edition o J2EE (Plataforma Java, Edición Empresarial)*

Codagen Architect

Permite la transformación de modelos y la automatización de la generación de código partiendo de los modelos CIM. Como integración con herramientas de modelado, Codagen incorpora plugins para Microsoft Visio, Rational Rose y Borland Together Control. Admite como entrada ficheros XMI y ficheros MDL de Rational Rose. Entre los estándares libres soportados tenemos únicamente Struts, pero como contrapartida, los lenguajes de salida que admite son Java, C#, C++ y Visual Basic, admitiendo también la incorporación de código propio al código generado. (13)

IQGen

El generador de modelos y código IQGen, de la empresa InnoQ está desarrollado en Java. Incorpora un entorno de modelado muy pobre, que se suple con la admisión de entrada de modelos en formato XMI. Puede generar salida en cualquier lenguaje (J2EE/EJB, C#, Cobol, entre otros). (13)

ArcStyler

Es un sistema basado en uso de cartuchos para descripción de transformaciones que permite generar aplicaciones de n capas codificadas en java/J2EE y c#/.NET a partir de diagramas UML y la especificación de los procesos del negocio. Permite extender las capacidades de transformación, generando nuevos cartuchos a partir de UML, cuyo objetivo sea cualquier plataforma o lenguaje. Usa Rose como herramienta de modelado UML, soportando directamente los ficheros MDL. No soporta diagramas de componentes ni diagramas de despliegue, pero admite código propio para la generación de código. Presenta dificultad para llevar a cabo la integración de modelos. (13)

El estudio de estos sistemas demuestra la necesidad de una nueva herramienta para la definición de modelos, independientemente de la plataforma o arquitectura que se utilice para el desarrollo de la aplicación final; que evite además las desventajas existentes en las herramientas mencionadas anteriormente, siendo esto de gran ayuda para los desarrolladores y facilitándoles su trabajo, ahorrando además gran parte del tiempo que invierten actualmente en la creación de una interfaz.

Con la creación de esta herramienta el país no tendría que comprar software propietario a países extranjeros para desarrollar una aplicación para la gestión de modelos de interfaces, logrando de esta forma un grado de independencia tecnológica.

1.4 Tecnologías y Herramientas

Las tecnologías, metodologías y herramientas utilizadas en el desarrollo del módulo fueron definidas a partir de las políticas establecidas por el Departamento de Soluciones Financieras del centro CEIGE, por tal motivo en este fragmento del capítulo no se definen dichas herramientas, sino que se realiza una resumida caracterización de cada una de ellas y de los beneficios que aportan en el trabajo de diseño e implementación del módulo implicado.

1.4.1 eXtensible Markup Language (XML)

XML, siglas en inglés de eXtensible Markup Language (Lenguaje de Marcas Extensible), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Su función principal es describir datos y no mostrarlos.

Las tecnologías XML son un conjunto de módulos que ofrecen servicios útiles a las demandas más frecuentes por parte de los usuarios. XML sirve para estructurar, almacenar e intercambiar información. Permite al programador y los soportes dedicar sus esfuerzos a las tareas importantes cuando trabaja con los datos, ya que tareas como la validación de estos o el recorrido de las estructuras corre a cargo del lenguaje y está especificada por el estándar, de modo que el programador no tiene que preocuparse por ello.

XML tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

1.4.2 Lenguaje de modelado

Un lenguaje de modelado de objetos se puede definir como el conjunto estandarizado de símbolos y de modos de disponerlos para modelar un diseño de software, o parte de este, orientado a objetos.

Unified Modeling Language (UML)

UML es un lenguaje, que permite modelar analizar y diseñar sistemas orientados a objetos. Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad (14). Ofrece un estándar para describir un panorama del sistema modelo, incluyendo aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables y aspectos conceptuales como los procesos de negocios y funciones del sistema.

UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema que involucra una gran cantidad de software. UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. La finalidad de los diagramas es presentar diversas perspectivas de un sistema, a las cuales se les conoce como modelo. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos. Es importante recalcar que UML no es una guía para realizar el análisis y diseño orientado a objetos, es decir, no es un proceso. UML es un lenguaje que permite la modelación de sistemas con tecnología orientada a objetos.

1.4.3 Metodologías

Las metodologías imponen un proceso disciplinado sobre el desarrollo de software con el fin de hacerlo más predecible y eficiente. Lo hacen desarrollando un proceso detallado con un fuerte énfasis en planificar inspirado por otras disciplinas de la ingeniería.

Rational Unified Process (RUP)

El proceso unificado de desarrollo (RUP) es una metodología para la ingeniería de software que va más allá del mero análisis y diseño orientado a objetos para proporcionar una familia de técnicas que soportan el ciclo completo de desarrollo de software. El resultado es un proceso basado en componentes, dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

Su principal objetivo es desarrollar software de alta calidad y que cumpla con los requisitos de los usuarios dentro de una planificación y presupuestos definidos. Actualmente es una de las metodologías de desarrollo de software más exitosa, según la Organización Internacional de Estándar (ISO siglas en inglés).

RUP define cuatro fases para el desarrollo de un software:

- ✓ Inicio: se define y delimita el proyecto.
- ✓ Elaboración: su objetivo es obtener la arquitectura candidata del sistema.
- ✓ Construcción: obtención del producto, apoyado en una fuerte documentación.
- ✓ Transición: se alcanza el release del producto, listo para su instalación.

RUP también delimita el proceso de desarrollo en flujos de trabajo que se agrupan en dos:

De Desarrollo:

- ✓ Modelamiento del negocio: se definen y describen los procesos del negocio.
- ✓ Requerimientos: delimitación de los requisitos funcionales y no funcionales del proyecto.
- ✓ Análisis y diseño: se obtiene la guía, basada en los requerimientos, de lo que se debe programar.
- ✓ Implementación: define todos los aspectos relacionados con la implementación del sistema.
- ✓ Prueba (Testeo): busca deficiencias en el producto.

De Soporte:

- ✓ Administración del proyecto: definición de cómo se organizará todo el trabajo dentro del equipo de desarrollo.
- ✓ Administración de configuración y cambios: se conservan todos los elementos producidos, para analizar cuál o cuáles utilizar.
- ✓ Ambiente: se determinan los procesos, herramientas y procedimientos que guiarán al equipo de desarrollo. (15)

1.4.4 Herramienta CASE

El acrónimo CASE en inglés significa Computer Aided Software Engineering y según su traducción en español es Ingeniería de Software Asistida por Computadora. Las herramientas CASE es un sistema de software que intenta proporcionar ayuda automatizada a las actividades del proceso de software. Son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software.

Visual Paradigm

Es una herramienta UML profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. El software de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. La herramienta UML CASE también proporciona abundantes tutoriales de UML, demostraciones interactivas de UML y proyectos UML. (16)

1.4.5 Herramienta Ecore

EMF[9] es un marco de trabajo de Eclipse que unifica Java, XML y UML, permitiendo a los desarrolladores construir rápidamente aplicaciones robustas basadas en modelos simples. El metamodelo usado para representar modelos en EMF se llama Ecore. Ecore es en sí mismo, un metamodelo EMF, y así es su propio metamodelo, es decir, un metametamodelo. (17)

Ecore es un subconjunto pequeño y simplificado de UML (se utiliza un subconjunto de UML en vez del mismo porque éste es demasiado ambicioso modelando, más de lo que el núcleo de EMF puede soportar).

1.4.6 Lenguaje de programación

Java

Java es un lenguaje de desarrollo orientado a objetos, multiplataforma. Proporciona una colección de clases para su uso en aplicaciones de red, que permiten abrir sockets y establecer y aceptar conexiones con servidores o clientes remotos, facilitando así la creación de aplicaciones distribuidas. Es un lenguaje interpretado y compilado, esto quiere decir que su código fuente se transforma en algo similar al código máquina, los bytecodes. A su vez estos bytecodes se pueden ejecutar directamente sobre cualquier máquina a la cual se hayan portado el intérprete y el sistema de ejecución en tiempo real (run-time).

[9] *Siglas del inglés Eclipse Modelling Framework (Plataforma de Modelado de Eclipse)*

Además, es considerado un lenguaje robusto, fiable y seguro, para ello proporciona numerosas comprobaciones en compilación y en tiempo de ejecución.

1.4.7 Entorno de Desarrollo Integrado (IDE)

Eclipse

Eclipse es un entorno de desarrollo integrado (IDE) de código abierto independiente de una plataforma, es una aplicación de cliente enriquecido, emplea plugins para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Eclipse es, en el fondo, únicamente un armazón sobre la que se pueden montar herramientas de desarrollo. La arquitectura de plug-ins permite, además de integrar diversos lenguajes, introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo, tales como: herramientas UML, editores visuales de interfaces, ayuda en línea para librerías, entre otros.

SWT: Standard Widget Toolkit[10]

SWT es un conjunto de componentes para construir interfaces gráficas en Java, (widgets[11]) desarrollados por el proyecto Eclipse. Recupera la idea original de la biblioteca AWT[12] de utilizar componentes nativos, con lo que adopta un estilo más consistente en todas las plataformas.

SWT es un framework que crea a través de JNI (Java Native Interface) interfaces gráficas nativas del Sistema Operativo en donde ejecutemos nuestra aplicación SWT. Esto quiere decir que con el mismo código visualizaremos en cada Sistema Operativo (SO) nuestras ventanas como si hubieran sido creadas para ese SO en específico.

La interfaz del workbench (banco de trabajo) de Eclipse (vistas, editores, perspectivas, asistentes) depende de una capa intermedia de Interfaz Gráfica de Usuario (GUI) llamada JFace que simplifica la construcción de aplicaciones basadas en SWT. (18)

---

[10] *Toolkit: Conjunto de herramientas.*

[11] *Un widget es un objeto de la interfaz gráfica de usuario, responsable de la interacción con el usuario.*

[12] *El AWT (Abstract Windows Toolkit) es la parte de Java que se ocupa de construir interfaces gráficas del usuario.*

JFace

Es una librería que usa SWT para facilitar ciertas labores en la creación de GUIs o incluir nuevos widgets de más alto nivel. Básicamente añade funcionalidad a la librería básica de SWT. Es independiente del SO. Sin embargo SWT no depende de ningún código JFace o plataforma en absoluto.

JFace está diseñado para proporcionar la funcionalidad de interfaz de usuario de aplicación común en la parte superior de la biblioteca SWT. JFace no intenta "ocultar" SWT o reemplazar su función. Proporciona clases e interfaces que se encargan de muchas de las tareas comunes asociadas con la programación de una dinámica interfaz de usuario mediante SWT. (18)

JDT

Java Develop Tooling[13] (JDT) es uno de los principales subproyectos de Eclipse y es el que permite que la plataforma Eclipse se transforme en un potente IDE de Java, uno de los más reconocidos del mundo. Es el conjunto de plugin más avanzado y elaborado que posee Eclipse. El JDT asiste a los usuarios en los procesos de escribir, compilar, probar, depurar y editar programas escritos en el lenguaje de programación Java, incluyendo los plugins de Eclipse. El JDT está estructurado en tres componentes principales que contienen los paquetes del API:

- ✓ *JDT Core*: define el núcleo de los elementos Java y provee clases útiles para manipular archivos de extensión .class y el modelo de elementos Java.
- ✓ *JDT UI*: contiene la interfaz de usuario que provee el IDE.
- ✓ *JDT Debug*: brinda un conjunto de clases que permiten correr y depurar código Java.

PDE

El Plugin Development Enviroment o PDE, no es más que el Ambiente de Desarrollo para Plugin que provee Eclipse y uno de los principales subproyectos del mismo, junto con JDT. Está compuesto por un conjunto de herramientas que cubren todo el ciclo de vida completo del desarrollo de un plugin. Facilita

[13] *Java Develop Tooling: Desarrollar Herramientas de Java.*

todo el proceso de crear, desarrollar, probar, depurar, construir y desplegar plugins de Eclipse. Algunas de estas herramientas de PDE incluyen:

- ✓ *Editores del Manifiesto*: ofrece editores multipáginas basados en formularios, que administran centralmente todos los ficheros del manifiesto del plugin, o sea, el MANIFEST.MF y/o el plugin.xml.
- ✓ *Asistentes para la creación de nuevos proyectos*: facilitan todo el proceso de creación de un proyecto plugin.
- ✓ *Asistentes para importar*: provee un conjunto de asistentes que simplifican el proceso de importar plugins del sistema de archivos o file system.
- ✓ *Asistentes para exportar*: provee un conjunto de asistentes que construyen, empaquetan y exportan plugins con un simple clic a un formato de despliegue.
- ✓ *Launchers (Probadores)*: permiten probar y depurar aplicaciones de Eclipse, así como paquetes OSGi.
- ✓ *Vistas:* PDE provee cuatro vistas y una perspectiva de desarrollo, denominada Plug-in Development, que ayudan a los desarrolladores de plugins a inspeccionar los diferentes aspectos de su ambiente de desarrollo.
- ✓ *Herramientas Combinadas*: posee asistentes para externalizar y purificar los ficheros del manifiesto del plugin.
- ✓ *Herramientas de Conversión*: consiste en asistentes para facilitar el proceso de convertir un proyecto simple de Java o un archivo JARs simple, en un proyecto plugin.
- ✓ *Integración con el JDT*: los ficheros del manifiesto del plugin participan en búsquedas Java y en la reestructuración del código. (19)

1.4.8 Subversion (SVN)

Es un software de sistema de control de versiones. Una característica importante de Subversion es que, los archivos versionados no tienen cada uno un número de revisión independiente. En cambio, todo el repositorio tiene un único número de versión que identifica un estado común de todos los archivos del repositorio en cierto punto del tiempo. Subversion puede acceder al repositorio a través de redes, lo que le permite ser usado por personas que se encuentran en distintos ordenadores. A cierto nivel, la posibilidad de que varias personas puedan modificar y administrar el mismo conjunto de datos desde sus respectivas ubicaciones fomenta la colaboración. Se puede progresar más rápidamente sin un único conducto por el cual deban pasar todas las modificaciones. Y puesto que el trabajo se encuentra bajo el control de versiones, no hay razón para temer por que la calidad del mismo vaya a verse afectada, si se ha hecho un cambio incorrecto a los datos, simplemente deshaga ese cambio. (20)

Existen varias interfaces a Subversion, ya sea programas individuales como interfaces que lo integran en entornos de desarrollo.

SubEclipse

Subclipse es un plugin que permite integrar Subversion al Eclipse. Simplifica mucho las cosas ya que evita tener que manejar los comandos por consola o usar un cliente de SVN aparte, y además provee herramientas para resolver problemas entre versiones o bien seleccionar distintas versiones de un mismo archivo para compararlas entre sí.

1.4.9 Patrones

Un patrón es una solución a un problema en un contexto dado, codifica conocimiento específico acumulado por la experiencia en un dominio. Recurrente, lo que hace que sea relevante para otras soluciones. Es un conjunto de información que proporciona respuesta a un conjunto de problemas similares

Patrones de diseño

Un patrón de diseño es una solución estándar para un problema común de programación, es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular. Los patrones de diseño proponen una forma de reutilizar la experiencia de los desarrolladores, para ello clasifican y describen formas de solucionar problemas que ocurren de forma frecuente en el desarrollo.

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas de software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente. (21)

Patrones GoF

Los patrones Gof son patrones de diseño publicados en el libro Design Patterns: Elements of Reusable Object-Oriented Software por Gamma, Helm, Jonson y Vlissides conocidos mundialmente por “Gang of Four” o Pandilla de los cuatro.

El grupo de GoF clasificó los patrones en tres grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento. (22)

- Creacionales: patrones creacionales tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.

- Estructurales: los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.

- Comportamiento: los patrones de comportamiento ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

Patrón Singleton

El Singleton es quizás el más sencillo de los patrones que se presentan en el catálogo del GoF. Es también uno de los patrones más conocidos y utilizados. Su propósito es asegurar que sólo exista una instancia de una clase. El patrón Singleton garantiza que una clase sólo tenga una instancia y proporciona un punto de acceso global a ésta instancia.

A primera vista, uno puede pensar que pueden utilizarse clases con miembros estáticos para el mismo fin. Sin embargo, los resultados no son los mismos, ya que en este caso la responsabilidad de tener una única instancia recae en el cliente de la clase. El patrón Singleton hace que la clase sea responsable de su única instancia, quitando así este problema a los clientes. (23)

Patrones de asignación de responsabilidades

Los Patrones Generales de Software para Asignar Responsabilidades (GRASP) describen los principios fundamentales de diseño de objetos para la asignación de responsabilidades. Constituyen un apoyo para la enseñanza que ayuda a entender el diseño de objeto esencial y aplica el razonamiento para el diseño de una forma sistemática, racional y explicable.

Existen nueve patrones GRASP los cuales son: Experto, Creador, Alta Cohesión, Bajo Acoplamiento, Controlador, Polimorfismo, Fabricación Pura, Indirección y No Hables con Extraños.

Experto

Es el principio básico de asignación de responsabilidades. Nos indica que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo. De este modo obtendremos un diseño con mayor cohesión y así la información se mantiene encapsulada.

Creador

Ayuda a identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases.

Alta Cohesión

La información que almacena una clase debe de ser coherente y debe estar (en la medida de lo posible) relacionada con la clase.

Bajo Acoplamiento

Es la idea de tener las clases lo menos ligadas entre sí que se pueda. De tal forma que en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de las clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases.

Controlador

Sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado. Este patrón sugiere que la lógica de negocios debe estar separada de la capa de presentación, esto para aumentar la reutilización de código y a la vez tener un mayor control.

Polimorfismo

Siempre que se tenga que llevar a cabo una responsabilidad que dependa del tipo, se tiene que hacer uso del polimorfismo, cuando las alternativas o comportamientos relacionados varían según el tipo (clase),

asigne la responsabilidad para el comportamiento utilizando operaciones polimórficas a los tipos para los que varia el comportamiento. Asigna el mismo nombre a servicios en diferentes objetos.

Fabricación Pura

La fabricación pura se da en las clases que no representan un ente u objeto real del dominio del problema, sino que se ha creado intencionadamente para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código. Es la solución cuando el diseñador se encuentre con una clase poco cohesiva y no tenga otra clase en la que implementar algunos métodos. Es decir que es una clase que no existe en el problema como tal, pero que añadiéndola se logra mejorar estructuralmente el sistema. Como contraindicación al abusar de este patrón suelen aparecer clases función o algoritmo (que tienen un solo método).

Indirección

El patrón de indirección nos aporta mejorar el bajo acoplamiento entre dos clases asignando la responsabilidad de la mediación entre ellos a un tercer elemento (clase) intermedio.

No Hables con Extraños

Se refiere a no obtener una visibilidad temporal frente a objetos indirectos, que son de conocimiento de otros objetos pero no del cliente. Evita el alto acoplamiento y además que un cliente tenga que conocer la estructura interna para acceder a algo o solicitar algo. (24)

1.5 Conclusiones parciales

En el presente capítulo se abordaron varios conceptos generales que servirán de guía para la realización de este trabajo.

- Se realizó una descripción sobre el proceso de desarrollo de software dirigido por modelos así como un estudio y análisis sobre algunas herramientas ya existentes, lo que atendiendo a algunas desventajas que las mismas presentan permitirá desarrollar una herramienta con más calidad.

- Se hizo una breve caracterización sobre las tecnologías, lenguajes, herramientas y metodología a utilizar, definidas por el proyecto, para el desarrollo de la solución propuesta.

CAPÍTULO 2: PROPUESTA DE SOLUCIÓN

2.1 Introducción

En el siguiente capítulo se recogen los resultados obtenidos durante el proceso de desarrollo de la solución, los artefactos generados durante el diseño del sistema a desarrollar y se construirá el metamodelo de las interfaces.

2.2 Importancia del diseño

El diseño es el proceso que consiste en traducir los requisitos a una representación del software que va a ser implementado. Debe servir de base para que los programadores implementen el sistema, proporcionar información necesaria para las personas que mantendrán el sistema en el futuro, debe servir de base para que los miembros del equipo elaboren los casos de prueba, así como proporcionar información sobre las necesidades de instalación del sistema. (25)

2.3 Modelado del sistema

La parte más difícil de construir un sistema es precisamente saber qué construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer los requerimientos técnicos detallados. Entonces, la tarea más importante que el ingeniero de software hace para el usuario/cliente es la extracción iterativa y el refinamiento de los requerimientos del producto. (26)

La determinación de los requisitos de software constituye la base para las etapas futuras de desarrollo, todos los integrantes del equipo a partir de aquí conocen cuales son las funcionalidades que el sistema deberá tener, teniendo en cuenta además las propiedades que el usuario/cliente desea para su sistema.

2.3.1 Descripción de los Casos de Uso (CU) del sistema

Un caso de uso es una técnica para la captura de requisitos potenciales de un nuevo sistema o una actualización de software. Cada caso de uso proporciona uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para conseguir un objetivo específico. Mediante la descripción de los casos de uso del sistema se detalla la secuencia de eventos que los actores llevan a cabo para completar un proceso a través de la aplicación.

Como artefacto de entrada al diseño del módulo se utilizaron los casos de uso. A continuación se muestra la descripción abreviada de los mismos junto a las reglas del negocio que deben cumplir, las descripciones extendidas están disponibles en el Anexo #2.

1- CU "Crear Componente Vista"

El sistema debe permitir crear una vista a una entidad del sistema. Para ello se muestra una interfaz con los datos iniciales necesarios que la identifican. La vista creada debe ser salvada y debe ser posible editar la misma en cualquier momento. El identificador de la vista debe listar todos los nombres de vistas que se han creado en el módulo y permite modificar o agregar uno nuevo, además debe ser único y se debe validar que este no exista dentro del módulo seleccionado. La vista debe ser almacenada en el mismo módulo que la entidad seleccionada.

2- CU "Adicionar Componente a Vista"

El sistema debe permitir adicionar componentes existentes a la vista que está siendo editada. Hay componentes de dos tipos: los comunes, que son los que son elementos externos a las entidades y los atributos de las entidades los que pueden formar parte de una interfaz, es decir al seleccionar la entidad a la que se le desea crear la vista, pueden ser utilizados sus atributos como posibles componentes para formar parte de la vista . El proceso de seleccionar el componente y entrar los datos de configuración del mismo es el objetivo de esta funcionalidad.

Para adicionar un componente tipo Fila el usuario debe seleccionar la fila por la que se va insertar, así como el tipo de inserción, Antes de selección o Después de selección. Para poder adicionar un componente en una fila hay que tener en cuenta el ancho del mismo que debe ser menor que el espacio disponible. Al adicionar un componente campo si su tipo de dato es primitivo y se selecciona una fuente predefinida entonces la fuente a seleccionar en las opciones del componente debe tener un tipo de retorno del mismo tipo que el atributo. Al adicionar un componente campo si está asociado con un tipo de relación de composición y se escoge el componente vista incluida entonces el ancho de la caja debe ser mayor o igual que el ancho de la vista.

3- CU "Configurar Visor de Opciones"

El sistema debe permitir configurar la visualización de objetos en un visor. Un visor de objetos es un componente que tiene la responsabilidad de mostrar una lista de objetos en diferentes estilos. Se definen dos estilos: el estilo tabla que permite mostrar a los objetos en una tabla donde las columnas serían los atributos o características del objeto y un segundo estilo de visualización que permite mostrar los objetos en pequeñas vistas miniaturas previamente confeccionadas. Debe haber al menos una columna por defecto a mostrar en el estilo de tabla.

4- CU "Editar Componente Vista"

El sistema debe permitir editar una vista del sistema. Para ello se muestra una interfaz con los datos necesarios y además se explican las reglas de validación que deben cumplirse. Al cargar la vista por primera vez se le debe incluir una fila inicial. Para adicionar un componente debe haber siempre una fila seleccionada o pivote. Para importar filas a la vista se debe tener seleccionada una fila pivote. Si la cantidad de filas en la vista es 1 no es posible eliminarla ya que siempre debe haber una fila en la vista. Si se elimina un componente dentro de una fila, se debe deben correr los demás componente hacia la izquierda o la derecha según la alineación. Si se elimina una fila las filas inferiores subirían a ocupar el espacio disponible.

Al eliminar un componente compuesto, entonces se deben eliminar todos los componentes. Para poder realizar un movimiento se debe tener seleccionado el componente que se desea mover. En caso específico de la opción Mover se debe además seleccionar la fila destino. Las filas que son referencias a otras vistas no pueden ser modificadas, lo único que se permite es subirlas o bajarlas pero todas en conjunto a la misma vez.

2.4 Arquitectura de la solución

La arquitectura de software es el diseño de más alto nivel de la estructura de un sistema, donde se establecen los fundamentos para que analistas, diseñadores, programadores, etc. trabajen en una línea común que permita alcanzar los objetivos del sistema de información, cubriendo todas las necesidades.

Se refiere a la abstracción en la representación de los componentes involucrados en un proyecto de software, esta se puede dividir en diversos niveles para facilitar su comprensión y aislar las funcionalidades.

El módulo está estructurado siguiendo un patrón de arquitectura por capas, el cual cuenta con 4 capas lógicas que dan un alto nivel de encapsulamiento de las responsabilidades, permitiendo reducir al máximo el acoplamiento y aumentar la reutilización entre las mismas. Esta distribución de las capas permite que se realicen grandes cambios sin tener que realizar cambios en las demás capas.

Los beneficios de trabajar un sistema en capas son (27):

- Se puede entender una capa como un todo, sin considerar las otras.
- Las capas se pueden sustituir con implementaciones alternativas de los mismos servicios básicos
- Se minimizan dependencias entre capas.
- Las capas posibilitan la estandarización de servicios
- Luego de tener una capa construida, puede ser utilizada por muchos servicios de mayor nivel.

2.4.1 Vista lógica del módulo Interfaces del plugin Génesis

El modelo de capas de una arquitectura organiza el sistema en capas, cada una de las cuales proporciona un conjunto de servicios. Este tipo de arquitectura soporta el desarrollo incremental de sistemas y evita que los cambios en una de las capas afecten directamente al resto, además se enfoca en la distribución de roles y responsabilidades de forma jerárquica proveyendo una forma muy efectiva de separación de responsabilidades. El rol indica el modo y tipo de interacción con otras capas, y la responsabilidad indica la funcionalidad que está siendo desarrollada. Para el desarrollo del sistema propuesto se combinan 4 capas esenciales, donde cada nivel se especializa en una actividad específica para evitar la dependencia a una sola tecnología o aplicación.

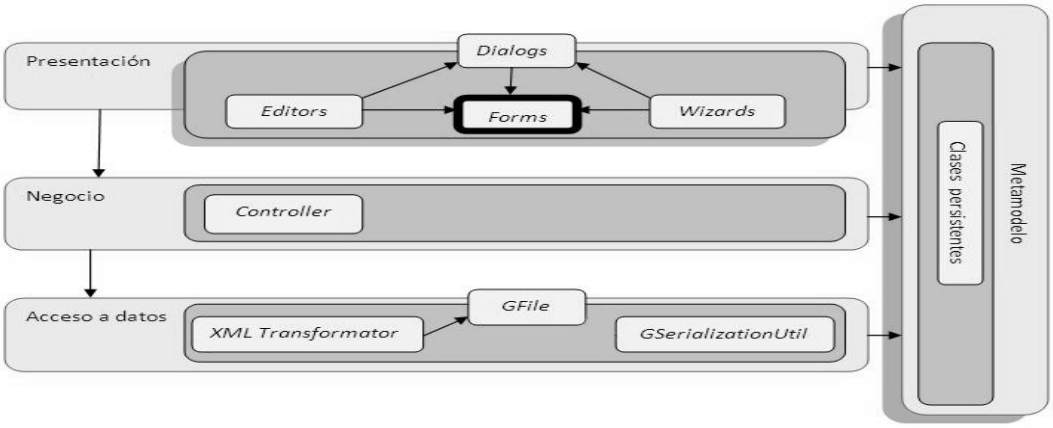


Figura 1: Estructura de la arquitectura del módulo por capas

Presentación

Referente a la interacción entre el usuario y el sistema. Su principal responsabilidad es mostrar información al usuario, interpretar los comandos de este y realizar algunas validaciones simples de los datos ingresados. Contiene los requerimientos y la lógica de la interfaz de usuario dentro de una capa separada que permite tanto su reutilización como su independencia de la lógica de negocio, permitiendo que se pueda modificar la interfaz sin que esto afecte al negocio y viceversa. Está compuesta por todas las interfaces de usuario y los componentes necesarios para su correcto funcionamiento.

Negocio

Esta capa reúne todos los aspectos del software que automatizan o apoyan los procesos de negocio que llevan a cabo los usuarios. Estos aspectos típicamente incluyen las tareas que forman parte de los procesos, las reglas y restricciones que aplican. (Es donde se ejecutan todas las reglas de negocio). Por otra parte en esta capa se recogen todos los servicios necesarios para darle solución a los requisitos de negocio que no pueden ser satisfechos por el metamodelo. Tienen la responsabilidad de manejar todas las operaciones sobre una entidad de negocio en específico, así como todas las entidades que por conceptos de composición se encuentran relacionadas con esta; además de contener las clases controladoras, que son las que manejan todas las operaciones sobre las entidades de negocio definidas. Controla la ejecución de la capa de acceso a datos y servicios externos. Se puede diseñar la lógica de la

capa de negocios para uso directo por parte de componentes de presentación o su encapsulamiento como servicio y llamada a través de una interfaz de servicios que coordina la conversación con los clientes del servicio o invoca cualquier flujo o componente de negocio.

Metamodelo

Ofrece servicios de persistencia y recuperación de información a las capas superiores; la persistencia permite al programador almacenar, transferir y recuperar el estado de los objetos. El metamodelo describe las entidades que participan en el sistema, las relaciones y el flujo de datos que existe entre ellas; dichas entidades se mapean en clases que se componen de propiedades y métodos. Los objetos del modelo de dominio no deben contener código para cargar el estado de base de datos ni tampoco para guardarlo, dicho código debe declararse en otras clases de la capa de negocio; una buena práctica para unificar este código es usar el patrón repositorio, el cual consiste en crear una capa de abstracción que actúe de puente entre la capa de datos y la capa de negocio. Las clases repositorio, deben poder invocarse desde la capa de presentación, y normalmente tenemos una por cada objeto.

Acceso a datos

Esta capa reúne todos los aspectos del software que tienen que ver con el manejo de los datos persistentes o sea, se ocupa de obtener y persistir los datos. La capa de acceso a datos está directamente relacionada con los servicios definidos en el negocio. Su principal función es realizar una implementación de las interfaces definidas en la capa de negocio y al mismo tiempo trabajar directamente con la fuentes de datos establecidas en el metamodelo. Contiene las clases entidades que representan lo que se gestiona en la aplicación.

2.4.2 Modelo Vista Controlador (MVC)

La arquitectura Model-View-Controller surgió como patrón arquitectónico para el desarrollo de interfaces gráficas de usuario. Su concepto se basa en separar el modelo de datos de la aplicación de su representación de cara al usuario y de la interacción de éste con la aplicación, mediante la división de la aplicación en tres partes fundamentales:

- *El modelo:* que contiene la lógica de negocio de la aplicación.

- *La vista*: que muestra al usuario la información que éste necesita.
- *El controlador*: que recibe e interpreta la interacción del usuario, actuando sobre modelo y vista de manera adecuada para provocar cambios de estado en la representación interna de los datos, así como en su visualización.

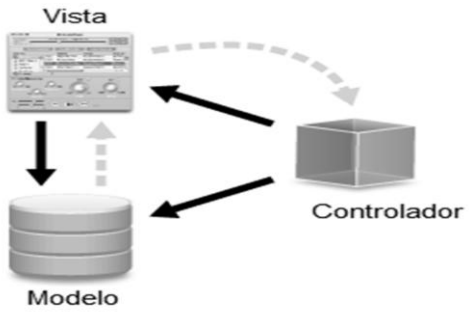


Figura 2: Estructura del patrón arquitectónico MVC

La principal ventaja de esta separación reside en la facilidad para realizar cambios en la aplicación puesto que cuando realizamos un cambio de bases de datos, programación o interfaz de usuario solo tocaremos uno de los componentes y que además podemos modificar uno de los componentes sin conocer cómo funcionan los otros. (28)

Aunque MVC se manifiesta habitualmente en aplicaciones web, el sistema propuesto hace referencia a este tipo de arquitectura, que ha demostrado ser muy apropiada para adaptarse bien a las tecnologías proporcionadas por la plataforma J2EE y otorga varias ventajas ya que al separar de manera clara la lógica de negocio (modelo) de la vista permite la reusabilidad del modelo, de modo que la misma implementación de la lógica de negocio que maneja una aplicación pueda ser usado en otras aplicaciones. En la solución propuesta se utiliza el estilo arquitectónico MVC permitiendo la definición de la estructura de la capa de presentación de la arquitectura utilizada y administrando el modo en que los datos son mostrados al usuario.

2.5 Patrones de diseño utilizados

Los patrones de diseño son soluciones simples y elegantes a problemas específicos y comunes del diseño orientado a objetos, basados en la experiencia y que se ha demostrado que funcionan (29). Existen problemas durante el desarrollo de software que se repiten o que son análogos, que responden a un cierto patrón, por lo que es deseable tener una colección de dichos patrones con soluciones óptimas para cada caso a modo de buenas prácticas. Durante el desarrollo del módulo se utilizaron un conjunto de patrones entre los que se destacan:

Singleton: se asegura que sólo se pueda crear una instancia de la clase y ofrece un punto global de acceso a esta instancia. El uso de este patrón permite que los servicios puedan ser creados sólo una vez. Un ejemplo de la utilización de este patrón se puede ver en la clase controladora GViewController.

Experto: este patrón fue utilizado en el diseño del módulo de manera general ya que cada método se implementa en la clase que conoce toda la información relacionado con este.

Controlador: la clase controladora del módulo, constituye un ejemplo de la aplicación de este patrón, la misma tendrá la responsabilidad de escuchar y responder a las peticiones realizadas por la capa de presentación y de comunicarse con la capa de negocio.

Alta cohesión: este patrón fue utilizado en el diseño del módulo de manera general ya que cada clase cuenta con la cantidad mínima de operaciones necesarias para realizar su funcionalidad.

Bajo acoplamiento: acoplamiento bajo significa que una clase no depende de muchas clases. Acoplamiento alto significa que una clase recurre a muchas otras clases. Un ejemplo de este patrón está presente en la clase controladora GViewController.

2.6 Modelo de diseño

El modelo de diseño es un modelo de objetos que describe la estructura estática (de datos) de los objetos del sistema (entidad, atributos y operaciones) y también sus relaciones, así como la realización física de los casos de uso centrándose en como los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar.

Además, el modelo de diseño sirve de abstracción de la implementación del sistema y es, de ese modo, utilizado como una entrada fundamental de las actividades de implementación.

Estructura

El sistema propuesto se ha desarrollado con una estructura que permite el encapsulamiento en paquetes que se encuentran relacionados para dar solución a un requerimiento en particular. Se caracteriza por contener elementos que permiten una implementación de forma gráfica lo más cercana posible al modelo del proceso de negocio.

Modelo de paquetes

Un paquete de diseño es usado para estructurar el modelo de diseño mediante su división en partes más pequeñas. Los paquetes son usados principalmente para administración de configuración y organización del modelo. Es de gran ayuda a la hora de realizar el diseño de un software el organizar por paquetes las clases y ficheros de un módulo. El modelo de paquetes del sistema propuesto se encuentra estructurado de la siguiente forma:



Figura 3: Estructura de paquetes del módulo Vistas

2.6.1 Diagrama de paquetes

Durante el desarrollo de software resulta muy conveniente agrupar clases y ficheros por diferentes criterios que ayudaran a una fácil comprensión de la aplicación, resultando conveniente el desarrollo de los diagramas de paquetes, los cuales muestran cómo está dividido el sistema en agrupaciones lógicas mostrando las dependencias entre estas.

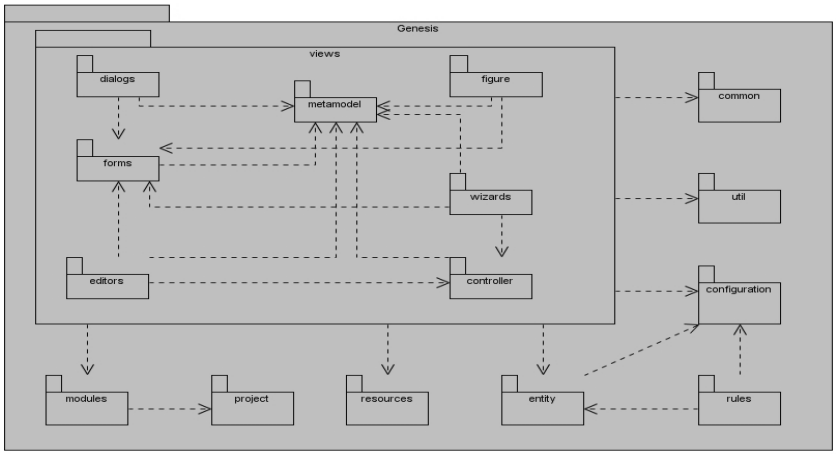


Figura 4: Diagrama de paquetes del sistema

Paquete common: contiene las clases que será utilizada indistintamente por cada módulo del sistema.

Paquete configuration: contiene las clases que describen los tipos de datos de las clases del dominio.

Paquete entity: contiene los paquetes necesarios para gestionar la creación de una entidad.

Paquete rules: contiene los paquetes necesarios para gestionar los diferentes tipos de validación que se pueden realizar sobre una entidad.

Paquete views: contiene los paquetes necesarios que actúan sobre una entidad para gestionar la creación de las vistas correspondientes.

Paquete modules: contiene las clases necesarias gestionar los editores en cada módulo.

Paquete project: contiene las clases necesarias para crear un proyecto.

Paquete forms: en este paquete se encuentran las interfaces e implementaciones del módulo correspondiente

Paquete metamodel: en este paquete están las clases del dominio del módulo.

Paquete editors: contiene clases que representan objetos a manipular en los formularios.

Paquete wizards: contiene las clases necesarias para gestionar la creación de un objeto por etapas.

Paquete dialogs: implementa las clases que hacen referencia al objeto a editar en el formulario que contienen.

Paquete figure: contiene las clases que crean los componentes y campos a mostrar cuando se está editando una vista.

Paquete util: en este paquete están las clases encargadas de manejar el objeto creado.

- ✓ GSerializationUtils: clona el objeto a modificar hasta terminar la operación y entonces persiste los datos.
- ✓ GFile: contiene el objeto persistido a transformar en un fichero xml y salvar.
- ✓ XML Transformator: transforma el objeto a un fichero xml.

Paquete resources: contiene las imágenes utilizadas en las interfaces de los diferentes módulos.

Paquete controller: contiene las clases necesarias que controlan la creación de un objeto.

2.6.2 Diagrama de clases visuales del diseño

En el diseño de clases se resume la definición de las clases que se pueden implementar en el software, se visualizan las relaciones entre ellas y se muestra gráficamente la interacción de los objetos para comunicarse entre sí. Un diagrama de clases es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Se utilizan para modelar la vista de diseño, que incluye modelar el vocabulario del sistema, las colaboraciones o los esquemas.

En el siguiente diagrama se muestran los atributos y métodos de cada clase identificada en la solución propuesta y se representa de una forma sencilla la colaboración y las responsabilidades de las distintas clases que conforman el sistema.

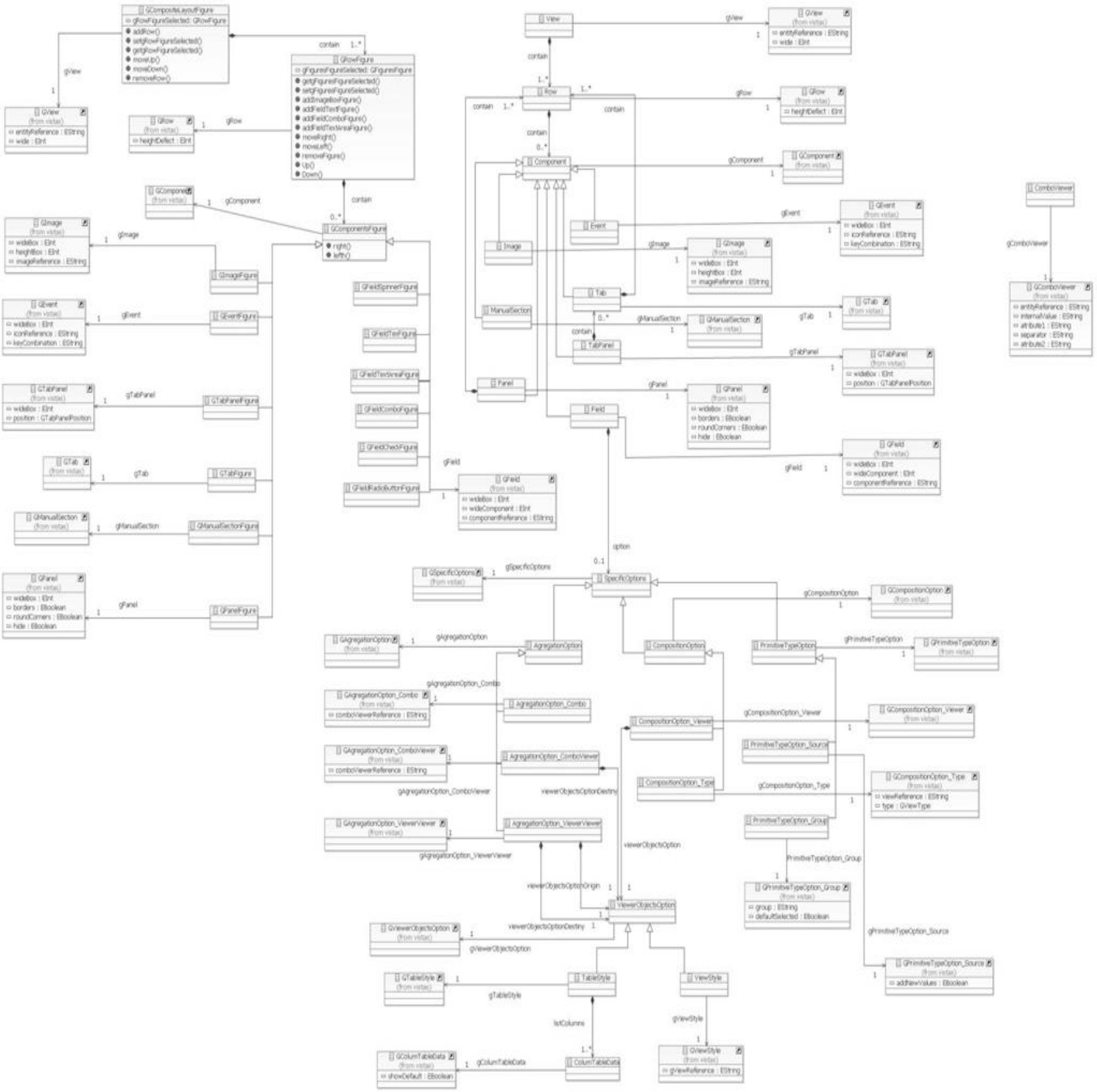


Figura 5: Diagrama de clases del diseño

2.6.3 Diagrama de interacción: diagrama de secuencia

Por cada escenario de los diagramas de clases del diseño es modelado un diagrama de interacción. Como diagrama de interacción fue seleccionado el de secuencia, el cual no es más que una representación que muestra, en determinado escenario de un caso de uso, los eventos generados por actores externos, su orden y los eventos internos del sistema. Contiene detalles de implementación del escenario, incluyendo los objetos y clases que se usan para implementar el escenario, y mensajes intercambiados entre los objetos.

Se realizará un diagrama de secuencia para la creación de una vista por la complejidad de los flujos presentes en el módulo.

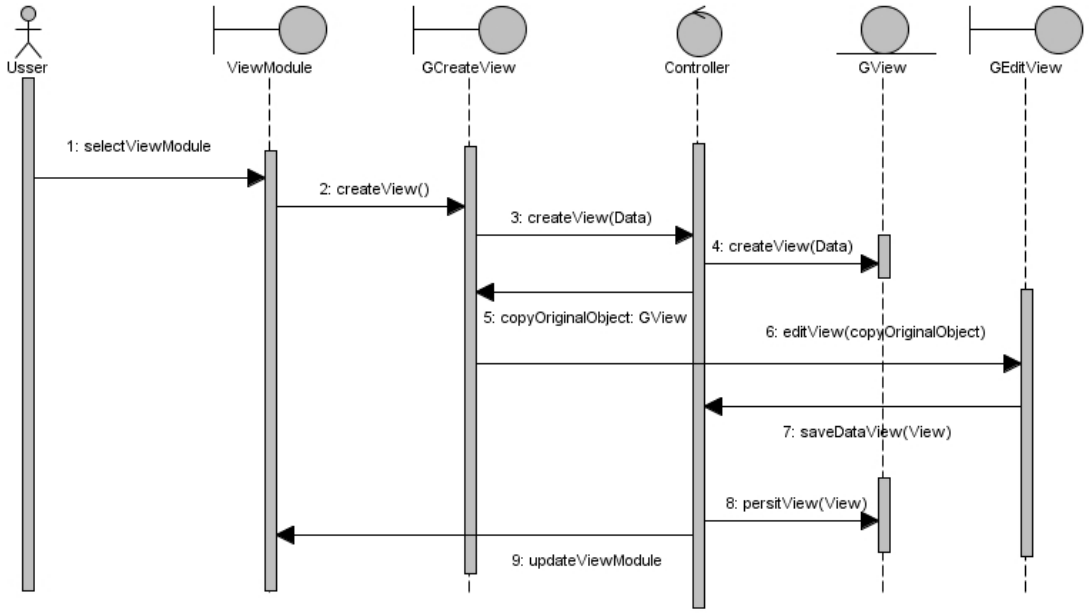


Figura 6: Diagrama de secuencia (Crear vista)

2.6.4 Diagrama de clases persistentes (metamodelo)

La persistencia es la capacidad de un objeto de mantener su valor en el espacio y en el tiempo. Las clases persistentes por lo general tienen como origen las clases entidad que modelan la información y el comportamiento asociado de algún fenómeno o concepto, como un objeto del mundo real o un suceso. El

diagrama de clases persistentes cuenta con todas la clases del modelo de datos incluyendo sus atributos, métodos y las relaciones que se establecen entre ellos. A continuación se muestra el diagrama de clases persistentes correspondientes al módulo.

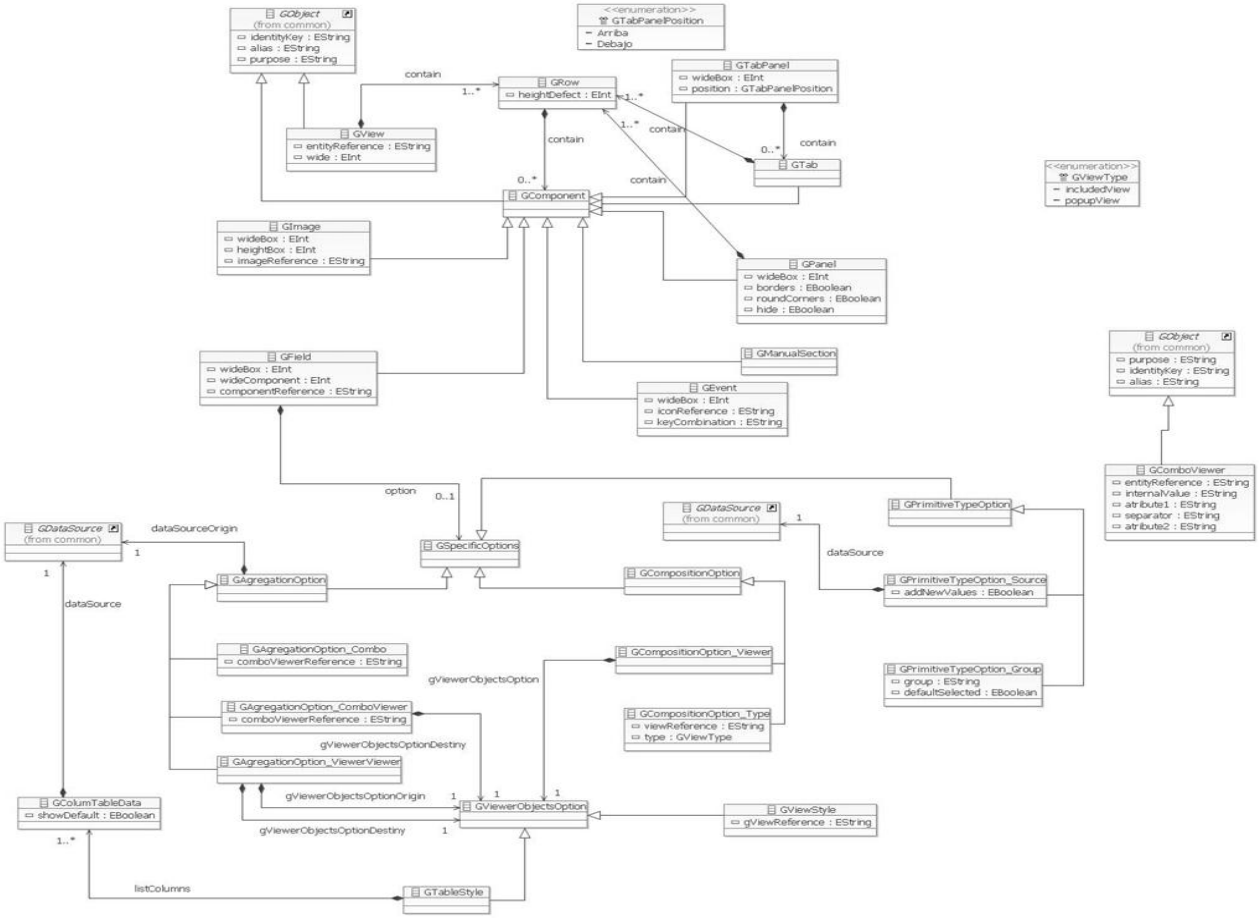


Figura 7: Diagrama de clases persistentes (metamodelo)

2.7 Conclusiones parciales

- En este capítulo fueron expuestos los artefactos generados durante el diseño de la solución propuesta, definiéndose las clases y servicios, así como la relación entre ellos.
- La definición de la arquitectura y de los patrones de diseño a utilizar permitió garantizar una mayor flexibilidad en la ejecución de la propuesta de solución, lográndose una independencia entre las capas de la misma.
- La identificación de las clases persistentes contribuyó al diseño del modelo de datos de la solución.
- Se especificaron los Casos de Uso del sistema y su descripción para lograr un mayor entendimiento de los mismos.

Una vez concluido el diseño de la solución puede darse paso a los flujos de implementación y prueba de la misma.

CAPÍTULO 3: IMPLEMENTACIÓN Y PRUEBA

3.1 Introducción

Como producto de software el módulo interfaces del plugin Génesis debe ser sometido a un conjunto de pruebas las cuales verifiquen y validen si constituye o no una solución adecuada que alcanza satisfacer las exigencias del entorno que motivaron su desarrollo.

En este capítulo se muestra el modelo de implementación que pone en práctica el diseño de la solución realizado en el capítulo anterior y se especifican las métricas para validar la solución propuesta, las pruebas realizadas al software que evalúan la calidad del sistema y se hace una valoración de las mismas según los resultados obtenidos.

3.2 Estándares de codificación

Los estándares de codificación se definen por el equipo de desarrollo para lograr generalización en la programación del software. La generalización de aspectos tan simples como el trato de las mayúsculas, ayuda a eliminar conflictos de funcionalidades implementadas con nombres iguales y guían de forma clara el proceso de desarrollo.

El mantenimiento del código es la facilidad con que el software puede modificarse para añadirle nuevas características, modificar las ya existentes, depurar errores o mejorar el rendimiento. Aunque la legibilidad y el mantenimiento son el resultado de muchos factores, una faceta del desarrollo de software en la que todos los programadores influyen especialmente es en la técnica de codificación. El mejor método para asegurar que un equipo de programadores mantenga un código de calidad es establecer un estándar de codificación sobre el que se efectuarán luego revisiones del código de rutina. A continuación se muestran algunos de los ejemplos de estándares que se tuvo en cuenta para el desarrollo del módulo interfaces del plugin Génesis.

Estilos para la capitalización

Se utilizan tres convenios para la capitalización de los identificadores:

Pascal: la primera letra en el identificador y la primera letra de cada subsiguiente palabra concatenada se capitalizan. Se pueden utilizar los identificadores de Pascal case en caso de tres o más caracteres. Por ejemplo: *BackColor.*

Camello: la primera letra en el identificador se pone en minúscula y la primera letra de cada subsiguiente palabra concatenada en mayúscula. Por ejemplo: *backColor.*

Mayúscula: todas las letras en el identificador se capitalizan. Esta convención se utiliza sólo para los identificadores que constan de dos o menos letras. Por ejemplo: *System.IO, System.Web.UI.*

Convención de nomenclatura

Variables: siempre comienzan con minúscula y en caso de nombres compuestos la primera letra de cada palabra comienza con mayúscula.

Clases: siempre comienzan con la letra “G”, seguido del nombre de la clase y en caso de nombres compuestos la primera letra de cada palabra comienza con mayúscula.

Funciones: siempre comienzan con minúscula y en caso de nombres compuestos la primera letra de cada palabra comienza con mayúscula. Los parámetros son separados por espacio luego de la coma que los separa.

Sensibilidad a mayúsculas

Para evitar confusiones y garantizar la interoperabilidad entre lenguajes, se siguen las siguientes reglas sobre el uso de mayúsculas y minúsculas:

- No utilizar nombres o identificadores que requieran ser case sensitivity.
- No utilizar *namespaces* que solo se diferencien con en el uso de las mayúsculas.

- No utilizar funciones con nombres de parámetros que se diferencian solo en el uso de las mayúsculas
- No utilizar *namespaces* con nombres de clases que se diferenciaran solo en el uso de las mayúsculas.
- No crear clases con propiedades que se diferencian solo en el uso de las mayúsculas.
- No crear clases con métodos que se diferencien solo en el uso de las mayúsculas.

Evitando confusión de nombre y tipo

Con el objetivo de incrementar la legibilidad del código se emplearon comentarios en todas las declaraciones de clases y funciones más complejas. También se organizó el código de forma estructurada, en bloques de código, para una mejor lectura del mismo.

Tratamiento de errores

El diseño de un sistema no solo debe tener en cuenta lo que debe ocurrir, sino que además debe realizar un análisis profundo de las diferentes situaciones que se puedan presentar y que constituyen algún tipo de violación o de situación en particular que provocaría un error dentro del sistema.

Las excepciones son parte del diseño aun cuando la gran mayoría de ellas son identificadas en el proceso de implementación, no obstante es necesario definir con antelación un mecanismo efectivo para su tratamiento.

3.3 Modelo de implementación

El modelo de implementación describe cómo los elementos del modelo de diseño se implementan en términos de componentes. Describe también cómo se organizan los componentes de acuerdo con los mecanismos de estructuración disponibles en el entorno de implementación y en el lenguaje o lenguajes de programación utilizados, y cómo dependen los componentes unos de otros, además de los recursos necesarios para poder ejecutar el sistema desarrollado.

3.3.1 Diagrama de componentes

Un diagrama de componentes describe los elementos físicos del sistema y sus relaciones. Se utiliza para mostrar las dependencias de compilación de los ficheros de código, relaciones de derivación entre ficheros de código fuente y ficheros que son resultados de la compilación, dependencias entre elementos de implementación y los correspondientes elementos de diseños que son implementados. (30)

Se encarga de mostrar las organizaciones y dependencias lógicas que existen entre los componentes de software, además pueden presentar paquetes utilizados para agrupar elementos del modelo. Este diagrama es utilizado para modelar la vista estática del sistema. A continuación se muestra el diagrama de componentes.

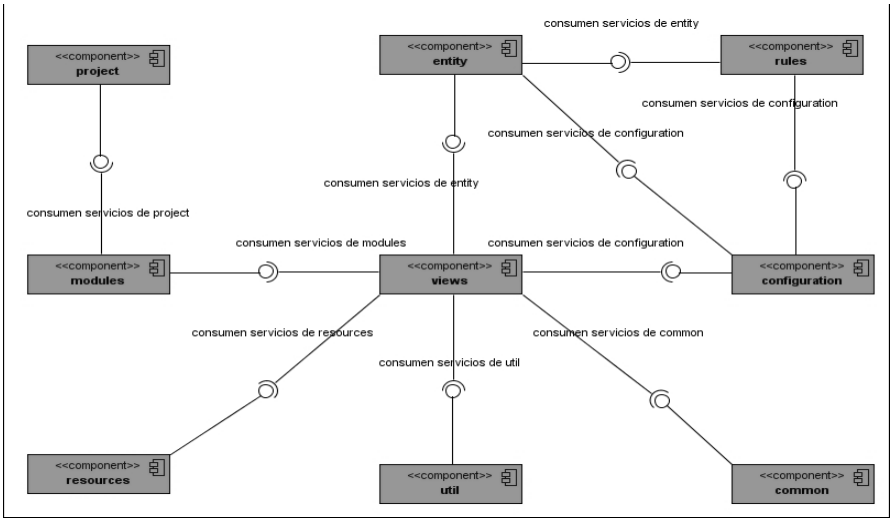


Figura 8: Diagrama de componentes

3.3.2 Diagrama de despliegue

Un Diagrama de Despliegue muestra las relaciones físicas de los distintos nodos que componen un sistema y el reparto de los componentes sobre dichos nodos. La vista de despliegue representa la disposición de las instancias de componentes de ejecución en instancias de nodos conectados por

enlaces de comunicación. Estos diagramas describen la arquitectura física del sistema durante la ejecución en términos de procesadores, dispositivos y componentes de software. (30)

El Modelo de Despliegue describe la distribución física del sistema entre los nodos de cómputo sobre los que se va a instalar el sistema. La siguiente figura refleja cómo quedaría dicha distribución para lograr un funcionamiento exitoso de la aplicación.



Figura 9: Diagrama de despliegue

3.4 Métricas de software

Las métricas de software son una medida cuantitativa que permite a los desarrolladores tener una visión profunda de la eficacia del proceso del software y de los proyectos que dirigen utilizando el proceso como un marco de trabajo. Se reúnen los datos básicos de calidad y productividad. Estos datos son entonces analizados, comparados con promedios anteriores, y evaluados para determinar las mejoras en la calidad y productividad. Las métricas son también utilizadas para señalar áreas con problemas de manera que se puedan desarrollar los remedios y mejorar el proceso del software. (31)

Las métricas empleadas están diseñadas para evaluar los siguientes atributos de calidad:

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

- *Reutilización:* consiste en el grado de reutilización presente en una clase o estructura de clase, dentro de un diseño de software.
- *Acoplamiento*: consiste en el grado de dependencia o interconexión de una clase o estructura de clase con otras, está muy ligada a la característica de Reutilización.
- *Complejidad del mantenimiento*: consiste en el grado de esfuerzo necesario a realizar para desarrollar un arreglo, una mejora o una rectificación de algún error de un diseño de software. Puede influir indirecta, pero fuertemente en los costes y la planificación del proyecto.
- *Cantidad de pruebas*: consiste en el número o el grado de esfuerzo para realizar las pruebas de calidad (Unidad) del producto (componente, modulo, clase, conjunto de clases, etc.) diseñado.

Las métricas escogidas como instrumento para evaluar la calidad del diseño descrito en el capítulo anterior y su relación con los atributos de calidad son las siguientes:

Tamaño Operacional de Clase (TOC)

Está dado por el número de métodos asignados a una clase y evalúa los siguientes atributos de calidad.

Tabla 1: Atributos de calidad evaluados por la métrica TOC

Atributo de calidad	Modo en que lo afecta
Responsabilidad	Un aumento del TOC implica un aumento de la responsabilidad asignada a la clase.
Complejidad de implementación	Un aumento del TOC implica un aumento de la complejidad de implementación de la clase.
Reutilización	Un aumento del TOC implica una disminución del grado de reutilización de la clase.

Para los cuales están definidos los siguientes criterios y categorías de evaluación:

Tabla 2: Criterios de evaluación para la métrica TOC

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

Relaciones entre Clases (RC)

Esta métrica está dada por la cantidad de relaciones de uso que existe entre las distintas clases que forman el diseño propuesto. Se le aplica a las mismas clases que le fue aplicada la métrica TOC. Los aspectos de calidad que se miden son: Acoplamiento, Complejidad de mantenimiento, Reutilización y Cantidad de pruebas.

Tabla 3: Atributos de calidad evaluados por la métrica RC

Atributo de calidad	Modo en que lo afecta
Acoplamiento	Un aumento del RC implica un aumento del Acoplamiento de la clase.
Complejidad de mantenimiento	Un aumento del RC implica un aumento de la complejidad del mantenimiento de la clase.
Reutilización	Un aumento del RC implica una disminución en el grado de reutilización de la clase.
Cantidad de pruebas	Un aumento del RC implica un aumento de la Cantidad de pruebas de unidad necesarias para probar una clase.

Para los cuales están definidos los siguientes criterios y categorías de evaluación:

Tabla 4: Criterios de evaluación para la métrica RC

Atributo de calidad	Categoría	Criterio
Acoplamiento	Ninguno	0
	Bajo	1
	Medio	2
	Alto	>2
Complejidad de mantenimiento	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio
Reutilización	Baja	>2*Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	<=Promedio
Cantidad de pruebas	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio

3.4.1 Resultados obtenidos de la aplicación de la métrica TOC

Tabla 5: Instrumento de evaluación de la métrica TOC

Clase	Cantidad de procedimientos	Responsabilidad	Complejidad	Reutilización
GComponentsFigure	7	Media	Media	Media
GCompositeLayoutFigure	15	Alta	Alta	Baja
GEventFigure	2	Baja	Baja	Alta
GFieldCheckFigure	2	Baja	Baja	Alta
GFieldComboFigure	2	Baja	Baja	Alta
GFieldRadioButtonFigure	2	Baja	Baja	Alta
GFieldSpinnerFigure	2	Baja	Baja	Alta
GFieldTexFigure	2	Baja	Baja	Alta
GFieldTextAreaFigure	4	Baja	Baja	Alta

GImageFigure	2	Baja	Baja	Alta
GManualSectionFigure	6	Baja	Baja	Alta
GPanelFigure	5	Baja	Baja	Alta
GRowFigure	33	Alta	Alta	Baja
GTabFigure	3	Baja	Baja	Alta
GTabPanelFigure	5	Baja	Baja	Alta

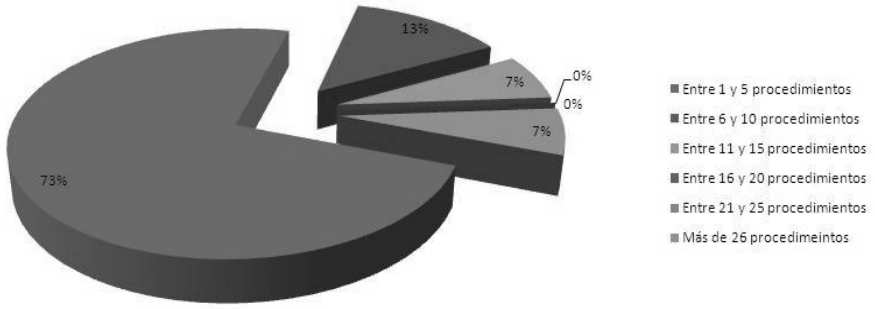


Figura 10: Resultados de la evaluación de la métrica TOC para los procedimientos

Una vez obtenidos los resultados de la evaluación del instrumento de medición de la métrica TOC, se puede concluir que el diseño propuesto tiene una calidad aceptable teniendo en cuenta que el 73% de las clases empleadas en el sistema posee 5 operaciones o menos lo que conlleva a evaluaciones positivas de los atributos de calidad involucrados (responsabilidad, complejidad de implementación y reutilización). A continuación se muestran los resultados obtenidos para los mismos.

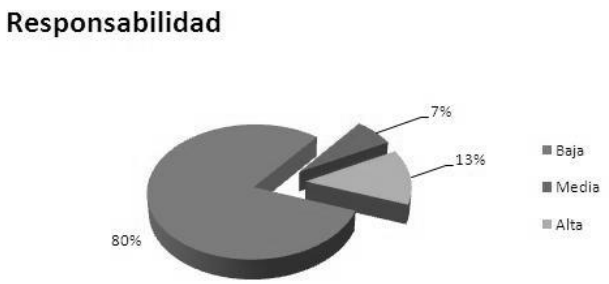


Figura 11: Resultados de la evaluación de la métrica TOC para el atributo Responsabilidad

Las clases evaluadas con el instrumento de evaluación de la métrica TOC confirman la calidad del diseño propuesto ya que el 80% de las mismas presenta una baja responsabilidad y un 7% presenta una responsabilidad media, lo que indica que las clases no se encuentran muy sobrecargadas, siendo así el 87% un promedio positivo para la evaluación del atributo involucrado.

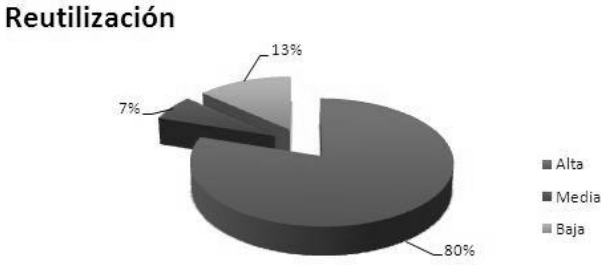


Figura 12: Resultados de la evaluación de la métrica TOC para el atributo Reutilización

La Figura 14 indica un 80% de alta reutilización presente en las clases evaluadas lo que conlleva a una evaluación positiva del atributo involucrado, confirmando la calidad del diseño. Asimismo se demuestra el bajo grado de dificultad que presentó implementar el diseño propuesto (Figura 15), ya que el 80% de las clases presentan baja complejidad de implementación, lo que indica que el diseño no es complejo.

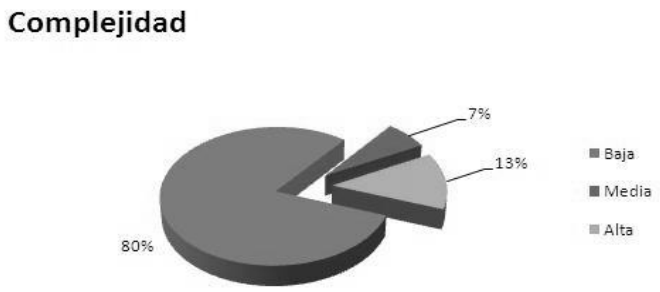


Figura 13: Resultados de la evaluación de la métrica TOC para el atributo Complejidad

3.4.2 Resultados obtenidos de la aplicación de la métrica RC

Tabla 6: Instrumento de evaluación de la métrica RC

Clase	Cantidad de relaciones de uso	Acoplamiento	Complejidad de mantenimiento	Reutilización	Cantidad de pruebas
GComponentsFigure	1	Bajo	Baja	Alta	Baja
GCompositeLayoutFigure	3	Alto	Media	Media	Media
GEventFigure	1	Bajo	Baja	Alta	Baja
GFieldCheckFigure	1	Bajo	Baja	Alta	Baja
GFieldComboFigure	1	Bajo	Baja	Alta	Baja
GFieldRadioButtonFigure	1	Bajo	Baja	Alta	Baja
GFieldSpinnerFigure	1	Bajo	Baja	Alta	Baja
GFieldTexFigure	1	Bajo	Baja	Alta	Baja
GFieldTextAreaFigure	1	Bajo	Baja	Alta	Baja
GImageFigure	1	Bajo	Baja	Alta	Baja
GManualSectionFigure	1	Bajo	Baja	Alta	Baja
GPanelFigure	1	Bajo	Baja	Alta	Baja

GRowFigure	1	Bajo	Baja	Alta	Baja
GTabFigure	1	Bajo	Baja	Alta	Baja
GTabPanelFigure	1	Bajo	Baja	Alta	Baja

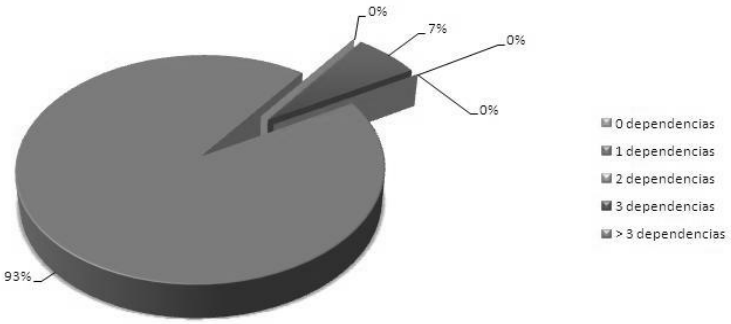


Figura 14: Resultados de la evaluación de la métrica RC para las dependencias

Una vez obtenidos los resultados de la evaluación del instrumento de medición de la métrica RC, se puede concluir que el diseño propuesto tiene una calidad aceptable teniendo en cuenta que el 93% de las clases empleadas posee menos de 3 dependencias de otras clases lo que conlleva a evaluaciones positivas de los atributos de calidad involucrados (acoplamiento, complejidad de mantenimiento, cantidad de pruebas y reutilización). A continuación se muestran los resultados obtenidos para los mismos.

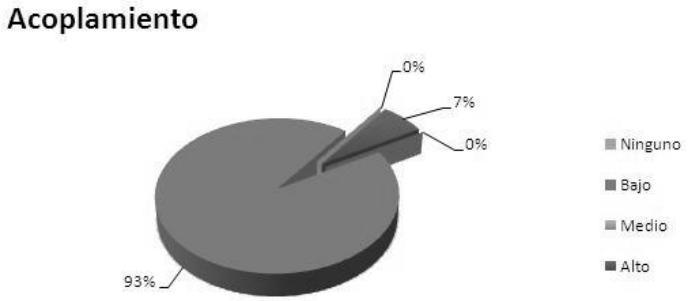


Figura 15: Resultados de la evaluación de la métrica RC para el atributo Acoplamiento

Las clases evaluadas con el instrumento de evaluación de la métrica RC confirman la calidad del diseño propuesto ya que el 93% de las mismas presenta un bajo grado de dependencia con otras clases, llevando a evaluaciones positivas del acoplamiento.



Figura 16: Resultados de la evaluación de la métrica RC para el atributo Complejidad de Mantenimiento

La Figura 18 indica un 93% de baja complejidad de mantenimiento presente en las clases evaluadas lo que indica un bajo grado de esfuerzo a realizar al desarrollar alguna mejora del diseño o rectificar algún error, lo que conlleva a una evaluación positiva del atributo involucrado, confirmando la calidad del diseño propuesto. Asimismo se demuestra la alta reutilización presente en las clases evaluadas (Figura 19), siendo esta de un 93%. Las características de reutilización y acoplamiento se encuentran estrechamente relacionadas.

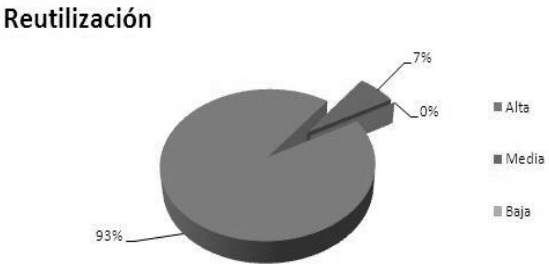


Figura 17: Resultados de la evaluación de la métrica RC para el atributo Reutilización



Figura 18: Resultados de la evaluación de la métrica RC para el atributo Cantidad de Pruebas

Los resultados de la evaluación del atributo cantidad de pruebas indica un bajo grado de esfuerzo para realizar las pruebas de calidad del producto diseñado, siendo este del 93%, lo que conlleva a una evaluación positiva del atributo involucrado, confirmando la calidad del diseño propuesto.

3.4.3 Matriz de inferencia de indicadores de calidad

La matriz inferencia de indicadores de calidad, también llamada matriz de cubrimiento, es una representación estructurada de los atributos de calidad y métricas utilizadas para evaluar la calidad del diseño propuesto. Dicha matriz permite conocer si los resultados obtenidos de las relaciones atributo/métrica es positivo o no, llevando estos resultados a una escalabilidad numérica donde, si los resultados son positivos se le asigna el valor de 1, si son negativos toma valor 0 y si no existe relación es considerada como nula y es representada con un guión simple (-). Luego se puede obtener un resultado general para cada atributo promediando todas sus relaciones no nulas.
A continuación se muestran los resultados obtenidos.

Tabla 7: Resultados de la evaluación de la relación atributo/métrica

Atributos/Métricas	TOC	RC	Promedio
Responsabilidad	1	(-)	1
Complejidad de implementación	1	(-)	1

Reutilización	1	1	1
Acoplamiento	(-)	1	1
Complejidad de mantenimiento	(-)	1	1
Cantidad de pruebas	(-)	1	1

Tabla 8: Rango de valores para la evaluación de la relación atributo/métrica

Categoría	Rango de valores
Malo	<=0.4
Regular	>0.4 y <0.7
Bueno	>=0.7

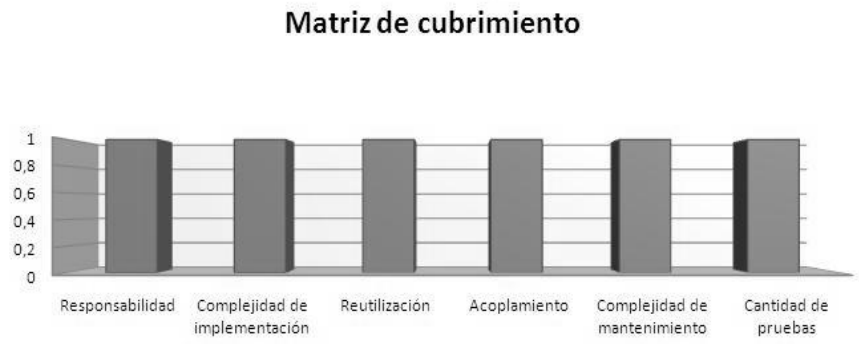


Figura 19: Resultados obtenidos de la evaluación de los atributos de calidad

Las métricas de software aplicadas posibilitaron estimar la calidad de los atributos internos del producto, demostrando una aceptable calidad de diseño. La solución propuesta contribuirá a la disminución de disturbios durante la implementación del componente, garantizando la reutilización y agilidad en el proceso de desarrollo de software. En sentido general el resultado de la aplicación de estas métricas demuestra que las clases no se encuentran muy sobrecargadas en responsabilidad y existe además bajo

acoplamiento entre las mismas. Indican además que el diseño no es complejo, pues la complejidad de mantenimiento es baja, así como la complejidad en las pruebas.

3.5 Pruebas de software

Las pruebas de software son los procesos que permiten verificar y revelar la calidad de un producto de software. Son un elemento crítico para la garantía de la calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación. Dichas pruebas son realizadas con el objetivo de detectar errores en el sistema, identificar posibles fallos de implementación, calidad o usabilidad, por lo que se llevan a cabo durante todo el ciclo de vida del producto.

Los casos de prueba especifican una forma de probar el sistema, incluyendo las entradas con las que se ha de probar, las condiciones bajo las que ha de probarse, así como los resultados esperados.

Un buen caso de prueba es aquel que tiene alta probabilidad de mostrar un error no descubierto hasta entonces. Una prueba tiene éxito si descubre un error no detectado hasta entonces. La prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el software. (32)

La fase de pruebas añade al producto final el valor para afirmar que ya se ha alcanzado la calidad requerida. Un gran porcentaje de los programas que se desarrollan tienen errores y es en la fase de pruebas donde se descubren, esa es la importancia fundamental de esta etapa, el objetivo específico de la fase de pruebas es encontrar el mayor número de errores.

3.5.1 Pruebas estructurales o de caja blanca

Las llamadas pruebas de caja blanca, también conocidas como técnicas de caja transparente o de cristal, proponen una manera de diseñar los casos de prueba centrándose en el comportamiento interno y la estructura del programa. De esta manera se examina solo la lógica interna del programa, dejando fuera los aspectos de rendimiento del mismo.

Según Pressman, mediante la prueba de la caja blanca el ingeniero del software puede obtener casos de prueba que cumplan una serie de requisitos:

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

Es por ello que se considera a la prueba de caja blanca como uno de los tipos de pruebas más importantes que se le aplican al 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.

Algunas técnicas de prueba de Caja Blanca son:

La cobertura de caminos o prueba del camino básico: esta técnica permite obtener una medida de la complejidad lógica de un diseño y usar la misma 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.

Pruebas sobre las expresiones lógico-aritméticas o prueba de condición: es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa.

Pruebas de camino de datos o de flujo de datos: se selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.

Comprobación de bucles o prueba de bucles: es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles.

Para la realización de las pruebas de caja blanca se utilizan las pruebas unitarias.

Pruebas unitarias: son una forma de probar pequeñas e individuales porciones de código. A través de ellas se verifica que cierto módulo o funcionalidad se ejecuta dentro de los parámetros y especificaciones concretadas en los requisitos. Las mismas permiten detectar efectivamente la inyección de defectos durante fases sucesivas de desarrollo o mantenimiento. Las pruebas unitarias típicamente son automatizadas, pero pueden llevarse a cabo de forma manual. Cuando son automatizadas es buena práctica que forme parte del repositorio que contiene al código probado. (33)

Para realizar las pruebas de caja blanca se utilizó la técnica del camino básico. Los pasos que se siguen para aplicar esta técnica son:

1. A partir del diseño o del código fuente, se dibuja el grafo de flujo asociado.
2. Se calcula la complejidad ciclomática del grafo.
3. Se determina un conjunto básico de caminos independientes.
4. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del camino básico.

A continuación se procede a enumerar las sentencias del código y a partir del mismo se construye el grafo de flujo asociado.

```
public void removeRow() {
    int pos = getPositionFigureSelected();//1
    if (listRows.size() != 1) {//2
        if (pos == listRows.size() - 1) {//3
            getListRows().get(pos - 1).selectGRowFigure();//4
            setgRowFigureSelected(getListRows().get(pos - 1));//4
            getListRows().remove(pos).dispose();//4
        } else {
            for (int i = pos + 1; i < listRows.size(); i++) {//5
                GRowFigure item = listRows.get(i);//6
                item.up((gRowFigureSelected.getBounds().height + 1));//6
                listRows.set(i, item);//6
            }
            listRows.get(pos + 1).selectGRowFigure();//7
            setgRowFigureSelected(listRows.get(pos + 1));//7
            getListRows().remove(pos).dispose();//7
        }
    }//8
}
```

Figura 20: Código fuente de la funcionalidad Eliminar Fila

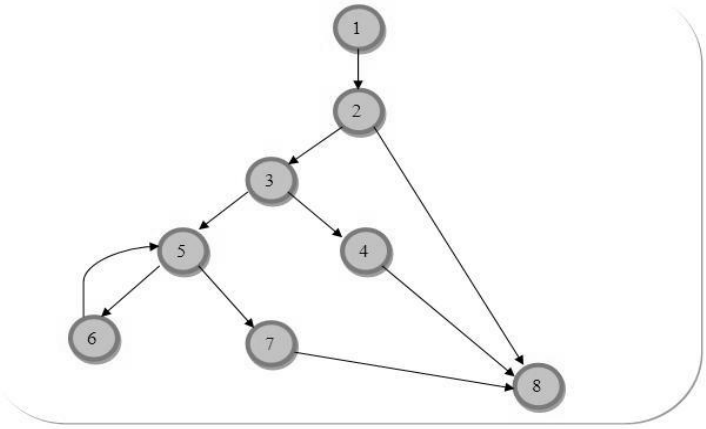


Figura 21: Grafo de flujo asociado a la funcionalidad Eliminar Fila

Luego de haber construido el grafo se realiza el cálculo de la complejidad ciclomática mediante las tres fórmulas descritas a continuación, las cuales deben arrojar el mismo resultado para asegurar que el cálculo de la complejidad es correcto.

1 -) V (G) = (A – N) + 2

Siendo “A” la cantidad total de aristas y “N” la cantidad total de nodos.
V (G) = (10 - 8) + 2
V (G) = 4

2 -) V (G) = P + 1

Siendo “P” la cantidad total de nodos predicados (son los nodos de los cuales parten dos o más aristas).
V (G) = 3 + 1
V (G) = 4

3 -) V (G) = R

Siendo “R” la cantidad total de regiones, para cada formula “V (G)” representa el valor del cálculo.
V (G) = 4

El cálculo efectuado mediante las tres fórmulas ha dado el mismo valor, dando como resultado 4, lo que indica que existen 4 posibles caminos por donde el flujo puede circular, y determina el número de pruebas que se deben realizar para asegurar que se ejecute cada sentencia al menos una vez. Seguidamente es necesario representar los caminos básicos por los que puede recorrer el flujo:

- Camino básico #1: 1 – 2 – 8
- Camino básico #2: 1 – 2 – 3 – 4 – 8
- Camino básico #3: 1 – 2 – 3 – 5 – 7 – 8
- Camino básico #4: 1 – 2 – 3 – 5 – 6 – 5 – 7 – 8

Para cada camino se realiza un caso de prueba.

Caso de prueba para el camino básico #1:

Descripción:

Se cumplirá el requisito de que habrá una fila seleccionada que es añadida por defecto al comenzar a editar una vista.

Condición de ejecución:

La vista a editar solo tendrá la fila añadida por defecto.

Entrada:

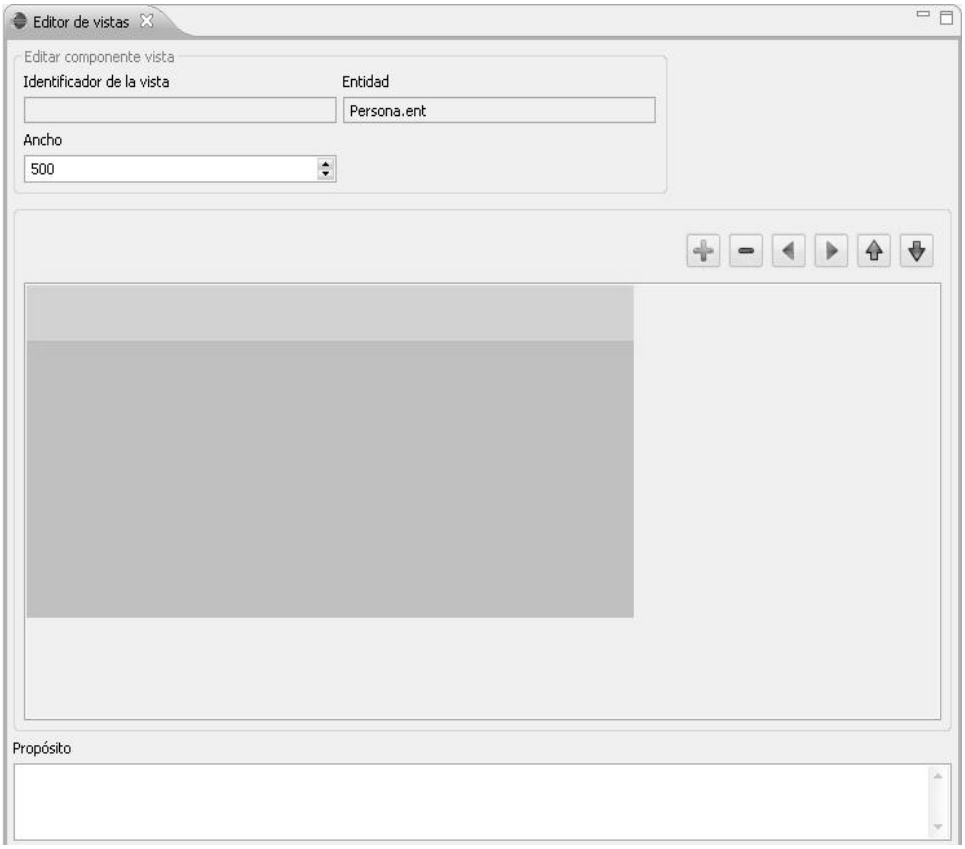


Figura 22: Entrada para el camino básico #1

Resultados esperados:

No pasa nada como resultado de intentar eliminar la fila añadida por defecto. La vista se mantiene exactamente igual a como se muestra en la entrada.

Caso de prueba para el camino básico #2:

Descripción:

Se cumplirá el requisito de que habrá una fila seleccionada y la vista a editar contiene más de una fila añadidas.

Condición de ejecución:

La fila seleccionada será la que está al final de la vista.

Entrada:

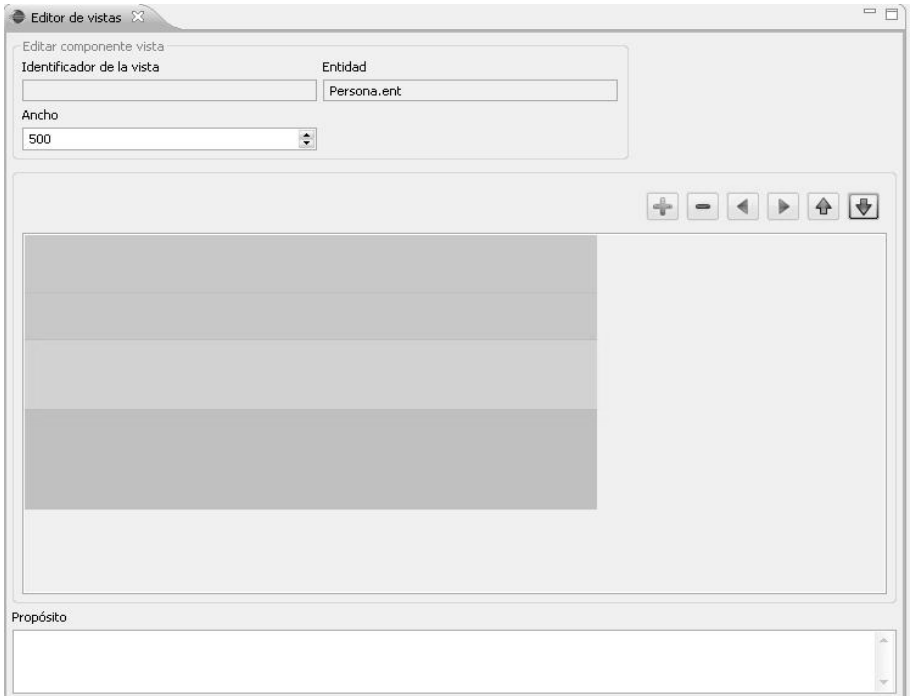


Figura 23: Entrada para el camino básico #2

Resultados esperados:

Se elimina la fila seleccionada y entonces se selecciona automáticamente la fila anterior a esta, como se muestra a continuación:

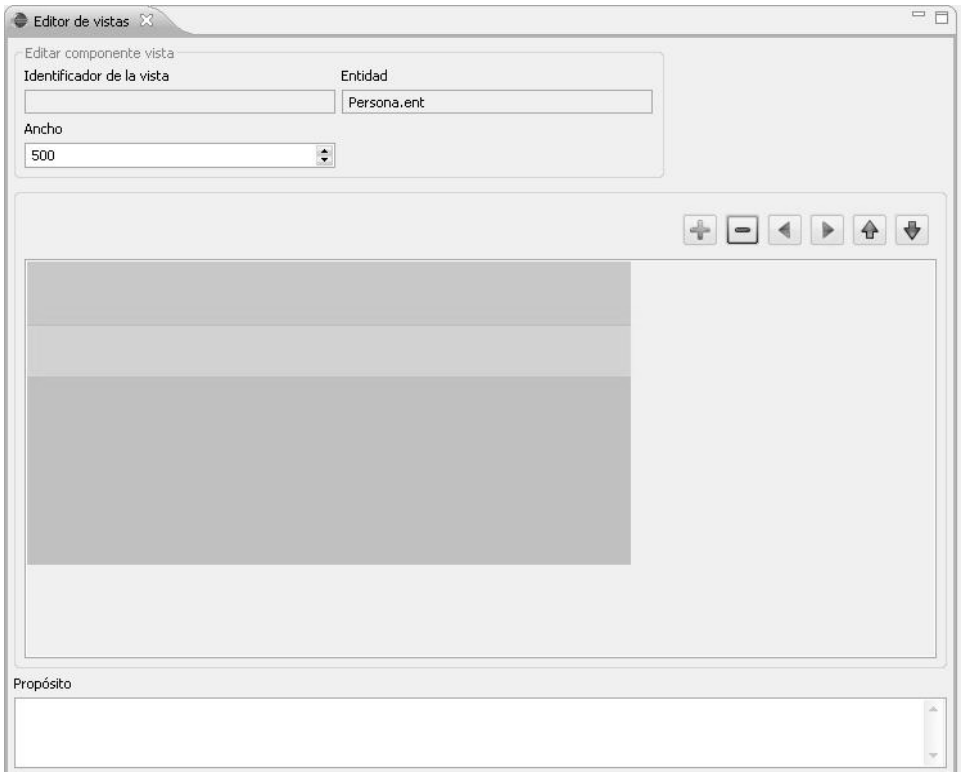


Figura 24: Resultados esperados para el camino básico #2

Caso de prueba para el camino básico #3:

Descripción:

Se cumplirá el requisito de que habrá una fila seleccionada y la vista a editar contiene más de una fila añadidas.

Condición de ejecución:

La fila seleccionada no estará al final de la vista.

Entrada:

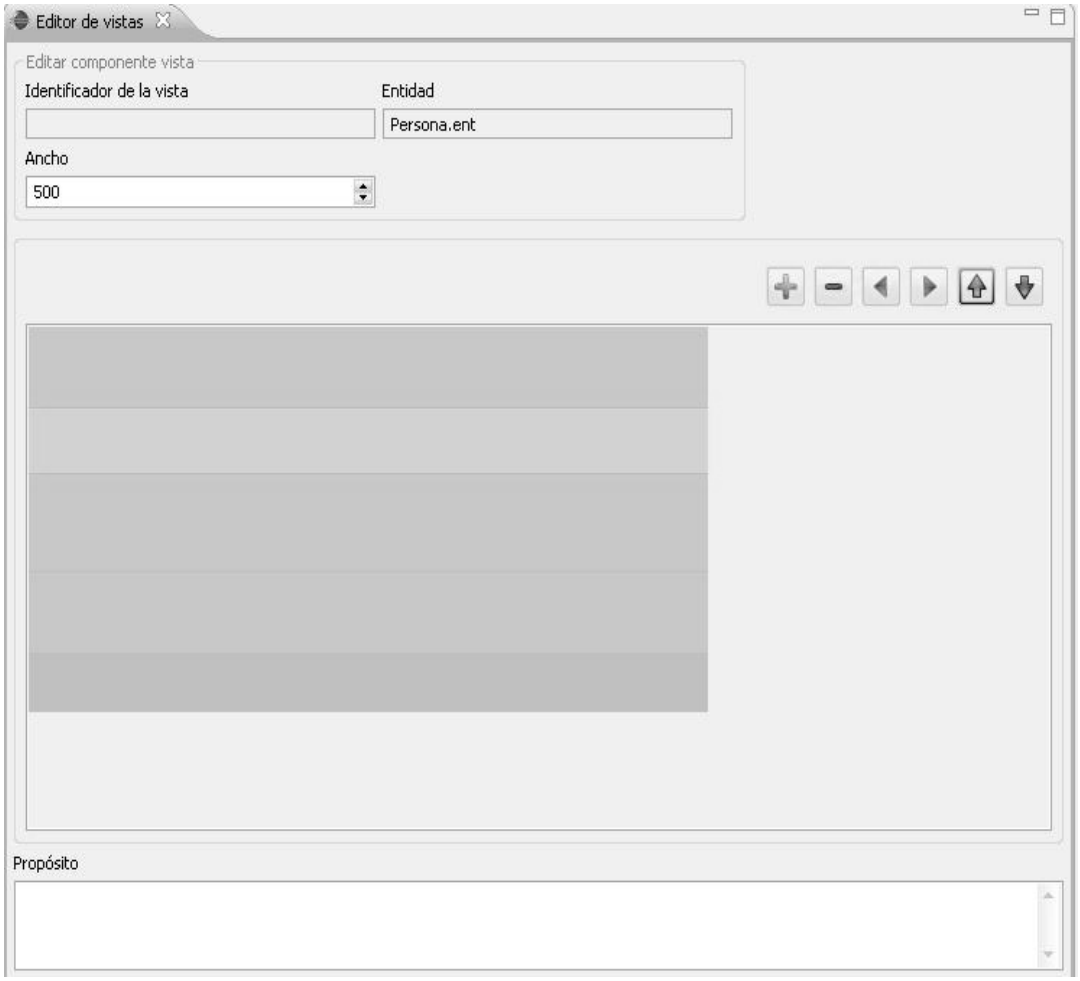


Figura 25: Entrada para el camino básico #3

Resultados esperados:

Se elimina la fila seleccionada, entonces se selecciona automáticamente la fila a continuación de esta y se corren hacia arriba, como se muestra a continuación:

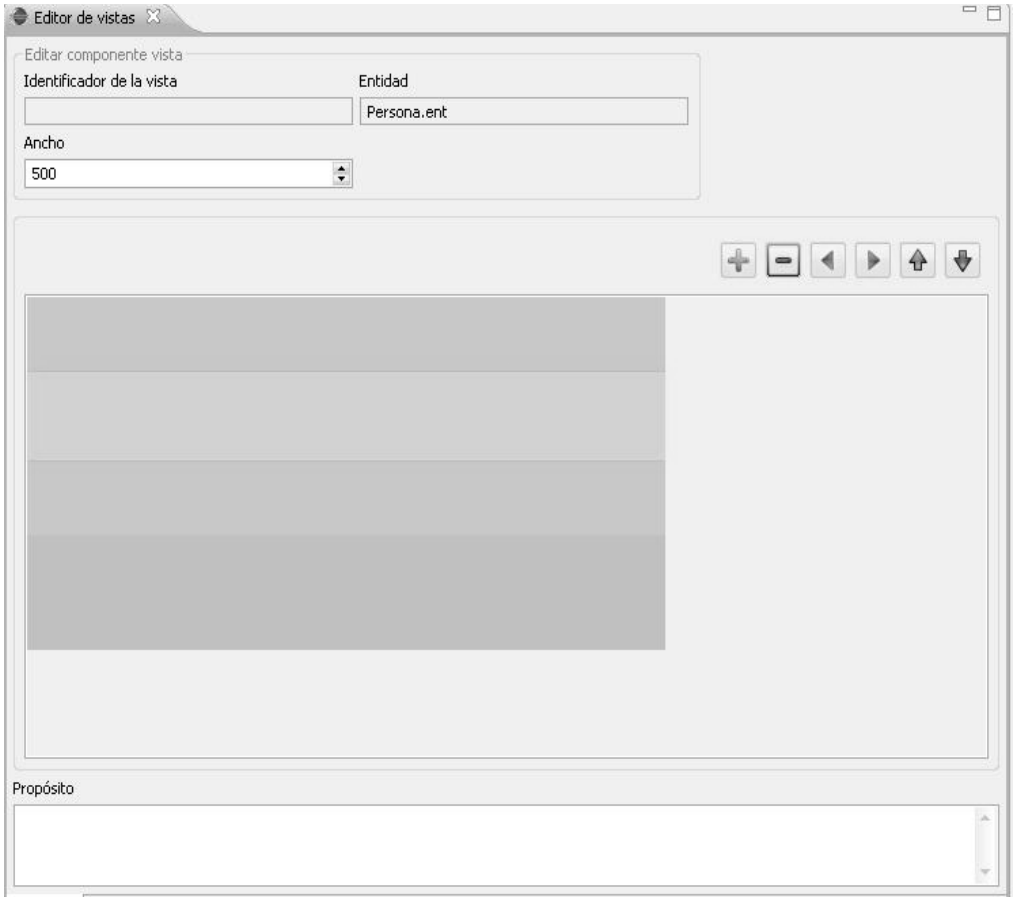


Figura 26: Resultados esperados para el camino básico #3

Caso de prueba para el camino básico #4:

Descripción:

Se cumplirá el requisito de que habrá una fila seleccionada y la vista a editar contiene más de una fila añadidas.

Condición de ejecución:

La fila seleccionada no estará al final de la vista.

Entrada:

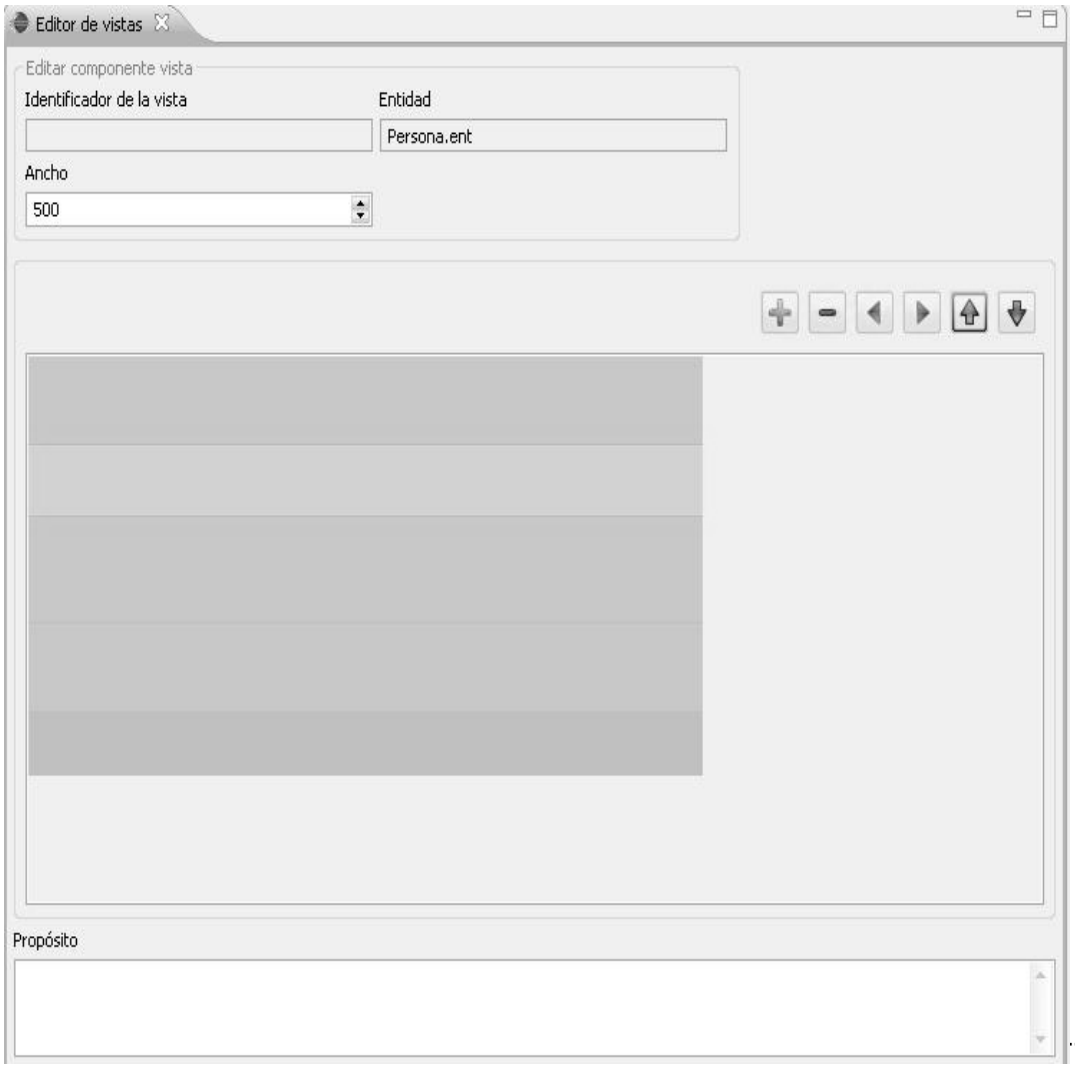


Figura 27: Entrada para el camino básico #4

Resultados esperados:

Se elimina la fila seleccionada, entonces se selecciona automáticamente la fila a continuación de esta y se corren hacia arriba, como se muestra a continuación:

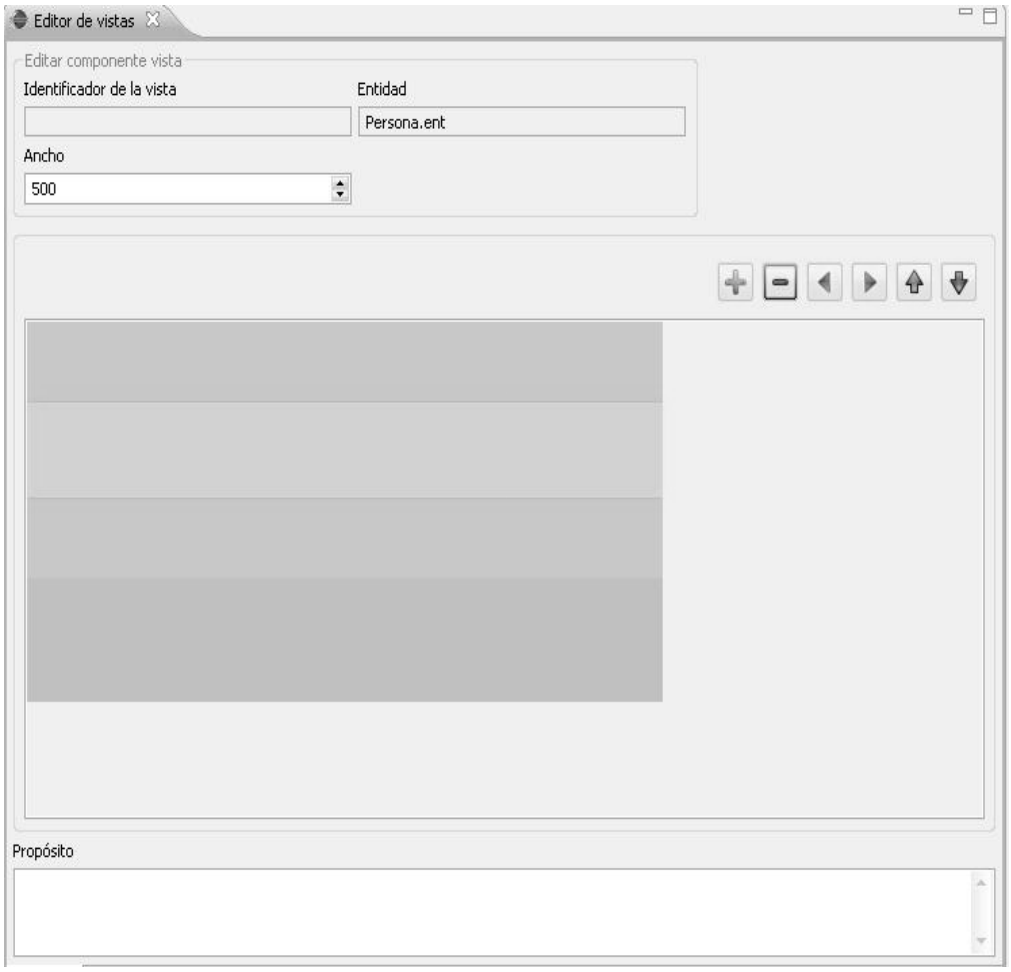


Figura 28: Resultados esperados para el camino básico #4

Luego de aplicar los distintos casos de pruebas, se pudo comprobar que el flujo de trabajo del método es correcto ya que cumple con las condiciones necesarias que se habían planteado.

3.5.2 Pruebas funcionales o de caja negra

Las pruebas de caja negra son las que se llevan a cabo sobre la interfaz del software, o sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa se mantiene. Estas pruebas permiten encontrar:

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

Estas pruebas se basan en la especificación del programa o componente a ser probado para elaborar los casos de prueba. El componente es visto como una "caja negra" cuyo comportamiento es desconocido y sólo puede ser evaluado estudiando sus entradas y las salidas obtenidas. También son conocidas como pruebas de comportamiento o pruebas inducidas por los datos.

Algunas técnicas utilizadas en la prueba de caja negra son:

Partición de equivalencia: técnica que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. La partición equivalente se dirige a una definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar.

Análisis de valores límite: la experiencia muestra que los casos de prueba que exploran las condiciones límite producen mejor resultado que aquellos que no lo hacen. Las condiciones límite son aquellas que se hallan en los márgenes de la clase de equivalencia, tanto de entrada como de salida. Por ello, se ha desarrollado el análisis de valores límite como técnica de prueba. Esta técnica lleva a elegir los casos de prueba que ejerciten los valores límite.

Grafos de causa-efecto: es una técnica que permite al encargado de la prueba validar complejos conjuntos de acciones y condiciones.

Para la aplicación de este tipo de pruebas se diseñaron un conjunto de escenarios de pruebas con el objetivo de evaluar el cumplimiento de los requisitos del sistema. Los escenarios empleados se muestran en el Anexo #3.

Después de probar los casos de pruebas descritos en la aplicación se obtuvo los resultados esperados, comprobándose el correcto funcionamiento de la misma.

3.7 Conclusiones parciales

En este capítulo fueron expuestos los artefactos generados como parte del modelo de implementación de la solución propuesta, tales como el diagrama de componentes y el diagrama de despliegue y los estándares de codificación. Se realizó una validación del diseño expuesto en el capítulo anterior mediante la aplicación de métricas para la evaluación de atributos de calidad, así como se describieron las pruebas estructurales y funcionales realizadas que permitieron comprobar el correcto funcionamiento del sistema.

En resumen:

- El uso de los estándares de codificación garantizó que la implementación realizada fuera de manera organizada, permitiendo un mayor entendimiento de la misma.
- Se desarrolló una primera versión del sistema, definiéndose los componentes que intervienen en la solución final de la aplicación así como la relación de los nodos físicos que la componen.
- Las pruebas realizadas validaron la calidad de las funcionalidades de la solución.

CONCLUSIONES

Con la realización del presente trabajo de diploma se concluye que:

- Actualmente las interfaces se crean manualmente ya que las tecnologías de generación existentes no producen una interfaz que cumpla totalmente con los requisitos deseados por los desarrolladores. La evaluación del estado del arte del desarrollo de software dirigido por modelos y el análisis realizado sobre algunas herramientas ya existentes para la gestión de modelos de interfaces, permitió identificar una serie de desventajas en las mismas, lográndose desarrollar una herramienta con más calidad y de gran ayuda para los desarrolladores, que les permitirá ahorrar gran parte del tiempo invertido en la creación de una interfaz. Además nuestro país alcanza de esta forma un grado de independencia tecnológica.

- El estudio de las tecnologías, lenguajes, herramientas y metodología a utilizar, definidas por el proyecto, fueron esenciales para el desarrollo de la solución, influyendo en la calidad del producto obtenido.

- La definición de la arquitectura y de los patrones de diseño a utilizar permitió garantizar una mayor flexibilidad en la ejecución de la propuesta de solución, lográndose una independencia entre las capas de la misma.

- El uso de los estándares de codificación garantizó que la implementación realizada fuera de manera organizada, permitiendo un mayor entendimiento de la misma.

- La realización de pruebas a las funcionalidades definidas garantizó la calidad del producto (software) obtenido en la primera iteración de desarrollo.

De manera general se puede concluir que los objetivos planteados en un inicio fueron alcanzados satisfactoriamente, desarrollándose una primera versión del sistema.

RECOMENDACIONES

Una vez concluido el proceso de desarrollo se recomienda llevar a cabo la construcción de una segunda versión donde se incorpore las funcionalidades:

- Importar Vista
- Crear Visor para Combo
- Editar Visor para Combo

Además que se realicen las tareas de diseño e implementación de un generador de interfaces que interprete los modelos creados y genere el código de aplicación necesario hacia una arquitectura específica.

REFERENCIAS BIBLIOGRAFICAS

1. **Toledo, Carlos Aimacaña.** *Interfaz de usuario.* Administración de bases de datos, Analista de Sistemas, Liderazgo de Proyecto, Soporte Técnico, Diseño Web. 2000.

2. **Gómez, Leopoldo Sebastián M.** Licenciado en Ciencias de la Computación (U.N.S.) – Perito Informático Oficial. Argentina : s.n.

3. **Visconti, Marcelo y Astudillo, Hernán.** *Fundamentos de Ingeniería de software.* Departamento de Informática Universidad Técnica Federico Santa María.

4. **Pérez, Yudid Fernández.** Monografías. *Enfoque Administrativo de la Ingeniería de Software.* [En línea] http://www.monografias.com/trabajos-pdf/enfoque-administrativo-ingenieria-software/enfoque-administrativo-ingenieria-software.pdf.

5. **Brown, Marcelo Pardo.** *Metamodelos.* Ing. Civil, Master en informática.

6. **Rodríguez, J. y García, J.** *Ingeniería de modelos con MDA.* Murcia, España : s.n., 2004. págs. http://www.dis.um.es/~jmolina/pfcs/proyecto-mda.pdf.

7. **Brown, Alan.** *An introduction to Model Driven Architecture.* pág. http://www.ibm.com/developerworks/rational/library/3100.html. Citado febrero 2011.

8. *Model-Driven Development in a lightweight way.* 2011. pág. http://www.openxava.org/web/guest/mdd.

9. **Kelly, Steven, Tolvanen, Juka-Pekka y Wiley, John.** *Domain Specific Languages.* 2008.

10. **Pastor, Oscar.** *Desarrollo Dirigido por Modelos.* Académico y Director del Centro de Investigación en Métodos de Producción de Software -PROS- de la Universidad Politécnica de Valencia, España., Profesor en la Universidad Politécnica de Valencia, España. Autor de más de 200 artículos publicados.

11. **Jacobson, I., Booch, G. y Rumbaugh, J.** *El Proceso Unificado de Desarrollo de Software.* Madrid : Pearson Educación : s.n., 2000.

12. *Reporte del Encuentro GeneXus.* 2009.

13. **Aranda, Andres y Rodríguez, Leidy Natalia González.** Grupo MDA. *Metodología MDA.* [En línea] [Citado el: 2 de marzo de 2011.] http://ingenieriasoftwaredos.wikispaces.com/.

14. **Ivar, Jacobson, James, Rumbaugh y Booch, Grady.** *El Proceso Unificado de desarrollo de Software.* 2000. pág. http://bibliodoc.uci.cu/pdf/reg00060.pdf.

15. **Colegtivo de profesores de la Universidad de las Ciencias Informáticas, EVA.** 2011. pág. http://eva.uci.cu/mod/resource/view.php?id=33748.

16. **UML, BPMN and Database Tools for Software Development.** Visual Paradigm for UML - UML tool for software application development. [En línea] 2005. [Citado el: 2011 de junio de 1.] http://www.visual-paradigm.com.

17. **Budinsky, Frank, y otros.** *Eclipse Modeling Framework.* Addison Wesley Professional. 2003.

18. **Abián, Miguel Angel.** *El Archipiélago Eclipse (Parte 3 de 4).* 2005. pág. http://www.javahispano.org/licencias/.

19. **Gallardo, David, Ed Burnette y Robert McGovern.** *Eclipse in Action: A Guide for Web Developers.* s.l. : Manning Publications, 2003.

20. **Collins-Sussman, Ben, W. Fitzpatrick, Brian y Pilato, C. Michael.** *Control de versiones con Subversion.* s.l. : Edición Primera, 2004. Este trabajo se publica bajo la licencia Creative Commons Attribution License. Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by/2.0/ o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305,.

21. **Scribd Inc.** *Patrones de Diseño.* [En línea] 2011. [Citado el: 1 de Junio de 2011.] http://es.scribd.com/doc/27239031/PATRONES-DE-DISENO#ad_unit=Doc_Sideboard_MediumRectangle_BTF_300x250&url=http%3A//es.scribd.com/doc/27239031/PATRONES-DE-DISENO%23ad_unit%3DDoc_Sideboard_MediumRectangle_BTF_300x250%26url%3Dhttp%253A//es.scribd.com/doc/27.

22. **Tedeschi, Nicolás.** *¿Qué es un Patrón de Diseño?* Analista de Sistemas de la Facultad de Ingeniería del Uruguay. 2011. Desde1995 trabaja con tecnologías Microsoft de tres capas. Es además Desarrollador 5 Estrellas de Microsoft MSDN y ha obtenido 3 Estrellas de ese programa. .

23. **Welicki, León.** *El Patrón Singleton.* Profesor Asociado de Ingeniería Web en el Máster en Ingeniería del Software, Universidad Pontificia de Salamanca. Madrid, España : s.n., 2011. Actualmente está realizando el Doctorado en Ingeniería Informática, su tesis doctoral trata sobre las Arquitecturas de Software y Paradigmas No Convencionales para Ingeniería Web. Trabaja como Arquitecto de Software. Cuenta con más de 12 años de experienc.

24. **Mora, Roberto Canales.** *Patrones de GRASP.* Ingeniero Técnico de Telecomunicaciones . 2003. Autor del Libro: Informática profesional, las reglas no escritas para triunfar en la empresa.

25. **Acevedo, Rodolfo Humberto Viallarroel.** *Académicos.* Doctor en Ingeniería Informática, Magister en Ingeniería Informática. , Universidad de Castilla La Mancha. España. 2011. pág. http://www.eici.ucm.cl/Academicos/R_Villarroel/descargas/ing_sw_1/Diseno_dominio_solucion.pdf.

26. **Informáticas, Colectivo de profesores de la Universidad de las Ciencias.** Entorno Virtual de Aprendizaje. Fase de Construcción. [En línea] 2011. [Citado el: 13 de junio de 2011.] http://eva.uci.cu/file.php/259/Curso_2010-2011/Semana_8/Conferencia_6/Materiales_Basicos/Implementacion.

27. **Arevalo, Maria Eugenia.** *Introducción al Patrón de Arquitectura por Capas.* 2010. págs. http://arevalomaria.wordpress.com/2010/12/02/introduccion-al-patron-de-arquitectura-por-capas/.

28. **Lago, Ramiro.** *Patrones de diseño software.* Jefe de Área de Desarrollo, Ministerio de Ciencia e Innovación, Fundación Española para la Ciencia y la Tecnología . 2007. págs. http://www.proactiva-calidad.com/java/patrones/index.html#algunos_patrones.

29. **Gamma, Erich, y otros.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley Professional illustrated edition, 1994.

30. **Guerrero, Luis A.** *Arquitectura física.* Santiago, Chile : s.n.

31. **Pressman, Roger S.** *Ingenieria del Software - Un Enfoque Practico.* McGraw-Hill Companies. 2002.

32. **Marañón Álvarez, Gonzalo.** *Características del lenguaje Java.* 2009.

33. **Beck, Kent.** XProgramming. [En línea] [Citado el: 20 de 5 de 2011.] http://www.xprogramming.com/testfram.htm.

34. **Mercovich, Eduardo.** *Ponencia sobre Diseño de Interfaces y Usabilidad.* Buenos Aires, Argentina : s.n., SIGGRAPH ’99.

35. **Suárez, Uriser Barbón y Machín Pérez, Edel.** *Diseño e implementación de la cartera de servicios del producto Cedrux sobre la plataforma de arquitectura distribuida basada en servicios Web.* Ciudad de La Habana : s.n., 2010. pág. 51.

36. **Ruiz Duran, Javier y González Marcaida, Eyonys.** *Diseño e implementación de una herramienta para la gestión de servicios Web sobre aplicaciones PHP.* Ciudad de La Habana : s.n., 2010.