Universidad de las Ciencias Informáticas

Desarrollo de una herramienta generadora de ficheros de mapeo, para la persistencia de objetos en esquemas relacionales basada en Doctrine.

Trabajo de Diploma para optar por el título de:

Ingeniero en Ciencias Informáticas

Autor: René R. Bauta Camejo

Tutor: Ing. Osmar Leyet Fernández

DEDICATORIA

A mis padres,

Por todo el amor, cariño, comprensión y dedicación;

Por todo el sacrificio que han hecho para que yo llegara a este momento.

A mi hermano,

Por mantener siempre en mi la chispa del conocimiento.

A mis hermanos, Yady y Pache,

Por haber estado conmigo en las buenas y malas;

Por cuidarme y entenderme.

A mi novia, Churri, por su comprensión, apoyo y por su amor;

Por haberle dedicado parte de su tiempo a este trabajo y a mí.

AGRADECIMIENTOS

A mi tutor, Osmar, gracias por todo hermano; más que tutor diría amigo y hermano. Este resultado también es tuyo. Gracias.

A mis colegas del equipo de Arquitectura. Rene nos enseñó que el resultado de uno era el resultado del equipo así que este trabajo también es el logro de esa gran familia que somos.

A René Lazo. Gracias por estar cada vez que te necesité. Eres ejemplo, te debo mucho de lo que he aprendido para mi vida, tanto profesional como personal. Siempre fuiste, más que profe, hermano y compañero. Gracias por tus enseñanzas.

A mis amigos de la vieja escuela. Gracias por responder cada vez que acudí a ustedes.

A todos los que están aquí presentes por haberme dedicado este tiempo.

Gracias a todos.

RESUMEN

El uso de framework para la persistencia de esquemas y de herramientas generadoras de ficheros de mapeo para estos se ha convertido en una práctica muy difundida en los equipos de desarrollo por lo que las herramientas de generación de código están ocupando un papel primordial en la producción de software. Con ellas se disminuye el tiempo en que se desarrolla un producto y las posibilidades de errores en la elaboración del mismo, además se incrementa la calidad del software creado y se facilita su mantenimiento. La Universidad de las Ciencias Informáticas (UCI) emplea en el desarrollo del producto Cedrux el framework de persistencia Doctrine. Este ORM (Object – Relational Mapping) genera algunos ficheros de mapeo que presentan una serie de deficiencias que atentan contra el tiempo de estimación para la implementación de la solución.

Este trabajo esta orientado a la creación de una herramienta capaz de corregir estas deficiencias y lograr la generación correcta y eficiente de los ficheros de mapeo necesarios para la persistencia de objetos relacionales mediante el uso de este ORM.

Luego de un análisis del estado del arte de las principales herramientas existentes para la generación de ficheros de mapeo, y basado en las deficiencias que presentan estas herramientas, se realizó el diseño de clases del sistema, dando respuesta al modelo de arquitectura seleccionado, obteniendo un sistema capaz de realizar la generación de estos ficheros, proponiéndolo como herramienta de generación para el uso de Doctrine como framework de persistencia en el desarrollo del producto Cedrux.

A raíz de esta propuesta, el mayor impacto radica en el logro de un sistema que garantice en mayor medida al equipo de desarrollo una eficiente generación de los ficheros necesarios para la persistencia mediante el ORM Doctrine.

Palabras Claves: Esquemas de persistencia, framework Doctrine, ficheros de mapeo, herramienta, generación de código, ORM.

ÍNDICE DE CONTENIDOS

INTRODUCCIÓN....................1

CAPÍTULO 1: Fundamentación Teórica....................7

1.1 Introducción....................7

1.2 Diseño de Software....................7

1.2.1 Evolución del diseño de software....................8

1.2.2 Principios de diseño....................9

1.3 Microsoft .NET....................11

1.3.1 Características de la plataforma .NET....................13

1.3.2 Visual Studio.NET....................14

1.3.3 .NET Framework....................16

1.4 Base de Datos Relacionales....................17

1.4.1 Base de Datos (BD)....................17

1.4.2 Características....................18

1.4.3 Modelo de Datos....................19

1.4.4 Modelo Relacional....................19

1.4.5 Base de Datos Relacional (BDR)....................20

1.5 Framework de Persistencia de Objetos Relacionales....................21

1.5.1 Framework....................21

1.5.2 Persistencia....................22

1.5.3 Esquemas de persistencia....................23

1.5.4 Framework de persistencia....................25

1.5.5 Framework de persistencia más usados....................26

1.6 Framework de Persistencia de Objetos Relacionales Doctrine (29)....................29

1.6.1 ¿Qué es Doctrine?....................29

1.6.2 Principales características de Doctrine....................29

1.6.3 Ventajas y desventajas de Doctrine....................31

1.7 Generadores de Ficheros de Mapeo de Objetos Relacionales....................32

1.7.1 ¿Qué es un generador de código?....................32

1.7.2 Tipos de generadores de código....................32

1.7.3 Funcionamiento de los generadores de código....................35

1.7.4 Técnicas de generación de código....................36

1.7.5 Ventajas de la generación de código....................37

1.7.6 Desventajas de la generación de código....................38

1.8 Selección de las Herramientas y Tecnologías a utilizar....................39

1.8.1 Metodologías de Desarrollo....................39

1.8.2 Herramientas CASE (35)....................44

1.8.3 Microsoft Visual C# .NET....................47

1.9 Conclusiones 50

CAPÍTULO 2: Descripción de la Solución Propuesta 51

2.1 Introducción 51

2.2 Descripción de la Solución (Doctrine Generator) 51

2.3 Arquitectura Base 53

2.3.1 Estilo arquitectónico empleado en la solución 53
2.3.2 Atributos de calidad que validan la arquitectura. 56
2.3.3 Vista vertical de la arquitectura de la solución propuesta. 58
2.3.4 Tecnología utilizada 59

2.4 Patrones 60

2.4.1 Patrones de Diseño 61
2.4.2 Patrones utilizados en la solución 62

2.5 Disciplina Modelo definida por AUP. Artefacto Modelo de Diseño 65

2.5.1 Definición del Modelo de Diseño. 66
2.5.2 Diseño de Clases 66
2.5.3 Descripción de las principales clases utilizadas en la solución 66
2.5.4 Paquetes de Clases del Diseño 73
2.5.5 Diagrama de Clases 73
2.5.6 Diagramas de Interacción 76
2.5.7 Diagrama de Componentes 79
2.5.8 Diagrama de Despliegue 80

2.6 Modelo de Implementación 81

2.7 Conclusiones 82

CAPÍTULO 3: Evaluación de la Solución Propuesta 83

3.1 Introducción 83

3.2 Métricas de validación para el diseño. 83

3.2.1 Consideraciones en la etapa del diseño 83

3.3 Métricas Orientadas a Objeto 85

3.3.1 Algunas características de las métricas orientadas a objeto (4) 85

3.4 Métricas Orientadas a Clases 87

3.5 Evaluación del modelo de diseño. 87

3.6 Métricas aplicadas a la solución propuesta 89

3.6.1 Árbol de Profundidad de Herencia (APH). 91
3.6.2 Número de Descendientes (NDD). 92
3.6.3 Tamaño de Clases (TC). 93
3.6.4 Número de Operaciones Redefinidas para una Sub-Clase (NOR). 96

3.7 Matriz de cubrimiento de los parámetros de calidad evaluados con las métricas propuestas. 96

3.8 Pruebas realizadas a la solución propuesta 98

3.8.1 Pruebas de Usuario 99

3.8.2 Casos de Prueba 100

3.9 Conclusiones 107

CONCLUSIONES GENERALES 108

RECOMENDACIONES 109

BIBLIOGRAFÍA 110

GLOSARIO DE TÉRMINOS 114

ANEXOS 119

ÍNDICE DE ILUSTRACIONES

Figura 1 Visual Studio.NET (8) 13
Figura 2 Arquitectura del Framework .NET 16
Figura 3 Esquema de Persistencia. (23) 25
Figura 4 Estructura del framework Doctrine (29) 30
Figura 5 Ejemplo de programación orientada a objetos 31
Figura 6 Funcionamiento de un generador de código 36
Figura 7 Estructura de carpetas generada por Donctrine Generator 52
Figura 8 Separación lógica en capas 54
Figura 9 Arquitectura de la solución propuesta 59
Figura 10 Diagrama de clases parcial 64
Figura 11 Estructura del patrón Singlenton. 65
Figura 12 Paquetes en que se agrupan las clases del diseño 73
Figura 13 Diagrama de Clases de las clases entidades 75
Figura 14 Diagrama de Clases para las clases generadoras 75
Figura 15 Diagrama de Colaboración para el escenario Adicionar Tabla del CU Gestionar Tablas 77
Figura 16 Diagrama de Colaboración para el escenario Eliminar Tabla del CU Gestionar Tablas 77
Figura 17 Diagrama de Secuencia para el escenario Adicionar Tabla del CU Gestionar Tablas 78
Figura 18 Diagrama de Secuencia para el escenario Eliminar Tabla del CU Gestionar Tablas 79
Figura 19 Diagrama de Componentes para la solución propuesta 80
Figura 20 Diagrama de Despliegue 81
Figura 21 Niveles de herencia existentes en el diseño propuesto 92
Figura 22 Representación de las clases de la capa de negocio según cantidad de operaciones 94
Figura 23 Representación de los resultados obtenidos en el instrumento agrupados en los intervalos definidos 95
Figura 24 Representación en % de los resultados obtenidos en el instrumento agrupados en los intervalos definidos 96
Figura 25 Impacto de los atributos de calidad en el diseño de la solución propuesta 98
Figura 26 Diagrama de Colaboración: Caso de Uso Generar Ficheros de Mapeo 123
Figura 27 Diagrama de Secuencia: Caso de Uso Generar Ficheros de Mapeo 123
Figura 28 Diagrama de Colaboración: Caso de Uso Gestionar ConsultaDQL. Escenario Adicionar ConsultaDQL 124
Figura 29 Diagrama de Colaboración: Caso de Uso Gestionar ConsultaDQL. Escenario Eliminar ConsultaDQL 124
Figura 30 Diagrama de Secuencia: Caso de Uso Gestionar ConsultaDQL. Escenario Adicionar ConsultaDQL 125
Figura 31 Diagrama de Secuencia: Caso de Uso Gestionar ConsultaDQL. Escenario Eliminar ConsultaDQL 125
Figura 32 Diagrama de Colaboración: Caso de Uso Gestionar Relación. Escenario Adicionar Relación 126
Figura 33 Diagrama de Colaboración: Caso de Uso Gestionar Relación. Escenario Eliminar Relación 126
Figura 34 Diagrama de Colaboración: Caso de Uso Gestionar Relación. Escenario Modificar Tipo Relación 127
Figura 35 Diagrama de Secuencia: Caso de Uso Gestionar Relación. Escenario Adicionar Relación 127
Figura 36 Diagrama de Secuencia: Caso de Uso Gestionar Relación. Escenario Eliminar Relación 128
Figura 37 Diagrama de Secuencia: Caso de Uso Gestionar Relación. Escenario Modificar Tipo Relación 128
Figura 38 Diagrama de Colaboración: Caso de Uso Gestionar Fichero de Esquemas. Escenario Adicionar Esquema 129
Figura 39 Diagrama de Colaboración: Caso de Uso Gestionar Fichero de Esquemas. Escenario Eliminar Esquema 129
Figura 40 Diagrama de Secuencia: Caso de Uso Gestionar Fichero de Esquemas. Escenario Adicionar Esquema 130
Figura 41 Diagrama de Secuencia: Caso de Uso Gestionar Fichero de Esquemas. Escenario Eliminar Esquema 130
Figura 42 Árbol de herencia para las superclases Generador y Relacion 133

ÍNDICE DE TABLAS

Tabla 1 Precio de licencias de TierDeveloper. 26
Tabla 2 Precio por licencia de CodeCharge. 27
Tabla 3 Precio por licencia de Visual Paradigm. 28
Tabla 4 Descripción de atributos de calidad observables vía ejecución (4) 56
Tabla 5 Descripción de atributos de calidad no observables vía ejecución (4) 57
Tabla 6 Aplicación del patrón GRASP Experto en la solución 62
Tabla 7 Descripción de la Clase BaseDatos 67
Tabla 8 Descripción de la Clase Esquema 67
Tabla 9 Descripción de la Clase Tabla 68
Tabla 10 Descripción de la Clase CampoTabla 68
Tabla 11 Descripción de la Clase Relacion 69
Tabla 12 Descripción de la Clase RelacionM_M 69
Tabla 13 Descripción de la Clase Proyecto 69
Tabla 14 Descripción de la Clase Fichero 70
Tabla 15 Descripción de la Clase Generador 70
Tabla 16 Descripción de la Clase GeneradorDomain 71
Tabla 17 Descripción de la Clase GeneradorDomainG 71
Tabla 18 Descripción de la Clase GeneradorBussines 72
Tabla 19 Descripción de la Clase Cargador 72
Tabla 20 Descripción de la Clase ManagerExcepcion 73
Tabla 21 Cumplimiento del atributo de calidad Funcionalidad de la norma ISO 9126 88
Tabla 22 Árbol de Profundidad de Herencia 90
Tabla 23 Número de Descendientes 90
Tabla 24 Tamaño de Clase 91
Tabla 25 Número de Descendientes 91
Tabla 26 Resultados para el parámetro de calidad Reutilización 93
Tabla 27 Resultado para el parámetro de calidad Abstracción 93
Tabla 28 Resultado para el parámetro de calidad Cohesión 93
Tabla 29 Resultado para el parámetro de calidad Cantidad de Pruebas 93
Tabla 30 Distribución de valores de tamaño de las clases de la capa de negocio 95
Tabla 31 Clasificación de los parámetros de calidad según impacto en el diseño propuesto 97
Tabla 32 Rangos para evaluar el impacto de los parámetros de calidad en el diseño propuesto 97
Tabla 33 Matriz de cubrimiento para los parámetros de calidad evaluados con las métricas aplicadas al diseño propuesto . 97
Tabla 34 Requisitos a probar del CP Crear Nuevo Proyecto 101
Tabla 35 Juego de datos para probar el del CP Crear Nuevo Proyecto 102
Tabla 36 Requisitos a probar del CP Abrir Proyecto Existente 102
Tabla 37 Juego de datos para probar el del CP Abrir Proyecto Existente 103
Tabla 38 Requisitos a probar del CP Generar Ficheros de Mapeo 103
Tabla 39 Juego de datos para probar el del CP Generar Ficheros de Mapeo 103
Tabla 40 Requisitos a probar del CP Construir ConsultaDQL 104
Tabla 42 Requisitos a probar del CP Adicionar Relación 105
Tabla 43 Juego de datos para probar el del CP Adicionar Relación 105

Tabla 44 Requisitos a probar del CP Modificar Relación *106*
Tabla 45 Juego de datos para probar el del CP Modificar Relación *106*
Tabla 46 Requisitos a probar del CP Eliminar Relación *106*
Tabla 47 Juego de datos para probar el del CP Eliminar Relación *107*
Tabla 48 Umbrales para el parámetro de calidad Reutilización *132*
Tabla 49 Umbrales para el parámetro de calidad Abstracción *132*
Tabla 50 Umbrales para el parámetro de calidad Cohesión *132*
Tabla 51 Umbrales para el parámetro de calidad Cantidad de Pruebas *132*
Tabla 52 Rango de valores de para la evaluación técnica de los atributos de calidad (Reutilización, Abstracción del diseño, Nivel de Cohesión y Cantidad de Pruebas) *132*
Tabla 53 Resultados de la evaluación de la métrica TOC y su influencia en los atributos de calidad (Responsabilidad, Complejidad de Implementación y Reutilización) *134*

INTRODUCCIÓN

Actualidad y necesidad del trabajo.

Las empresas desarrolladoras de software sea cual sea su tamaño, capital o presencia en el mercado persiguen un objetivo común: "desarrollar software de calidad a un costo y en un tiempo adecuados". Este es un reto difícil de lograr si se tiene en cuenta que las demandas de los clientes en muchos casos son superiores a las capacidades productivas de las empresas y que la producción de software de forma industrial es un mito que solo pocas entidades pueden respaldar. La solución a este conflicto se ha buscado en todas las direcciones, se han perfeccionado los procesos y las metodologías de desarrollo, se trabaja fuertemente en la capacitación de los recursos humanos y constantemente se desarrollan nuevas tecnologías y framework.

Precisamente la creación de framework que colaboren en el proceso de desarrollo de un software es una de las direcciones en las que más se ha trabajado y que ha reportado beneficios considerables en cuanto a reducción de tiempos y errores durante la fase de implementación en el proceso de desarrollo. Sin lugar a dudas la mejor opción siempre sería la de contar con una herramienta propia que se adecue a las necesidades concretas de un proyecto y que pueda ser modificada para corregir errores o adaptarse a los cambios de las tecnologías.

Los generadores de código son un ejemplo de herramientas que se han puesto en función de agilizar el proceso de desarrollo pero su construcción no es un proceso trivial. En el funcionamiento de un generador, aspectos claves como la carga del modelo de datos y los mecanismos de generación requieren de mucho esfuerzo por parte de los programadores y se cae en el riesgo de que una vez terminado no cumpla con las expectativas de los futuros usuarios.

En la actualidad, una práctica muy difundida entre los equipos de desarrollo para lograr agilizar el proceso de desarrollo es el uso de framework para la persistencia de esquemas en distintas plataformas. Estos framework conocidos como ORM, constituyen otro ejemplo de cuanto se ha avanzado en el desarrollo de herramientas que favorezcan el proceso de creación de un software. Los ORM permiten trabajar con los datos persistidos como si ellos fueran parte de una base de datos orientada a objetos virtual. Las bases de datos relacionales están pensadas para manejar conjuntos de datos, sin embargo los lenguajes orientados a objetos manejan objetos individuales, o listas (tuplas/arrays) de objetos, que suelen ser más complejos que los datos almacenados en las tablas de una base de datos relacional por lo que los ORM hacen función de adaptador entre la base de datos relacional y el modelo de objetos que maneja una aplicación.

Utilizar un ORM ofrece muchas ventajas entre las que se pueden mencionar las siguientes:

- ✓ Persistencia transparente, es decir, los objetos del dominio no saben nada acerca de la base de datos donde son persistidos, el framework lo resuelve en forma automática utilizando archivos de mapeo.
- ✓ Soporte de polimorfismo: Posibilita cargar jerarquías de objetos en forma polimórfica.
- ✓ Soporte de los 3 niveles de mapeo de herencia: Permite mapear toda una jerarquía de clases a una sola tabla, crear una tabla por cada clase concreta o crear una tabla por cada escalón de la jerarquía.
- ✓ Soporte completo de asociaciones: Soportan el mapeo de todos los tipos de relaciones que pueden existir en un modelo de objetos del dominio (asociaciones 1...1, 1...N, N...M, unidireccionales y bidireccionales).
- ✓ Soporte de carga de objetos Proxy: Es posible cargar objetos que solo contengan la clave del objeto completo.
- ✓ Soporte de múltiples dialectos SQL: Posibilita independencia respecto al tipo de base de datos utilizada. La aplicación puede persistir sus datos en SQL Server, en Oracle o en MySQL, simplemente cambiando la configuración correspondiente.

La UCI, no está ajena al uso de estos framework y en varios de sus proyectos productivos hacen uso de ellos para realizar la persistencia de la información en bases de datos relacionales. En el desarrollo del producto Cedrux se emplea el lenguaje PHP. Para este lenguaje existen varios framework para la persistencia de objetos en esquemas relacionales, entre los que podemos citar a PDO y Doctrine; siendo este último el que se emplea para estos fines en el desarrollo del producto antes mencionado.

El ORM Doctrine genera un grupo de ficheros .PHP que garantizan de cierta forma el mapeo del esquema relacional con el que se necesite interactuar pero que a su vez no cubre todas las necesidades de los equipos de desarrollo de este proyecto ya que presentan una serie de dificultades en cuanto a las salidas de los ficheros de mapeo generados tales como la ausencia de las relaciones existentes entre las tablas, la ausencia de la secuencia en la generación de los campos que son llaves primarias en la Base de Datos y que la contienen y la imposibilidad de personalización del mapeo, que luego tienen que ser necesariamente corregidas por los desarrolladores de forma manual; proceso tedioso y que afecta el tiempo de estimación para la implementación de las soluciones.

Por lo cual se impone la creación de una nueva herramienta capaz de satisfacer las necesidades parcialmente o en su totalidad de los equipos de desarrollo, en la generación de los ficheros de mapeo, para la persistencia de datos mediante el framework Doctrine, estandarizando su uso en el proyecto que desarrolla el producto Cedrux.

Situación Problemática

Los equipos de desarrollo que están implementando la solución Cedrux que necesitan acceso a la base de datos relacional del proyecto, usando el framework de persistencia Doctrine, están teniendo dificultades a la hora de utilizar los ficheros de mapeo generados por este framework, debido a las dificultades en cuanto a las salidas que presentan estos ficheros; teniendo luego que personalizar los mismos de forma manual, proceso engorroso y lento que atenta contra las estimaciones de tiempo hechas para la implementación de la solución. Se desea desarrollar una herramienta que corrija estas dificultades y que permita a los equipos de desarrollo la personalización de estos ficheros.

Atendiendo a lo anteriormente descrito el **Problema de la Investigación** se formula de la siguiente forma:

La generación de los ficheros de mapeo para la persistencia de objetos en esquemas relacionales mediante el framework Doctrine, introduce errores en la implementación de las soluciones del producto Cedrux.

Objeto de Estudio

Persistencia de objetos en esquemas de Base de Datos Relacionales.

Para dar solución a esta problemática se determinó el siguiente **Objetivo General:**

Desarrollar una herramienta capaz de generar los ficheros de persistencia de objetos en esquemas relacionales mediante el framework Doctrine, para evitar la introducción de errores en la implementación de las soluciones del producto Cedrux.

Del Objetivo General se desglosan los siguientes **Objetivos Específicos:**

- ✓ Elaborar del marco teórico de la investigación.
- ✓ Desarrollar la aplicación propuesta.
- ✓ Analizar los resultados obtenidos de la validación del diseño propuesto y de la calidad del sistema.

Campo de Acción

Herramientas para la generación de ficheros de mapeo en Base de Datos Relacionales.

La investigación cuenta con la siguiente **Hipótesis:**

Si se realiza el diseño e implementación de una herramienta generadora de los ficheros de mapeo para el framework Doctrine, entonces se evita la introducción de errores en la implementación de las soluciones del producto Cedrux.

En correspondencia con el objetivo se establecieron las siguientes **Tareas:**

- ✓ Estudio del estado del arte de las principales herramientas y frameworks relacionados con la investigación.
- ✓ Selección de la metodología y herramientas para el desarrollo de la aplicación.
- ✓ Desarrollo el modelo de diseño del sistema.
- ✓ Desarrollo del modelo de implementación del sistema.
- ✓ Diseño e implementación de los algoritmos a usar en la aplicación.
- ✓ Aplicación de métricas para validar el diseño propuesto.
- ✓ Evaluación de los resultados arrojados por las métricas aplicadas al diseño.
- ✓ Realización de pruebas de aceptación a la aplicación desarrollada.
- ✓ Evaluación de los resultados arrojados por las pruebas de aceptación realizadas a la aplicación desarrollada.

Métodos Científicos

Métodos Teóricos

Histórico – Lógico
Está vinculado al conocimiento de las distintas etapas de los objetos en su sucesión cronológica. Para conocer la evolución y desarrollo del objeto o fenómeno de investigación se hace necesario revelar su historia, las etapas principales de su desenvolvimiento y las conexiones históricas fundamentales. (1)

Mediante este método científico, se realizó un estudio del estado del arte, sobre los principales framework para la persistencia de esquemas que se utilizan en la actualidad, así como de las herramientas generadoras de los ficheros de mapeo para la realización de esta persistencia de esquemas. Permitió analizar la evolución de estas herramientas en materia de arquitectura y diseño, y cuál es la tendencia actual.

Analítico – Sintético
El uso de este método permite distinguir los elementos de un fenómeno y se procede a revisar ordenadamente cada uno de ellos por separado. Consiste en la extracción de las partes de un todo, con el objeto de estudiarlas y examinarlas por separado. Estas operaciones no existen independientes una de la otra; el análisis de un objeto se realiza a partir de la relación que existe entre los elementos que conforman dicho objeto como un todo; y a su vez, la síntesis se produce sobre la base de los resultados previos del análisis. (2)

El uso del método científico analítico – sintético permitió realizar un estudio por separado de cada una de las herramientas generadoras de ficheros de mapeos para la persistencia de esquemas que más se utilizan en la actualidad, se definió que particularidades presentaban en común y se estableció una series de parámetros, atendiendo principalmente a las características relacionadas con sus objetivos fundamentales, para establecer una comparación entre ellas y tomar los resultados arrojados por dicha comparación, como datos de gran interés para la actual investigación.

Inducción – Deducción
Se estudian los caracteres y conexiones necesarios del objeto de investigación, relaciones de causalidad, entre otros. Este método se apoya en métodos empíricos como la observación y la experimentación.

Mediante la aplicación del mismo se desarrolló un estudio con los principales generadores de ficheros de mapeo para la persistencia de esquemas para PHP, revisando sus características propias, y basado en estas características se definieron, características o cualidades que debe tener o cumplir el sistema que se propone en el presente trabajo.

Métodos Empíricos

Observación
Mediante el método científico de la observación, se prestó atención a la situación actual existente en el proyecto ERP – Cuba, en cuanto a las dificultades existentes a la hora de generar los ficheros de mapeo para la persistencia de esquemas mediante el framework Doctrine, así como la necesidad de

creación de una herramienta con este objetivo, para corregir la mayor cantidad de estas dificultades como fuese posible.

Para una mejor comprensión del trabajo, el mismo se estructuró de la siguiente manera:

Un Capítulo 1 en el que se realiza la fundamentación teórica del trabajo, el mismo incluye un estudio del arte del tema tratado a nivel internacional, nacional y de la Universidad, de las tendencias, técnicas, tecnologías, metodologías y software usados en la actualidad. Además se hacen algunas consideraciones sobre el flujo de trabajo relacionado específicamente con el diseño.

Un Capítulo 2 en el que se describe la solución propuesta haciendo referencia a la arquitectura base, estilo arquitectónico y patrones de diseño empleados en la solución así como algunas especificaciones propias de los flujos de trabajo de Diseño e Implementación definidos por la metodología de desarrollo de software seleccionada.

Y finalmente un Capítulo 3 que se enfoca a la evaluación de los resultados obtenidos. En el se abordan algunas métricas de diseño aplicadas a la solución así como los resultados de algunas pruebas realizadas a la herramienta.

CAPÍTULO 1: Fundamentación Teórica

1.1 Introducción

En este capítulo se hace referencia al uso de framework de persistencia de Objetos Relacionales para resolver la lógica de la persistencia de los datos, se exponen brevemente algunas de las principales características de estos framework. Se explica que es y en qué consiste Doctrine para la persistencia de los objetos relacionales. Se realiza un análisis de algunas de las herramientas que actualmente se usan para la generación de los ficheros de mapeo para los esquemas de persistencia. Se mencionan además algunas consideraciones sobre el flujo de trabajo relacionado específicamente con el diseño según los puntos de vista de algunas de las principales metodologías de desarrollo, tendencias y tecnologías actuales para este entorno de negocio, así como su evolución.

1.2 Diseño de Software

El diseño es la primera etapa técnica del proceso de ingeniería del software, es iterativo y consiste en producir una representación técnica del software que se va a desarrollar. Sobre el diseño se asienta la calidad del software. Se representa a un alto nivel de abstracción, un nivel que se puede seguir hasta requisitos específicos de datos, funcionales y de comportamiento.

Con el diseño se pretende construir un sistema que:

- ✓ Satisfaga determinada especificación del sistema.
- ✓ Se ajuste a las limitaciones impuestas por el medio de destino.
- ✓ Respete requisitos sobre forma, rendimiento utilización de recursos y coste.
- ✓ El diseño es la primera etapa técnica del proceso de Ingeniería del Software, consiste en producir un modelo o representación técnica del software que se va a desarrollar.
- ✓ El diseño es el proceso sobre el que se asienta la calidad del software.
- ✓ El diseño de software es un proceso iterativo a través del cual se traducen los requisitos en una representación del software. (3)

El diseño del software es el proceso de aplicar técnicas y principios para concebir un producto con suficientes detalles como para permitir su interpretación y realización física. La etapa del diseño del sistema encierra cuatro etapas:

- ✓ Diseño de los datos: Trasforma el modelo de dominio de la información, creado durante el análisis, en las estructuras de datos necesarios para implementar el Software.
- ✓ Diseño Arquitectónico: Define la relación entre cada uno de los elementos estructurales del programa.
- ✓ Diseño de la Interfaz: Describe como se comunica el Software consigo mismo, con los sistemas que operan junto con el y con los operadores y usuarios que lo emplean.
- ✓ Diseño de procedimientos: Transforman los elementos estructurales de un programa en una descripción procedimental del software.

1.2.1 Evolución del diseño de software

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

Hoy en día, se ha hecho hincapié en un diseño de software basado en la arquitectura del software. La programación orientada a objetos, como paradigma es una filosofía de la que surge una cultura nueva que incorpora técnicas y metodologías diferentes, en ella el universo computacional está poblado por objetos, cada uno responsable de sí mismo, y comunicándose con los demás por medio de mensajes. Cada objeto representa una instancia de alguna clase, y estas clases son miembros de una jerarquía de clases unidas vía relaciones de herencia.

La evolución histórica de los paradigmas de programación se podría presentar de la siguiente forma:

- ✓ Programación estructurada.
- ✓ Programación modular.
- ✓ Programación orientada a objetos.
- ✓ Programación orientada a aspectos.

- ✓ Programación orientada a componentes.

Independientemente del modelo de diseño que se utilice, un ingeniero del software deberá aplicar un conjunto de principios fundamentales y conceptos básicos para el diseño a nivel de componentes, de interfaz, arquitectónico y de datos. (4)

1.2.2 Principios de diseño

A lo largo del diseño se evalúa la calidad del desarrollo del proyecto con un conjunto de revisiones técnicas. Este debe implementar todos los requisitos explícitos contenidos en el modelo de análisis y debe acumular todos los requisitos implícitos que desea el cliente. Debe ser una guía que puedan leer y entender los que construyan el código y los que prueban y mantienen el software. Además debe proporcionar una completa idea de lo que es el software, enfocando los dominios de datos, funcional y comportamiento desde el punto de vista de la Implementación.

La importancia del diseño del software se puede definir en una sola palabra, calidad. Dentro del diseño es donde se fomenta la calidad del proyecto y este es la única manera de materializar con precisión los requerimientos del cliente.

Para evaluar la calidad de una presentación del diseño, se deben establecer criterios técnicos para un buen diseño como son:

- ✓ Debe presentar una organización jerárquica que haga un uso inteligente del control entre los componentes del software.
- ✓ Debe ser modular, es decir, se debe hacer una partición lógica del software en elementos que realicen funciones y sub funciones específicas.
- ✓ Debe producir módulos que presenten características de funcionamiento independiente.
- ✓ Debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y el entorno exterior.
- ✓ Debe producir un diseño usando un método que pudiera repetirse según la información obtenida durante el análisis de requisitos de software.

Estos criterios no se consiguen por casualidad. El proceso de diseño del software exige buena calidad a través de la aplicación de principios fundamentales de diseño, metodología sistemática y una revisión

exhaustiva. Cuando se va a diseñar un sistema se debe tener presente que el proceso de un diseño incluye, concebir y planear algo en la mente, así como hacer un modelo o croquis.

Los principios básicos de diseño hacen posible que el ingeniero del software navegue por el proceso de diseño. El proceso de diseño no es infalible y absoluto, un buen diseñador deberá tener en cuenta enfoques alternativos, juzgando todos los que se basan en los requisitos del problema, los recursos disponibles para realizar el trabajo y los conceptos de diseño. (5)

El diseño no deberá inventar nada que ya esté inventado. Los sistemas se construyen utilizando un conjunto de patrones de diseño, muchos de los cuales probablemente ya se han encontrado antes. Estos patrones deberán elegirse siempre como una alternativa para reinventar. Hay poco tiempo y los recursos son limitados. El tiempo de diseño se deberá invertir en la representación verdadera de ideas nuevas y en la integración de esos patrones que ya existen. (5)

El diseño deberá minimizar la distancia intelectual entre el software y el problema como si de la misma vida real se tratara. Es decir, la estructura del diseño del software (siempre que sea posible) imita la estructura del dominio del problema. (6) (7)

El diseño deberá presentar uniformidad e integración. Un diseño es uniforme si parece que fue una persona la que lo desarrolló por completo. Las reglas de estilo y de formato deberán definirse para un equipo de diseño antes de comenzar el trabajo sobre el diseño. Un diseño se integra si se tiene cuidado a la hora de definir interfaces entre los componentes del diseño. (6) (7)

El diseño deberá estructurarse para admitir cambios. Los conceptos de diseño estudiados hacen posible un diseño que logra este principio. (6) (7)

El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes. Un software bien diseñado no deberá nunca explotar, deberá diseñarse para adaptarse a circunstancias inusuales y si debe terminar de funcionar, que lo haga de forma suave. (4) (8)

El diseño no es escribir código y escribir código no es diseñar. Incluso cuando se crean diseños procedimentales para componentes de programas, el nivel de abstracción del modelo de diseño es mayor que el código fuente. Las únicas decisiones de diseño realizadas a nivel de codificación se enfrentan con pequeños datos de implementación que posibilitan codificar el diseño procedimental. (4) (8)

El diseño deberá evaluarse en función de la calidad mientras se va creando, no después de terminarlo. Para ayudar al diseñador en la evaluación de la calidad se dispone de conceptos de diseño y de medidas de diseño. (4) (8)

El diseño deberá revisarse para minimizar los errores conceptuales. A veces existe la tendencia de centrarse en minucias cuando se revisa el diseño, olvidándose del bosque por culpa de los árboles. Un equipo de diseñadores deberá asegurarse de haber afrontado los elementos conceptuales principales antes de preocuparse por la sintaxis del modelo del diseño. (4) (8)

1.3 Microsoft .NET

.NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, independiente de plataforma de hardware y que permita un rápido desarrollo de aplicaciones. Basado en ella, la compañía intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el sistema operativo hasta las herramientas de mercado.

Durante la conferencia para desarrolladores de software profesionales llamada PDC (Professional Developers Conference) celebrada por Microsoft en Orlando en Julio de 2000, se presentó la nueva apuesta de futuro de esta compañía para el desarrollo de software: la plataforma .NET. (4)

.NET podría considerarse una respuesta de Microsoft al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de Sun Microsystems (8) y a los diversos framework de desarrollo web basados en PHP. Su propuesta es ofrecer una manera rápida y económica, a la vez que segura y robusta, de desarrollar aplicaciones – o como la misma plataforma las denomina, soluciones – permitiendo una integración más rápida y ágil entre empresas y un acceso más simple y universal a todo tipo de información trascendiendo los limites de los dispositivos y fortaleciendo la conectividad de internet en sus aplicaciones (9). Con .NET el programador no trabaja directamente contra un sistema operativo concreto como Windows, sino que lo hace frente a una máquina virtual (el CLR o Common Language Runtime) que le ofrece los servicios que ante le proporcionaba el sistema operativo de forma más simplificada y adecuada a los tiempos actuales. Además, es viable para todas sus aplicaciones, ya que no necesita pensar en dos infraestructuras separadas—una para aplicaciones Web y otra para aplicaciones internas o de escritorio. La Plataforma .NET permite la creación y uso de servicios de aplicaciones, procesos y sitios Web basados en XML, que compartan y combinen información y funcionalidad entre ellos por diseño, en cualquier plataforma o dispositivo inteligente, para proveer soluciones a la medida para empresas e individuos. Incluye una

familia de productos integral, construida en estándares de la industria y de Internet, que provee servicios Web XML para cada aspecto del desarrollo (herramientas), administración (servers), uso (bloques ensamblables y clientes inteligentes) y experiencias (experiencias de usuario).

La plataforma .NET de Microsoft está diseñada para que se puedan desarrollar componentes software utilizando casi cualquier lenguaje de programación, de forma que lo que escribamos en un lenguaje pueda utilizarse desde cualquier otro de la manera más transparente posible (utilizando servicios web como middleware). En vez de limitar el uso de un único lenguaje de programación, permite cualquier lenguaje de programación, siempre y cuando se adhiera a unas normas comunes establecidas para la plataforma .NET en su conjunto. Existen compiladores de múltiples lenguajes para la plataforma .NET: Visual Basic .NET, C#, Managed C++, Oberon, Component Pascal, Eiffel, Smalltalk, Cobol, Fortran, Scheme, Mercury, Mondrian/Haskell, Perl, Python, SML.NET (9). La plataforma .NET permite utilizar una amplia gama de lenguajes de programación:

C#: Un nuevo lenguaje creado para la plataforma .NET. Se puede considerar una versión "segura" de C++. Es un lenguaje de programación orientado a objetos que pretende facilitar el desarrollo de componentes software robusto y duradero.

Visual Basic .NET: Moderniza y simplifica el lenguaje de programación Visual Basic, con algunas novedades sintácticas, herencia simple, tratamiento de hebras y manejo de excepciones.

Este conjunto de nuevas tecnologías podrán resumirse en las siguientes:

- ✓ Plataforma .NET.
- ✓ SDK de la plataforma .NET.
- ✓ Visual Studio.NET.
- ✓ Servicios Web.
- ✓ Servidores para empresas.

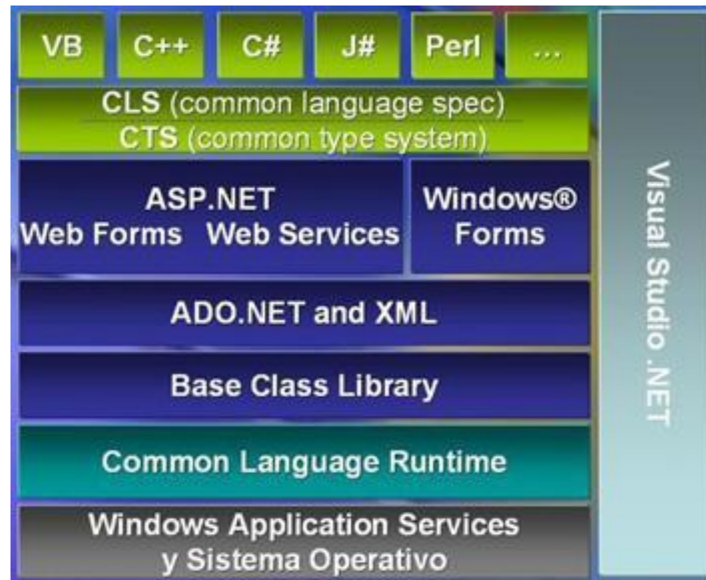


Figura 1 Visual Studio.NET (8)

1.3.1 Características de la plataforma .NET

Para resumir las características de la plataforma .NET basta con enumerar los servicios que el Common Language Runtime o CLR por sus siglas en ingles brinda. Este es el lenguaje insignia de .NET Framework (marco de trabajo .NET) y pretende reunir las ventajas de lenguajes como C, C++ y Visual Basic en uno solo. El CLR es el verdadero núcleo del framework de .NET, entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios del sistema operativo.

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

Para ejecutarse se necesita un segundo paso, un compilador JIT (Just-In-Time) es el que genera el código máquina real que se ejecuta en la plataforma del cliente. De esta forma se consigue con .NET independencia de la plataforma de hardware. La compilación JIT la realiza el CLR a medida que el programa invoca métodos. El código ejecutable obtenido se almacena en la memoria caché del ordenador, siendo recompilado de nuevo sólo en el caso de producirse algún cambio en el código fuente.

El CLR es el encargado de proveer lo que se llama co – administrado, es decir, un entorno que provee servicios automáticos al código que se ejecuta. Entre los servicios que brinda podemos mencionar:

- ✓ Sencillo modelo de programación.
- ✓ Tratamiento homogéneo de errores mediante excepciones.
- ✓ Desarrollo interlenguaje.
- ✓ Ejecución multiplataforma.
- ✓ Gestión automática de memoria con recolección de basura.
- ✓ Aislamiento de procesos.
- ✓ Soporte multihilo.
- ✓ Seguridad avanzada basada en el usuario y la procedencia del código.
- ✓ Interoperabilidad con código antiguo.
- ✓ Pone fin al infierno de las DLL. (10) (9)

1.3.2 Visual Studio.NET

"Visual Studio.Net es la culminación no sólo de una fenomenal herramienta de desarrollo, sino también de una plataforma de servicios que a partir de ahora, será componente fundamental de las futuras versiones de Windows"[1]. (11)

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

Es un conjunto de aplicaciones completo para la creación tanto de aplicaciones de escritorio como de aplicaciones Web de empresa para trabajo en equipo. Aparte de generar aplicaciones de escritorio de alto rendimiento, se pueden utilizar las eficaces herramientas de desarrollo basado en componentes y otras tecnologías de Visual Studio para simplificar el diseño, desarrollo e implementación en equipo de

1 Francisco Charte Ojeda, importante escritor de libros sobre informática de España. Cuenta con muchos años de experiencia trabajando con una editorial tan importante como Anaya.

soluciones para empresa. Constituye un conjunto completo de herramientas de desarrollo para la construcción de aplicaciones Web ASP, servicios Web XML, aplicaciones para escritorio y aplicaciones móviles. Visual Basic .NET, Visual C++ .NET, Visual C# .NET y Visual J# .NET utilizan el mismo entorno de desarrollo integrado (IDE), que les permite compartir herramientas y facilita la creación de soluciones en varios lenguajes. Asimismo, dichos lenguajes aprovechan las funciones de .NET framework, que ofrece acceso a tecnologías clave para simplificar el desarrollo de aplicaciones Web ASP y servicios Web XML.

Visual Studio .NET proporciona diversas plantillas de proyecto que pueden utilizarse para iniciar el desarrollo de aplicaciones distribuidas sin tener que empezar de cero. Las plantillas de empresa definen la estructura inicial de una aplicación distribuida, y proporcionan una guía de arquitectura y tecnología para el diseño de la aplicación. Aparte de las plantillas de empresa predefinidas, se pueden crear plantillas personalizadas que los programadores pueden utilizar en un entorno de equipo. (12)

Visual Studio 2005 se empezó a comercializar a través de Internet a partir del 4 de Octubre de 2005 y llegó a los comercios a finales del mes de Octubre en inglés. En castellano no salió hasta el 4 de Febrero de 2006. Microsoft eliminó .NET, pero eso no indica que se alejara de la plataforma .NET, de la cual se incluyó la versión 2.0.

La actualización más importante que recibieron los lenguajes de programación fue la inclusión de tipos genéricos, similares en muchos aspectos a las plantillas de C#. Con esto se consigue encontrar muchos más errores en la compilación en vez de en tiempo de ejecución, incitando a usar comprobaciones estrictas en áreas donde antes no era posible. C++ tiene una actualización similar con la adición de C++/CLI como sustituto de C# manejado.

Se incluye un diseñador de implantación, que permite que el diseño de la aplicación sea validado antes de su implantación. También se incluye un entorno para publicación web y pruebas de carga para comprobar el rendimiento de los programas bajo varias condiciones de carga.

Visual Studio 2005 también añade soporte de 64-bit. Aunque el entorno de desarrollo sigue siendo una aplicación de 32 bits Visual C++ 2005 soporta compilación para x86-64 (AMD64 e Intel 64) e IA-64 (Itanium). El SDK incluye compiladores de 64 bits así como versiones de 64 bits de las librerías.

1.3.3 .NET Framework

Un framework no es mas que una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la que podemos añadirle las últimas piezas para construir una aplicación concreta. Entre las ventajas que nos ofrece podemos mencionar que acelera el proceso de desarrollo, permite la reutilización de código ya existente y promueve buenas prácticas de desarrollo como el uso de patrones. (13)

El Framework de .Net es una infraestructura sobre la que se reúne todo un conjunto de lenguajes y servicios que simplifican enormemente el desarrollo de aplicaciones. Mediante esta herramienta se ofrece un entorno de ejecución altamente distribuido, que permite crear aplicaciones robustas y escalables. Los principales componentes de este entorno son:

- ✓ Lenguajes de compilación
- ✓ Biblioteca de clases de .Net
- ✓ CLR (Common Language Runtime)

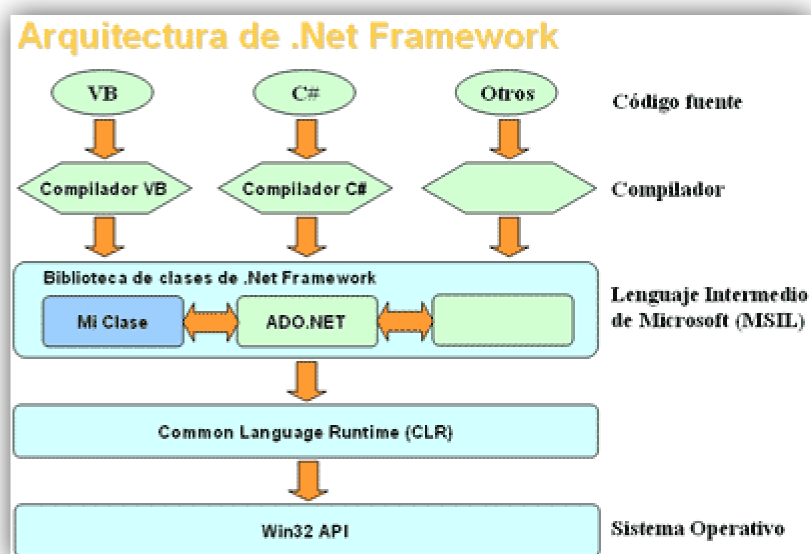


Figura 2 Arquitectura del Framework .NET

Actualmente, el framework de .Net es una plataforma no incluida en los diferentes sistemas operativos distribuidos por Microsoft, por lo que es necesaria su instalación previa a la ejecución de programas

creados mediante .Net. El framework se puede descargar gratuitamente desde la web oficial de Microsoft. .Net framework soporta múltiples lenguajes de programación y aunque cada lenguaje tiene sus características propias, es posible desarrollar cualquier tipo de aplicación con cualquiera de estos lenguajes. Existen más de 30 lenguajes adaptados a .Net, desde los más conocidos como C# (C Sharp), Visual Basic o C++ hasta otros lenguajes menos conocidos como Perl o Cobol. A continuación se resumen las ventajas más importantes que proporciona .Net framework:

- ✓ Código administrado: El CLR realiza un control automático del código para que este sea seguro, es decir, controla los recursos del sistema para que la aplicación se ejecute correctamente.
- ✓ Interoperabilidad multilenguaje: El código puede ser escrito en cualquier lenguaje compatible con .Net ya que siempre se compila en código intermedio (MSIL).
- ✓ Compilación just-in-time: El compilador JIT incluido en el framework compila el código intermedio (MSIL) generando el código máquina propio de la plataforma.
- ✓ Recolector de Basura: El CLR proporciona un sistema automático de administración de memoria denominado recolector de basura (garbage collector). El CLR detecta cuándo el programa deja de utilizar la memoria y la libera automáticamente. De esta forma el programador no tiene por que liberar la memoria de forma explícita aunque también sea posible hacerlo manualmente (mediante el método dispose () liberamos el objeto para que el recolector de basura lo elimine de memoria).
- ✓ Seguridad de acceso al código: Se puede especificar que una pieza de código tenga permisos de lectura de archivos pero no de escritura.
- ✓ Despliegue: Por medio de los ensamblados resulta mucho más fácil el desarrollo de aplicaciones distribuidas y el mantenimiento de las mismas. El framework realiza esta tarea de forma automática mejorando el rendimiento y asegurando el funcionamiento. (14)

1.4 Base de Datos Relacionales

1.4.1 Base de Datos (BD)

Una base de datos es un “almacén” que nos permite guardar grandes cantidades de información de forma organizada para que luego podamos encontrar y utilizar fácilmente.

Desde el punto de vista informático, la base de datos es un sistema formado por un conjunto de datos almacenados en discos que permiten el acceso directo a ellos y un conjunto de programas que manipulen ese conjunto de datos. (15)

Desde el punto de vista más formal, se podría definir una base de datos como un conjunto de datos estructurados, fiables y homogéneos, organizados independientemente en máquina, accesibles a tiempo real, compartibles por usuarios concurrentes que tienen necesidades de informaciones diferentes y no predecibles en el tiempo. (16)

De forma sencilla se puede definir como Base de Datos a un conjunto de información relacionada que se encuentra agrupada o estructurada.

La idea general es que se trata de una colección de datos que cumplen las siguientes propiedades:

- ✓ Están estructurados independientemente de las aplicaciones y del soporte de almacenamiento que los contiene.
- ✓ Presentan la menor redundancia posible.
- ✓ Son compartidos por varios usuarios y/o aplicaciones.

1.4.2 Características

Entre las principales características de los sistemas de base de datos podemos mencionar: (17)

- ✓ Independencia lógica y física de los datos.
- ✓ Redundancia mínima.
- ✓ Acceso concurrente por parte de múltiples usuarios.
- ✓ Distribución espacial de los datos.
- ✓ Integridad de los datos.
- ✓ Consultas complejas optimizadas.
- ✓ Seguridad de acceso y auditoría.
- ✓ Respaldo y recuperación.

- ✓ Acceso a través de lenguajes de programación estándar.

1.4.3 Modelo de Datos

Un modelo de datos es "un conjunto de conceptos, reglas y convenciones que nos permiten describir y en ocasiones manipular los datos de un cierto mundo real que deseamos almacenar en la base de datos". (18)

Colección de conceptos bien definidos matemáticamente que ayudan a expresar las propiedades estáticas y dinámicas de una aplicación con un uso de datos intensivo. (4)

- ✓ Propiedades estáticas: entidades (u objetos), propiedades (o atributos) de esas entidades, y relaciones entre esas entidades.
- ✓ Propiedades dinámicas: operaciones sobre entidades, propiedades o relaciones entre propiedades.

1.4.4 Modelo Relacional

Kim Waldén y Jean-Marc Nerson en su libro confeccionaron una descripción muy sencilla y directa del modelo relacional, donde se define que la responsabilidad de las bases de datos relacionales es modelar la información basándose en relaciones definidas en conjuntos finitos de valores llamados dominios. (4)

La estructura fundamental del modelo relacional es precisamente esa, "relación", es decir una tabla bidimensional constituida por líneas (tuplas) y columnas (atributos). Las relaciones representan las entidades que se consideran interesantes en la base de datos. Cada instancia de la entidad encontrará sitio en una tupla de la relación, mientras que los atributos de la relación representarán las propiedades de la entidad.

En este modelo todos los datos son almacenados en relaciones, y como cada relación es un conjunto de datos, el orden en el que estos se almacenen no tiene mayor relevancia a diferencia de otros modelos como el jerárquico y el de red. Esto tiene la considerable ventaja de que es más fácil de entender y de utilizar por un usuario no experto. La información puede ser recuperada o almacenada por medio de «consultas» que ofrecen una amplia flexibilidad y poder para administrar la información.

Este modelo considera la base de datos como una colección de relaciones. De manera simple, una relación representa una tabla que no es más que un conjunto de filas, cada fila es un conjunto de

campos y cada campo representa un valor que interpretado describe el mundo real. Cada fila también se puede denominar tupla o registro y a cada columna también se le puede llamar campo o atributo.

Para manipular la información utilizamos un lenguaje relacional, actualmente se cuenta con dos lenguajes formales el Álgebra relacional y el Cálculo relacional. El Álgebra relacional permite describir la forma de realizar una consulta, en cambio, el Cálculo relacional sólo indica lo que se desea devolver.

1.4.5 Base de Datos Relacional (BDR)

Las bases de datos relacionales se basan en el modelo relacional, cuya estructura principal es la relación, es decir una tabla bidimensional compuesta por filas y columnas. Una base de datos relacional es una base de datos en donde todos los datos visibles al usuario están organizados estrictamente como tablas de valores, y en donde todas las operaciones de la base de datos operan sobre estas tablas. Estas bases de datos son percibidas por los usuarios como una colección de relaciones normalizadas de diversos grados que varían con el tiempo.

Entre las ventajas de las bases de datos relacionales se encuentran que:

- ✓ Garantiza herramientas para evitar la duplicidad de registros, a través de campos claves o llaves.
- ✓ Garantiza la integridad referencial: Así al eliminar un registro elimina todos los registros relacionados dependientes.
- ✓ Favorece la normalización por ser más comprensible y aplicable.

Características

Algunas de las características de las Base de Datos Relacionales son: (19)

- ✓ Una base de datos relacional se compone de varias tablas o relaciones.
- ✓ No pueden existir dos tablas con el mismo nombre.
- ✓ Cada tabla es a su vez un conjunto de registros, filas o tuplas.
- ✓ Cada registro representa un objeto del mundo real.
- ✓ Cada una de estos registros consta de varias columnas, campos o atributos.

- ✓ No pueden existir dos columnas con el mismo nombre en una misma tabla.
- ✓ Los valores almacenados en una columna deben ser del mismo tipo de dato.
- ✓ Todas las filas de una misma tabla poseen el mismo número de columnas.
- ✓ No se considera el orden en que se almacenan los registros en las tablas.
- ✓ No se considera el orden en que se almacenan las tablas en la base de datos.
- ✓ La información puede ser recuperada o almacenada por medio de sentencias llamadas «consultas».

Principios básicos del diseño de Bases de Datos Relacionales

- ✓ Reflejar la estructura del problema en el mundo real.
- ✓ Ser capaz de representar todos los datos esperados, incluso con el paso del tiempo.
- ✓ Evitar el almacenamiento de información redundante.
- ✓ Proporcionar un acceso eficaz a los datos.
- ✓ Mantener la integridad de los datos a lo largo del tiempo.
- ✓ Ser claro, coherente y de fácil comprensión.

1.5 Framework de Persistencia de Objetos Relacionales

1.5.1 Framework

En el desarrollo de software, un framework es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un framework puede incluir soporte de programas, bibliotecas y un lenguaje de scripting entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Un framework representa una arquitectura de software que modela las relaciones generales de las entidades del dominio. Provee una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio.

Los framework son diseñados con el intento de facilitar el desarrollo de software, permitiendo a los diseñadores y programadores pasar más tiempo identificando requerimientos de software que tratando con los tediosos detalles de bajo nivel de proveer un sistema funcional. Por ejemplo, un equipo que usa Apache Struts para desarrollar un sitio web de un banco puede enfocarse en cómo los retiros de ahorros van a funcionar en lugar de preocuparse de cómo se controla la navegación entre las páginas en una forma libre de errores. Sin embargo, hay quejas comunes acerca de que el uso de framework añade código innecesario y que la preponderancia de otros competitivos y complementarios significa que el tiempo que se pasaba programando y diseñando ahora se gasta en aprender a usar framework. (20)

Pero un framework, a veces, puede ser más de lo que se necesita para ciertos proyectos. Siendo plataformas genéricas diseñadas para casi cualquier tipo de desarrollo, para un proyecto pequeño o urgente la cantidad de opciones y dificultad de instalación de un framework pueden ser abrumadoras. De cualquier modo, un framework es la forma en que un desarrollador decide solucionar sus proyectos, un ambiente de trabajo. En esencia un framework es la aplicación rigurosa de los principios básicos que hacen la diferencia entre un programa bien construido y uno malo: economía, modularización, separación de tareas. Incluso si se decide no ocupar un framework existente, aplicando estos principios al trabajo tarde o temprano se termina creando el propio. (4)

Fuera de las aplicaciones en la informática, un framework puede ser considerado como el conjunto de procesos y tecnologías usados para resolver un problema complejo. Es el esqueleto sobre el cual varios objetos son integrados para una solución dada.

1.5.2 Persistencia

Existen diferentes definiciones para el término persistencia, según distintos puntos de vistas y autores. A continuación se presentan dos definiciones que con claridad y sencillez concretan el concepto de persistencia de objeto.

El primero y más antiguo dice así: “Es la capacidad del programador para conseguir que sus datos sobrevivan a la ejecución del proceso que los creó, de forma que puedan ser reutilizados en otro proceso. Cada objeto, independiente de su tipo, debería poder llegar a ser persistente sin traducción explícita. También debería ser implícito que el usuario no tuviera que mover o copiar los datos expresamente para ser persistentes”. (21)

Esta definición recuerda que es tarea del programador, determinar cuando y como una instancia pasa a ser persistente o deja de serlo, o cuando debe ser nuevamente reconstruida; asimismo, que la transformación de un objeto en su imagen persistente y viceversa debe ser transparente para el programador y que todos los tipos, clases deberían tener la posibilidad de que sus instancias perduren.

La segunda definición dice así: Persistencia es "la capacidad de un lenguaje de programación o entorno de desarrollo de programación para almacenar y recuperar el estado de los objetos de forma que sobrevivan a los procesos que los manipulan". (22)

Esta definición indica que el programador no debería preocuparse por el mecanismo interno que hace un objeto ser persistente, sea este mecanismo soportado por el propio lenguaje de programación usado, o por utilidades de programación para la persistencia como librerías, framework o compiladores.

En definitiva, el programador debería disponer de algún medio para poder convertir el estado de un objeto, a una representación adecuada sobre un soporte de información que permitirá con posterioridad revivir o reconstruir el objeto, logrando que como programadores, no exista la preocupación de cómo esta operación es llevada a cabo.

La persistencia de la información es la parte más crítica en una aplicación de software. Si la aplicación está diseñada con orientación a objetos, la persistencia se logra por: serialización del objeto o, almacenando en una base de datos. Las bases de datos más populares hoy en día son relacionales. El modelo de objetos difiere en muchos aspectos del modelo relacional. La interface que une esos dos modelos se llama marco de mapeo relacional-objeto (ORM en inglés). (4)

1.5.3 Esquemas de persistencia

Los sistemas de aplicaciones de negocios de arquitectura compleja son los más usuales, con lo que surge la problemática de guardar objetos en un mecanismo persistente. A dichos objetos los llamaremos objetos persistentes. Existen distintos tipos de almacenamiento como son las bases de datos relacionales, las orientadas a objetos y otro tipo de mecanismos como ficheros planos, bases jerárquicas. (23)

Para conseguir esa persistencia de objetos se diseña un esquema o estructura de persistencia (framework), el cual es un conjunto de clases reutilizables y expansibles que forman un subsistema y

que mediante servicios afines que prestan a los objetos persistentes son capaces de traducirlos a registros que se guardarán en la base de datos y viceversa. (23)

Los principios de un esquema son:

- ✓ Son un conjunto de clases cohesivo que colaboran para prestar servicio a un núcleo invariable de un subsistema lógico.
- ✓ Poseen clases concretas y abstractas.
- ✓ El usuario definirá subclases a las existentes para que utilice y amplíe los servicios ofrecidos.
- ✓ Las subclases añadidas serán llamadas por los métodos concretos o abstractos de las clases existentes en el esquema.

Los esquemas tienen la característica de ser muy reutilizables con lo que el grado de rehúso del software aumenta (23). Además cuando estemos diseñando el esquema hemos de tener en cuenta que ha de ser fácil de usar, y sobre todo fácil de extender a otros mecanismos de persistencia que en un momento dado no ofrezca el esquema.

En un sistema persistente, el sistema ha de enfrentarse a todos los problemas de recuperación y almacenamiento de objetos en el almacenamiento persistente, siendo un proceso totalmente transparente al usuario (23).

- ✓ Se utiliza para trabajar con bases de datos relacionales, una API de servicios de datos orientados a registros (Microsoft ODBC) u otro mecanismo de almacenamiento.
- ✓ No se utiliza en bases de datos orientadas a objetos.
- ✓ En general, un esquema debe traducir los objetos a registros para guardarlos en una base de datos y viceversa. (23)

Un Esquema de Persistencia debe ofrecer los siguientes servicios:

- ✓ Almacenamiento y recuperación de objetos.
- ✓ Transacciones del tipo commit y rollback.
- commit - completar la transacción de guardar.

- rollback - deshacer la transacción, restaurar el estado anterior.

El diseño de un Esquema de Persistencia debe considerar lo siguiente:

- ✓ Extensible para otros medios de almacenamiento.
- ✓ Realizar la menor cantidad posible de modificaciones al código. (23)

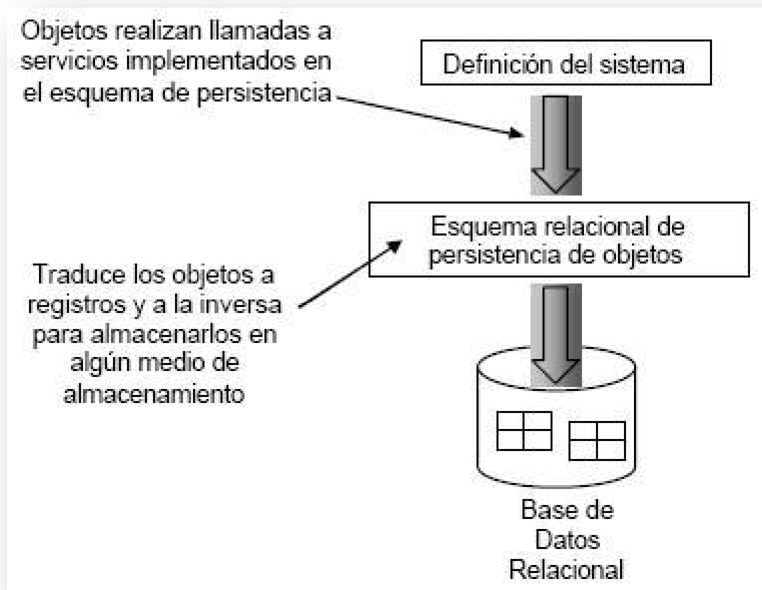


Figura 3 Esquema de Persistencia. (23)

1.5.4 Framework de persistencia

Un framework de persistencia permite una abstracción en gran medida del sistema de almacenamiento utilizado, pudiendo en muchas ocasiones mezclar almacenamiento en bases de datos relacionales, archivos XML o incluso datos provenientes de servicios web, todo ello de forma completamente transparente para las capas superiores.

Utilizar un framework de persistencia ofrece entre otras las siguientes ventajas: (4)

- ✓ Persistencia transparente: Los objetos del dominio no saben nada acerca de la base de datos donde son persistidos, el framework lo resuelve en forma automática utilizando archivos de mapeo expresados en XML.
- ✓ Soporte de polimorfismo: Puede cargar jerarquías de objetos en forma polimórfica.

- ✓ Soporte de los 3 niveles de mapeo de herencia: Puede mapear toda una jerarquía de clases a una sola tabla, crear una tabla por cada clase concreta o crear una tabla por cada escalón de la jerarquía.
- ✓ Soporte completo de asociaciones: Los framework de persistencia soportan el mapeo de todos los tipos de relaciones que pueden existir en un modelo de objetos del dominio (asociaciones 1.1, 1...N, N...M, unidireccionales y bidireccionales).
- ✓ Soporte de carga de objetos Proxy: Puede cargar objetos que solo contengan la clave del objeto completo.
- ✓ Soporte de múltiples dialectos SQL.

1.5.5 Framework de persistencia más usados

Existe una variedad de herramientas para la generación de código tanto comercial como libre. A continuación se muestran las de mayores prestaciones y presencia en el mercado.

TierDeveloper: Es una herramienta propietaria que permite la creación de la capa de acceso a datos para tecnología .NET. Soporta cuatro gestores de base de datos (SQL Server, Oracle, IBM Db2, Microsoft Access). Permite obtener un diagrama de objetos a partir de la estructura de la base de datos. Permite definir un diagrama propio de objeto y cómo se almacenará cada clase en la base de datos. Genera una clase por cada tabla de la base de datos y una clase para manejar toda la capa de acceso a datos. Genera una interfaz WEB para comprobar que funcionan las clases generadas. Entre sus desventajas está su elevado costo y que sólo soporta tecnologías propietarias. (26)

TierDeveloper Ver 5.6	Precio (USD)
Licencia por un año de uso para un desarrollador	$795
Licencia por tiempo indefinido para un desarrollador	$1495
Licencia por tiempo indefinido para tres desarrolladores Licencia por tiempo indefinido para cinco desarrolladores Licencia por tiempo indefinido para diez desarrolladores	$3995 $5995 $9995

Tabla 1 Precio de licencias de TierDeveloper.

CodeCharge Studio: Es una de las herramientas de mayores prestaciones disponibles en el mercado. Está orientada principalmente a la creación de aplicaciones WEB aunque puede ser utilizada para crear aplicaciones de escritorio. Sus principales características son:

- ✓ Soporta varios gestores de base de datos (Microsoft SQL Server, Oracle, DB2, MYSQL y Microsoft Access.).
- ✓ Brinda facilidades para la creación de reportes.
- ✓ Implementa mecanismo de cache para los reportes.
- ✓ Disponible en varios idiomas.
- ✓ Permite la creación de interfaces de usuarios.
- ✓ Soporta hojas de estilo CSS.
- ✓ Soporta integración con SubVersion.
- ✓ Soporta múltiples lenguajes de programación (PHP/JAVA/PERL/ASP).

En cuanto a su interacción con el código se clasifica en Activo y utiliza una aproximación estructural. Su principal desventaja es que no establece una división por capas, en la aplicación que genera por lo que se hace difícil la comprensión del código por parte de los desarrolladores. Es una herramienta propietaria y su costo es elevado. (6)

Licencia	Características	Precio (USD)
CodeCharge Studio 3.1		
Perpetua	Incluye Soporte y actualizaciones gratis.	$499 por usuario.
No Perpetua	30 días de soporte y actualizaciones gratis.	$279 al año por usuario.
CodeCharge Studio Personal Edition 3.1		
Perpetua	30 días de soporte y actualizaciones gratis.	$199 por usuario.
No Perpetua	30 días de soporte y actualizaciones gratis.	$139 al año por usuario.
Consultas y accesoria técnica		
	Asistencia técnica personalizada	$60 por hora.

Tabla 2 Precio por licencia de CodeCharge.

Visual Paradigm: Es una herramienta Visual para el modelado UML. Está diseñada para una amplia gama de usuarios entre los que se incluyen ingenieros de software, analistas de sistemas, analistas de negocio, arquitectos y desarrolladores. Está orientada a la creación de diseños usando el paradigma de programación orientada a objetos. Visual Paradigm incluye una herramienta llamada Visual Architect que permite la generación de código para el manejo de la base de datos. Con esta herramienta se puede generar código para los lenguajes PHP, JAVA y C# y para los gestores de base de datos DB2, Informix, SQL Server, MYSQL, Oracle y PostgreSql. Entre sus limitaciones está que el código generado hace uso de librerías propias, suprimiendo al desarrollador la posibilidad de escoger qué librería usar para acceder a los datos.

El código generado no es de fácil comprensión para los desarrolladores. Es una herramienta propietaria que tiene un costo elevado. (27)

Licencia	Precio (USD)
Enterprise	$1399
Professional	$699
Standard	$299
Modeler	$99

Tabla 3 Precio por licencia de Visual Paradigm.

TriActive JDO (TJDO): Es una implementación libre de la especificación JDO 1.0.1 de Sun. Implementa el lenguaje de consultas JDOQL, aunque permite consultas SQL directas. Permite la auto-creación de todos los elementos del esquema necesarios (tablas, claves, índices) y la auto-validación de la estructura del esquema en tiempo de ejecución. Tiene la capacidad de mapear clases de Java a Vistas SQL. (28)

PHP Object Generator: (POG) es un generador libre para aplicaciones WEB realizadas con PHP. Permite la generación de los métodos elementales (select, insert, delete, update) en forma de funciones. Es uno de los más conocidos, y no en vano, ya que el generador online es excelente, uno ingresa el nombre del objeto y las columnas de la tabla a la que se desea mapear el objeto. Tiene soporte para PHP4 y PHP5.

EZPDO: es una de las más completas y activamente desarrolladas librerías. Algo curioso de esta librería es que utiliza los comentarios para indicar relaciones y tipos de datos de cada objeto. Vale la pena tenerla en cuenta.

PDO: Formaliza las conexiones a bases de datos mediante la creación de una interfaz uniforme. Esto permite a los desarrolladores a crear un código portable a numerosas bases de datos y plataformas.

1.6 Framework de Persistencia de Objetos Relacionales Doctrine (29)

Las BD siguen una estructura relacional. PHP 5 por el contrario es orientado a objetos. Por este motivo, para acceder a la base de datos como si fuera orientada a objetos, es necesario contar con una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz es lo que se conoce como ORM.

Un ORM consiste en una serie de objetos que permiten acceder a los datos y que contienen en su interior cierta lógica de negocio. Es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos tales como la herencia y el polimorfismo.

1.6.1 ¿Qué es Doctrine?

Doctrine es un ORM que posee una poderosa capa de abstracción de BD. Una de sus características es la opción de escribir consultas de BD en un objeto apropiado orientado al dialecto SQL (Structured Query Language) y que se le denomina Lenguaje de Consulta de Doctrine o DQL del inglés Doctrine Query Language, inspirado por el Lenguaje de Consulta de Hibernate (HQL por sus siglas en inglés). Este proporciona a los desarrolladores una poderosa alternativa al SQL que mantiene la flexibilidad sin requerir duplicación de código innecesario.

1.6.2 Principales características de Doctrine

Doctrine es un framework para el mapeo objeto – relacional para PHP que está dividido en dos capas principales, la DBAL del inglés Database Abstraction Layer y el ORM.

La imagen que se muestra a continuación refleja cómo las capas de Doctrine trabajan juntos.

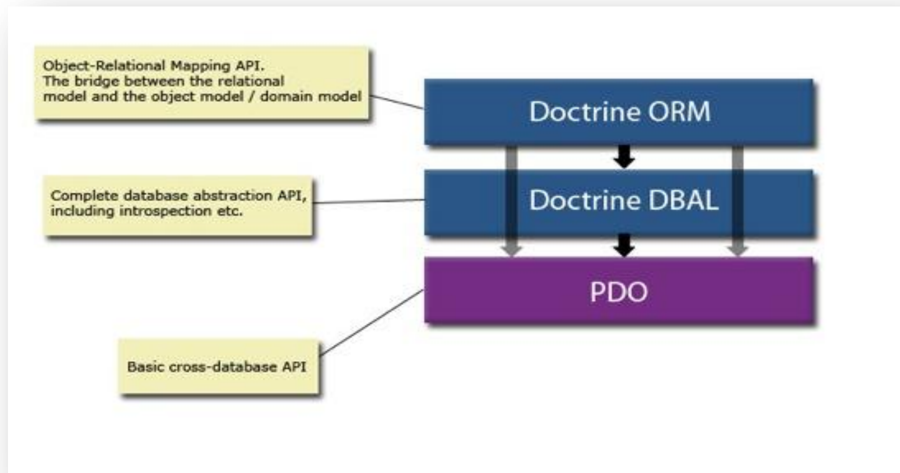


Figura 4 Estructura del framework Doctrine (29)

Doctrine es un framework que está principalmente construido alrededor de los patrones de **`ActiveRecord`**, **`Data Mapping`** y **`Meta Data Mapping`**.

A través de una clase base especifica llamada **`Doctrine_Record`**, todas las clases hijas obtienen la interfaz típica ActiveRecord (salvar, eliminar, etc.) y esto permite a Doctrine fácilmente participar y monitorear el ciclo de vida de sus registros. El trabajo real, sin embargo, es mayormente reenviado a otros componetes como la clase **`Doctrine_Table`**. Esta clase tiene la típica interfaz de mapeo de datos **`createQuery()`**, **`find(id)`**, **`findAll()`**, **`findBy*()`**, **`findOneBy*()`**.

El enfoque ActiveRecord viene con sus típicas limitaciones. La más obvia es la ejecución por una clase para extender una clase base especifica con el objetivo de hacerla persistente. En general el diseño de modelo de dominio es mucho más restringido que el diseño del modelo relacional. Aunque pueden haber excepciones. Cuando tratamos con la herencia, Doctrine provee algunas estrategias sofisticadas de mapeo que permiten al modelo de dominio divergir un poco del modelo relacional.

Después de mencionados los inconvenientes, es hora de mencionar las ventajas de el enfoque ActiveRecord. Aparte de proporcionar un modelo de programación más simple, resulta que la gran similitud del modelo relacional y el modelo de dominio orientado a objetos también tiene una ventaja y es que hace relativamente más fácil proveer una herramienta de generación que pueda crear el modelo de dominio fuera de un esquema relacional existente.

1.6.3 Ventajas y desventajas de Doctrine

Una de las ventajas de utilizar estas capas de abstracción de objetos/relacional es que evita utilizar una sintaxis específica de un sistema de bases de datos concreto. Esta capa transforma automáticamente las llamadas a los objetos en consultas SQL optimizadas para el sistema gestor de bases de datos que se está utilizando en cada momento. De esta forma, es muy sencillo cambiar a otro sistema de bases de datos completamente diferente en mitad del desarrollo de un proyecto. Estas técnicas son útiles por ejemplo cuando se debe desarrollar un prototipo rápido de una aplicación y el cliente aun no ha decidido el sistema de bases de datos que más le conviene. El prototipo se puede realizar utilizando SQLite y después se puede cambiar fácilmente a MySQL, PostgreSQL u otro gestor cuando el cliente se haya decidido. El cambio se puede realizar modificando solamente una línea en un archivo de configuración.

La capa de abstracción utilizada encapsula toda la lógica de los datos. El resto de la aplicación no tiene que preocuparse por las consultas SQL y el código SQL que se encarga del acceso a la base de datos es fácil de encontrar. Los desarrolladores especializados en la programación con bases de datos pueden localizar fácilmente el código.

Utilizar objetos en vez de registros y clases en vez de tablas tiene otra ventaja: se pueden definir nuevos métodos de acceso a las tablas. Por ejemplo, si se dispone de una tabla llamada `Cliente` con 2 campos, **`Nombre`** y **`Apellido`**, puede que sea necesario acceder directamente al nombre completo (**`NombreCompleto`**). Con la programación orientada a objetos, este problema se resuelve añadiendo un nuevo método de acceso a la clase **`Cliente`** de la siguiente forma:

```
public function getNombreCompleto()
{
  return $this->getNombre().' '.$this->getApellido();
}
```

Figura 5 Ejemplo de programación orientada a objetos

Entre muchas otras cosas tienes la posibilidad de exportar una base de datos existente a sus clases correspondientes y también a la inversa, es decir convertir clases (convenientemente creadas siguiendo las pautas del ORM) a tablas de una base de datos.

Por otro lado, como la librería es bastante grande ésta tiene un método para ser 'compilada' al pasar a producción.

Su principal ventaja es el rendimiento en ejecución y la forma tan concisa en la que se pueden escribir consultas muy complejas. Otra de las ventajas de Doctrine es que se puede utilizar las herramientas de sincronización y generación durante todo el proceso de desarrollo.

1.7 Generadores de Ficheros de Mapeo de Objetos Relacionales

1.7.1 ¿Qué es un generador de código?

Un generador de código es una herramienta capaz de generar código de forma automática. Por lo general generan código en algún lenguaje de tercera generación, listo para compilar o interpretar. Estas herramientas hacen de forma automática el trabajo que a los programadores les tomaría mucho más tiempo lograr de forma manual. Para su trabajo los generadores parten de un modelo que puede ser un diagrama de clases o la estructura de una base de datos. El código generado va a depender del modelo que se le entregue y de los algoritmos y patrones que tenga implementado el generador. La mayoría de estas herramientas poseen plantillas de lenguajes scripts para representar los algoritmos y patrones implementados, las cuales el usuario puede modificar y adaptar según sus necesidades. (30) (31)

1.7.2 Tipos de generadores de código

Según su interacción con el código ya generado se clasifican en: Activos y Pasivos.

Los generadores activos son aquellos que permiten generar varias veces sobre el mismo código generado a partir de cambios en la entrada. Estos generadores definen espacios de código seguros donde el programador puede hacer los cambios que desee sin que éstos se pierdan en las sucesivas generaciones de código.

Los generadores pasivos generan el código una vez y no vuelven a tener interacción con él. Tienen la desventaja de que si se corrige un error en los mecanismos de generación o se cambia el diseño y se vuelve a generar se pierde todo lo que se codificó manualmente. (31)

Según la aproximación que usan para generar el código se clasifican en: Estructurales, de Comportamiento y Traductivos.

Aproximación Estructural: Genera bloques de código (como interfaces de clases) desde modelos estáticos y relaciones entre objetos. Las primitivas de trabajo en estos modelos son clases, atributos, tipos y asociaciones. Algunas herramientas usan un motor de traducción y plantillas preexistentes (que pueden ser modificadas y adaptadas) para especificar correspondencias con un código fuente en particular. Escritas en un lenguaje de script, las plantillas guían la traducción de los modelos en estructuras de código, como cabeceras de clases o declaraciones de funciones. Los lenguajes de script permiten a los diseñadores seguir estándares de codificación, personalizar la arquitectura del código generado y crear plantillas nuevas para lenguajes no soportados. Los generadores suelen ofrecer opciones para definir qué objetos y lenguajes usar, en el proceso de generación.

La generación de código estructural es incompleta pero ahorra esfuerzo de codificación manual y proporciona un marco de trabajo inicial consistente con los modelos. Las plantillas de traducción aportan un modesto grado de reutilización. La mayoría de las aplicaciones son adecuadas para esta aproximación. (32)

Aproximación de comportamiento: Generan código completo a partir de modelos de máquinas de estados y la especificación de acciones en un lenguaje de alto nivel. La especificación del comportamiento se basa en máquinas de estados aumentadas con especificación de acciones. Algunos métodos que modelan comportamiento con máquinas de estados, añaden código (como C++ o un lenguaje propietario) para representar las acciones que ocurren durante la transición de estado. Junto con modelos de estructuras de objetos y mecanismos de comunicación, esta técnica permite a las herramientas generar código para el modelo completo de la aplicación. Un beneficio de esta técnica es la capacidad para simular y verificar el comportamiento del sistema basado en modelos antes de que el código sea generado. En los diagramas de estados los usuarios especifican el código explícito para manejar transiciones de eventos. Los lenguajes empleados incluyen C++, C e incluso ensamblador. (31)

En contraste con la aproximación estructural, la codificación del comportamiento se ha hecho durante el modelado de objetos. En la aproximación de comportamiento, la programación se reduce a especificar manejadores de eventos (como las acciones de una máquina de estados) y objetos que son codificados a mano (por ejemplo, optimizados por motivos de rendimiento o eficiencia). Estas herramientas de generación ofrecen poco control al usuario sobre la arquitectura y no usan ingeniería inversa, debido a que todo el código proviene de los modelos. Esta aproximación fomenta un uso continuado de los modelos a lo largo de todo el ciclo de vida del sistema. Los desarrolladores deben

adoptar una visión de máquina de estados de la funcionalidad del sistema además de una visión de la estructura de objetos del sistema. Una especificación completa de comportamiento es ejecutable y por tanto permite ser probada y depurada.

La calidad del código generado es buena debido a que los generadores han madurado en respuesta a la retroalimentación de los clientes. Las aplicaciones típicas de este enfoque están en la industria de las telecomunicaciones además de muchos otros tipos de sistemas que pueden ser modelados con máquinas de estados. (31)

Aproximación traductiva: Estas se basan en que los modelos de aplicación y arquitectura son independientes uno del otro. Un modelo de aplicación completo, con estructura de objetos, comportamiento y comunicaciones es creado usando el método de Análisis Orientado a Objetos.

Como en el caso de la aproximación de comportamiento, los desarrolladores pueden simular el comportamiento del sistema antes de generar el código.

Un modelo de arquitectura (un conjunto de patrones llamados plantillas o arquetipos) es desarrollado con una herramienta que soporte esta aproximación. Entonces, un motor de traducción genera el código para la aplicación de acuerdo con las reglas de correspondencia en la arquitectura. Las aproximaciones traductivas ofrecen una reutilización significativa debido a que la aplicación y el modelo de arquitectura son independientes.

A la hora de especificar un sistema, el sistema es particionado en dominios.

Un domino es modelado por un modelo de información de objetos, un modelo de estados para cada objeto y especificaciones de acciones para cada estado, con el objetivo de permitir su simulación y la generación del código. La especificación de acciones se realiza en un lenguaje propietario de cada herramienta y no en un lenguaje de alto nivel. La simulación es llevada a cabo interpretando la especificación de las acciones. El modelo de aplicación es verificado por un mecanismo de simulación antes de la generación del código.

Un modelo de arquitectura es un conjunto completo de reglas de traducción que establecen una correspondencia de diagramas de objetos con un código fuente. La correspondencia debe ser completa, esto es, todo objeto usado en el modelo de aplicación es traducido. Normalmente las correspondencias gestionan concurrencia (por ejemplo: múltiples hilos de ejecución, multi-tarea, mono-tarea), manejo de eventos (colas, comunicación entre procesos o flujos de entrada/salida) y datos

(estructuras, mecanismos de almacenamiento y persistencia). Cualquier lenguaje destino puede ser soportado con la aproximación traductiva.

Construir un modelo de arquitectura es un proceso de desarrollo en sí mismo. Las correspondencias de traducción son escritas en lenguajes de scripts propietarios. Aunque construir una arquitectura requiere un esfuerzo similar a construir un compilador dirigido por tablas, afortunadamente, hay algo de ayuda. Los fabricantes de herramientas proporcionan arquitecturas genéricas que los desarrolladores pueden modificar. Existen arquitecturas desarrolladas por terceros.

Los desarrolladores pueden reutilizar una arquitectura de otros productos en la misma plataforma para amortizar su esfuerzo. Los fabricantes están mejorando las herramientas de construcción de arquitecturas, las cuales pueden incluir librerías de plantillas para ayudar a la composición.

Dado un modelo de aplicación y una arquitectura, el modelo de traducción extrae los objetos identificados del repositorio de modelos, realiza sustituciones y genera código de acuerdo a los scripts de reglas de correspondencias. El desarrollador controla totalmente la generación del código en la aproximación traductiva y el código es potencialmente completo.

Con la aproximación traductiva, el modelo de aplicación se convierte en el principal artefacto de software, desplazando al código fuente. La combinación de arquitecturas y motor de traducción es análoga a un conjunto de compiladores para un lenguaje de alto nivel. (31)

1.7.3 Funcionamiento de los generadores de código

Los generadores implementan las siguientes fases en este orden:

Carga: La fase de carga consiste en la lectura desde un repositorio (puede ser un fichero binario, XML, una base de datos, o un diagrama UML) del modelo a traducir y crear una representación de éste en memoria. La representación de este modelo en memoria, no tiene porqué ser completa (cargar toda la información de modelo), ni tampoco seguir la misma estructura del modelo. Al contrario, la carga y las estructuras pueden ser adaptadas para cargar sólo la información necesaria y disponerla del modo que sea más conveniente para la tarea de traducción a realizar. (32)

Inferencia: Si se han definido una serie de mecanismos de inferencia, que completan la información de modelado, éstos se ejecutan. Las estructuras del modelo en memoria son completadas y extendidas. Es necesario llevar a cabo este proceso antes de proceder a la generación propiamente dicha. El

proceso es responsable de realizar pre cálculos útiles y de disponer adecuadamente la información para la fase posterior. (32)

Generación: La última fase es la de generación. En función del código destino a producir, se recorren secuencialmente, y de modo anidado, los elementos de modelo en sucesivas pasadas. Por ejemplo: para cada clase, para cada servicio y para cada argumento. Como resultado de esta fase se obtiene el código generado. (32)

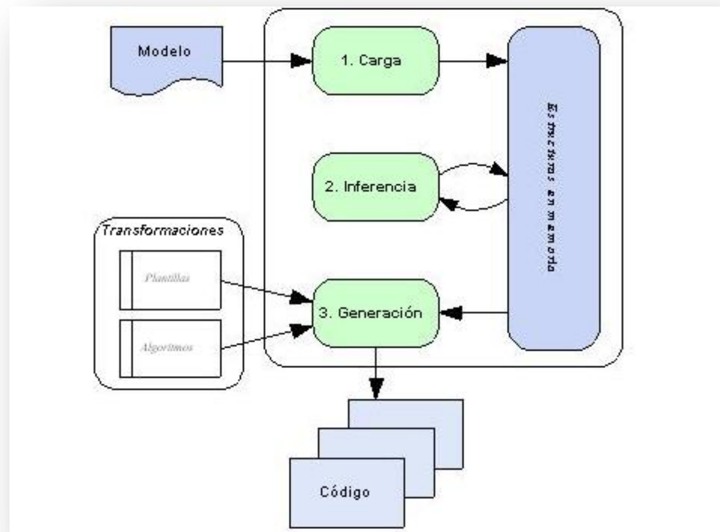


Figura 6 Funcionamiento de un generador de código

1.7.4 Técnicas de generación de código

Se reconocen cuatro técnicas generales utilizadas por los generadores de códigos, ellas son: la Clonación, la Concatenación de cadenas, las Plantillas y el Análisis de expresiones mediante gramáticas.

Clonación: Si el código destino a producir contiene ficheros que permanecen constantes independientemente del modelo a traducir, la traducción puede llevarse a cabo por medio de clonación, o copia directa del fichero origen al directorio de generación. Las librerías de funciones genéricas o ficheros binarios representando imágenes o iconos constantes pueden ser tratados de este modo: construidos una única vez y añadidos mediante un proceso de copia al producto final. (32)

Concatenación de cadenas: La concatenación de cadenas es un modo sencillo de ir construyendo el código destino desde un lenguaje de programación clásico. El nombre proviene del proceso de ir concatenando cadenas de texto que contienen todo el código a producir. Finalmente, la cadena es volcada a un fichero. Se dispone de toda la potencia del lenguaje de programación para determinar que código debe ser generado, permitiendo cálculos, sustituciones y procesados complejos. Sin embargo, entre las desventajas más sobresalientes, figura la necesidad de proteger los caracteres de control propios del lenguaje empleado para la generación. Por ejemplo: dentro de una cadena de texto en C o C++ no es posible usar el caracter comillas " (indicaría el final de la cadena) y para su empleo es necesario protegerlo con el caracter de escape \". Del mismo modo, deben protegerse otros caracteres como por ejemplo el retorno de carro (\n). El mayor inconveniente de estas protecciones radica en que dificulta la lectura del código objetivo dentro del código del generador. (32)

Plantillas: La generación mediante plantillas puede ser empleada cuando el código destino a producir tiene un patrón de repetición bien caracterizado, de modo que todos los ficheros generados son idénticos salvo los datos procedentes directamente del modelo a generar. En este caso puede definirse una plantilla genérica a partir de ejemplares del código objetivo. En esta plantilla, las dependencias del modelo son sustituidas por marcadores con nombre único. Aparejado a la plantilla, podemos definir un proceso de generación que, dado un elemento del modelo, la plantilla sea instanciada a código mediante un proceso de sustitución de cadenas: los marcadores son sustituidos por los datos del modelo correspondiente. (32)

Análisis de expresiones mediante gramáticas: Existen ocasiones donde las estrategias previas de generación se vuelven insuficientes. En particular, un caso muy claro es el del tratamiento de expresiones que siguen una gramática dada. Aquí las técnicas de compilación clásicas son las más adecuadas para producir el código necesario. La expresión puede ser convertida en una estructura con forma arbórea en memoria: Un analizador léxico, sintáctico y semántico realizan este trabajo. Después, un algoritmo puede recorrer dicho árbol que representa la expresión para analizarla y decidir el tipo de código a producir. El algoritmo incluso puede aplicar optimizaciones si el caso lo permite. (32)

1.7.5 Ventajas de la generación de código

Muchos autores coinciden en que la generación de código ofrece cuatro ventajas principales: Calidad, Consistencia, Productividad y Abstracción. (32) (33)

Calidad: La calidad del código generado está en correspondencia con la calidad de las plantillas y del proceso que se usa para la generación. A medida que son detectados errores y es mejorado el código de las plantillas la calidad del código generado aumenta. Se pueden imponer reglas de estilo al código generado para aumentar su homogeneidad y legibilidad. Se puede generar la documentación del código así como comentarios que faciliten su mantenimiento. (32) (33)

Consistencia: El código generado es extremadamente consistente. El nombre de las variables, métodos y clases es formado de la misma forma a lo largo de todo el código. La relación entre el modelo y el código generado permite que también haya consistencia entre la documentación y el código, la cual es muy difícil de mantener en un proceso de desarrollo tradicional. (32) (33)

Productividad: Es fácil reconocer los beneficios de la generación de código respecto a la productividad. Se comienza con un diseño de entrada e instantáneamente se obtiene una implementación de salida que responde al diseño. Por otra parte estos beneficios son mucho más apreciados cuando regeneras el código debido a un cambio en el diseño y no pierdes todo el trabajo que ya está hecho. Por otra parte libera a los programadores del trabajo tedioso y repetitivo permitiéndoles enfocarse en los aspectos que sí requieren de toda su creatividad e inteligencia. (32) (33)

Abstracción: Muchos generadores construyen el código basados en modelos abstractos. Por ejemplo, se puede generar una capa de acceso a datos, a través de un XML que represente las tablas, los atributos y sus relaciones. Entre las ventajas que esto brinda está la portabilidad a distintas plataformas ya que elevando el nivel de abstracción no se cae en especificaciones únicas de una plataforma, sino que permite centrarse en el modelado de los datos o del negocio. Incluso si surgiera una nueva tecnología sólo habría que cambiar el generador y volver a generar la aplicación para que la implemente. (32) (33)

1.7.6 Desventajas de la generación de código

Esfuerzo extra en educación: Se necesita educar a los desarrolladores en las ventajas que ofrece el generador y de qué forma deben emplearlo. (31)

Mantenimiento: cuando se usa un generador hay que darle mantenimiento constantemente. Si es desarrollado por terceros se tiene que estar al tanto de las versiones nuevas que salen y de los errores que se le van detectando. En cualquier caso hay que dedicarle esfuerzos a resolver los errores que pueda traer el generador y a agregarle las nuevas funcionalidades que van surgiendo en el mercado. Si es un generador poco usado se corre además el riesgo de que sus desarrolladores dejen de darle

soporte y que por tanto en poco tiempo se vuelva obsoleto. Por otra parte el grado de sofisticación de estas herramientas dificulta mucho su mantenimiento. (33)

Dominios reducidos: La generación de código tradicionalmente se ha aplicado a dominios muy específicos y bien conocidos donde es posible anticiparse a la variabilidad de problemas que pueden encontrarse en ese dominio. Los dominios o áreas de aplicación demasiado grandes o fuera de ámbito no se benefician de la aplicación de técnicas de generación de código. (31)

Resistencia por parte de los desarrolladores al uso de generadores: Hay programadores convencionales que ven la generación de código como una amenaza para su trabajo. Ante lo cual, toman una postura defensiva o de rechazo. Otros afirman que el uso de generadores va contra las buenas prácticas de diseño y critican mucho la idea de copiar y pegar sobre plantillas. (31)

1.8 Selección de las Herramientas y Tecnologías a utilizar

1.8.1 Metodologías de Desarrollo

En un proyecto de desarrollo de software la metodología define Quién debe hacer Qué, Cuándo y Cómo debe hacerlo. Una metodología es un proceso. No existe una metodología de software universal. Las características de cada proyecto (equipo de desarrollo, recursos) exigen que el proceso sea configurable. Existen dos grupos en los cuáles se dividen estas metodologías: los métodos tradicionales y los procesos ágiles con marcadas diferencias.

En la actualidad existen varias metodologías Orientadas a Objetos (OO) basadas en UML (Unified Model Language) entre las que se encuentran Rational Unified Process (RUP), Agile Unified Process (AUP) y Extreme Programming (XP). A continuación una breve descripción de estas metodologías de desarrollo de software.

Rational Unified Process (RUP)

RUP es un proceso de desarrollo de software que junto al UML constituyen la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos. RUP es en realidad un refinamiento realizado por Rational Software del más genérico Proceso Unificado. (7)

RUP define claramente quien, cómo, cuándo y qué debe hacerse en el proyecto. Como ventajas fundamentales posee:

- ✓ Es guiado por casos de uso que orientan el proyecto a la importancia para el usuario y lo que este quiere.
- ✓ Es un proceso iterativo incremental donde divide el proyecto en mini proyectos donde los casos de uso y la arquitectura cumplen sus objetivos de manera más depurada.
- ✓ Es centrado en la arquitectura que relaciona la toma de decisiones que indican cómo tiene que ser construido el sistema y en qué orden.
- ✓ Utiliza UML para el modelado visual que permite modelar, documentar, especificar y construir artefactos del sistema.

Como RUP es un proceso, en su modelación define como sus principales elementos:

Trabajadores ("quién"): Define el comportamiento y responsabilidades (rol) de un individuo, grupo de individuos, sistema automatizado o máquina, que trabajan en conjunto como un equipo. Ellos realizan las actividades y son propietarios de elementos.

Actividades ("cómo"): Es una tarea que tiene un propósito claro, es realizada por un trabajador y manipula elementos.

Artefactos ("qué"): Productos tangibles del proyecto que son producidos, modificados y usados por las actividades. Pueden ser modelos, elementos dentro del modelo, código fuente y ejecutables.

Flujo de actividades ("Cuándo"): Secuencia de actividades realizadas por trabajadores y que produce un resultado de valor observable.

Agile Unified Process (AUP) (34)

AUP es una adaptación de UP ágil formalizada por Scott W. Ambler[2] que se preocupa especialmente de la gestión de riesgos. Propone que aquellos elementos con alto riesgo obtengan prioridad en el proceso de desarrollo y sean abordados en etapas tempranas del mismo. Para ello, se crean y mantienen listas identificando los riesgos desde las primeras etapas del proyecto. Especialmente relevante en este sentido es el desarrollo de prototipos ejecutables durante la base de elaboración del

[2] *Scott W. Ambler: Escritor de varios libros sobre el desarrollo de software orientado a objetos y sobre metodologías de desarrollo agiles como Agile Model Driven Development (AMDD), Agile Database Techniques, Agile UP y Enterprise Unified Process (EUP). Actualmente trabaja con el Grupo de Software de la IBM.*

producto, donde se demuestre la validez de la arquitectura para los requisitos clave del producto y que determinan los riesgos técnicos.

Al igual que en RUP, en AUP se establecen cuatro fases que transcurren de manera consecutiva y que acaban con hitos claros alcanzados:

- ✓ **Inicio** (Concepción): El objetivo de esta fase es obtener una comprensión común cliente-equipo de desarrollo del alcance del nuevo sistema y definir una o varias arquitecturas candidatas para el mismo.
- ✓ **Elaboración**: El objetivo es que el equipo de desarrollo profundice en la comprensión de los requisitos del sistema y en validar la arquitectura.
- ✓ **Construcción**: Durante la fase de construcción el sistema es desarrollado y probado al completo en el ambiente de desarrollo.
- ✓ **Transición**: el sistema se lleva a los entornos de preproducción donde se somete a pruebas de validación y aceptación y finalmente se despliega en los sistemas de producción.

El proceso AUP establece un Modelo más simple que el que aparece en RUP por lo que reúne en una única disciplina las disciplinas de Modelado de Negocio, Requisitos y Análisis y Diseño. El resto de disciplinas (Implementación, Pruebas, Despliegue, Gestión de Configuración, Gestión y Entorno) coinciden con las restantes de RUP.

El cambio más significativo es la simplicidad. AUP tiene un pequeño número de productos – artefactos que deben producir, como el código fuente y un conjunto de pruebas de regresión, como parte de su sistema. La racionalización de las fases reduce el tiempo de mercado. La agilidad pretende maximizar la suma de trabajo que no se hace: un documento escrito en 50 páginas se reduce a cinco páginas; sin embargo no tendrá que escribir un documento de cinco páginas cuando una pizarra boceto lo hará.

AUP se basa en el conocimiento de que los desarrolladores son capaces y saben lo que están haciendo, pero pueden necesitar un recordatorio de vez en cuando. Como la mayoría de los desarrolladores no leen los procedimientos detallados, se describe la disciplina AUP de la forma más sencilla posible, con enlaces suplementarios para cualquier persona que la use.

AUP reemplaza la disciplina de administración de configuración y cambio de RUP con una disciplina de administración de configuración e incluye las actividades de configuración del cambio y la disciplina de

Gestión de Proyectos de RUP en el Modelo. En el Modelo de la disciplina, una solicitud de cambio o informe de defecto se trata como otro requisito a priorizar por el interesado y puesto en la pila.

La disciplina de implementación adopta técnicas ágiles como pruebas de programación, la refactorización del código, refactorización de base de datos, construcciones continuas y pruebas de regresión continuas. La necesidad de los desarrolladores de trabajar en estrecha colaboración con las partes interesadas y los modeladores ágiles en el equipo también está descrita explícitamente. Esta disciplina también hace esfuerzos de administración de bases de datos más explícita, a través del método ágil de datos.

AUP define algunos productos ágiles de trabajo tales como:

- ✓ El código fuente para el sistema.
- ✓ Una colección completa de pruebas (unidad, sistema y aceptación), y el código que las ejecuta en el orden adecuado.
- ✓ Instalación de Sistema y scripts de instalación.
- ✓ Documentación de Sistema y Notas del release entregadas como parte del sistema para ayudar a trabajar a las partes interesadas y los desarrolladores.
- ✓ Un modelo de requisitos mínimos (por ejemplo, el modelo de CU, modelo de dominio, glosario, etc) sólo lo que sea suficiente para comprender y, a continuación, construir lo que los usuarios necesitan.
- ✓ Un mínimo del Modelo Diseño (por ejemplo, un modelo de objeto y / o modelo de datos, un modelo de despliegue, y una vista de la documentación del sistema)

AUP se basa en las siguientes filosofías:

- ✓ El personal sabe lo que está haciendo. El personal no va a leer una documentación del proceso detallada, sino que prefieren una guía a alto nivel o un tiempo de entrenamiento. El producto AUP proporciona enlaces a muchos detalles que pueden ser vistos si se está interesado, pero es mejor no forzar a leerlos.
- ✓ Simplicidad.- Se describe todo usando solo unas cuantas páginas, no millones de ellas.

- ✓ Agilidad.- AUP se conforma con los valores y los principios del desarrollo ágil del software y de la alianza ágil.
- ✓ Foco de actividades de alto valor.- Centrarse en las actividades que van a suceder realmente, no en cada cosa que pueda suceder en el proyecto.
- ✓ Independencia de la herramienta.- Se puede utilizar cualquier conjunto de herramientas con la metodología AUP. La recomendación es que se utilicen las herramientas que satisfacen lo mejor posible el trabajo, que son a menudo herramientas simples (como muchas soluciones de código abierto).

Extreme Programming (XP) (35)

Es un conjunto de técnicas y prácticas para el desarrollo de software. Se encuadra dentro de la familia de metodologías ligeras, en la búsqueda de métodos sencillos para obtener software de calidad. Sus principios básicos son dos: la mejora de la comunicación con los usuarios, para retroalimentar el proceso de desarrollo; y obtener cuanto antes un programa que haga algo, partiendo de esto para ir añadiendo incrementalmente nuevas características. Estas técnicas se aplican a proyectos con un equipo de desarrollo medio grande, para solucionar un problema no trivial. Algunas de las medidas que proponen no tienen sentido para proyectos pequeños. El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos:

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

En XP se comienza en pequeño y se añaden funcionalidades con la retroalimentación continua y no antes de que sean necesarias; el manejo de cambios es importante en el proceso; el costo del cambio no depende de la fase o etapa y el cliente o el usuario se convierte en miembro del equipo. Además la metodología XP se basa en:

Pruebas Unitarias: pruebas realizadas a los principales procesos, de tal manera que si se quiere adelantar en algo hacia el futuro, se puedan hacer pruebas de las fallas que pudieran ocurrir. Es como adelantarse a obtener los posibles errores.

Re fabricación: se basa en la reutilización de código, para lo cual se crean patrones o modelos estándares, siendo más flexible al cambio.

Programación en pares: una particularidad de esta metodología es que propone la programación en pares, la cual consiste en que dos desarrolladores participen en un proyecto en una misma estación de trabajo. Cada miembro lleva a cabo la acción que el otro no está haciendo en ese momento.

1.8.2 Herramientas CASE (35)

Las Herramientas CASE[3] constituyen un conjunto de programas y ayudas que dan asistencia a los analistas, ingenieros de software y desarrolladores, durante todos los pasos del ciclo de vida de desarrollo de un software.

CASE se define también como:

- ✓ Conjunto de métodos, utilidades y técnicas que facilitan la automatización del ciclo de vida del desarrollo de sistemas de información, completamente o en alguna de sus fases.
- ✓ La sigla genérica para una serie de programas y una filosofía de desarrollo de software que ayuda a automatizar el ciclo de vida de desarrollo de los sistemas.
- ✓ Una innovación en la organización, un concepto avanzado en la evolución de tecnología con un potencial efecto profundo en la organización. Se puede ver al CASE como la unión de las herramientas automáticas de software y las metodologías de desarrollo de software formales.

Estas herramientas pueden proveer muchos beneficios en todas las etapas del proceso de desarrollo de software, algunas de ellas son:

- ✓ Verificar el uso de todos los elementos en el sistema diseñado.
- ✓ Automatizar el dibujo de diagramas.

[3] Computer-Aided Software Engineering es un sistema de ayuda al analista, o administrador de bases de datos. Se inicia a principios de los 80 con la introducción de la documentación asistida por computadoras y de herramientas de diagramación. Estas fueron creadas para desarrollar diagramas estructurados basados en metodología de análisis y diseño estructurado.

- ✓ Ayudar en la documentación del sistema.
- ✓ Ayudar en la creación de relaciones en la Base de Datos.
- ✓ Generar estructuras de código.

Actualmente existen Herramientas CASE tales como:

- ✓ Visual Paradigm para UML.
- ✓ Rational Rose.
- ✓ Enterprise Architect.
- ✓ Umbrello.
- ✓ ArgoUML.

Visual Paradigm

Visual Paradigm es una herramienta CASE profesional que soporta la última versión de UML 2.1 así como el ciclo de vida completo del desarrollo de software, análisis y diseño orientados a objetos, construcción, pruebas y despliegue, exportación desde Rational Rose, exportación/importación XML, generación de informes y edición de figuras. Visual Paradigm tiene disponible distintas versiones: Enterprise, Professional, Standard, Modeler, Personal y Community (que es gratuita). (35)

Visual Paradigm ofrece además:

- ✓ Diseño centrado en casos de uso y enfocado al negocio que genera un software de mayor calidad.
- ✓ Uso de un lenguaje estándar común a todo el equipo de desarrollo que facilita la comunicación.
- ✓ Capacidades de ingeniería directa (versión profesional) e inversa.
- ✓ Modelo y código que permanece sincronizado en todo el ciclo de desarrollo.
- ✓ Disponibilidad de múltiples versiones, para cada necesidad.

- ✓ Disponibilidad de integrarse en los principales IDE[4].
- ✓ Disponibilidad en múltiples plataformas.
- ✓ Ingeniería Inversa Java, C++, Esquemas XML, XML[5], NET exe/dll, CORBA IDL.
- ✓ Ingeniería de ida y vuelta.

Rational Rose Enterprise Edition

Rational Rose Enterprise es una herramienta visual para el análisis y diseño de aplicaciones que abarca todas las etapas de desarrollo de un software. Utiliza UML como lenguaje de modelado y soporta la generación de código a partir de modelos en Ada, ANSI C++, C++, CORBA, Java™/J2EE™, Visual C++® y Visual Basic®.

Características adicionales incluidas:

- ✓ Soporte para análisis de patrones ANSI C++, Rose J y Visual C++ basado en "Design Patterns: Elements of Reusable Object-Oriented Software"
- ✓ Característica de control por separado de componentes modelo que permite una administración más granular y el uso de modelos
- ✓ Soporte de ingeniería Forward y/o reversa para algunos de los conceptos más comunes de Java 1.5
- ✓ Soporte Enterprise Java Beans™ 2.0
- ✓ Capacidad de análisis de calidad de código
- ✓ El Add-In para modelado Web provee visualización, modelado y las herramientas para desarrollar aplicaciones de Web

---

[4] Entorno de Desarrollo Integrado o en inglés Integrated Development Environment (IDE). Entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica GUI

[5] XML, sigla en inglés de eXtensible Markup Language (lenguaje de marcas extensible), es un metalenguaje extensible de etiquetas desarrollado

- ✓ Modelado UML para trabajar en diseños de base de datos, con capacidad de representar la integración de los datos y los requerimientos de aplicación a través de diseños lógicos y físicos
- ✓ Capacidad de crear definiciones de tipo de documento XML (DTD) para el uso en la aplicación.
- ✓ Soporta ingeniería inversa para los lenguajes C++ y JAVA.
- ✓ Genera código para los lenguajes JAVA/J2EE™, C++, Ada, ANSI C++, CORBA, Visual Basic y Visual C++.
- ✓ Genera documentación a partir de los modelos creados.
- ✓ Domina el mercado de herramientas para el análisis y diseño orientado a objetos. (36)

1.8.3 Microsoft Visual C# .NET

Son muchos los lenguajes de programación que han aparecido a lo largo de la historia de las computadoras y a pesar de que con cada uno se pueden lograr hacer grandes aplicaciones, como todo, tienen sus ventajas y desventajas. Una de las desventajas de estos programas es que entre ellos existen maneras muy diferentes de estructurar el código, además de que cada programa maneja sus propias librerías y sintaxis, además de que las funciones en algunos casos se tienen que crear desde cero, lo que conlleva mas tiempo a la hora de programar.

Por todos estos aspectos la empresa de Microsoft creo un lenguaje que podría ser la respuesta a todo este tipo de inconvenientes, un lenguaje llamado C# el cual es uno de los lenguajes base de su plataforma .NET. Lo que pretende este lenguaje es ser muy versátil en su uso y eficiente en su aplicación conformando un lenguaje que reúne las mejores características de los lenguajes más utilizados y conjuntándolos en un solo lenguaje mejorado: C#.

Características:

Facilidad de uso: El ambiente de trabajo es muy cómodo ya que tiene un ambiente amigable y clásico de las aplicaciones de Windows. C# ahorra muchos pasos “tediosos” de otros lenguajes como la creación de funciones complejas desde cero y declaración de variables globales.

Programación orientada a objetos: Esta forma de programación ahorra mucho código, lo cual indica que partes de código son reutilizables para no volverlas a escribir, con lo cual se afirma que C# presenta las características necesarias para considerarlo como un lenguaje orientado a objetos, tales

son: encapsulación, herencia y polimorfismo; además una de las mejoras que presenta este lenguaje con respecto a este tipo de programación es que para evitar confusiones no existen variables o funciones globales, sino que se definen dentro de los tipos de datos. En cuanto a la herencia, esta solo puede ser herencia simple, con lo cual se evitan confusiones que si fuera herencia múltiple.

Administración de memoria: C# tiene la característica de inicializar los datos o variables declaradas en el programa, además de que también de forma automática libera la memoria cuando el mismo programa lo cree conveniente. Es decir tiene constructores y destructores, y estos actúan automáticamente a menos que se manipulen desde el código.

Seguridad en el manejo de datos: C# tiene la característica de estar comprobando que efectivamente los tipos de datos que se estén manejando correspondan a los validados para las funciones que han sido creadas; así también vigila que no se produzcan errores en operaciones matemáticas, además de que también impide el uso de variables que no han sido inicializadas. Todo esto permite que no se produzcan errores en el momento de la ejecución.

Sistema de tipos unificado: Todos los tipos de datos que se definan siempre se derivarán, incluso de forma implícita, de una clase base común llamada System.Object, por lo que dispondrán de todos los miembros definidos en ésta clase.

La ventaja de que todos los tipos se deriven de una clase común es que facilita el diseño de colecciones genéricas que puedan almacenar objetos de cualquier tipo.

Uso de operadores: Este lenguaje permite de forma automática la manera en que pueden trabajar los operadores, ya sea de tipo lógico, aritmético, etc. Es decir dependiendo del contexto de donde se encuentre el operador, el programa detecta que tipo de uso debe tener el operador.

Compatible: C# no sólo mantiene una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes, sino que el runtime de lenguaje común también ofrece la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32.

Ventajas:

Las ventajas que ofrece C# frente a otros leguajes de programación son:

Declaraciones en el espacio de nombres: al empezar a programar algo, se puede definir una o más clases dentro de un mismo espacio de nombres.

Tipos de datos: en C# existe un rango más amplio y definido de tipos de datos que los que se encuentran en C, C++ o Java.

Atributos: cada miembro de una clase tiene un atributo de acceso del tipo público, protegido, interno, interno protegido y privado.

Pase de parámetros: aquí se puede declarar a los métodos para que acepten un número variable de parámetros. De forma predeterminada, el pase de parámetros es por valor, a menos que se use la palabra reservada ref, la cual indica que el pase es por referencia.

Métodos virtuales y redefiniciones: antes de que un método pueda ser redefinido en una clase base, debe declararse como virtual. El método redefinido en la subclase debe ser declarado con la palabra override

Propiedades: un objeto tiene intrínsecamente propiedades, y debido a que las clases en C# pueden ser utilizadas como objetos, C# permite la declaración de propiedades dentro de cualquier clase.

Inicializador: un inicializador es como una propiedad, con la diferencia de que en lugar de un nombre de propiedad, un valor de índice entre corchetes se utiliza en forma anónima para hacer referencia al miembro de una clase.

Control de versiones: C# permite mantener múltiples versiones de clases en forma binaria, colocándolas en diferentes espacios de nombres. Esto permite que versiones nuevas y anteriores de software puedan ejecutarse en forma simultánea.

Desventajas:

Las desventajas que se derivan del uso de este lenguaje de programación son que en primer lugar se tiene que conseguir una versión reciente de Visual Studio .NET, por otra parte se tiene que tener algunos requerimientos mínimos del sistema para poder trabajar adecuadamente tales como contar con Windows NT 4 o superior, tener alrededor de 4 gigas de espacio libre para la pura instalación, etc.

Además para quien no está familiarizado con ningún lenguaje de programación, le costará más trabajo iniciarse en su uso, y si se quiere consultar algún tutorial más explícito sobre la programación en C# se tendría que contar además con una conexión a Internet.

1.9 Conclusiones

En este capítulo se hizo un análisis y un estudio de los principales framework de persistencia utilizados en el proceso de mapeo de objetos relacionales, con el fin de persistir dichos objetos en esquemas relacionales de bases de datos, enfatizando principalmente en el framework Doctrine, así como de los principales generadores de ficheros de mapeos de esquemas relacionales. Se definió la metodología, tecnología y las herramientas a utilizar para el desarrollo de la solución. Partiendo del análisis anterior se llegó a la conclusión de:

Utilizar la plataforma Microsoft .NET para el desarrollo de la solución propuesta, como lenguaje de programación Microsoft Visual C# .NET, como metodología para el desarrollo de software AUP y como herramienta CASE Rational Enterprise Edition, debido a las ventajas y facilidades que brindan y que son enumeradas en el epígrafe 1.9.

La situación actual de los generadores de ficheros de mapeo de esquemas relacionales, lleva a que se plantee el desarrollo de un completo diseño e implementación de una herramienta generadora de ficheros de mapeo basada en Doctrine, que abarque las principales ventajas de las ya existentes y cubra la mayor parte de las necesidades de los equipos de desarrollado que utilizan dicho framework de persistencia, teniendo como puntos positivos la agilización en el proceso de desarrollo de software así como la estandarización de una herramienta para la generación de ficheros de mapeos propia del proyecto ERP – Cuba de la Universidad de las Ciencias Informáticas para la implementación de las soluciones para el producto Cedrux.

CAPÍTULO 2: Descripción de la Solución Propuesta

2.1 Introducción

En el presente Capítulo se describe la solución propuesta, se ofrece una explicación de la arquitectura empleada, especificando además el estilo arquitectónico escogido, cada una de sus partes y como se integran las mismas, así como las especificaciones de las clases necesarias correspondientes al diseño. También se hace referencia a los patrones de diseño empleados en cada una de las capas correspondientes. Para lograr una mayor documentación de respaldo al software desarrollado, se elaboran los artefactos necesarios correspondientes a la metodología de desarrollo de software seleccionada, en este caso AUP.

2.2 Descripción de la Solución (Doctrine Generator)

Doctrine Generator es una aplicación para la persistencia de objetos relacionales, basada en el ORM Doctrine, que ofrece funcionalidades de mapeo que apoyan el trabajo de los equipos de desarrollo del proyecto ERP – Cuba. Doctrine Generator fue diseñado con el principio de brindar a los desarrolladores de este proyecto que necesiten persistir su información mediante Doctrine, un ambiente de desarrollo amigable y fácil de usar que cubra sus necesidades respecto a la persistencia de esta información. La herramienta proporciona la posibilidad de generar la misma lógica de negocio que genera el framework Doctrine con un buen rendimiento en cuanto a funcionamiento corrigiendo algunos de sus problemas en las salidas como por ejemplo la ausencia de las relaciones entre las tablas del esquema con el que se está interactuando y la imposibilidad de personalizar estas salidas.

El desarrollo de esta herramienta pretende cubrir parcialmente o en su totalidad los problemas que tienen los equipos de desarrollo del proyecto ERP – Cuba, que utilizan el framework Doctrine para la persistencia de datos, a la hora de generar los ficheros de mapeo de los esquemas relacionales, brindándole la posibilidad al desarrollador de configurar a su gusto la forma en que el sistema generará las salidas para los ficheros de mapeo. Para complementar lo antes dicho el sistema permite al desarrollador personalizar su propio mapeo, en cuanto a relaciones que existen entre los objetos presentes en el mismo, relaciones como las de uno a uno, uno a mucho y mucho a mucho. Teniendo en cuenta que en el proyecto se desarrollan las soluciones empleando el lenguaje PHP con PostgreSQL como gestor de bases de datos, la herramienta permite conectividad con este gestor, permitiendo el mapeo de los esquemas existentes y generando los correspondientes ficheros.

Si es la primera vez que se ejecuta, el sistema brinda una interfaz donde el usuario debe especificar los esquemas que existen en la base de datos o al menos aquellos con los que se piensa interactuar. Una vez definido esto ofrece la posibilidad de iniciar un proyecto a través de una interfaz en la que se establece el nombre del proyecto y la ubicación del mismo. Además se puede configurar la conexión con el gestor antes mencionado y obtener una lista de los esquemas a los que se tiene acceso de acuerdo a los datos introducidos cuando se configuró la conexión, brindando la opción de seleccionar por esquema las tablas que se desean mapear haciendo así mas personalizado el trabajo de los desarrolladores. Una vez definido estos aspectos se cargan los objetos seleccionados para el mapeo. Este proyecto después es guardado (*.dg) y puede ser reeditado posteriormente. Doctrine Generator cuenta con una serie de interfaces de personalización de los objetos, en las que el desarrollador podrá editar a su gusto las salidas de las clases generadas para dichos objetos en el lenguaje antes mencionado (**ver anexo I**). El sistema genera la siguiente estructura de carpeta y brinda la facilidad de seleccionar el lugar donde se generará:

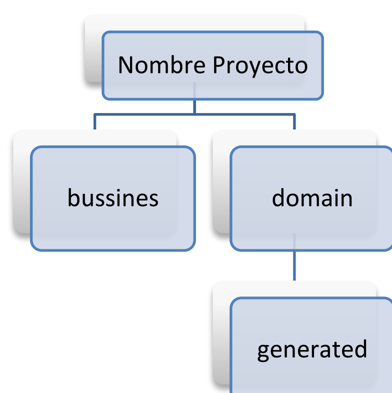


Figura 7 Estructura de carpetas generada por Donctrine Generator

Para cada objeto el sistema generará las clases (*.php) con el prefijo “Base” seguido del nombre de la clase que es el nombre del objeto; con el sufijo “Model” antecedido por el nombre de la clase y otra con el nombre del objeto dentro de las carpetas “generated”, “bussines” y “domain” respectivamente. (**Ver anexo IX y Ver anexo X**)

Otra interfaz importante es la de la configuración de las relaciones de los objetos (**ver anexo II**), en la que el desarrollador especifica las relaciones que puede tener el objeto con otros objetos mapeados, las que podrá configurar a su gusto para el objeto, atendiendo a la lógica de negocio implementada para su sistema.

Doctrine Generator cuenta además con una interfaz que permite configurar algunas funcionalidades que se añadirán en las clases generadas (**ver anexo III**) disminuyendo el contacto directo entre el desarrollador y el fichero evitando de esta forma la introducción de errores en el mismo a partir del proceso de personalización manual.

En los epígrafes siguientes se realiza una descripción del proceso de desarrollo de la herramienta, donde se recogen puntos clave del desarrollo como la arquitectura de la aplicación, patrones de diseño utilizados, algunos artefactos que propone la metodología seleccionada y el diseño de clases del sistema.

2.3 Arquitectura Base

La Arquitectura de Software es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión y la supresión o diferimiento del detalle inherente a la mayor parte de las abstracciones (37).

La Arquitectura de Software de un programa o sistema es la estructura o estructuras del sistema, la cual comprende los componentes de software, las propiedades externas visibles de estos elementos y las relaciones entre ellos. (38)

2.3.1 Estilo arquitectónico empleado en la solución

Un estilo arquitectónico es una abstracción de tipos de elementos y aspectos formales a partir de diversas arquitecturas específicas. Encapsula decisiones esenciales sobre los elementos arquitectónicos y enfatiza restricciones importantes de los elementos y sus relaciones posibles.

Los estilos expresan la arquitectura en el sentido más formal y teórico, describen entonces una clase de arquitectura, o piezas identificables de las arquitecturas empíricamente dadas. Esas piezas se encuentran repetidamente en la práctica, trasuntando la existencia de decisiones estructurales coherentes. Una vez que se han identificado los estilos, es lógico y natural pensar en re-utilizarlos en situaciones semejantes que se presenten en el futuro. Igual que los patrones de diseño, todos los estilos tienen un nombre: cliente-servidor, modelo-vista-controlador, tubería-filtros, arquitectura en capas, arquitectura orientada a servicios. (39)

Arquitectura en capas

Este estilo arquitectónico define cómo organizar el modelo de diseño a través de capas, que pueden estar físicamente distribuidas, lo cual quiere decir que los componentes de una capa sólo pueden hacer referencia a componentes en capas inmediatamente inferiores. Este estilo es importante porque simplifica la comprensión y la organización del desarrollo de sistemas complejos, reduciendo las dependencias de forma que las capas más bajas no son conscientes de ningún detalle o interfaz de las superiores. Además, ayuda a identificar qué se puede reutilizar, y proporciona una estructura que apoya a la toma de decisiones sobre qué partes comprar y qué partes construir. Cuando se emplea este estilo, es posible usar diferentes niveles o cantidad de capas (2 o más capas).

A la hora de plantear el diseño de esta aplicación, el primer paso es conseguir separar conceptualmente las tareas que el sistema debe desempeñar entre las distintas capas lógicas y en base a la naturaleza de tales tareas. Se ha de partir de la separación inicial en tres capas, diferenciando que proceso de los que hay que modelar responde a tareas de presentación, cual a negocio y cual a acceso a datos. Es posible durante el diseño de la solución identificar más capas de acuerdo a la diversidad de procesos. En caso de identificar algún proceso lógico que abarque responsabilidades adjudicadas a dos o más capas distintas, es probable que dicho proceso deba ser explotado en subprocesos iterativamente, hasta alcanzar el punto en el que no exista ninguno que abarque más de una capa lógica. A continuación de describen las tres capas primarias de la solución.

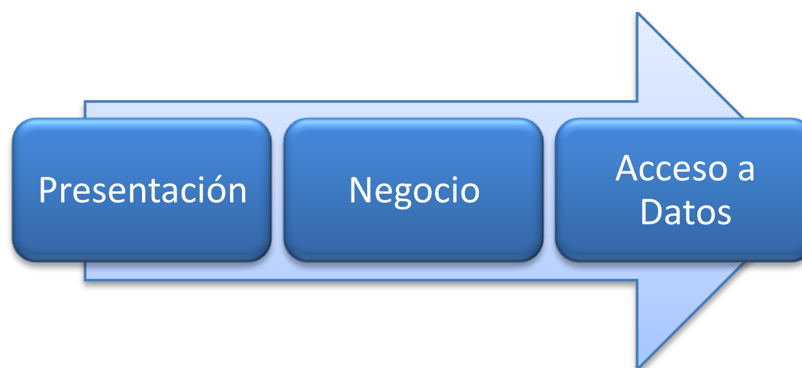


Figura 8 Separación lógica en capas

Capa de presentación

Esta capa es la responsable de todos los aspectos relacionados con la interfaz de usuario de la aplicación. Además en esta capa se resuelven cuestiones como: (4)

- ✓ Navegabilidad del sistema, mapa de navegación.
- ✓ Formateo de los datos de salida: Resolución del formato más adecuado para la presentación de resultados.
- ✓ Validación de los datos de entrada, en cuanto a formatos, longitudes máximas.
- ✓ Interfaz gráfica con el usuario.

Capa de negocio

En esta capa es donde se deben implementar todas aquellas reglas obtenidas a partir del análisis funcional del proyecto. Así mismo, debe ser completamente independiente de cualquiera de los aspectos relacionados con la presentación de la misma. Por otro lado, la capa de negocio ha de ser también completamente independiente de los mecanismos de persistencia empleados en la capa de acceso a datos. Cuando la capa de negocio requiera recuperar o persistir entidades o cualquier conjunto de información, lo hará siempre apoyándose en los servicios que ofrezca la capa de acceso a datos para ello. De esta forma, la sustitución del motor de persistencia no afecta lo más mínimo a esta parte del sistema. Las responsabilidades que conviene abordar en esta capa son:

- ✓ Implementación de los procesos de negocio identificados en el análisis del proyecto.
- ✓ Control de acceso a los servicios de negocio.
- ✓ Publicación de servicios de negocio.
- ✓ Invocación a la capa de persistencia.

Los procesos de negocio son los que determinan que, como y cuando se debe persistir en el repositorio de información. Los servicios ofertados por la interfaz de la capa de acceso a datos son invocados desde la capa de negocio en base a los requerimientos de los procesos en ella implementados. (4)

Esta capa cuenta con un paquete de clases gestoras que se encargan de manipular toda la información de cada una de las entidades y que son quienes interactúan con la capa de acceso a datos. Cuenta además con un paquete de clases generadoras que son las responsables de formatear y generar en la ubicación del proyecto los ficheros por cada una de las tablas que se desean mapear.

Capa de acceso a datos

La capa de acceso a datos es la responsable de la gestión de la persistencia de la información manejada en las capas superiores. En el sistema más específicamente es la responsable de cargar toda la información necesaria para el posterior procesamiento de estos datos por la capa de negocio. (4)

2.3.2 Atributos de calidad que validan la arquitectura.

Los modelos de arquitecturas deben responder a unas series de aspectos para su posible validación ya sean por grupos especializados de calidad, o para la satisfacción del equipo de desarrollo de que la arquitectura a implementar asegurara la fiabilidad, usabilidad y posterior éxito del software a desarrollarse, así como para el aseguramiento al cliente de la salida del producto acordado respondiendo a los intereses por los cuales el mismo decidió adoptar dicha solución informática. (4) Por ejemplos mencionaremos algunos de los aspectos a los que se hacen referencias.

Descripción de atributos de calidad observables vía ejecución:

Atributo de calidad	Descripción
Disponibilidad	Es la medida de disponibilidad del sistema para el uso.
Confidencialidad	Es la ausencia de acceso no autorizado a la información.
Funcionalidad	Habilidad del sistema para realizar el trabajo para el cual fue concebido.
Desempeño	Es el grado en el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria.
Confiabilidad	Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo.
Seguridad externa	Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información.
Seguridad interna	Es la medida de la habilidad del sistema para resistir a intentos de uso no autorizados y negación del servicio, mientras se sirve a usuarios legítimos.

Tabla 4 Descripción de atributos de calidad observables vía ejecución (4)

Descripción de atributos de calidad no observables vía ejecución:

Atributo de calidad	Descripción
Configurabilidad	Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema.
Integrabilidad	Es la medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados.
Integridad	Es la ausencia de alteraciones inapropiadas de la información.
Interoperabilidad	Es la medida de la habilidad de que un grupo de partes del sistema trabajen con otro sistema. Es un tipo especial de integrabilidad.
Modificabilidad	Es la habilidad de realizar cambios futuros al sistema.
Mantenibilidad	Es la capacidad de someter a un sistema a reparaciones y evolución. Capacidad de modificar el sistema de manera rápida y a bajo costo.
Portabilidad	Es la habilidad del sistema para ser ejecutado en diferentes ambientes de computación. Estos ambientes pueden ser hardware, software o una combinación de los dos.
Reusabilidad	Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones.
Escalabilidad	Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental.
Capacidad de prueba	Es la medida de la facilidad con la que el software, al ser sometido a una serie de pruebas, puede demostrar sus fallas. Es la probabilidad de que, asumiendo que tiene al menos una falla, el software fallará en su próxima ejecución de prueba.

Tabla 5 Descripción de atributos de calidad no observables vía ejecución (4)

Ahora cada sistema debe adoptar cuales de estos aspectos, debe responder como prioridad, es decir en cuales aspectos debe responder con más fuerza el sistema que se desea desarrollar, atendiendo claro está, al tipo de software que se esté desarrollando y a las características especificas de dicho software. Debido a las características de la aplicación que se está desarrollando, se adopta como principales aspectos a tener en cuenta los siguientes:

- ✓ Funcionalidad
- ✓ Integrabilidad
- ✓ Modificabilidad
- ✓ Mantenibilidad
- ✓ Reusabilidad

2.3.3 Vista vertical de la arquitectura de la solución propuesta.

La vista vertical de la arquitectura del sistema que se propone, es un estilo en capas, estructurado de la siguiente forma; una primera capa, la de Acceso a Datos, (figura 9), que se encargará de la comunicación con el servidor de datos de la aplicación. Esta capa contiene un grupo de clases que implementan un conjunto de funcionalidades que garantizan la comunicación de las clases de la capa de acceso a datos con el servidor de datos y a su vez sirven de comunicación entre la capa de negocio y la capa de acceso a datos.

La capa de Negocio, (figura 9), contiene las entidades del modelo de diseño que dan descripción y modelado al negocio presente en la aplicación que se propone, así como las entidades de tipo gestores que mantienen todo el control y seguimiento de los distintos hilos de negocios que se crean. En ellas están presentes unas series de patrones de diseño, abordados y explicados en epígrafe posteriores, garantizando fortaleza y rapidez en la aplicación propuesta.

También se cuenta con la capa de Presentación (figura 9), que contiene los diversos controles elaborados para el desarrollo del software, así como las distintas interfaces de usuarios y pautas de diseños aplicadas a las mismas, están presentes también en dicha capa los distintos flujos de navegación que seguirá el usuario mediante la interacción con la aplicación.

Otra capa presente en la solución es la de Generación de Salidas, (figura 9), en la misma están presentes una serie de entidades encargadas de generarle al usuario en el lenguaje PHP los ficheros de salidas. Similar a la capa de Acceso a Datos, en esta las entidades presentes y encargadas de generar ficheros específicos deben implementar una serie de funcionalidades que garantizan estándares de implementación por parte de las entidades de la capa de Generación de Salidas.

Verticalmente a las capas antes descritas, se encuentra el tratamiento de errores, garantizando así el manejo de los errores que se identificaron podía producirse y la captura de otros que no se identificaron. También de forma vertical a la arquitectura propuesta se encuentra el .Net framework, como framework a utilizar por todo el sistema.

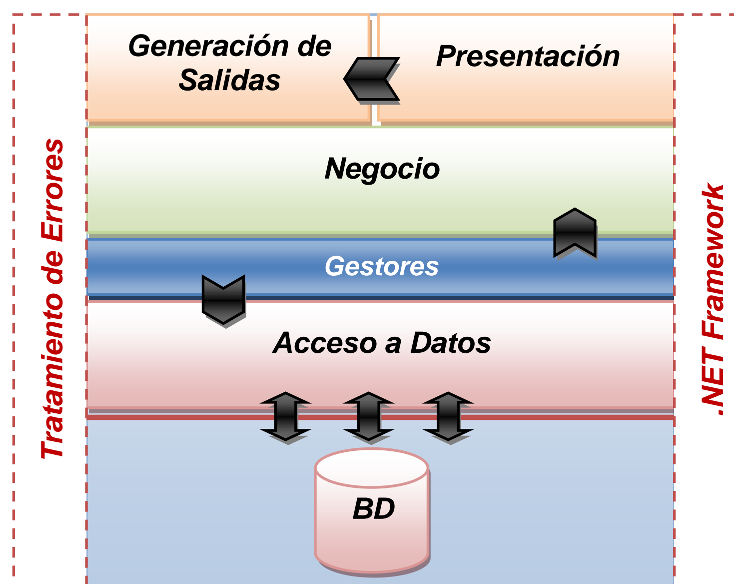


Figura 9 Arquitectura de la solución propuesta

2.3.4 Tecnología utilizada

El principal objetivo de la solución que se propone es poner a disposición de los equipos de desarrollo del proyecto ERP – Cuba una herramienta capaz de generar los ficheros de mapeos que necesita el framework Doctrine para la persistencia de esquemas de objetos relacionales con un número reducido de errores y permitiendo que los desarrolladores personalicen estos ficheros permitiendo así el cumplimiento de las estimaciones del tiempo de implementación hechas por la dirección del proyecto. La necesidad inminente de contar con esta herramienta condiciona la rapidez con la que debe ser desarrollada. En la línea de Arquitectura del proyecto existe el Grupo de Infraestructura que se dedica a desarrollar soluciones informáticas sobre la plataforma .NET para apoyar el desarrollo del proyecto en general y que tienen un gran dominio de esta plataforma. Esta plataforma presenta como interfaz de desarrollo (IDE) Microsoft Visual Studio .NET, en varias de sus versiones. Este IDE independientemente que soporta varios lenguajes de programación, contiene como lenguaje estrella, y creado especialmente para esta plataforma, Visual C#, lo que implica ser el más usado sobre dicho IDE, lenguaje que brinda toda una series de facilidades y soporte al paradigma de programación orientado a objeto, así como posee un fuerte respaldo en el .NET framework.

Estas razones unido a los elementos expuestos anteriormente fundamentan la decisión de implementar la herramienta que se propone teniendo en cuenta su objetivo y características, usando la plataforma .NET, con la utilización de IDE Microsoft Visual Studio 2005, como lenguaje de programación Visual C# 2.0 y sobre .NET framework 2.0 del entorno de desarrollo.

Para el control del código y toda la documentación generada durante el proceso de desarrollo, se propone Subversión, como repositorio y controlador de versiones de la documentación del sistema y TortoiseSVN como cliente de este repositorio. Para la generación de los artefactos propuestos por la metodología a utilizar, en este caso AUP, se utilizó Rational Rose Enterprise Edition, unido a Microsoft Project como herramienta de estimación y planificación del tiempo de desarrollo, así como otras herramientas auxiliares del paquete de Office 2007. Para el diseño gráfico de la aplicación se utilizaron herramientas como Adobe Photoshop CS3 y AAA Logo para el diseño del logo.

2.4 Patrones

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma".[6]

Los patrones de diseño proporcionan esquemas para refinar subsistemas o componentes de un sistema, ayudan a los diseñadores a reutilizar con éxito los diseños; su objetivo es guardar la experiencia en diseños de programas orientados a objetos y hacer más fácil la reutilización de los diseños y arquitecturas; ayudan a elegir entre diseños alternativos, hacen a un sistema reutilizable y evitan alternativas que comprometen la reutilización. Estos son una disciplina de resolución de problemas reciente en la ingeniería del software que ha emergido en mayor medida de la comunidad de orientación a objetos, aunque pueden ser aplicados en cualquier ámbito de la informática y las ciencias en general. Los patrones tienen raíces en muchas áreas y constituyen un modelo que podemos seguir para realizar algo, surgen de la experiencia de seres humanos al tratar de lograr ciertos objetivos. Los patrones capturan la experiencia existente y probada para promover buenas prácticas.

El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones. Pueden referirse a distintos niveles de abstracción, desde un proceso de desarrollo hasta la utilización eficiente de un lenguaje de programación. Citando a James Coplien[7], un buen patrón debería:

***

6 Christopher Alexander: Arquitecto, reconocido por sus diseños destacados de edificios en California, Japón y México.

7 Escritor, profesor e investigador en el campo de la Ciencia de la Computación que ha hecho importantes aportes en el campo del diseño de software.

- ✓ Solucionar un problema.
- ✓ Ser un concepto probado.
- ✓ Describir participantes y relaciones entre ellos.
- ✓ Tener un componente humano alto: estética y utilidad.

El concepto y clasificación de los patrones será retomado en capítulos posteriores. A continuación se relacionan algunos aspectos que se requiere se conozcan en este punto.

2.4.1 Patrones de Diseño

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular. Identifican: Clases, Instancias, Roles, Colaboraciones y la distribución de responsabilidades. Podemos clasificar a los patrones según su propósito en: (35)

Patrones de Creación: Creación de instancias.

Patrones Estructurales: Relaciones entre clases, combinación y formación de estructuras mayores.

Patrones de Comportamiento: Interacción y cooperación entre clases.

Patrones de Creación: Los patrones de creación abstraen la forma en la que se crean los objetos, permitiendo tratar las clases a crear de forma genérica dejando para más tarde la decisión de qué clases crear o cómo crearlas. Según donde se tome dicha decisión se puede clasificar a los patrones de creación en patrones de creación de clase (la decisión se toma en los constructores de las clases y usan la herencia para determinar la creación de las instancias) y patrones de creación de objeto (se modifica la clase desde el objeto).

Patrones Estructurales: Tratan de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos y éstas están determinadas por las interfaces que soportan los objetos. Estudian cómo se relacionan los objetos en tiempo de ejecución. Sirven para diseñar las interconexiones entre los objetos.

Patrones de Comportamiento: Los patrones de comportamiento estudian las relaciones entre llamadas de los diferentes objetos, normalmente ligados con la dimensión temporal.

Patrones GRASP: Asignar correctamente las responsabilidades es muy importante en el diseño orientado a objetos. Los patrones GRASP describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones. Dentro de este grupo de patrones podemos encontrar los siguientes: Experto, Creador, Bajo Acoplamiento, Alta Cohesión, Controlador, Fabricación Pura, Indirección, Variaciones Protegidas, No hables con extraños y Polimorfismo.

2.4.2 Patrones utilizados en la solución

Experto:

El patrón GRASP Experto asigna una responsabilidad al experto en información o sea a la clase que cuenta con la información necesaria para cumplir la responsabilidad.

Resuelve el problema sobre cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño orientado a objetos.

Un modelo de clase puede definir docenas y hasta cientos de clases de software, y una aplicación tal vez requiera el cumplimiento de cientos o miles de responsabilidades. Durante el diseño orientado a objetos, cuando se definen las interacciones entre los objetos, se toman decisiones sobre la asignación de responsabilidades a las clases. Si se hace en forma adecuada, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y se presenta la oportunidad de reutilizar los componentes en futuras aplicaciones.

Por ejemplo en la solución propuesta se necesita gestionar las relaciones de una tabla con el resto de las tablas de la base de datos. Esta responsabilidad se le da a la clase GestorTabla.

Clase	Responsabilidades
Tabla	Contiene una colección de relaciones que gestiona a través de la clase GestorTabla
GestorTabla	Maneja las relaciones que contiene una tabla con el resto de las tablas de la base de datos con que se está interactuando.

Tabla 6 Aplicación del patrón GRASP Experto en la solución

Entre los beneficios que brinda este patrón se encuentran:

- ✓ Conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un bajo acoplamiento, lo que favorece al hecho de tener sistemas más robustos y de fácil mantenimiento.
- ✓ El comportamiento se distribuye entre las clases que cuentan con la información requerida, alentando con ello definiciones de clase "sencillas" y más cohesivas que son más fáciles de comprender y de mantener. Así se brinda soporte a una alta cohesión.

Creador:

Este patrón asigna a la clase B la responsabilidad de crear una instancia de clase A en uno de los siguientes casos:

- ✓ B agrega los objetos A.
- ✓ B contiene los objetos A.
- ✓ B registra las instancias de los objetos A.
- ✓ B utiliza específicamente los objetos A.
- ✓ B tiene los datos de inicialización que serán transmitidos a A cuando este objeto sea creado (así que B es un Experto respecto a la creación de A).
- ✓ B es un creador de los objetos A.

Resuelve el problema de definir quién debería ser responsable de crear una nueva instancia de alguna clase. La creación de objetos es una de las actividades mas frecuentes en un sistema orientado a objetos. En consecuencia, conviene contar con un principio general para asignar las responsabilidades concernientes a ella. El diseño, bien asignado, puede soportar un bajo acoplamiento, una mayor claridad, el encapsulamiento y la reutilizabilidad.

En la solución propuesta, quién debería encargarse de crear una instancia Fichero. Desde el punto de vista de este patrón, se debe buscar una clase que agregue, contenga y realice otras operaciones sobre este tipo de instancias. En la figura siguiente se muestra un diagrama de clases parcial que refleja que un Proyecto contiene (en realidad, agrega) muchos objetos Fichero. Por ello, el patrón Creador sugiere que Proyecto es idónea para asumir la responsabilidad de crear las instancias

Fichero. Pero la clase Proyecto usa los servicios que le brinda la clase GestorProyecto para gestionar todo lo referente a los ficheros que esta contiene por lo que esta clase es en realidad la idónea para asumir la responsabilidad de crear este tipo de instancias.

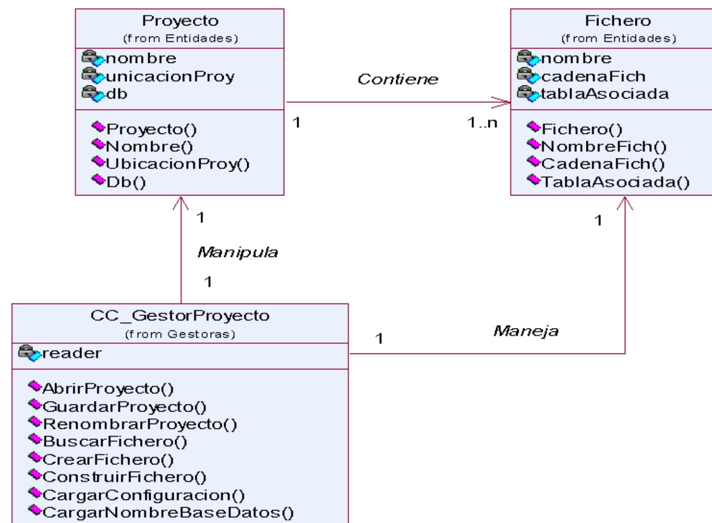


Figura 10 Diagrama de clases parcial

Singlenton/Solitario/Instancia Única:

El patrón de diseño singleton (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado). (40)

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

Ventajas:

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

- ✓ Permite refinamientos en las operaciones y en la representación, mediante la especialización por herencia de "Solitario".
- ✓ Es fácilmente modificable para permitir más de una instancia y, en general, para controlar el número de las mismas (incluso si es variable). (40)

Figura 11 Estructura del patrón Singlenton.

El patrón singleton provee una única instancia global gracias a que:

- ✓ La propia clase es responsable de crear la única instancia.
- ✓ Permite el acceso global a dicha instancia mediante un método de clase.
- ✓ Declara el constructor de clase como privado para que no sea instanciable directamente.

La implementación del patrón Singleton en la solución que se propone, responde a la necesidad de tener creado solamente una instancia de la entidad Proyecto ya que una vez que un usuario, comienza el flujo de navegación de la aplicación, al crear un proyecto, toda las restantes operaciones y modificaciones que realizará, lo hará sobre el mismo objeto "Proyecto" creado al inicio, de ahí la necesidad de mantener una y solo una instancia al usuario de la entidad "Proyecto".

2.5 Disciplina Modelo definida por AUP. Artefacto Modelo de Diseño

El diseño de software orientado a objetos es difícil, y el diseño de software reutilizable orientado a objetos lo es aún más. Se deben identificar los objetos persistentes, clasificarlos dentro de las clases con la granularidad correcta, definir interfaces de clases, jerarquías de herencia y establecer relaciones claves dentro de ellos. El diseño debe ser específico al problema que se tiene entre manos, pero suficientemente general para adaptarse a problemas y requerimientos futuros. Se debe evitar el

rediseño o por lo menos minimizarlo. Antes de que un diseño sea terminado, usualmente se tratan de reutilizar este, varias veces, modificándolo cada vez. (41)

2.5.1 Definición del Modelo de Diseño.

AUP propone que el artefacto Modelo de Diseño básicamente contenga:

- ✓ Paquetes de Diseño: Los paquetes del diseño permiten la organización de los elementos del modelo haciendo que los diagramas sean simples y fácil de entender.
- ✓ Diagramas: Los diagramas de clases con una breve descripción, de interacción (colaboración y/o secuencia) o de estado, de componentes y de despliegue.
- ✓ Vista de la documentación del sistema: Documentación técnica para el personal responsable de dar mantenimiento al software.
- ✓ Modelo de Interfaz de Usuario: Describe las interfaces de usuario del sistema.

2.5.2 Diseño de Clases

El diseño de las clases debe ajustarse al problema en cuestión y debe ser lo suficientemente adaptable a los cambios futuros, minimizando todo lo posible el rediseño. Teniendo en cuenta lo antes expuesto se realizó un diseño de clases donde se identificaron las clases pertinentes y se clasificaron las relaciones claves existentes entre ellas, dándole respuesta al problema.

El objetivo primario de la solución propuesta es el de generar clases persistentes (*.php) junto con el correspondiente mapeo de la misma en el esquema con el que el framework Doctrine interactúa. Por tal motivo el diseño debe responder a un modelo de clases que sea capaz de almacenar y gestionar la información necesaria para crear este tipo de archivos, como son los atributos de las clases que se van a generar, los tipos de relaciones que representan estos atributos con otras clases, la información de la tabla con la que persistirá la información de dicha clase, nombres de las propiedades de estos atributos y las funcionalidades necesarias para obtener la información del esquema de persistencia al que se le realizará el mapeo.

2.5.3 Descripción de las principales clases utilizadas en la solución

Nombre de la Clase:	BaseDatos
Descripción:	Modela la estructura de la Base de Datos

Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
NombreBD	String	Nombre de la base de datos con que se interactúa.
Lesquemas	List<Esquema>	Lista de esuqemas que contiene la base de datos.
TablasMapear	List<string>	Lista de tablas que serán mapeadas.
Funcionalidades		

Tabla 7 Descripción de la Clase BaseDatos

Nombre de la Clase:	Esquema	
Descripción:	Modela los esquemas de la base de datos.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
NombreEsq	String	Nombre del esquema
Ltablas	List<Tabla>	Lista de tablas que están contenidas dentro del esquema.
Funcionalidades		

Tabla 8 Descripción de la Clase Esquema

Nombre de la Clase:	Tabla	
Descripción:	Representa las clases mapeadas, contiene la información necesaria para generar una clase persistente.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
NombreTabla	String	Nombre de la tabla
Alias	String	Alias con el que se podrá referenciar a la tabla.
LcamTabla	List<CampoTabla>	Lista de campos de la tabla
Llaves	List<CampoTabla>	Lista de llaves de la tabla, tanto llaves primarias como foráneas.
Lrelaciones	List<Ralacion>	Lista de relaciones que contiene las tablas

		con otras tablas.
Consulta	List<CosnultaDQL>	Lista de consultas que posee la tabla, estas consultas son añadidas por el usuario en tiempo de ejecución.
Funcionalidades		

Tabla 9 Descripción de la Clase Tabla

Nombre de la Clase:	CampoTabla	
Descripción:	Almacena la información necesaria que se obtiene de los campos de una tabla.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
NombCamp	String	Nombre del atributo.
TipoDato	String	Tipo de dato del atributo.
Estado	String	Contiene la información de si el atributo puede ser nulo o no.
Tamanno	int	Longitud que tiene el atributo.
Llave	String	Indica que tipo de llave es el atributo, primaria o foránea.
ColumnaDef	String	Valor por defecto que contiene el atributo.
Funcionalidades		

Tabla 10 Descripción de la Clase CampoTabla

Nombre de la Clase:	Relacion	
Descripción:	Almacena la información referente a las relaciones.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
NombreRe	String	Nombre de la relación.
Origen	String	Nombre del campo que pertenece a la tabla que da origen a la relación.
Destino	String	Nombre del campo que pertenece a la tabla

		con la que se establece la relación.
RelacTabla	String	Nombre de la tabla con la que se establece la relación.
Tipo	String	Tipo de la relación que se establece, puede ser 1 – 1, 1 – n o n – m.
Funcionalidades		

Tabla 11 Descripción de la Clase Relacion

Nombre de la Clase:	RelacionM_M	
Descripción:	Clase que hereda de la clase Relacion y que representa la relación de mucho a mucho entra tablas.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
TablaPuente	Tabla	Representa la tabla que actúa como puente en la relación n – m.
Funcionalidades		

Tabla 12 Descripción de la Clase RelacionM_M

Nombre de la Clase:	Proyecto	
Descripción:	Clase a través de la cual se realiza todo el flujo de información manejada por la herramienta. Contiene los ficheros que se generarán, la ubicación donde serán generados, un nombre y la base de datos con la que se trabaja.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
Nombre	String	Nombre del proyecto.
UbicacionProy	String	Ubicación donde se guardarán los ficheros que se generen.
ListaFich	List<Fichero>	Contiene la lista de ficheros que serán generados por la herramienta.
Funcionalidades		

Tabla 13 Descripción de la Clase Proyecto

Nombre de la Clase:	Fichero	
Descripción:	Clase que modela el fichero que será generado.	
Tipo de Clase:	Entidad	
Propiedad	Tipo dato	Descripción
NombreFich	String	Nombre del fichero, tiene similitud con el nombre de la tabla a la que está asociado..
CadenaFich	String	Contiene la cadena que se escribirá dentro del fichero.
TablaAsociada	Tabla	Tabla que esta asociada al fichero.
Funcionalidades		

Tabla 14 Descripción de la Clase Fichero

Nombre de la Clase:	Generador	
Descripción:	Clase que presenta algunas funcionalidades que formatean el fichero que se va a generar	
Tipo de Clase:	Generadora	
Propiedad	Tipo dato	Descripción
Funcionalidades		
FixNombre	String	Eliminar los espacios en blanco, los underscore y capitaliza las que están después de los underscores.
BuscarUltimoEspacioEnBlanco	Int	Devuelve la posición del último espacio en blanco de una cadena.
CrearDirectorio	String	Crea la estructura de carpetas en la que se generarán los ficheros y devuelve la ubicación.
Persistir	Void	Persiste la información en un fichero.
Capitalizar	String	Capitaliza la primera letra de la cadena.

Tabla 15 Descripción de la Clase Generador

Nombre de la Clase:	GeneradorDomain	
Descripción:	Clase que hereda de la clase Generador y que permite la generación de los ficheros del domain.	
Tipo de Clase:	Generadora	
Propiedad	Tipo dato	Descripción
Funcionalidades		
Generar	Void	Genera un fichero *.php vacío.
GenerarF	Void	Escribe en un fichero formateada según establece Doctrine una cadena con las relaciones y algunas funcionalidades.
Observaciones		
Esta clase presenta otras funcionalidades que contribuyen a generar el fichero según las opciones que seleccione el usuario.		

Tabla 16 Descripción de la Clase GeneradorDomain

Nombre de la Clase:	GeneradorDomainG	
Descripción:	Clase que hereda de la clase Generador y que presenta un grupo de funcionalidades que establecen el formato del fichero a generar.	
Tipo de Clase:	Generadora	
Propiedad	Tipo dato	Descripción
Funcionalidades		
GenerarB	Void	Genera un fichero *.php vacío
GeneradorB	Void	Escribe en un fichero formateada según establece Doctrine una cadena con la estructura de la tabla asociada a ese fichero.
GenerarFunctionsetTableDefinition	Void	Escribe en el fichero la estructura de la tabla asociada al fichero.

Tabla 17 Descripción de la Clase GeneradorDomainG

Nombre de la Clase:	GeneradorBussines	
Descripción:	Clase que hereda de la clase Generador y que presenta un grupo de funcionalidades que establecen el formato del fichero a generar.	
Tipo de Clase:	Generadora	
Propiedad	Tipo dato	Descripción
Funcionalidades		
GenerarM	Void	Genera un fichero *.php vacío
GeneradorM	Void	Escribe en un fichero formateada según establece Doctrine una cadena con algunas funcionalidades.
Observaciones		
Esta clase presenta otras funcionalidades que contribuyen a dar formato al fichero generado.		

Tabla 18 Descripción de la Clase GeneradorBussines

Nombre de la Clase:	Cargador	
Descripción:	Clase que permite cargar de un fichero XML algunos datos que necesita la herramienta.	
Tipo de Clase:	Gestora	
Propiedad	Tipo dato	Descripción
NombreSchema	List<string>	Lista de esquemas que están en el fichero schemas.xml
Funcionalidades		
CargarConfiguracion	Void	Carga para la lista nombreschema los nombres de los esquemas que están en el fichero schemas.xml
Observaciones		

Tabla 19 Descripción de la Clase Cargador

Nombre de la Clase:	ManagerExcepcion
Descripción:	Clase que permite manejar y controlar las excepciones

Tipo de Clase:	Gestora	
Propiedad	Tipo dato	Descripción
Funcionalidades		
TratarExcepcion	Void	Tiene 2 sobre cargas, en dependencia de los parámetros que se le pasen devuelve una excepción.
LanzarMensaje	Void	Muestra un cartel con el mensaje que se le pase como parámetro.
LanzarPregunta	DialogResult	Lanza una pregunta a través de un DialogResult.

Tabla 20 Descripción de la Clase ManagerExcepcion

2.5.4 Paquetes de Clases del Diseño

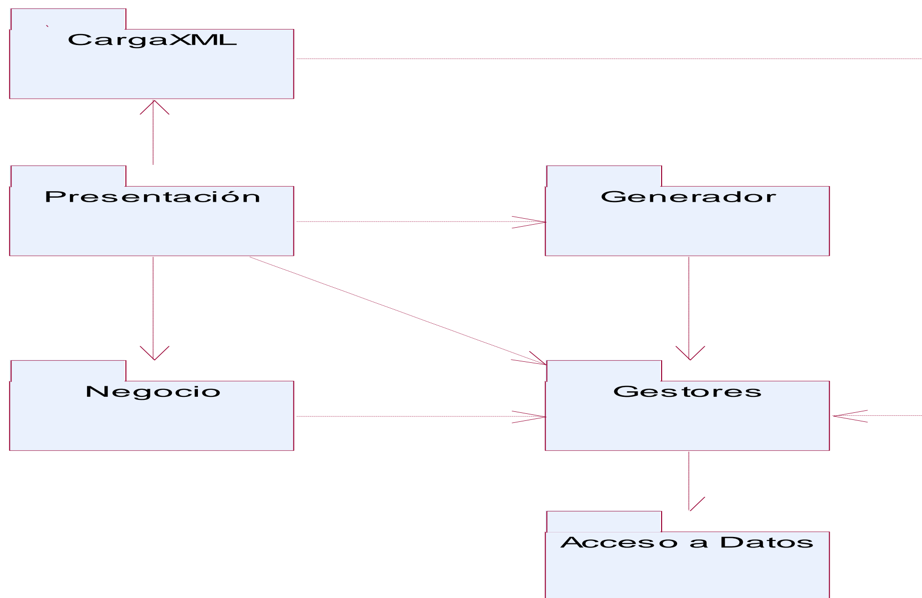


Figura 12 Paquetes en que se agrupan las clases del diseño

2.5.5 Diagrama de Clases

Una clase de diseño es una construcción similar en la implementación del sistema cuyo lenguaje de diseño es el mismo que el lenguaje de programación. Las operaciones, atributos, tipos, visibilidad (public, protected y private), etc. se pueden especificar con la sintaxis del lenguaje elegido.

Las relaciones entre estas clases se traducen de manera directa al lenguaje: generalización, herencia, asociaciones, agregaciones, atributos. Los métodos tienen correspondencia directa con el correspondiente método en la implementación de las clases. Se pueden postergar algunos requisitos a implementación (por ejemplo: manera de nombrar los atributos y operaciones). Además una clase de diseño puede proporcionar interfaces si tiene sentido hacerlo en el lenguaje de programación.

El diagrama de clases de diseño muestra la especificación para las clases de software de una aplicación de forma tal que describa gráficamente sus especificaciones y la de las interfaces en la aplicación. Incluye la siguiente información:

- ✓ Clases, asociaciones y atributos.
- ✓ Interfaces, con sus operaciones y constantes.
- ✓ Métodos.
- ✓ Navegabilidad.
- ✓ Dependencias.

Un diagrama de clases de diseño muestra definiciones de entidades de software más que conceptos del mundo real. Forma parte de la vista estática del sistema. Será en el diagrama de clases de diseño donde se definan las características de cada una de las clases, interfaces, colaboraciones y relaciones de dependencia y generalización.

Una clase de diseño y sus objetos, y de ese modo también los subsistemas que contienen las clases de diseño, a menudo participan en varias realizaciones de casos de uso. También se puede dar el caso de algunas operaciones, atributos y asociaciones sobre una clase específica que son relevantes para sólo una realización de caso de uso.

Esto es importante para coordinar todos los requisitos que diferentes realizaciones de casos de uso imponen a una clase, a sus objetos y a los subsistemas que contiene. Para manejar todo esto, se utilizan diagramas de clases conectados a una realización de caso de uso, mostrando sus clases participantes, subsistemas y sus relaciones. De esta forma podemos guardar la pista de los elementos participantes en una realización del caso de uso. (7)

A continuación se muestra el Diagrama de Clases para las clases entidades y generadoras.

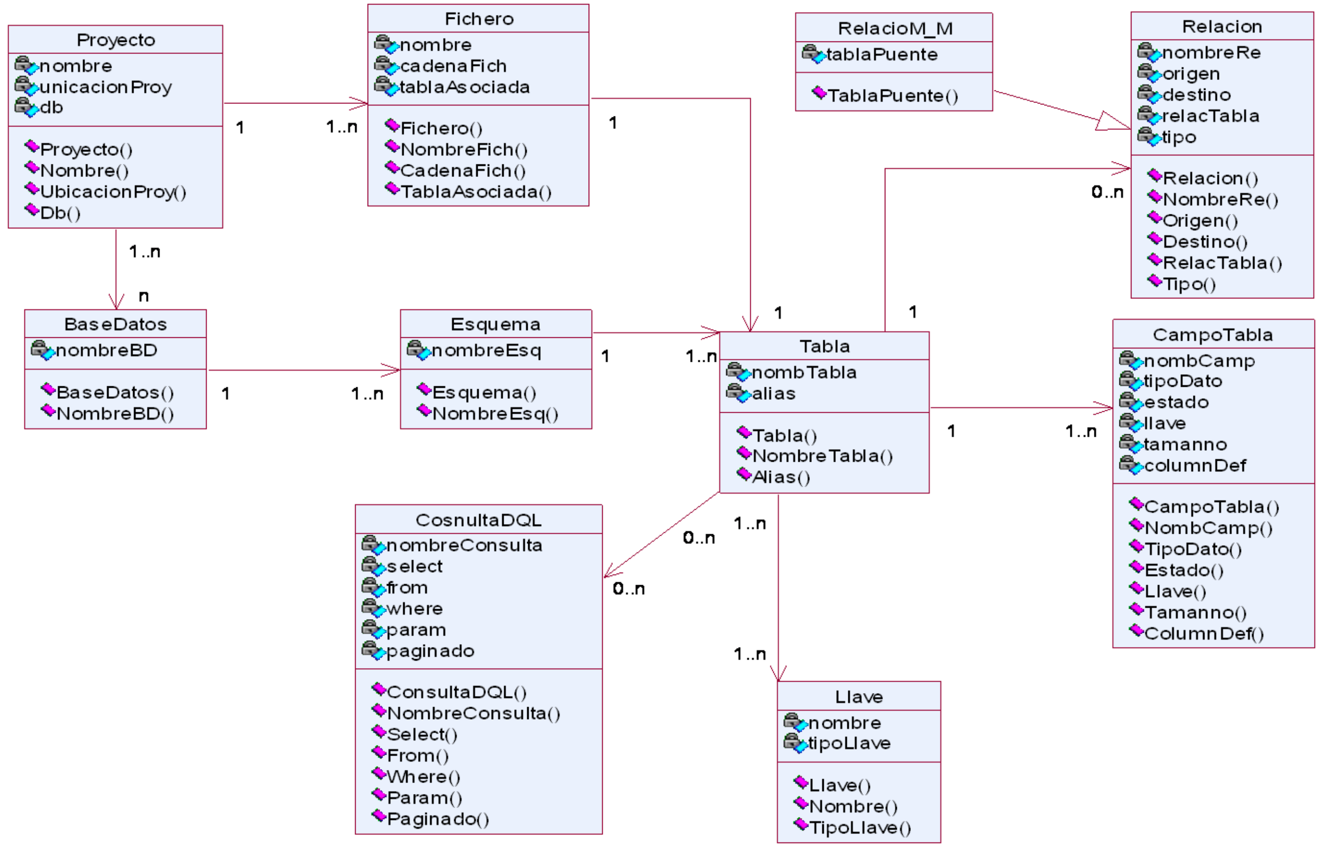


Figura 13 Diagrama de Clases de las clases entidades

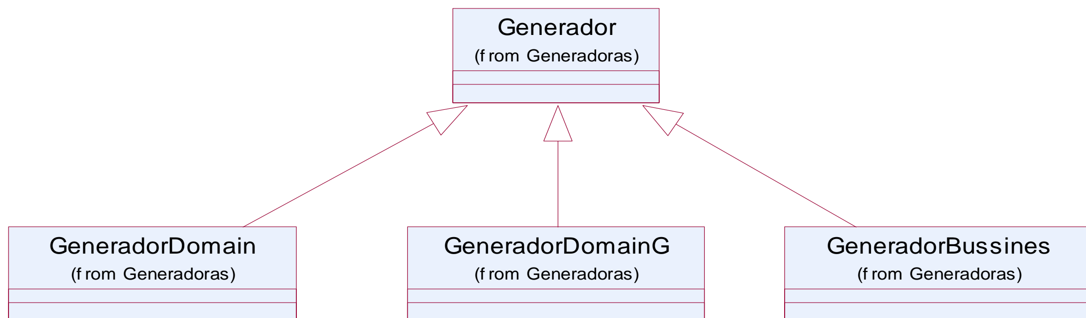


Figura 14 Diagrama de Clases para las clases generadoras

2.5.6 Diagramas de Interacción

Los Diagramas de Interacción muestran una interacción concreta: un conjunto de objetos y sus relaciones, junto con los mensajes que se envían entre ellos ordenados según el tiempo en que tienen lugar. Los objetos pueden existir sólo durante la ejecución de la interacción, se puede crear o puede ser destruido durante la ejecución de la interacción.

Estos diagramas modelan además el comportamiento dinámico del sistema y el flujo de control en una operación. Describen la interacción entre objetos que interactúan a través de mensajes para cumplir ciertas tareas. Se utilizan para mostrar la relación entre los distintos objetos que participan en un escenario a través de mensajes. Las interacciones proveen un “comportamiento” y típicamente implementan un Caso de Uso.

En este tipo de diagramas también intervienen los mensajes, que son la forma en que se comunican los objetos: el objeto origen solicita (llama a) una operación del objeto destino. Existen distintos tipos de mensajes según cómo se producen en el tiempo entre los que se encuentran los simples, síncronos y asíncronos

Existen dos tipos de diagramas de interacción en UML: Diagramas de Secuencia (dimensión temporal) y Diagramas de Colaboración (dimensión estructural).

Diagramas de colaboración.

El diagrama de colaboración es un diagrama de interacción que enfatiza la organización estructural de los objetos que participan en una interacción. Proporcionan la representación principal de un escenario, ya que las colaboraciones se organizan entorno a los enlaces de unos objetos con otros. Este tipo de diagramas se utilizan más frecuentemente en la fase de diseño, es decir, cuando se diseña la implementación de las relaciones que resalta la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de colaboración se caracterizan además por:

- ✓ Dan una visión clara del flujo de control en el contexto en el que se desarrollan.
- ✓ Son útiles en la fase exploratoria para identificar objetos.
- ✓ La distribución de los objetos en el diagrama permite observar adecuadamente la interacción de un objeto con respecto a los demás.
- ✓ La estructura estática viene dada por los enlaces; la dinámica por el envío de mensajes.

Un diagrama de colaboración es muy similar a un diagrama de secuencia en el propósito que alcanza; es decir demuestra la interacción dinámica de los objetos en un sistema.

A continuación se muestran los diagramas de colaboración para el Caso de Uso Gestionar Tablas que contiene los escenarios Adicionar Tabla y Eliminar Tabla.

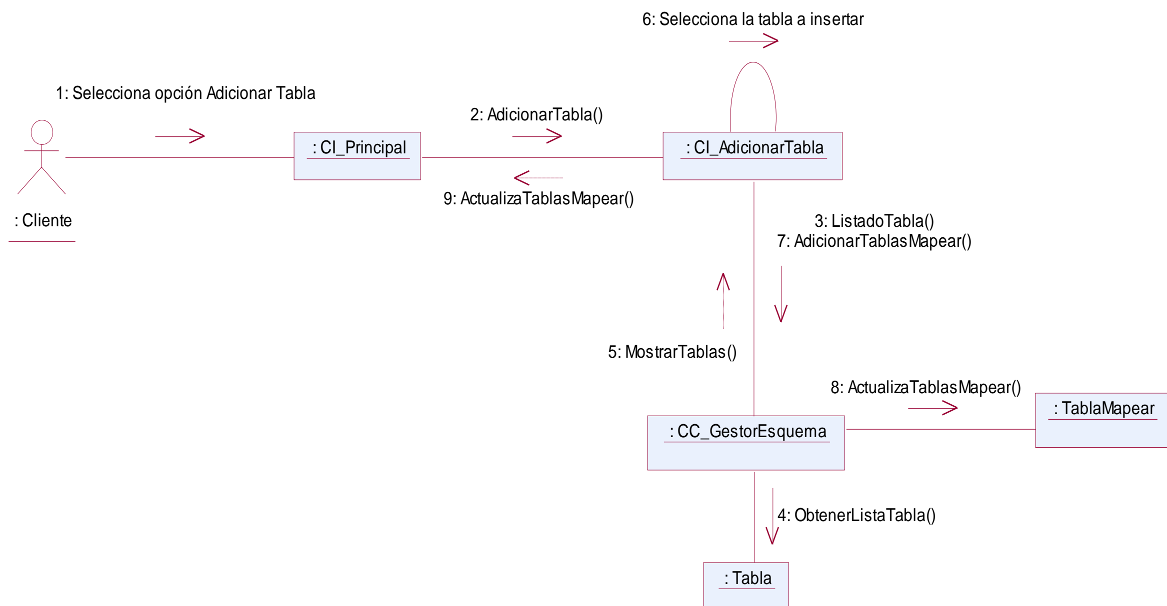


Figura 15 Diagrama de Colaboración para el escenario Adicionar Tabla del CU Gestionar Tablas

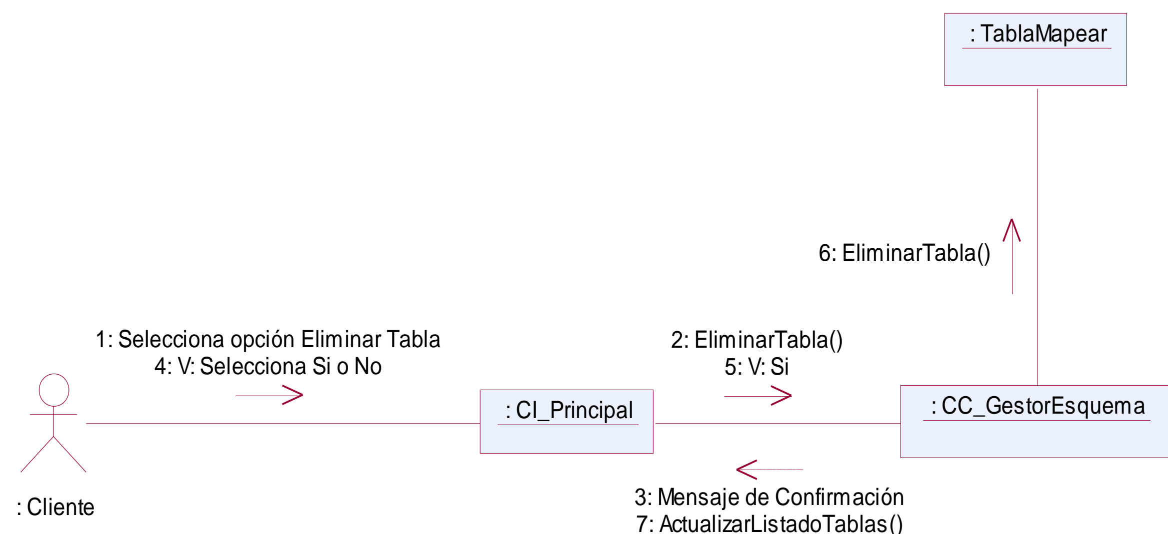


Figura 16 Diagrama de Colaboración para el escenario Eliminar Tabla del CU Gestionar Tablas

Los diagramas de colaboración para los Casos de Uso Generar Ficheros de Mapeo, Gestionar Consulta DQL, Gestionar Relación y Gestionar Fichero de Esquemas se muestran en los anexos. (**Ver anexos del V al VIII**)

Diagrama de secuencias.

El es un diagrama de interacción que consta de objetos que se representan del modo usual: rectángulos con nombre (subrayado), mensajes representados por líneas continuas con una punta de flecha y el tiempo representado como una progresión vertical. En los diagramas de secuencia, se muestran las interacciones entre objetos mediante transferencias de mensajes entre objetos o subsistemas. Cuando se dice que un subsistema "recibe" un mensaje, se quiere decir en realidad, que es un objeto de una clase del subsistema el que envía el mensaje. El nombre del mensaje debería indicar una operación del objeto que recibe la invocación o de una interfaz que el objeto proporciona. (7)

Cuando el diagrama de secuencias solo se centra en un escenario (una instancia) dentro del caso de uso se conoce como diagrama de secuencias de instancias, si por el contrario tuviera en cuenta todos los escenarios de un caso de uso, se trata de un diagrama de secuencias genérico.

A continuación se muestra el diagrama de secuencia para el Caso de Uso Gestionar Tablas que contiene los escenarios Adicionar Tabla y Eliminar Tabla.

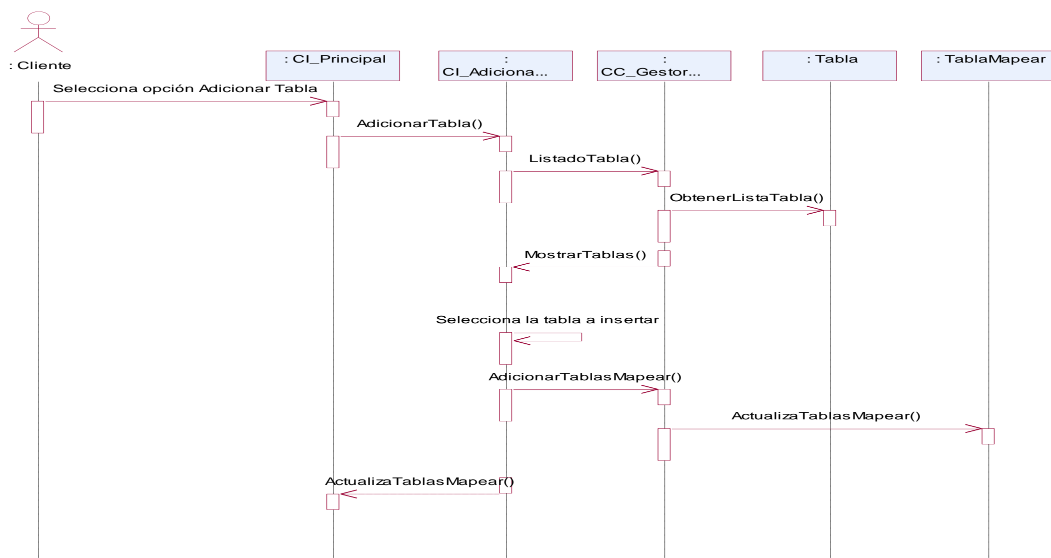


Figura 17 Diagrama de Secuencia para el escenario Adicionar Tabla del CU Gestionar Tablas

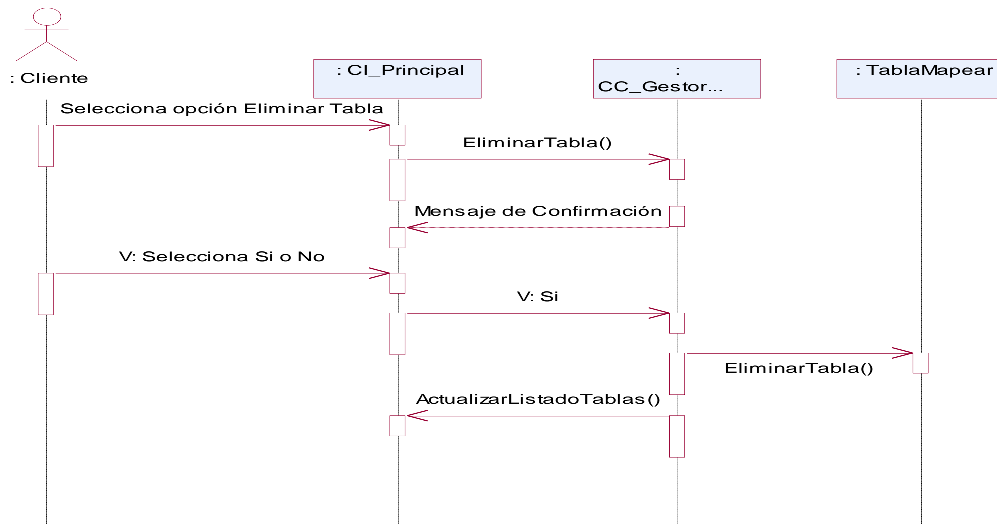


Figura 18 Diagrama de Secuencia para el escenario Eliminar Tabla del CU Gestionar Tablas

Los diagramas de secuencia de los Casos de Uso Generar Ficheros de Mapeo, Gestionar Consulta DQL, Gestionar Relación y Gestionar Fichero de Esquemas se muestran es los anexos. (**Ver anexos del V al VIII**)

2.5.7 Diagrama de Componentes

Los diagramas de componentes son usados para estructurar el modelo de implementación en términos de subsistemas de implementación y mostrar las relaciones entre sus elementos. El uso más importante de estos diagramas es mostrar la estructura de alto nivel del modelo, especificando los subsistemas de implementación y sus dependencias.

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

Dado que los diagramas de componentes muestran los componentes software que constituyen una parte reusable, sus interfaces, y sus interrelaciones, en muchos aspectos se puede considerar que un

diagrama de componentes es un diagrama de clases a gran escala. Cada componente en el diagrama debe ser documentado con un diagrama de componentes más detallado, un diagrama de clases, o un diagrama de casos de uso.

Un paquete en un diagrama de componentes representa una división física del sistema. Los paquetes se organizan en una jerarquía de capas donde cada capa tiene una interfaz bien definida. Un ejemplo típico de una jerarquía en capas de este tipo es: Interfaz de usuario; Paquetes específicos de la aplicación; Paquetes reusables y Paquetes hardware y del sistema operativo. A continuación se muestra el Diagrama de Componente para la solución propuesta:

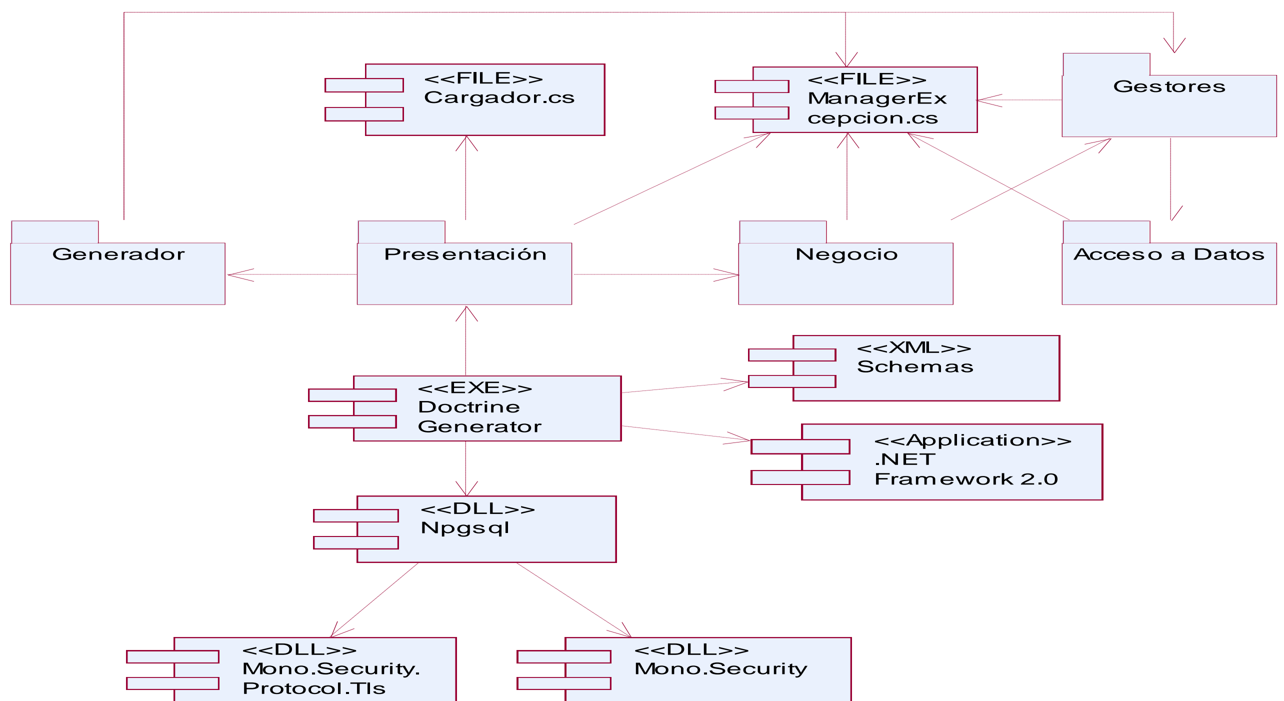


Figura 19 Diagrama de Componentes para la solución propuesta

2.5.8 Diagrama de Despliegue

El diagrame de despliegue muestra la configuración de los nodos de procesamiento en tiempo de ejecución, los vínculos de comunicación entre ellos, y las instancias de los componentes y objetos que residen en ellos. Consiste en: nodos[8]; dispositivos[9] y conectores[10]: El propósito del modelo de

[8] Elementos de procesamiento con al menos un procesador, memoria, y posiblemente otros dispositivos.

despliegue es capturar la configuración de los elementos de procesamiento y las conexiones entre estos elementos en el sistema. Permite el mapeo de procesos dentro de los nodos, asegurando la distribución del comportamiento a través de aquellos nodos que son representados.

A continuación se muestra el Diagrama de Despliegue para la solución propuesta:

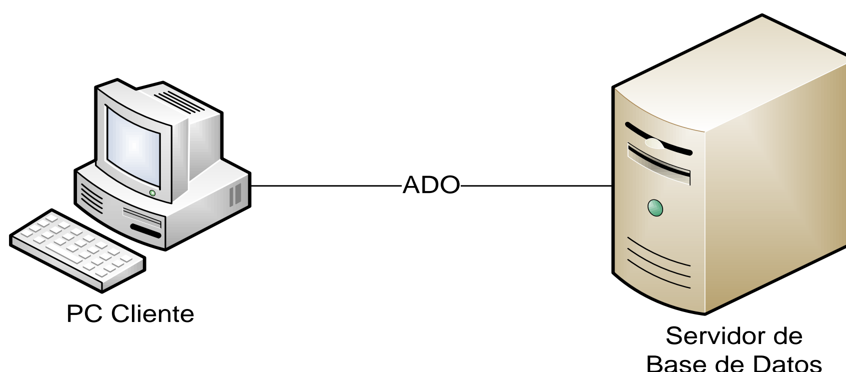


Figura 20 Diagrama de Despliegue

2.6 Modelo de Implementación

Los propósitos de la implementación son los de planificar las integraciones del sistema necesarias en cada iteración siguiendo un enfoque incremental que da lugar a un sistema que se implementa en una sucesión de pasos pequeños y manejables, implementar las clases y subsistemas encontrados durante el diseño, y finalmente probar los componentes individualmente e integrarlos enlazándolos en uno o más ejecutables antes de ser enviados para ser integrados y llevar a cabo las comprobaciones del sistema.

El modelo de implementación es comprendido por un conjunto de componentes y subsistemas que constituyen la composición física de la implementación del sistema. Entre los componentes podemos encontrar datos, archivos, ejecutables, código fuente y los directorios. Fundamentalmente, se describe la relación que existe desde los paquetes y clases del modelo de diseño a subsistemas y componentes físicos. Este artefacto describe cómo se implementan los componentes, congregándolos en subsistemas organizados en capas y jerarquías, y señala las dependencias entre éstos.

[9] Nodos estereotipados sin capacidad de procesamiento en el nivel de abstracción que se modela.
[10] Expresa el tipo de conector o protocolo utilizado entre el resto de los elementos del modelo.

Esta disciplina tiene su mayor pero durante la fase de construcción. Durante esta fase se hacen varias iteraciones liberando al final de cada una de ellas una versión del producto que bien puede ser un release para el desarrollo o un release para la producción. Cada release de producción constituye una versión del sistema.

2.7 Conclusiones

A lo largo del capítulo se abordaron temas de interés correspondientes a las etapas de diseño e implementación de la herramienta para la generación de ficheros de mapeo. Se expusieron los Diagramas de Interacción realizados para los casos de uso Gestionar Tablas, Generar Ficheros de Mapeo, Gestionar Consulta DQL, Gestionar Relación y Gestionar Fichero de Esquemas.

El análisis de estos diagramas arrojó como resultado el Modelo de Clases del Diseño compuesto por las clases de diseño y sus relaciones, además se definieron los Diagramas de Componente y Despliegue y otro grupo de artefactos definidos por la metodología seleccionada. Una vez concluida la etapa de diseño se inicia la de implementación del sistema.

A lo largo del capítulo se abordaron temas de interés para la mejor comprensión de las bases del sistema para la generación de ficheros de mapeo. Como resultado quedó definida la arquitectura del mismo sin obviar las diferentes capas por las que está compuesta. Fueron expuestos los resultados obtenidos durante el desarrollo de la herramienta, específicamente en las etapas de diseño e implementación que se reflejan en los diagramas presentados.

CAPÍTULO 3: Evaluación de la Solución Propuesta

3.1 Introducción

En el presente Capítulo se muestran algunas métricas que se aplican actualmente para validar la calidad en el diseño de software y se definen cuáles se aplicaron al diseño de la solución propuesta. Estas proporcionan una medida de cuán evolucionado se encuentra el desarrollo de la aplicación informática desde la visión interna que proporcionan los parámetros que estas definen. Además se describen las pruebas de aceptación realizadas al software y sus resultados.

3.2 Métricas de validación para el diseño.

Las métricas han de ser utilizadas para el control de los proyectos. No son ni estándares ni universales, sino que cada proyecto debe seleccionar sus propias métricas en dependencia de sus características.

Las métricas de software se pueden clasificar como:

- ✓ Métricas orientadas a la función y Métricas orientadas al tamaño.
- ✓ También se pueden clasificar según la información que entregan:
- ✓ Métricas de productividad, las que se centran en el rendimiento del proceso de ingeniería de software.
- ✓ Métricas de calidad, proporcionan una indicación de cómo se ajusta el software a los requisitos explícitos e implícitos del cliente.
- ✓ Métricas técnicas, que se centran más en el software que en el proceso a través del cual se ha desarrollado (por ejemplo grado de modularidad o grado de complejidad lógica). (35)

3.2.1 Consideraciones en la etapa del diseño

Como el análisis, el diseño de software posee una serie de métodos para apoyarlo. Cada uno de ellos posee su propia notación y heurística para acompañar el desarrollo de este proceso, pero todos ellos se basan en una serie de principios.

Los datos y los algoritmos que los manipulan deben crearse como un conjunto de abstracciones interrelacionadas. Creando abstracciones de datos y procedimientos, los modelos de los componentes

software poseen características que tienden a una alta calidad. Una abstracción es auto contenida, generalmente implementa una estructura de datos o algoritmo bien acotado, puede tener acceso a través de una interface simple, los detalles de su operación interna no necesitan ser conocida para ser utilizada en forma efectiva, y es inherentemente reusable.

Los detalles internos del diseño de las estructuras de datos y los algoritmos deben ocultarse de otros componentes software que hacen uso de dichas estructuras de datos o algoritmos. El ocultamiento de la información sugiere que los módulos sean caracterizados por decisiones de diseño que cada uno oculta de todos los demás. El ocultamiento implica que se puede alcanzar la modularidad efectiva definiendo un conjunto de módulos independientes que se comunican unos con otros pasando sólo aquella información que es necesaria para el funcionamiento del software. El uso de ocultamiento de la información como un criterio de diseño provee grandes beneficios cuando se requieren modificaciones (por ejemplo durante la evaluación, o más tarde, durante el mantenimiento). Debido a que la mayor parte de los datos y los procedimientos están ocultos para otras partes del software, los errores inadvertidos que se puedan introducir durante las modificaciones son menos propensos a propagarse a otros sitios del software.

Los módulos deben exhibir independencia. Los módulos deben estar muy poco acoplados con otros módulos y del ambiente externo, además deben poseer cohesión funcional. El software con modularidad efectiva es más fácil de desarrollar debido a que las funciones pueden ser divididas y las interfaces simplificadas (considere las ramificaciones cuando el desarrollo es conducido por un equipo). Los módulos independientes son más fáciles de mantener y evaluar debido a que los efectos secundarios causados por las modificaciones del diseño y la codificación son limitados; la propagación de errores se reduce y hace posible la creación de los módulos reusables.

Los algoritmos deben diseñarse utilizando un conjunto restringido de constructores lógicos. Este es un enfoque de diseño ampliamente conocido como programación estructurada, que fue propuesto para limitar el proceso del diseño de software a un número pequeño de operaciones predecibles. El uso de los constructores de la programación estructurada (secuencia, bifurcación e iteración), reduce la complejidad del programa, aumenta la legibilidad, evaluación y mantenimiento. El uso de un número limitado de constructores lógicos también ayuda a la comprensión humana (42).

3.3 Métricas Orientadas a Objeto

Las métricas orientadas a objetos se han introducido para ayudar a un ingeniero del software a usar el análisis cuantitativo, para evaluar la calidad en el diseño antes de que un sistema se construya. El enfoque de las métricas orientadas a objetos está en la clase, piedra fundamental en la arquitectura orientada a objetos. (43)

El Software Orientado a Objetos (OO) es fundamentalmente distinto del software que se desarrolla utilizando métodos convencionales. Las métricas para sistemas OO deben de ajustarse a las características que distinguen el software OO del software convencional. Estas métricas hacen hincapié en el encapsulamiento, la herencia, complejidad de clases y polimorfismo. Por lo tanto las métricas OO se centran en métricas que se pueden aplicar a las características de encapsulamiento, ocultamiento de información, herencia y técnicas de abstracción de objetos que hagan única a esa clase. (44)

Se conoce que las medidas y las métricas son componentes clave de cualquier disciplina de la ingeniería; la ingeniería de software orientada a objetos no es una excepción. Lamentablemente, la utilización de métricas para sistemas orientados a objetos ha progresado con mucha más lentitud que la utilización de los demás métodos OO. Sin embargo, a medida que los sistemas OO van siendo más habituales, resulta fundamental que los ingenieros del software dispongan de mecanismos cuantitativos para estimar la calidad de los diseños y la efectividad de los programas 00. (44)

Los objetivos principales de las métricas orientadas a objetos son los mismos que los existentes para las métricas surgidas para el software estructurado: (44)

- ✓ Comprender mejor la calidad del producto
- ✓ Estimar la efectividad del proceso
- ✓ Mejorar la calidad del trabajo realizado en el nivel del proyecto.

3.3.1 Algunas características de las métricas orientadas a objeto (4)

Las métricas para cualquier producto de ingeniería son reguladas por las características únicas del producto. El software orientado a objetos es fundamentalmente diferente del software desarrollado con el uso de métodos convencionales. Por esta razón, las métricas para sistemas orientados a objetos

deben ser afinadas a las características que distinguen al software orientado a objeto del software convencional.

Existen 5 características definidas por Berard que regulan las métricas especializadas: Localización, Encapsulación, Ocultamiento de Información, Herencia y Técnicas de Abstracción de Objetos.

Localización: La localización es una característica que indica la manera en que la información se concentra en un programa.

Encapsulamiento: Empaquetamiento o ligamento de una colección de elementos. Ejemplos de bajo nivel de encapsulación incluyen registros y matrices, y subprogramas (procedimientos, funciones, subrutinas y párrafos), son mecanismos de nivel medio para la encapsulación. Para los sistemas orientados a objetos, la encapsulación engloba las responsabilidades de una clase, incluyendo sus atributos (y otras clases para objetos agregados) y operaciones, y los estados de las clases, definidos por valores de atributos específicos.

La encapsulación influye en las métricas, cambiando el enfoque de las mediciones de un módulo simple, a un paquete de datos (atributos) y modelos de procesos (operaciones).

Ocultación de la Información: La ocultación de la información suprime (u oculta) los detalles operacionales de un componente de programa. Solo se proporciona la información necesaria para acceder al componente a aquellos otros componentes que deseen acceder.

Un sistema orientado a objetos bien diseñado debe implementar ocultación de información. Por esta razón, las métricas que proporcionan una indicación del grado de ocultación logrado suministran un indicio de la calidad del diseño orientado a objetos.

Herencia: La herencia es un mecanismo que habilita las responsabilidades de un objeto, para propagarse a otros objetos. La herencia ocurre a través de todos los niveles de una jerarquía de clases. Ya que la herencia es una característica vital en muchos sistemas orientados a objetos, muchos métodos orientados a objetos se centran en ella.

Abstracción: La abstracción es un mecanismo que permite al desarrollador concentrarse en los detalles esenciales de un componente de programa (ya sean datos o procesos), presentando poca atención a los detalles de bajo nivel. Como Bernard declara << la abstracción es un concepto relativo. A medida que se mueve a niveles más altos de abstracción, se ignoran más y más detalles, es decir, se tiene una visión más general de un concepto o elemento. A medida que se mueve a niveles de

abstracción más bajos, se introducen más detalles, es decir, se tiene una visión más específica de un concepto o elemento.

3.4 Métricas Orientadas a Clases

En la actualidad uno de los conjuntos de métricas más aplicados, propuestas por Chidamber y Kemerer y un total de seis métricas, son las conocidas como “La serie de métricas CK”:

- ✓ Métodos ponderados por clase (MPC).
- ✓ Árbol de profundidad de herencia (APH).
- ✓ Número de descendiente (NDD).
- ✓ Acoplamiento entre clases objeto (ACO).
- ✓ Respuesta para una clase (RPC).
- ✓ Carencia de cohesión en los métodos (CCM).

Otra de las propuestas de métricas más aplicadas son las métricas propuestas por Lorenz y Kidd, separándolas en cuatro amplias categorías:

- ✓ Tamaño
- ✓ Herencia
- ✓ Valores internos
- ✓ Valores externos

3.5 Evaluación del modelo de diseño.

Son varios los puntos de vista relacionados con la calidad del software. Desde metodologías hasta las distintas normas de calidad, que pueden estar orientados tanto a los procesos de desarrollo como a los productos de software. No es objetivo del presente trabajo abundar sobre los temas de calidad, pero si desarrollar una evaluación del diseño obtenido en la solución propuesta.

Para ello se realizó en una primera fase un mapeo entre las características funcionales asociadas a los casos de usos de Doctrine Generator y el diseño de clases construido para dar solución a estos

requisitos funcionales. De este modo se garantizó el chequeo de cumplimiento del atributo interno de calidad más importante que plantea la norma ISO 9126 (45), la Funcionalidad, que consiste en la capacidad del software de proveer las funciones que cumplen con las necesidades implícitas y explícitas cuando el mismo es utilizado bajo ciertas condiciones (46). Demostrándose que las características funcionales más importantes definidas para la solución han sido cubiertas en el diseño propuesto.

Caso Uso	Requerimiento funcional asociado al caso de uso	Clase el diseño que lo cubre	Método de la clase que lo cubre
Gestionar Tablas	Adicionar tablas.	GestorEsquema	AdicionarTabla(tabla: Tabla): void
	Eliminar tablas.		EliminarTabla(pos: int): void
	Buscar tablas.		BuscarTabla(nombre:string): Tabla
	Cargar tablas a mapear.		BuscarTablasMapear(nombre: string): int
	Buscar tablas por alias.		BuscarTablaAlias(alias: string): string
	Actualizar datos de las tablas.		ActualizarDatos(): void
	Actualizar datos de una tabla.		ActualizarDatosTabla(tabla: Tabla): void
Gestionar Consulta DQL	Adicionar Consulta DQL.	GestorTabla	AdicionarConsulta(consulta: COnsultaDQL): void
	Eliminar Consulta DQL.		EliminarConsulta(nombre: string, tabla: Tabla): void
Gestionar Relación	Crear Relaciones.	GestorTabla	CrearRelacion(tabla: Tabla): void
	Eliminar Relación.		EliminarRelacion(relac: Relacion, tabla: Tabla): void
	Modificar Tipo Relación.		ModificarRelacion(relac: Relacion, tabla: Tabla, nuevoTipo: string): void
Gestionar Fichero de Esquemas	Adicionar esquemas.	Cargador	EscribirXML(esquemas: List<Esquemas>): void
	Eliminar esquemas.		
Generar Ficheros de Mapeo	Generar ficheros de mapeo.	Generador	GeneradorM(fichero: Fichero): void GeneradorF(fichero: Fichero, limite: int, inicio: int): void GeneradorB(fichero: Fichero): void

Tabla 21 Cumplimiento del atributo de calidad Funcionalidad de la norma ISO 9126

En la segunda fase de evaluación de la calidad del diseño se aplicaron algunas métricas básicas inspiradas en el estudio de la calidad del diseño orientado a objeto referenciadas por Pressman teniendo en cuenta que este estudio brinda un esquema sencillo de implementar y que a la vez cubre los principales atributos de calidad de software. Siendo esto la principal razón de la concepción de las métricas inspiradas en lo propuesto por Pressman.

Los atributos de calidad que se abarcan dichas métricas son:

1. Responsabilidad. Consiste en la responsabilidad asignada a una clase en un marco de modelado de un dominio o concepto, de la problemática propuesta.
2. Complejidad del diseño. Consiste en la complejidad que posee una estructura de diseño de clases.
3. Complejidad de implementación. Consiste en el grado de dificultad que tiene implementar un diseño de clases determinado.
4. Reutilización. Consiste en el grado de reutilización de presente en una clase o estructura de clase, dentro de un diseño de software.
5. Acoplamiento. Consiste en el grado de dependencia o interconexión de una clase o estructura de clase, con otras, esta muy ligada a la característica de Reutilización.
6. 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.
7. 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.
8. Nivel de Cohesión. Consiste en el grado de especialización de las clases concebidas para modelar un dominio o concepto específico.
9. Abstracción del diseño. Consiste en la capacidad de modelar lo más cercano posible a la realidad un concepto o dominio determinado.

3.6 Métricas aplicadas a la solución propuesta

Uno de los conjuntos de métricas más ampliamente referenciados, es el propuesto por Chidamber y Kemerer, normalmente conocidas como la serie de métricas CK, estos autores han propuesto seis (6) métricas basadas en clases para sistemas orientados a objetos. (43)

Para la validación del diseño del software se escogieron dos (2) de las series de métricas más conocidas y divulgadas a nivel mundial, la primera fue la serie de métricas CK, la cual de sus seis (6) métricas plantadas, se escogieron dos (2) en particular, por su importancia.

Árbol de Profundidad de Herencia (APH): Esta métrica se define como la máxima longitud del nodo a la raíz del árbol. (47)

Atributo que afecta	Modo en que lo afecta
Complejidad de Mantenimiento	Un aumento del APH implica una disminución de la complejidad del mantenimiento de la clase.
Complejidad de Diseño	Un aumento del APH implica una disminución de la complejidad del diseño de la clase.
Reutilización	Un aumento del APH implica un aumento en el grado de reutilización de la clase.

Tabla 22 Árbol de Profundidad de Herencia

Número de Descendientes (NDD): Las subclases inmediatamente subordinadas a una clase de la jerarquía de clases se denominan sus descendientes. (47)

Atributo que afecta	Modo en que lo afecta
Abstracción del diseño	Un aumento del ND puede diluir la abstracción del diseño de clases.
Cantidad de pruebas	Un aumento del ND implica un aumento de la cantidad de pruebas de unidad necesarias para probar una clase.
Reutilización	Un aumento del ND implica un aumento en el grado de reutilización de la clase.
Nivel de Cohesión	Un aumento del ND implica una disminución en el nivel de cohesión de la clase.

Tabla 23 Número de Descendientes

La segunda serie que se escogió fue las métricas propuestas por Lorenz y Kidd, donde estos en su libro de métricas basadas en clases separan estas en cuatro (4) grandes categorías, de las cuales solo dos (2) de ellas se escogieron para la medición de este diseño.

Tamaño de Clases (TC): Las clases pueden medirse determinado el total de operaciones, tanto heredadas como privadas de la instancia que se encapsulan dentro de una clase, más el total de atributos, atributos tanto heredados como privados de la instancia encapsulados por la clases. (43)

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

Tabla 24 Tamaño de Clase

Número de Operaciones Redefinidas para una Sub-Clase (NOR): Es el caso de que una subclase reemplaza una operación heredada de su superclase por una versión especializada para su propio uso, a esto se le denomina redefinición. (43)

Atributo que afecta	Modo en que lo afecta
Abstracción del diseño	Un aumento del NOR implica que se ha logrado una buena definición de la abstracción del diseño de clases.
Cantidad de pruebas	Un aumento del NOR implica un aumento de la Cantidad de pruebas de unidad necesarias para probar una clase.
Complejidad del mantenimiento	Un aumento del NOR implica un aumento de la complejidad del mantenimiento de la clase.

Tabla 25 Número de Descendientes

A continuación se explicará cada una de estas métricas y se realizará el análisis de los resultados obtenidos al aplicarlas al diseño planteado anteriormente.

3.6.1 Árbol de Profundidad de Herencia (APH).

A medida que el APH crece, es posible que clases de más bajos niveles hereden muchos métodos, esto conlleva dificultades potenciales, cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clases profunda, también conduce a una complejidad mayor del diseño. Por el lado positivo, los valores de APH grandes implican un gran número de métodos que se reutilizarán. (43)

Por su parte, algunos autores sugieren que un umbral de 6 niveles como indicador es un abuso en la herencia en distintos lenguajes de programación.

Después de aplicar la métrica APH se obtuvo que el mayor nivel de herencia entre las clases del diseño es 2 (**ver anexo XII**), lo cual se encuentra dentro del umbral definido para determinar que el diseño no es complejo, que existe un bajo acoplamiento entre las clases y no es difícil su mantenimiento. Además se puede concluir que el diseño de la herramienta Doctrine Generator tiene

una calidad aceptable teniendo en cuenta que la profundidad de herencia presente en ella es siempre de 2, el cual es el valor mínimo posible. Al analizar además los atributos de calidad Complejidad de Mantenimiento y Complejidad de Diseño se puede decir que sin duda se tienen buenos índices debido a que solo se cuenta como máximo con un nivel 2 de profundidad en la herencia. Sin embargo el atributo Reutilización no posee indicadores positivos debido al bajo nivel de profundidad de herencia.

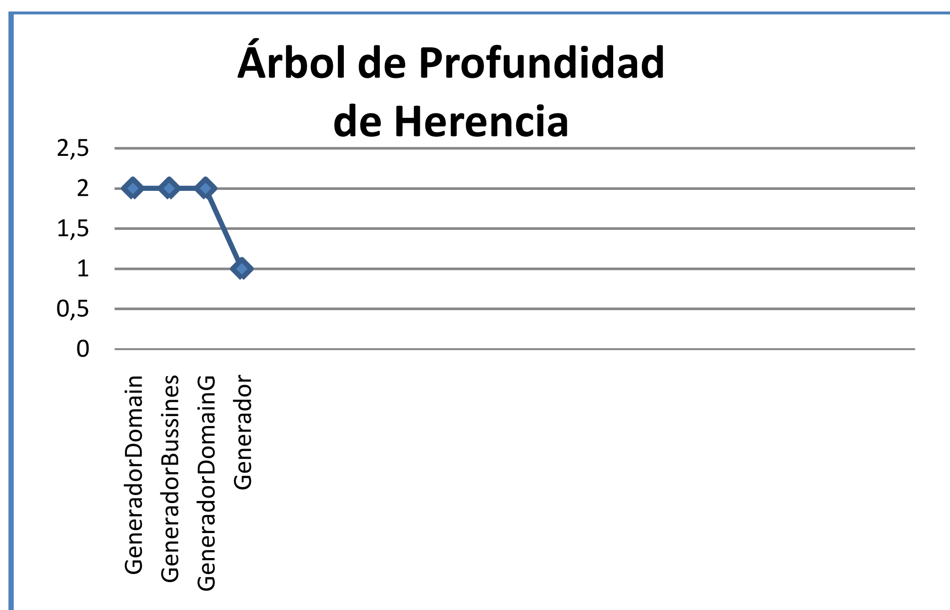


Figura 21 Niveles de herencia existentes en el diseño propuesto

3.6.2 Número de Descendientes (NDD).

A medida que el número de descendientes crece la reutilización se incrementa, pero además cuando el NDD crece, la abstracción representada por la clase predecesora puede diluirse. Esto significa que existe la posibilidad de que algunos descendientes no sean miembros realmente apropiados de la clase predecesora. A medida que el NDD crece, la cantidad de pruebas se incrementará también. (43)

Con esta métrica se evalúan algunos parámetros de calidad como la reutilización, la abstracción, la cohesión y la cantidad de pruebas. En los anexos (**ver anexo XI**) se muestran los umbrales definidos para estos parámetros y los rangos de valores para la evaluación técnica de estos parámetros.

Después de haber aplicado la métrica de NDD, se llegó a la conclusión de que el sistema presenta como máximo nivel de descendientes 1 (**ver anexo XII**), este nivel se encuentra en los umbrales que se proponen para esta métrica.

A continuación se muestran los resultados por parámetro de calidad.

REUTILIZACIÓN	Cantidad de Clases
Baja	3
Media	0
Alta	1

Tabla 26 Resultados para el parámetro de calidad Reutilización

ABSTRACCIÓN	Cantidad de Clases
Indefinida	0
Afectada	0
Definida	4

Tabla 27 Resultado para el parámetro de calidad Abstracción

COHESIÓN	Cantidad de Clases
Baja	0
Media	0
Alta	4

Tabla 28 Resultado para el parámetro de calidad Cohesión

CANTIDAD DE PRUEBAS	Cantidad de Clases
Baja	4
Media	0
Alta	0

Tabla 29 Resultado para el parámetro de calidad Cantidad de Pruebas

Con estos resultados se puede arribar a la conclusión de que el diseño propuesto tiene una calidad aceptable teniendo en cuenta que solo se emplea la herencia en casos bien identificados a partir de las necesidades del negocio o del diseño. Solo 2 clases poseen descendientes y en ninguno de estos 2 casos la cantidad de descendientes supera la cantidad de 3.

Valorando el atributo de calidad Reutilización se puede decir que por lo antes explicado los índices de Reutilización se mantienen bajos representados por un 75%. En cuanto al atributo Abstracción de la clase base se muestra como existe una tendencia a la conservación de la abstracción. Tanto para la Cohesión de la jerarquía de clases como para el atributo Cantidad de Pruebas los índices son positivos favoreciendo esto al diseño.

3.6.3 Tamaño de Clases (TC).

Los valores grandes para esta métrica, indican que la clase debe tener bastante responsabilidad. Esto reducirá la reutilización de esta clase y complicará la implementación y las pruebas. Se pueden calcular los promedios para el número de atributos y operaciones de clase. Cuando menor sea el valor del promedio para el tamaño será más posible que las clases dentro del sistema puedan ser

reutilizadas. Las medidas o umbrales para los parámetros de calidad han sido una polémica a nivel mundial en el diseño de sistemas.

El tamaño general de una clase se puede determinar empleando medidas para saber el número total de operaciones (tanto operaciones heredadas como privadas de la instancia) que están encapsuladas dentro de la clase así como encontrando el número de atributos (tanto atributos heredados como atributos privados de la Instancia) que están encapsulados en la clase. Si existen valores grandes de TC éstos mostrarán que una clase puede tener demasiada responsabilidad, lo cual reducirá la reutilizabilidad de la clase y complicará la implementación y la comprobación, por otra parte cuanto menor sea el valor medio para el tamaño, más probable es que las clases existentes dentro del sistema se puedan reutilizar ampliamente. La mayor cantidad de clases que involucran procesos presentes en la herramienta se encuentra en la Capa de Negocios, fue a esta capa a la que se le aplicó la métrica del tamaño de clase (TC).

En los anexos (**ver anexo XIII**) se muestra la tabla de resultados de la evaluación técnica y su influencia en los parámetros de calidad Responsabilidad, Complejidad de Implementación y Reutilización.

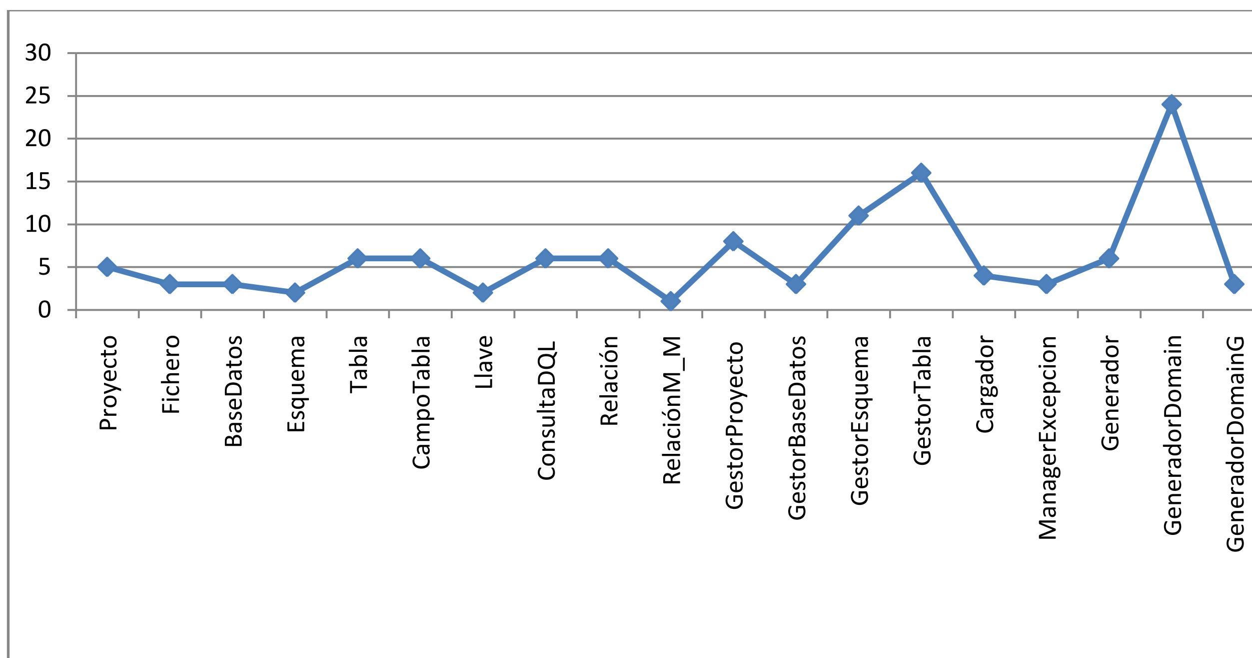


Figura 22 Representación de las clases de la capa de negocio según cantidad de operaciones

Teniendo en cuenta las medidas o umbrales de referencia en lo que respecta al número de operaciones y/o atributos de las clases se establece que un tamaño de clase pequeño es aquel que

tiene un valor menor o igual que 20. Un tamaño de clase medio es aquel cuyos valores exceden a 20 y son menores o incluyen a 30 y un tamaño de clase grande es aquel que es mayor que este último valor (30). Así se concluye que la Capa de Negocio cuenta con 20 clases, para un promedio de cantidad de atributos de 2.6 y un promedio de cantidad de operaciones de 5.9.

Los valores de tamaño quedan distribuidos de la siguiente manera:

Umbral	Tamaño	Cantidad de clases
Pequeño	<=20	19
Medio	>20 y <=30	1
Grande	>30	0

Tabla 30 Distribución de valores de tamaño de las clases de la capa de negocio

Como se puede observar las clases están clasificadas entre pequeñas y medianas, siendo el 95% clases pequeñas, lo cual brinda un resultado positivo según los parámetros de calidad Responsabilidad, Complejidad de Implementación y Reutilización propuestos para esta métrica tratada en este sub-epígrafe.

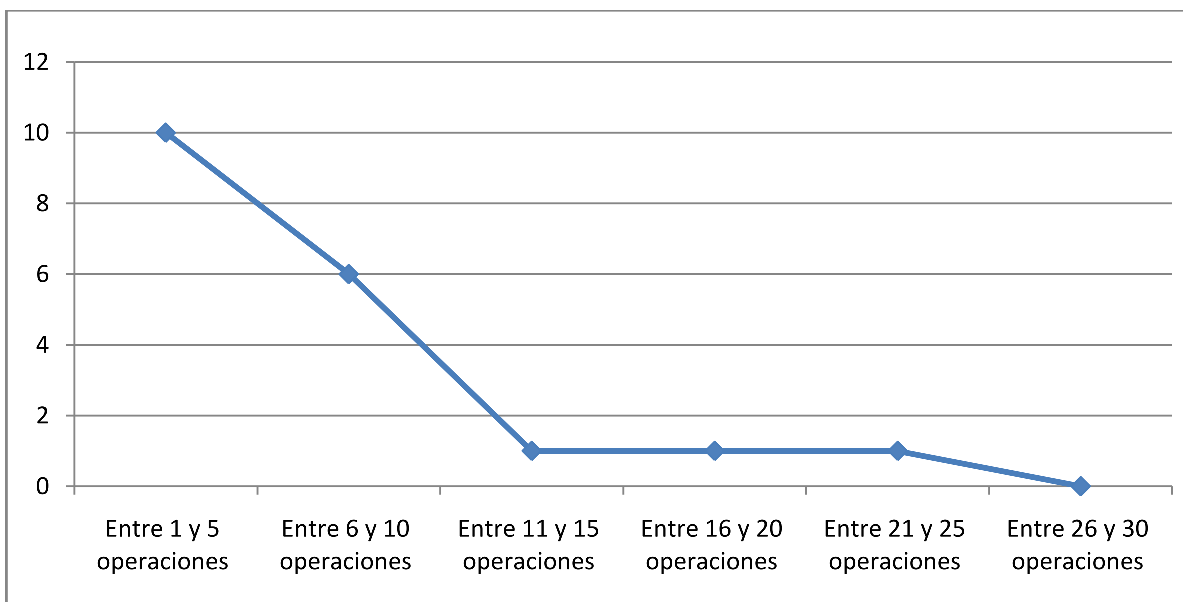


Figura 23 Representación de los resultados obtenidos en el instrumento agrupados en los intervalos definidos

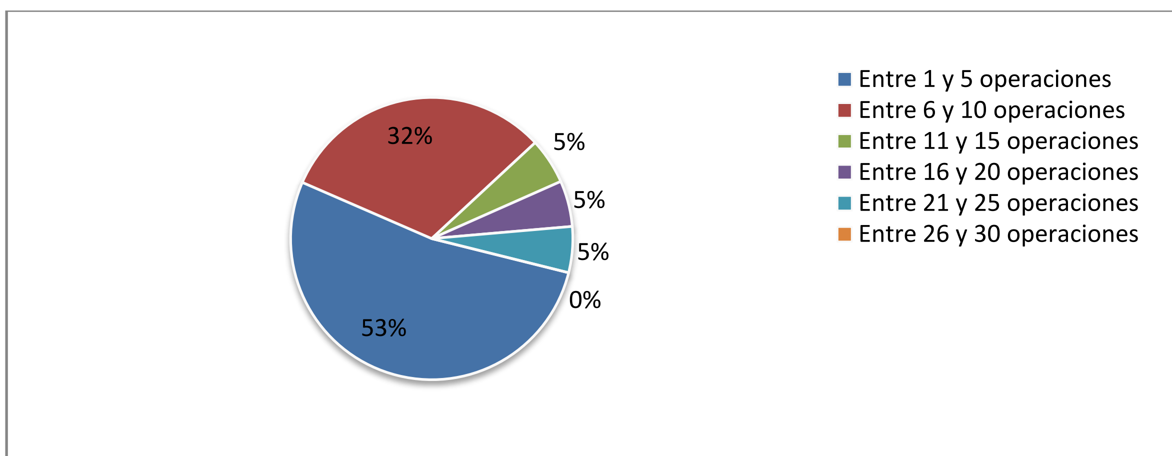


Figura 24 Representación en % de los resultados obtenidos en el instrumento agrupados en los intervalos definidos

3.6.4 Número de Operaciones Redefinidas para una Sub-Clase (NOR).

Los valores grandes para el NOR, generalmente indica un problema en el diseño, o sea si el NOR es grande el diseñador ha violado la abstracción representada por la superclase. Esto provoca una débil jerarquía de clases y un software orientado a objetos, que puede ser difícil de probar y modificar. (43)

A partir de los datos obtenidos después de aplicarle al sistema la métrica NOR se obtuvo que de 38 clases que tiene el sistema ninguna subclase reemplaza operaciones definidas en las superclases. Además los indicadores se comportan de forma adecuada para los atributos de calidad Complejidad del Mantenimiento, Cantidad de Pruebas y Violación de la Abstracción representada por la superclase. Con estos resultados podemos concluir que:

- ✓ Existe una jerarquía adecuada.
- ✓ El software puede ser fácil de probar y modificar sin afectar tanto el tiempo que requieran los cambios.

3.7 Matriz de cubrimiento de los parámetros de calidad evaluados con las métricas propuestas.

La matriz de cubrimiento o matriz de inferencia de indicadores de calidad es el resumen de los resultados obtenidos al aplicar las métricas mencionadas en el epígrafe anterior. Esta matriz es una representación estructurada de los atributos de calidad y métricas utilizadas para evaluar la calidad del diseño de la solución propuesta. La misma permite conocer si el resultado obtenido de la relación atributo/métricas es positivo o negativo. Llevando estos resultados a una escalabilidad numérica

donde, si los resultados son positivos tendrá un valor de 1, si son negativos de 0 y si no existe relación alguna tomará valor -1.

Una vez completado los datos de dicha relación se determina el promedio de los valores obtenidos de la relación atributo/métrica (solo se toman en consideración las que arrojan un resultado distinto de -1). Este valor representa el impacto que tiene cada atributo en el diseño de la solución determinando si su impacto fue bueno, regular o malo. Al desarrollar este procedimiento con los resultados que se obtuvieron una vez se aplicaron las métricas en el diseño propuesto se obtuvo la matriz de cubrimiento que se representa a continuación.

Calificativo	Valor
Positivo	1
Negativo	0
Nulo	-1

Tabla 31 Clasificación de los parámetros de calidad según impacto en el diseño propuesto

Calificativo	Rango
Malo	<= 0.4
Regular	>0.4 y <=0.7
Bueno	>0.7

Tabla 32 Rangos para evaluar el impacto de los parámetros de calidad en el diseño propuesto

Atributos de calidad evaluados	Métricas aplicadas al diseño de la solución propuesta				
	APH	NDD	TC	NOR	PROMEDIO
Responsabilidad	1	-1	-1	-1	1
Complejidad en el diseño	-1	1	-1	-1	1
Complejidad en la implementación	1	1	-1	-1	1
Reutilización	1	-1	0	-1	0,5
Complejidad del mantenimiento	-1	1	-1	1	1
Cantidad de pruebas	-1	-1	1	1	1
Nivel de cohesión	-1	-1	1	-1	1
Abstracción del diseño	-1	-1	1	1	1

Tabla 33 Matriz de cubrimiento para los parámetros de calidad evaluados con las métricas aplicadas al diseño propuesto

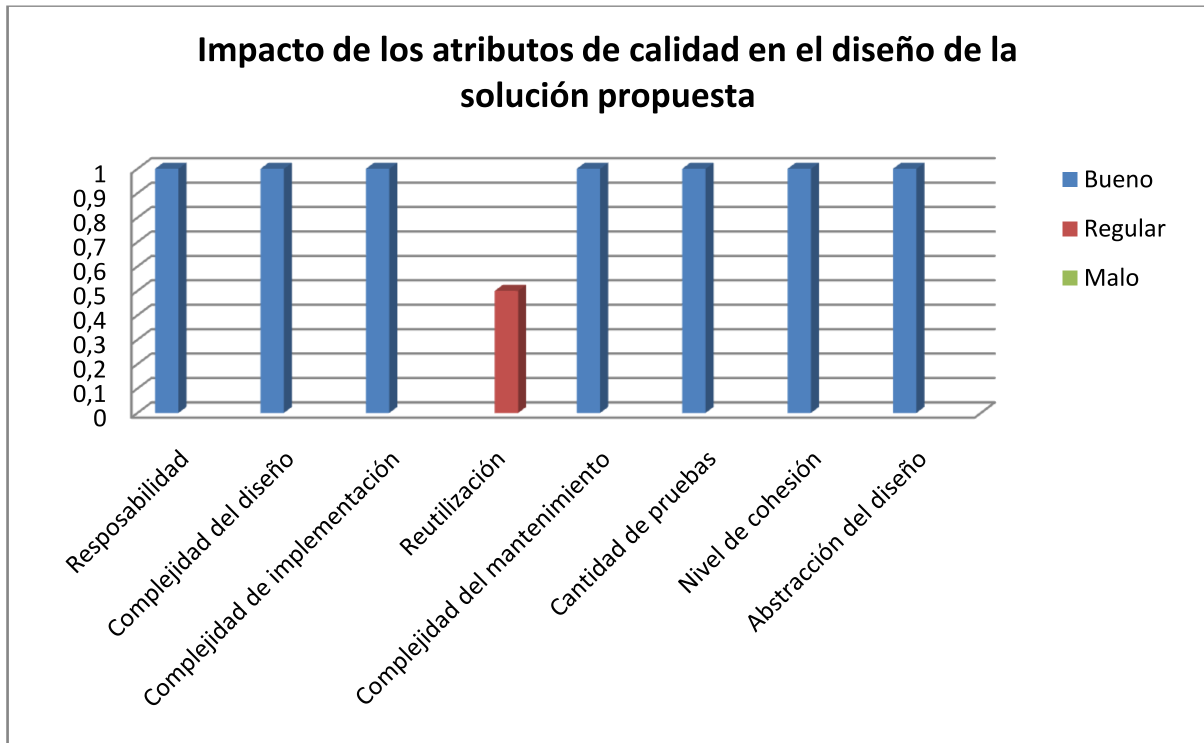


Figura 25 Impacto de los atributos de calidad en el diseño de la solución propuesta

3.8 Pruebas realizadas a la solución propuesta

En la creación de software se requiere gran esfuerzo mental por parte de los desarrolladores por lo que la posibilidad de cometer errores es muy alta. No se puede asegurar que un producto es ciento por ciento fiable ni que cumplirán al máximo con las expectativas de los clientes. La realización de pruebas es uno de los métodos que se emplea para garantizar la calidad y el buen funcionamiento de un producto. Estas no confirman la ausencia de errores en el software, solo brindan una medida de cómo responderá el mismo ante algunas situaciones determinadas. (30)

Algunos aspectos respecto a las pruebas:

- ✓ **El objetivo es encontrar defectos:** El propósito principal de las pruebas es validar la corrección de cualquier artefacto que se este probando. En otras palabras, las pruebas exitosas encuentran defectos.
- ✓ **Se pueden validar todos los artefactos**: Se pueden probar todos los artefactos no solamente el código fuente. Como mínimo es posible revisar los modelos y documentos y por lo tanto encontrar y corregir los defectos antes que lleguen al código.

- ✓ **Probar contra el riesgo de un artefacto:** Entre más riesgoso es algo, más es necesario que sea revisado y probado.
- ✓ **Una prueba vale mil opiniones:** Usted puede decirme que su aplicación funciona, pero mientras no se muestren los resultados de las pruebas, no será una opinión de fiar.

3.8.1 Pruebas de Usuario

Pruebas de Aceptación

El usuario comprueba en su propio entorno de explotación si acepta el software como está o precisa ser necesario aplicar nuevas optimizaciones y soluciones de fallas.

- ✓ En las pruebas funcionales del software todo se ejecutaba según lo requerido.
- ✓ Las pruebas de rendimiento, tanto el uso CPU, uso memoria, ping a distancias y otros variables son aceptables para una ejecución fluida y de excelente percepción para el usuario final.

Al sistema se le aplicaron pruebas de aceptación con el objetivo de verificar las funcionalidades de la solución propuesta. Para esto se utilizó el método de caja negra usando la técnica de partición de equivalencia.

Pressman presenta la partición equivalente como un método de prueba de caja negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba (CP). Un caso de prueba ideal descubre de forma inmediata una clase de errores que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico. La partición equivalente se dirige a la definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar. (48)

Una clase de equivalencia representa un conjunto de estados válidos o no válidos para condiciones de entrada. Típicamente, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica.

El objetivo de partición equivalente es reducir el posible conjunto de casos de prueba en uno más pequeño, un conjunto manejable que evalúe bien el software. Se toma un riesgo porque se escoge no probar todo. Así que se necesita tener mucho cuidado al escoger las clases. (49)

La partición equivalente es subjetiva. Dos probadores quienes prueban un programa complejo pueden llegar a diferentes conjuntos de particiones.

En el diseño de casos de prueba para partición equivalente se procede en dos pasos (50):

Se identifican las clases de equivalencia. Las clases de equivalencia son identificadas tomando cada condición de entrada (generalmente una oración o una frase en la especificación) y repartiéndola en dos o más grupos.

Es de notar que dos tipos de clases de equivalencia están identificados: las clases de equivalencia válidas representan entradas válidas al programa, y las clases de equivalencia inválidas que representan el resto de los estados posibles de la condición (es decir, valores erróneos de la entrada).

Se define los casos de prueba. El segundo paso es el uso de las clases de equivalencia para identificar los casos de prueba. El proceso es como sigue: se asigna un número único a cada clase de equivalencia. Hasta que todas las clases de equivalencia válidas han sido cubiertas por los casos de prueba, se escribe un nuevo caso de prueba que cubra la clase de equivalencia válida.

Por último hasta que los casos de prueba hallan cubierto todas las clases de equivalencia inválidas, se escribe un caso de la prueba que cubra una, y solamente una, de las clases de equivalencia inválidas descubiertas. A continuación se detallan los casos de pruebas definidos para la herramienta.

3.8.2 Casos de Prueba

CP #1: Crear Nuevo Proyecto

1. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Crear Nuevo Proyecto	Se crea un nuevo proyecto definiendo nombre y ubicación del mismo, las opciones de conexión y los esquemas y tablas con los que se va a interactuar.	EP 1.1: Iniciar un nuevo proyecto.	El formulario se muestra cuando se ejecuta la aplicación o cuando se presiona el botón Crear Nuevo Proyecto de la interfaz principal. Se introduce el nombre del proyecto y la ubicación donde se generarán los ficheros. Se presiona el botón Siguiente o la pestaña Opciones de Conexión. Se introducen los datos para establecer la conexión. Se presiona el botón Siguiente. Se seleccionan los esquemas

			para trabajar. Se seleccionan las tablas a mapear. Se presiona el botón Finalizar.
		EP 1.2: Iniciar un nuevo proyecto sin asignarle nombre.	El formulario se muestra cuando se ejecuta la aplicación o cuando se presiona el botón Crear Nuevo Proyecto de la interfaz principal. Se presiona el botón Siguiente sin introducir nombre al proyecto. Se presiona el botón Aceptar o se cierra la ventana que muestra el mensaje de información.
		EP 1.3: Iniciar nuevo proyecto introduciendo errores en los datos de la conexión.	El formulario se muestra cuando se ejecuta la aplicación o cuando se presiona el botón Crear Nuevo Proyecto de la interfaz principal. Se introduce el nombre del proyecto y la ubicación donde se generarán los ficheros. Se presiona el botón Siguiente o la pestaña Opciones de Conexión. Se introducen los datos del módulo que se desea adicionar en el formulario introduciendo errores en los datos. Se presiona el botón Aceptar o se cierra la ventana que muestra el error de conexión.
		EP 1.4: Iniciar nuevo Proyecto sin seleccionar esquemas ni tablas.	El formulario se muestra cuando se ejecuta la aplicación o cuando se presiona el botón Crear Nuevo Proyecto de la interfaz principal. Se introduce el nombre del proyecto y la ubicación donde se generarán los ficheros. Se presiona el botón Siguiente o la pestaña Opciones de Conexión. Se introducen los datos para establecer la conexión. Se presiona el botón Siguiente. Se presiona el botón Finalizar.
		EP 1.5: Cancelar.	El formulario se muestra cuando se ejecuta la aplicación o cuando se presiona el botón Crear Nuevo Proyecto de la interfaz principal. Se introducen o no datos. Se presiona el botón Cancelar.

Tabla 34 Requisitos a probar del CP Crear Nuevo Proyecto

2. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 1.1	Nuevo Proyecto	Para el botón **Finalizar**: Se inicia un nuevo proyecto y se guarda la información introducida, se cierra la ventana y se muestra la interfaz principal.	Se muestra la interfaz principal con el árbol de objetos de mapeo cargado con la información seleccionada.
EP 1.2	Nuevo Proyecto (nombre vacío)	Para el botón **Siguiente:** Se deja en blanco el campo de texto nombre del proyecto y sale un mensaje que indica que no puede estar en blanco.	Se muestra el mensaje "Debe asignar un nombre al proyecto"
EP 1.3	Nuevo Proyecto (/*-/-?!)	Para el botón **Siguiente:** Se muestra el mensaje "ERROR DE CONEXIÓN"	Se muestra el mensaje "ERROR DE CONEXIÓN"
EP 1.4	Nuevo Proyecto()	Para el botón **Finalizar**: Se inicia un nuevo proyecto y se guarda la información introducida, se cierra la ventana y se muestra la interfaz principal con el árbol de objetos de mapeo con el nombre del proyecto.	Se muestra la interfaz principal con el árbol de objetos de mapeo solo con el nombre del proyecto.
EP 1.5	NA	Se cancela la operación.	Se cancela la operación y se muestra la interfaz principal.

Tabla 35 Juego de datos para probar el del CP Crear Nuevo Proyecto

CP #2: Abrir Proyecto Existente

1. Condiciones de ejecución.
   - ✓ Debe existir un fichero (*.dg) con la información de un proyecto guardado con anterioridad.

2. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Abrir Proyecto Existente	Se abre un proyecto a través de un cuadro de diálogo.	EP 2.1: Abrir Proyecto Existente.	Se ejecuta la aplicación. Se presiona el botón Abrir Proyecto Existente de la ventana Nuevo Proyecto, del menú Archivo de la interfaz principal o de la barra de acceso rápido.

Tabla 36 Requisitos a probar del CP Abrir Proyecto Existente

3. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 2.1	Abrir Proyecto	Para el botón **Abrir**: Se carga un proyecto guardado con anterioridad y se muestra en el árbol de objetos de mapeo los datos de las tablas contenidas en el proyecto.	Se muestra la interfaz principal con el árbol de objetos de mapeo cargado con la información.

Tabla 37 Juego de datos para probar el del CP Abrir Proyecto Existente

CP #3: Generar Ficheros de Mapeo

1. Condiciones de ejecución.
   - ✓ Deben estar un proyecto iniciado.
   - ✓ Deben estar definidas las relaciones entre las tablas cargadas en el proyecto.
2. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Generar Ficheros de Mapeo	Se generan los ficheros de mapeo para cada una de las tablas contenidas en el proyecto especificando las opciones de generación.	EP 3.1: Generar Ficheros de Mapeo.	Se seleccionan las opciones de generación. Se presiona el botón Siguiente. Se muestra un listado de los ficheros generados y la ubicación de los mismos.

Tabla 38 Requisitos a probar del CP Generar Ficheros de Mapeo

3. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 3.1	Generar Ficheros de Mapeo	Para el botón **Siguiente**: Se generan los ficheros de mapeo de acuerdo a las opciones seleccionadas anteriormente.	Se muestra una interfaz con un listado de los ficheros generados y la ubicación de los mismos.

Tabla 39 Juego de datos para probar el del CP Generar Ficheros de Mapeo

CP #4: Construir ConsultaDQL

1. Condiciones de ejecución.
   - ✓ Deben estar seleccionada una tabla del árbol de objetos de mapeo.

2. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Construir ConsultaDQL	Se construye una consulta DQL que se añade al fichero de mapeo correspondiente a la tabla que esté seleccionada.	EP 4.1: Construir ConsultaDQL.	Se selecciona una tabla del árbol de objetos de mapeo de la interfaz principal. Se presiona el botón Construir ConsultaDQL del menú de acceso rápido o del menú Generar. Se construye la consulta. Se presiona el botón Aceptar.
		EP 4.2: Construir ConsultaDQL con paginado.	Se selecciona una tabla del árbol de objetos de mapeo de la interfaz principal. Se presiona el botón Construir ConsultaDQL del menú de acceso rápido o del menú Generar. Se construye la consulta. Se activa el paginado. Se presiona el botón Aceptar.
		EP 4.3: Cancelar.	Se selecciona una tabla del árbol de objetos de mapeo de la interfaz principal. Se presiona el botón Construir ConsultaDQL del menú de acceso rápido o del menú Generar. Se presiona el botón Cancelar.

Tabla 40 Requisitos a probar del CP Construir ConsultaDQL

3. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 4.1	Construir ConsultaDQL.	Para el botón **Aceptar**: Se crea una nueva ConsultaDQL, se selecciona no cuando se pide activar el paginado, se adiciona a la tabla seleccionada y se cierra la ventana.	Se muestra la interfaz principal y se actualiza el árbol de objetos de mapeo.
EP 4.2	Construir ConsultaDQL con paginado.	Para el botón **Aceptar**: Se crea una nueva ConsultaDQL, se selecciona si cuando se pide activar el paginado, se adiciona a la tabla seleccionada y se cierra la ventana.	Se muestra la interfaz principal y se actualiza el árbol de objetos de mapeo.
EP 4.3	NA	Se cancela la operación.	Se cancela la operación y se muestra la interfaz principal.

Tabla 41 Juego de datos para probar el del CP Construir ConsultaDQL

CP #5: Adicionar Relación

1. Condiciones de ejecución.
   - ✓ Debe estar iniciado un proyecto.
   - ✓ Debe estar seleccionada una tabla del árbol de objetos de mapeo.
2. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Adicionar Relación	Se adiciona una relación a una tabla seleccionada del árbol de objetos de mapeo.	EP 5.1: Adicionar Relación	Se selecciona una tabla del árbol de objetos de mapeo. Se selecciona la opción Adicionar Relación del menú secundario. Se introducen los datos de la relación.
		EP 5.2: Adicionar Relación con campos vacíos.	Se selecciona una tabla del árbol de objetos de mapeo. Se selecciona la opción Adicionar Relación del menú secundario. Se introducen los datos de la relación dejando campos vacíos. Se presiona el botón Aceptar o se cierra la ventana que muestra el mensaje de información.
		EP 5.3: Cancelar	Se selecciona la opción Adicionar Relación del menú secundario. Se introducen o no los datos en el formulario. Se presiona el botón Cancelar.

Tabla 42 Requisitos a probar del CP Adicionar Relación

3. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 5.1	Adicionar Relacion	Para el botón **Adicionar**: Se adiciona la relación creada y se guarda la información.	Se limpian los campos para introducir nuevas relaciones.
EP 5.2	Adicionar Relacion (Vacío)	Para el botón Adicionar: Se deja en blanco algún campo y se mantiene en la misma ventana de adicionar relación.	Se mantiene en la ventana de adicionar relación.
EP 5.3	NA	Se cancela la operación.	Se cierra la ventana de Adicionar Relación y se muestra la interfaz principal.

Tabla 43 Juego de datos para probar el del CP Adicionar Relación

CP #6: Modificar Relación

1. Condiciones de ejecución.
   - ✓ Debe estar seleccionada una tabla en el árbol de objetos de mapeo.
   - ✓ Es necesario que exista la relación y esta debe estar seleccionada.
2. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Modificar Relación	Se selecciona una relación y se modifica su tipo.	EP 6.1: Modificar Relación	Se selecciona la relación que se va a modificar. Se presiona el botón izquierdo del mouse y se selecciona la opción Modificar Tipo Relación. Se selecciona el tipo nuevo. Se presiona el botón Aceptar.

Tabla 44 Requisitos a probar del CP Modificar Relación

3. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 6.1	Modificar Relación	Para el botón **Aceptar**: Se modifica el tipo de la relación seleccionada correspondiente a la tabla seleccionada y se actualiza el árbol de objetos de mapeo.	Se cierra la ventana de modificar relación, se muestra la interfaz principal con los datos actualizados.

Tabla 45 Juego de datos para probar el del CP Modificar Relación

CP #7: Eliminar Relación

1. Condiciones de ejecución.
   - ✓ Debe estar seleccionada una tabla en el árbol de objetos de mapeo.
   - ✓ Debe estar seleccionada una relación existente.
2. Requisitos a probar

Nombre del requisito	Descripción general	Escenarios de pruebas (EP)	Flujo del escenario
Eliminar Relación	Se selecciona una relación y se elimina.	EP 7.1: Eliminar Relación	Se selecciona la relación que se desea eliminar. Se presiona el botón secundario del ratón y se selecciona la opción Eliminar Relación. Se actualiza el listado de relaciones.

Tabla 46 Requisitos a probar del CP Eliminar Relación

3. Juegos de datos a probar

Id escenario	Nombre	Respuesta del sistema	Resultado de la prueba
EP 7.1	Eliminar Relación	Para la opción **Eliminar Relación**: Se elimina la relación seleccionada correspondiente a la tabla seleccionada y se actualiza el árbol de objetos de mapeo.	Se muestra la interfaz principal con los datos actualizados.

Tabla 47 Juego de datos para probar el del CP Eliminar Relación

El principal objetivo del diseño de casos de prueba es obtener un conjunto de pruebas que tengan la mayor probabilidad de descubrir los defectos del software. Para validar los requisitos funcionales del sistema se definieron 7 Casos de Prueba que posibilitaron el descubrimiento y la corrección de un gran número de errores antes de que la aplicación fuese entregada a los clientes.

3.9 Conclusiones

En este capítulo se logró aplicar una serie de métricas para la validación del diseño propuesto, en el cual se determinó que este no presenta una alta complejidad estructural, de datos, ni del sistema en general, permitiendo que las pruebas no sean complejas.

La métrica de Tamaño de Clases evidencia que la mayoría de las clases son reutilizables, la implementación no es complicada y las pruebas no son complejas, al igual que la profundidad de los niveles de herencia están acorde con los umbrales definidos por los autores consultados permitiendo este un bajo acoplamiento y que el sistema no sea complejo. Además como el número de subclases que redefinen métodos no es alto, da la posibilidad de que el sistema pueda ser probado fácilmente y se pueda modificar en corto tiempo.

La matriz de cubrimiento o matriz de inferencia de los atributos de calidad permitió valorar el impacto que tuvieron los atributos que se midieron con las métricas aplicadas en el diseño de la solución propuesta. Permitiendo además concluir que el diseño propuesto no presenta complejidad, es fácil de implementar y dar mantenimiento, tiene una alta cohesión y es fácil de probar.

Por último se logró aplicar a la herramienta pruebas de caja negra mediante el método partición de equivalencia lo que permitió verificar que las funcionalidades implementadas responden a las necesidades y propósitos de los clientes.

CONCLUSIONES GENERALES

Se realizó un estudio del arte de los temas relacionados con la investigación que sentó las bases para el posterior desarrollo de la misma.

Se dio cumplimiento al objetivo de la investigación mediante la creación de una herramienta para la generación de ficheros de mapeo mediante el framework Doctrine.

La utilización de métricas para la validación del diseño propuesto permitió evaluar la calidad del mismo.

La aplicación de las pruebas de aceptación a la aplicación desarrollada permitió constatar la calidad de la misma.

RECOMENDACIONES

Se recomienda para futuras iteraciones y versiones del sistema propuesto:

Ampliar a otros lenguajes de programación la generación de las clases e interfaces.

Realizar las plantillas necesarias para el uso de XSLT, para la generación de clases e interfaces en distintos lenguajes.

Integrar Doctrine Generator con la aplicación N-HiberGen para ampliar las conexiones a otros gestores y la generación de ficheros a lenguajes como Java y C# 2005.

BIBLIOGRAFÍA

1. **Metodología de Investigación [En línea]. [En línea] [Citado el: 5 de 11 de 2008.] http://www.aibarra.org/investig/tema0.htm.**

2. Bunge, Mario. **La Ciencia, Su método y su filosofía. 2006.**

3. **Introducción al Diseño Estructurado. [En línea]. [En línea] http://www.chaco.gov.ar/UTN/disenodesistemas/apuntes/de/Unidad_1.html.**

4. Leyet Fernández, Osmar y Rodríguez Lorenzo, Iosmel. **Desarrollo de una herramienta generadora de ficheros de mapeo para la persistencia de objetos relacionales basada en NHibernate. La Habana : s.n., 2008.**

5. **Los Modelos Dinámicos y la Ingeniería del Software [En línea]. [En línea] [Citado el: 5 de 12 de 2008.] http://www.sc.ehu.es/jiwdocoj/remis/docs/modelos.html.**

6. Rumbaugh, J Jacobson y Booch, Greddy. **The Unified Modeling Language. s.l. : Addison-Wesley, 1998.**

7. Jacobson, I., Booch, G y Rumbaugh, J. **El Proceso Unificado de Desarrollo de Software. s.l. : Addison Wesley, 2000. Vol. I..**

8. López Góngora, Dalgis Rogelio y García Fernández, Daniel Mariano. **Implementación de módulo de reportes de Sistemas para la gestión económica de los Registros y Notarias de la República Boivariana de Venezuela. 2007.**

9. **Microsoft. [En línea] . [En línea] [Citado el: 15 de 12 de 2008.] http://www.microsoft.com/net/.**

10. **ciberaula. [En línea]. [En línea] [Citado el: 10 de 12 de 2008.] http://www.ciberaula.com/curso/puntonet/que_es/.**

11. **Department of Software and Computing Systems . [En línea] University of Alicante _ EPSA. [Citado el: 15 de 12 de 2008.] http://www.dlsi.ua.es/asignaturas/dpaa/tema1.pdf.**

12. **El Placer del Saber. [En línea]. [En línea] 2008. [Citado el: 08 de 12 de 2008.] http://elplacerdelsaber.com.ar/2008/04/visual-studio-net-2003.html.**

13. **Departamento de Lenguajes y Sistemas Informáticos. [En línea] [Citado el: 15 de 12 de 2008.] http://www.lsi.us.es/~javierj/investigacion_ficheros/Framework.pdf.**

14. **Página oficial de .Net Framework . [En línea] http://msdn.microsoft.com/netframework/ .**

15. **Maestros del Web. [En línea] [Citado el: 19 de enero de 2009.] http://www.maestrosdelweb.com/principiantes/%C2%BFque-son-las-bases-de-datos/.**

16. **Monografías.com. [En línea] [Citado el: 16 de enero de 2009.] http://www.monografias.com/trabajos5/basede/basede.shtml.**

17. ELIES. [En línea] [Citado el: 15 de enero de 2009.] http://elies.rediris.es/elies9/4-1-2.htm.

18. Mundo Geek. [En línea] [Citado el: 10 de enero de 2009.] http://mundogeek.net/archivos/2004/08/26/modelo-de-datos/.

19. Communications of the ACM. págs. 377-387. Vols. 13, issue=6.

20. Buen Master. [En línea] [Citado el: 20 de enero de 2009.] http://buenmaster.com/?a=554.

21. M. P. ATKINSON, F. BANCILHON, D. J. DEWITT, K. R. DITTRICH, D. **The object-oriented database system. s.l. : SIGMOD Conference, May 1990.**

22. MEYER, BERTRAND. **Object Oriented Software Construction 2nd Edition. s.l. : Prentice Hall, 1997.**

23. Visconti, Marcello y Astudillo, Hernán. **Fundamentos de Ingeniería de Software. Esquemas, Patrones y Persistencia.**

24. Departamento de Informático de la Universidad e Vigo. **Sitio Web del Departamento de Informático de la Universidad e Vigo. Departamento de Informático de la Universidad e Vigo. [En línea] [Citado el: 19 de 12 de 2008.] http://www.lsi.uvigo.es/lsi/erosello/imo/articulos/rendimiento.pdf.**

25. Motores de Persistencia. [En línea] [Citado el: 19 de 12 de 2008.] http://www.ufg.edu.sv/ufg/theorethikos/Julio04/mdp6.html.

26. LARMAN, Craig. **"UML y Patrones". s.l. : Prentice Hall, 2000. ISBN-84-205-3438-2.**

27. Sturm, Jack. **Desarrollo de soluciones XML. s.l. : McGraw Hill, 2001. ISBN: 8448131363.**

28. Motores de Persistencia. [En línea] . [En línea] http://petra.euitio.uniovi.es/~i1643233/MotoresDePersistencia-v02.pdf.

29. Doctrine - PHP Object Relational Mapper. [En línea] [Citado el: 17 de enero de 2009.] http://www.doctrine-project.org/.

30. Rodríguez Luque, David y Rodríguez Luque, Daniel. **Herramienta para la generación de código de aplicaciones WEB. La Habana : s.n., 2007.**

31. HERRINGTON, J. **Code Generation In Action. Manning, 2006. 280 p. .**

32. MORENO, P. J. M., Ed. **Especificación de interfaz de usuario: De los requisitos a la generación de código. Universidad de Valencia : s.n., 2003.**

33. Software Acumen Limited, Code Generator Models, [en línea]. Software Acumen Limited, Code Generator Models. [En línea] febrero de 2005. http://www.codegeneration.net/tiki-index.PHP?page=ModelsIntroduction .

34. Agil Unified Process. [En línea] http://www.ambysoft.com/unifiedprocess/agileUP.html.

35. Maribel Silva Muñoz, Sándor Rodríguez Prieto. **Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela. La Habana : s.n., 2008.**

36. IBM Corporation, “IBM Rational Rose” [en línea]. IBM Corporation, “IBM Rational Rose” [en línea]. [En línea] 2005. [Citado el: 29 de enero de 2009.] http://www-306.ibm.com/software/rational/ .

37. Clements, Paul. **A Survey of Architecture Description Languages. Proceedings of the International Workshop on Software Specification and Design. Alemania : s.n., 1996.**

38. Clements, Paul y Bachmann, Felix. **Software Architecture in Practice. 2do Edition. s.l. : Weslwy, Addison.**

39. Arquitectura de Software. [En línea] [Citado el: 3 de marzo de 2009.] http://siona.udea.edu.co/~aoviedo/Arquitectura%20de%20Software/Arquitectura%20de%20Software.htm.

40. Trowbridge, David, y otros. **Enterprise Solution Patterns Using Microsoft .NET.**

41. Gamma, E., y otros. **Design Patterns. s.l. : Addison-Wesley, 1995.**

42. Dirección de Información. [En línea] Junio de 2000. [Citado el: 12 de marzo de 2009.] http://bibliodoc.uci.cu/pdf/controldecalidad.pdf.

43. Pressman, Roger S. **Ingenería de Software. Un enfoque práctico. 1998. Vol. I.**

44. Métricas para Sistemas Orientados a Objetos. [En línea] [Citado el: 8 de abril de 2009.] http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/gonzalez_d_h/capitulo6.pdf.

45. Figueroa, María Antonieta Abud. **Revista UPIICSA. [En línea] [Citado el: 2009 de mayo de 10.] http://www.revistaupiicsa.20m.com/Emilia/RevEneAbr04/Antonieta1.pdf.**

46. ARREGUI, J. J. O. **Revisión Sistemática de Métricas de Diseño Orientado a Objetos. Madrid, España : s.n., Septiembre, 2005.**

47. Chidamber, S. R. y Kemerer, C. F. **A Metrics Suite for Object-Oriented Design. s.l. : IEEE Trans. Software Engineering, Junio de 1994.**

48. Pressman., Roger S. **Ingeniería del software, un enfoque práctico, pages 281-322. . s.l. : McGrawHill, 2002. Quinta Edition.**

49. Patton, Ron. **Software Testing, page 408. s.l. : Sams Publishing, 2005.**

50. Myers, Glenford J. **The art of sotware Testing, page 234. . s.l. : Wiley, 2004. Second Edition.**

51. Architect's and Develoepers Microsoft Technology. [En línea]. [En línea] http://peruti.wordpress.com/2007/10/18/%C2%BFque-es-un-framework-%C2%BFcomo-puedo-desarrollar-un-framework-propio.

52. Taller Introductorio. [En línea] . [En línea] http://ange.nireblog.com/post/2008/03/04/taller-introductorio.

53. Larman, Craig. **UML y patrones. Introducción al análisis y diseño orientado a objetos. [En línea] [Citado el: 14 de febrero de 2009.]**

54. Diseño y Programación Orientada a Objetos. [En línea] [Citado el: 14 de febrero de 2009.] http://www.info-ab.uclm.es/asignaturas/42579/pdf/04-Capitulo4a.pdf.

55. Fidel Gil, Javier Albrigo, Javier Do Rosario. **SISTEMAS DE GESTIÓN DE BASE DE DATOS . SGBD / DBMS. [En línea] 14 de febrero de 2005. [Citado el: 20 de febrero de 2009.] http://alfa.facyt.uc.edu.ve/computacion/pensum/cs0347/download/exposiciones/1/SGBD.pdf.**

56. López, Alien Rodríguez. **Diseño e Implementación de la solución informática para la Gestión de las Negativas Registrales en la Dirección General de Registros y Notarías de la República Bolivariana de Venezuela. Trabajo de Diploma para optar por el título de Ingeniero Informático. Caracas-Venezuela : s.n., 2008.**

57. Espinoza, Humberto. **PostgreSQL. Una Alternativa de DBMS Open Source. [En línea] [Citado el: 20 de febrero de 2009.] http://www.lgs.com.ve/pres/PresentacionES_PSQL.pdf.**

58. Roberth G. Figueroa, Camilo J. Solís y Armando A. Cabrera. **METODOLOGÍAS TRADICIONALES VS METODOLOGÍAS ÁGILES. [En línea] [Citado el: 24 de febrero de 2009.]**

GLOSARIO DE TÉRMINOS

Arquitectura de software: Consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información. La Arquitectura de Software establece 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.

Base de datos o banco de datos: Es un conjunto de datos pertenecientes al mismo contexto y almacenados sistemáticamente para su posterior uso.

Bases de datos Relacionales: Son las más utilizadas hoy en día. No poseen métodos para almacenamiento de objetos. Se requieren de servicios especiales para almacenar objetos en las tablas.

Calidad de Software: "Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente".

Clase: Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase).

Caso de Prueba: Conjunto de condiciones o variables bajo las cuáles el analista determinará si el requisito de una aplicación es parcial o completamente satisfactorio.

Diseño de Software: El proceso de aplicar distintas técnicas y principios con el propósito de definir un producto con los suficientes detalles como para permitir su realización física. El diseño es la primera etapa técnica del proceso de Ingeniería del Software, consiste en producir un modelo o representación técnica del software que se va a desarrollar.

Doctrine: Doctrine es un ORM (Objet Relational – Mapping) alternativo a Propel y que pronto estará integrado en Symfony. Posee una poderosa capa de abstracción de Base de Datos (BD). Una de sus características es la opción de escribir consultas de BD en un objeto apropiado orientado al dialecto SQL llamado y que se le denomina Lenguaje de Consulta de Doctrine o DQL del inglés Doctrine Query Language, inspirado por el Lenguaje de Consulta de Hibernate (HQL por sus siglas en inglés). Este

proporciona a los desarrolladores de una poderosa alternativa al SQL que mantiene la flexibilidad sin requerir duplicación de código innecesario.

Esquema de persistencia: Es un conjunto reutilizable de clases que presentan servicios a los objetos persistentes. Se utiliza para trabajar con bases de datos relacionales, una API de servicios de datos orientados a registros (Microsoft ODBC) u otro mecanismo de almacenamiento. Debe traducir los objetos a registros para guardarlos en una base de datos y viceversa.

Esquema relacional de persistencia de objetos: Traduce los objetos a registros y a la inversa para almacenarlos en algún medio de almacenamiento.

Estilos Arquitectónicos: Indican los tipos de componentes y conectores involucrados. Patrones y restricciones de interconexión o composición entre ellos. Asociados a cada estilo hay una serie de propiedades que lo caracterizan.

Framework: Es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un Framework puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Framework de persistencia: Permite abstraernos en gran medida del sistema de almacenamiento utilizado, pudiendo en muchas ocasiones mezclar almacenamiento en bases de datos relacionales, archivos XML o incluso datos provenientes de servicios web, todo ello de forma completamente transparente para las capas superiores.

Lenguaje de modelado de objetos: Es un conjunto estandarizado de símbolos y de modos de disponerlos para modelar (parte de) un diseño de software orientado a objetos. El uso de un lenguaje de modelado es más sencillo que la auténtica programación, pues existen menos medios para verificar efectivamente el funcionamiento adecuado del modelo.

Mapeo: Relación entre una clase y su almacenamiento persistente (p.ej. una tabla de la BD), y entre los atributos del objeto y los campos (columnas) de un registro.

Métrica: En el campo de la ingeniería del software una métrica es cualquier medida o conjunto de medidas destinadas a conocer o estimar el tamaño u otra característica de un software o un sistema de información, generalmente para realizar comparativas o para la planificación de proyectos de desarrollo.

Métricas de Diseño: Son medidas aplicadas al diseño de una solución y que permiten evaluar el diseño teniendo en cuenta parámetros de calidad tales como la reutilización, la abstracción, la responsabilidad y la complejidad de las clases que componen el diseño.

Métricas de Calidad: Es el término que describe muchos y muy variados casos de medición. Siendo una métrica una medida estadística (no cuantitativa como en otras disciplinas ejemplo física) que se aplica a todos los aspectos de calidad de software, los cuales deben ser medidos desde diferentes puntos de vista como el análisis, construcción, funcional, documentación, métodos, proceso, usuario, entre otros.

Metodología de desarrollo de software: Las metodologías de desarrollo de software son un conjunto de procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos software.

Metodología Ágil: Se basan en retrasar las decisiones y la planificación adaptativa; permitiendo potenciar aún más el desarrollo de software a gran escala. Estas metodologías ponen en relevancia que la capacidad de respuesta a un cambio es más importante que el seguimiento estricto de un plan. Como parte de ésta aparecen Extreme Programing (XP), SCRUM y Ágil Unified Process (AUP).

Modelo relacional: para la gestión de una base de datos es un modelo de datos basado en la lógica de predicado y en la teoría de conjuntos. Es el modelo más utilizado en la actualidad para modelar problemas reales y administrar datos dinámicamente. Su idea fundamental es el uso de «relaciones». Estas relaciones podrían considerarse en forma lógica como conjuntos de datos llamados «tuplas».

Objetos-complejos: Materializaciones de estructuras complejas de objetos.

Objeto-persistente: Objeto instanciado en memoria que debe ser almacenado en un medio no volátil.

Objeto-Relacional: Este modelo combina las ventajas de los dos anteriores: la base de datos es relacional por lo que conserva su rapidez y eficiencia, pero permite hacer uso de nuevos elementos que modelan los objetos a esta base de datos relacional, con lo que el analista y diseñador ve un modelo orientado a objetos.

Patrones: Normas de comportamiento, características que identifican una situación. En informática un patrón es una solución a un problema de diseño no trivial que es efectiva y reutilizable. Es posible aplicar a diferentes problemas de diseño en distintas circunstancias.

Patrones de diseño: (Design patterns) son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño es una solución a un problema de diseño.

PHP: Lenguaje de programación orientado a objetos.

PostgreSQL: Es un servidor de base de datos relacional orientada a objetos de software libre, liberado bajo la licencia BSD. Como muchos otros proyectos open source, el desarrollo de PostgreSQL no es manejado por una sola compañía sino que es dirigido por una comunidad de desarrolladores y organizaciones comerciales las cuales trabajan en su desarrollo. Dicha comunidad es denominada el PGDG (PostgreSQL Global Development Group).

Proceso de desarrollo de software: "es aquel en que las necesidades del usuario son traducidas en requerimientos de software, estos requerimientos transformados en diseño y el diseño implementado en código, el código es probado, documentado y certificado para su uso operativo". Concretamente "define quién está haciendo qué, cuándo hacerlo y cómo alcanzar un cierto objetivo"

Programación Orientada a Objetos (OOP por sus siglas en inglés de Object Oriented Programming) como paradigma, "es una forma de pensar, una filosofía, de la cual surge una cultura nueva que incorpora técnicas y metodologías diferentes. Pero estas técnicas y metodologías, y la cultura misma, provienen del paradigma, no lo hacen. La OOP como paradigma es una postura ontológica: el universo computacional está poblado por objetos, cada uno responsabilizándose por sí mismo, y comunicándose con los demás por medio de mensajes".

Pruebas de Software: Son los procesos que permiten verificar y revelar la calidad de un producto software.

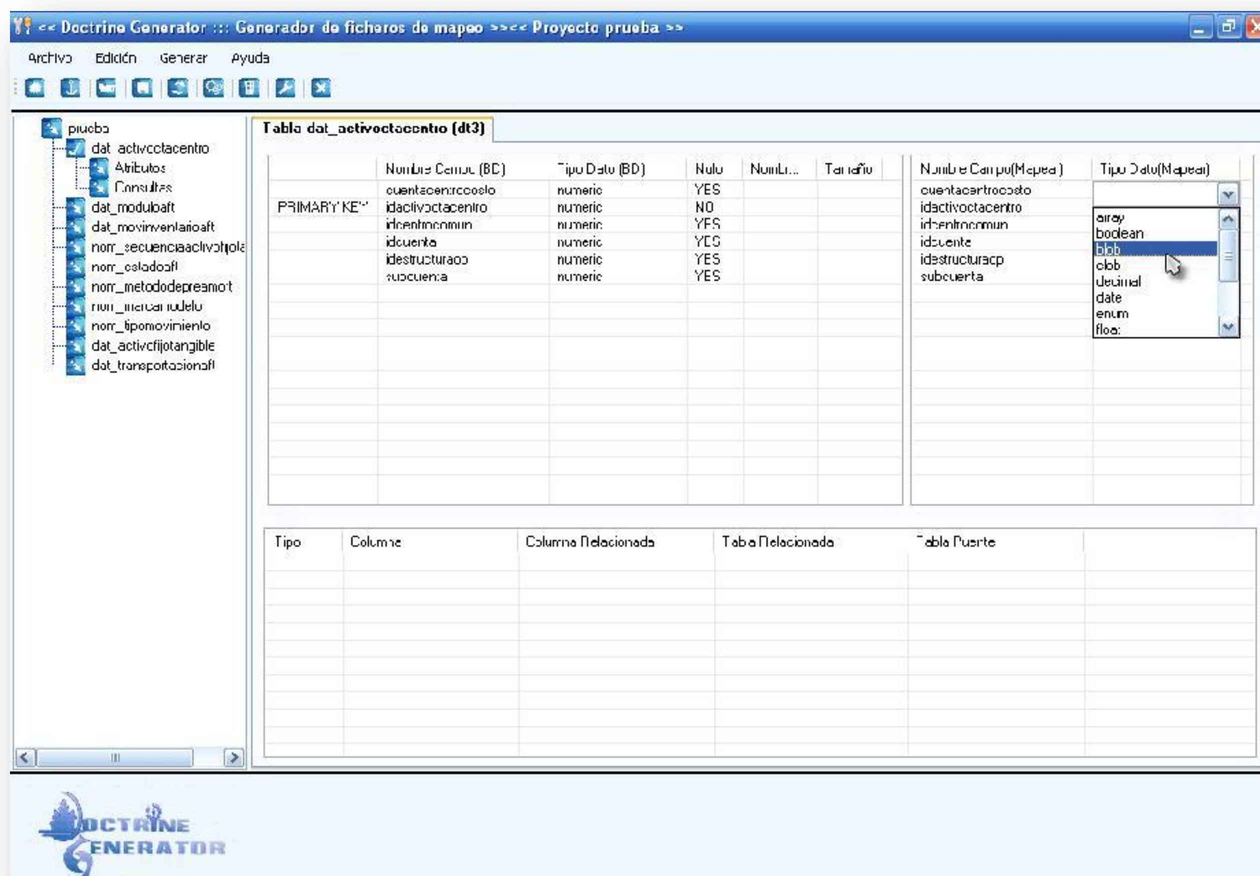
Pruebas de Aceptación: Estas pruebas las realiza el cliente. Son básicamente pruebas funcionales, sobre el sistema completo, y buscan una cobertura de la especificación de requisitos y del manual del usuario. Estas pruebas no se realizan durante el desarrollo, pues sería impresentable al cliente; sino que se realizan sobre el producto terminado e integrado o pudiera ser una versión del producto o una iteración funcionad pactada previamente con el cliente.

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). XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes

necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

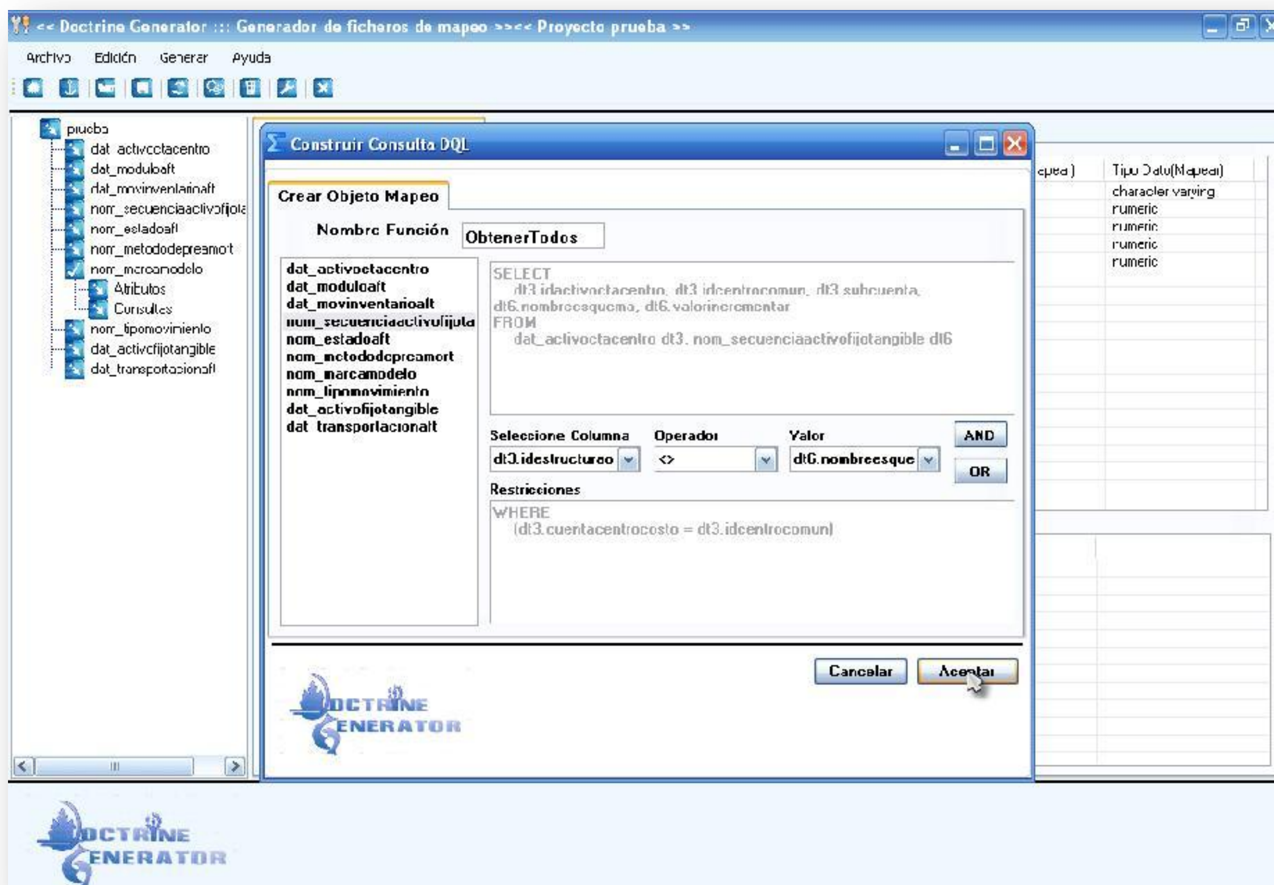
ANEXOS

Anexo I: Interfaz que permite personalizar las salidas de los ficheros y la configuración de las relaciones.

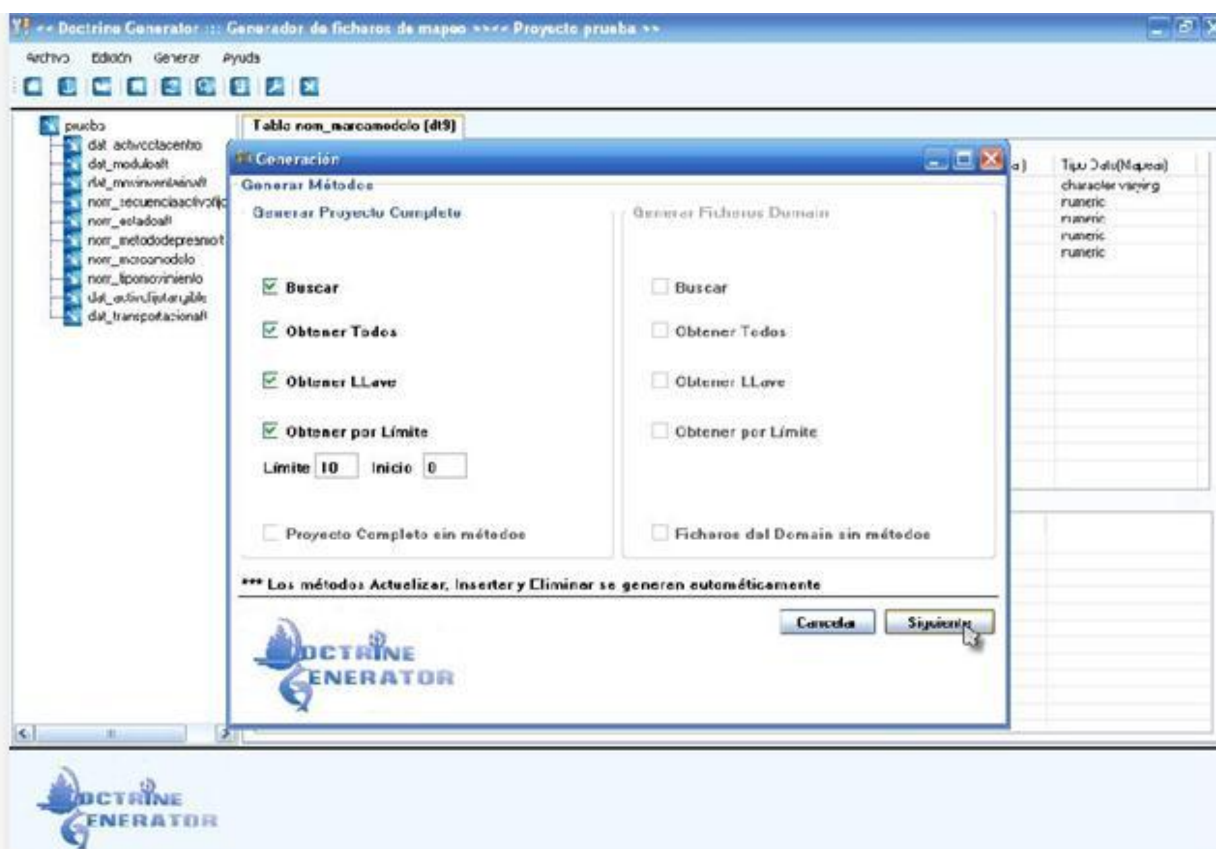


Anexo II: Interfaz que permite la configuración de las relaciones.

Anexo III: Interfaz que permite la configuración algunas funcionalidades que se le añadirán a los ficheros generados.



Anexo IV: Interfaz que permite la configuración de los ficheros que serán generados.



Anexo V: Diagramas de Colaboración y Secuencia para el Caso de Uso Generar Ficheros de Mapeo.

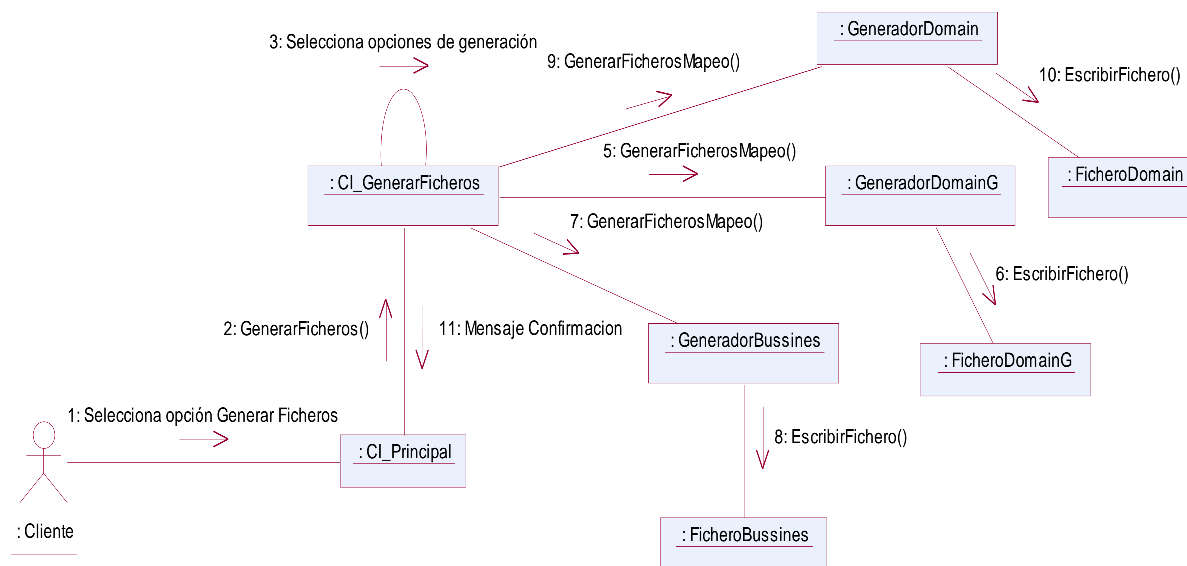


Figura 26 Diagrama de Colaboración: Caso de Uso Generar Ficheros de Mapeo

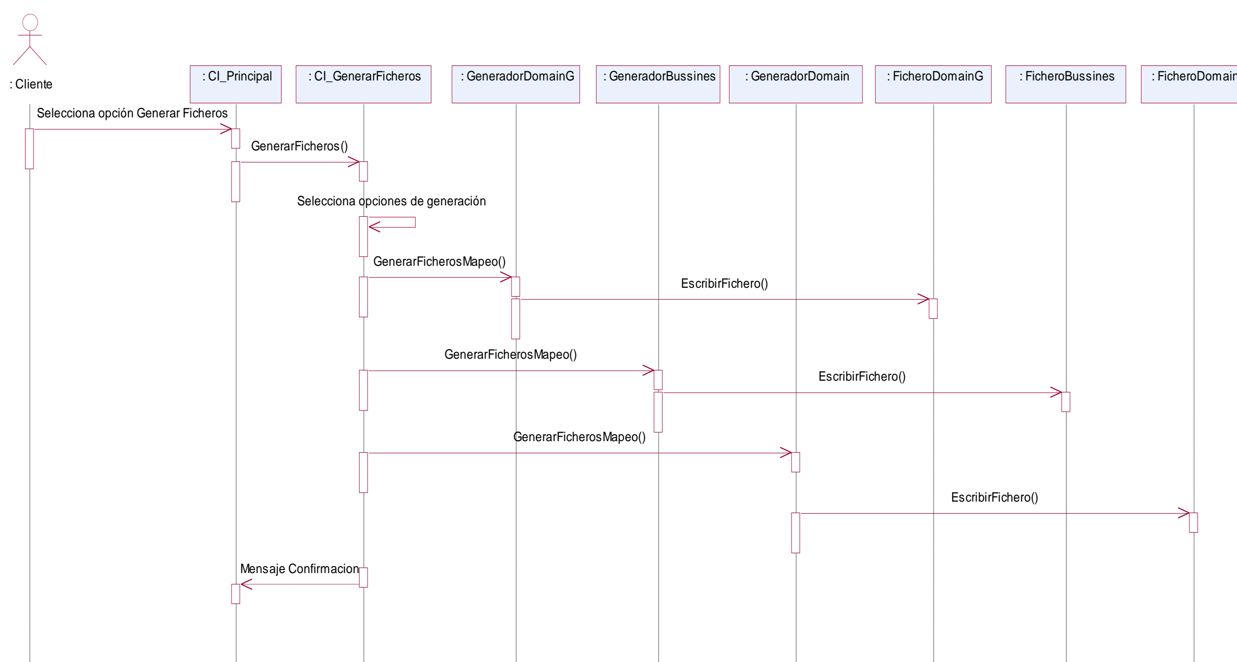


Figura 27 Diagrama de Secuencia: Caso de Uso Generar Ficheros de Mapeo

Anexo VI: Diagramas de Colaboración y Secuencia para el Caso de Uso Gestionar Consulta DQL.

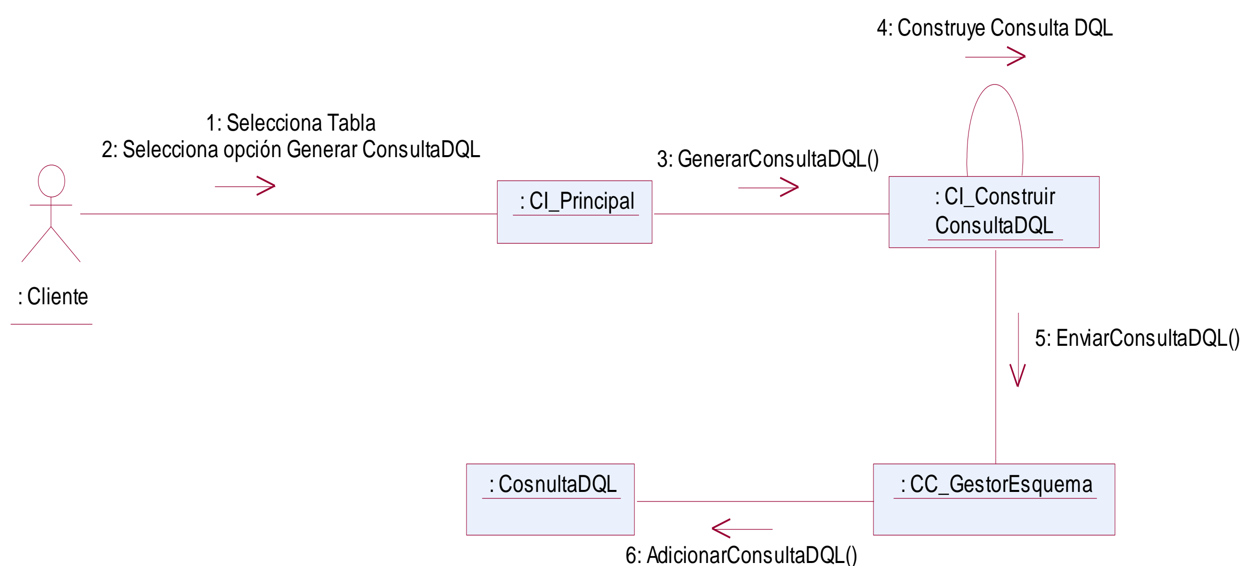


Figura 28 Diagrama de Colaboración: Caso de Uso Gestionar ConsultaDQL. Escenario Adicionar ConsultaDQL

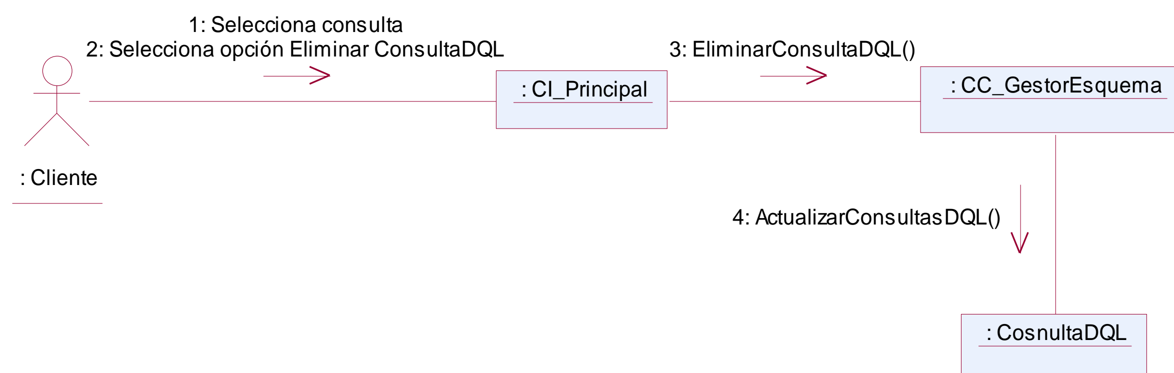


Figura 29 Diagrama de Colaboración: Caso de Uso Gestionar ConsultaDQL. Escenario Eliminar ConsultaDQL

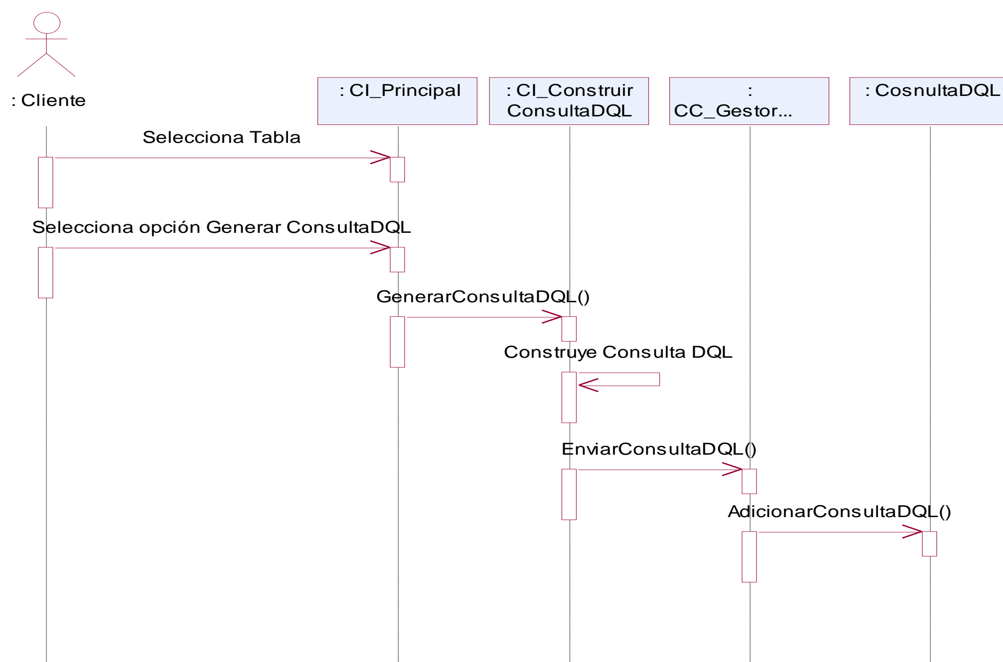


Figura 30 Diagrama de Secuencia: Caso de Uso Gestionar ConsultaDQL. Escenario Adicionar ConsultaDQL

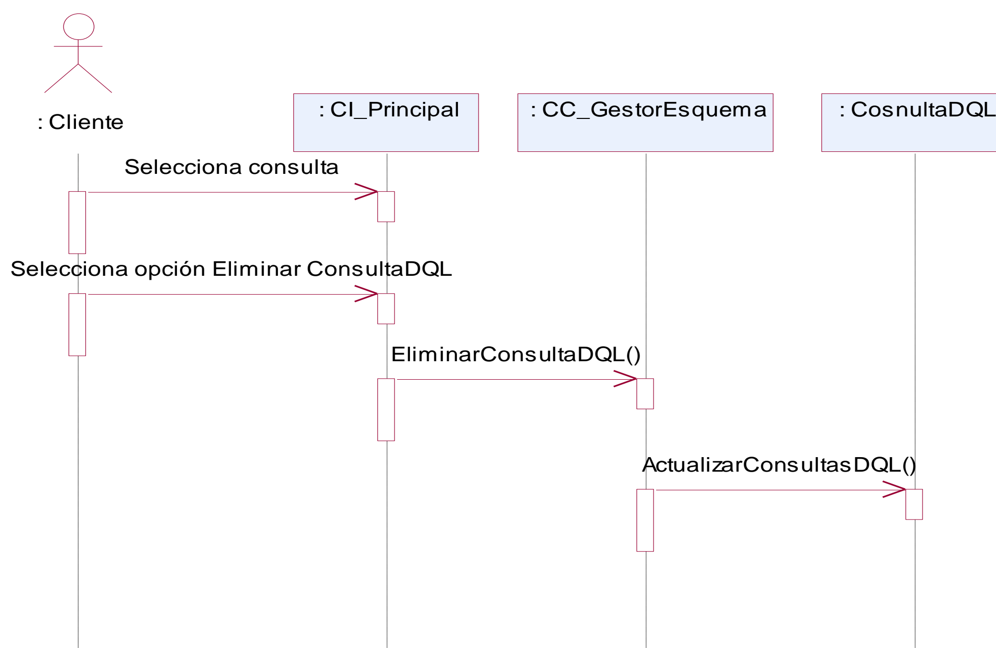


Figura 31 Diagrama de Secuencia: Caso de Uso Gestionar ConsultaDQL. Escenario Eliminar ConsultaDQL

Anexo VII: Diagramas de Colaboración y Secuencia para el Caso de Uso Gestionar Relación.

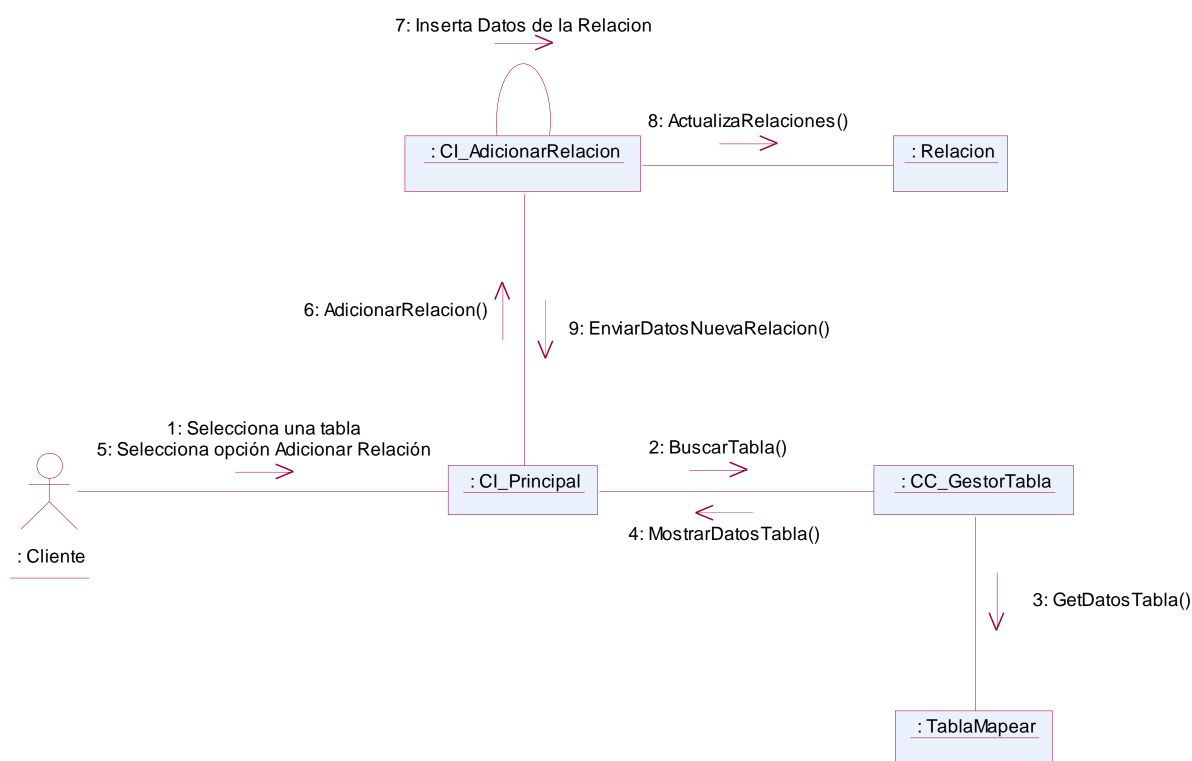


Figura 32 Diagrama de Colaboración: Caso de Uso Gestionar Relación. Escenario Adicionar Relación

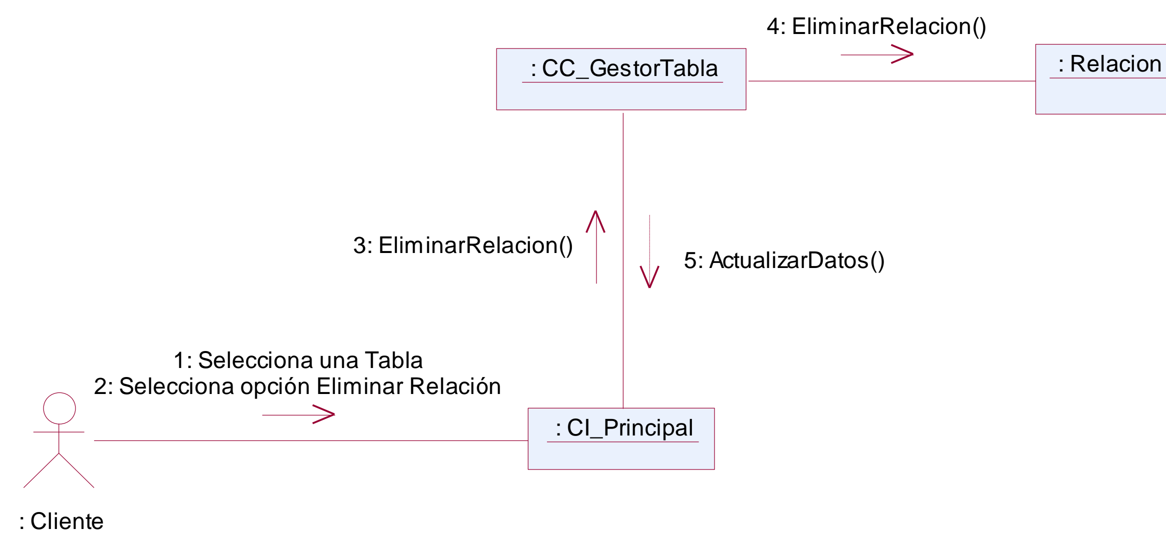


Figura 33 Diagrama de Colaboración: Caso de Uso Gestionar Relación. Escenario Eliminar Relación

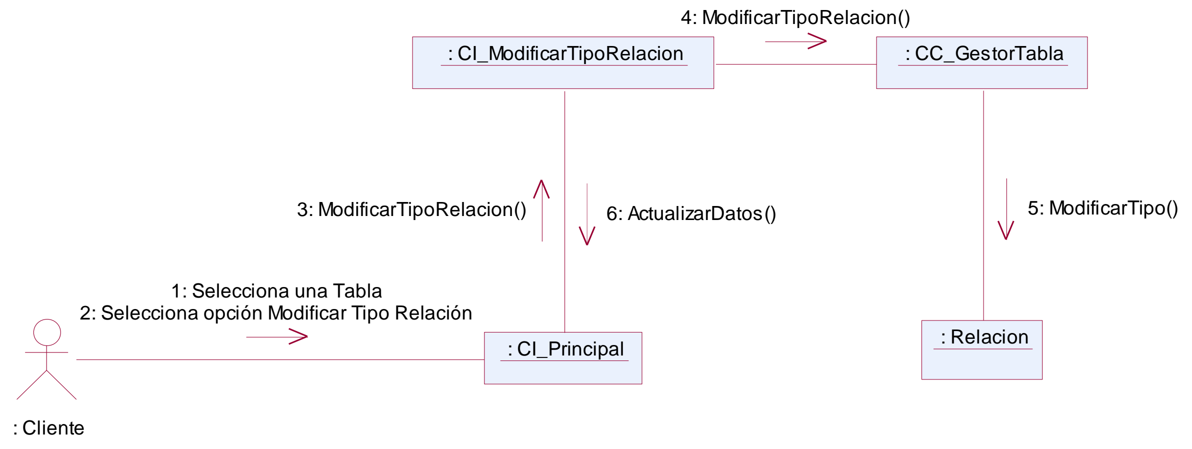


Figura 34 Diagrama de Colaboración: Caso de Uso Gestionar Relación. Escenario Modificar Tipo Relación

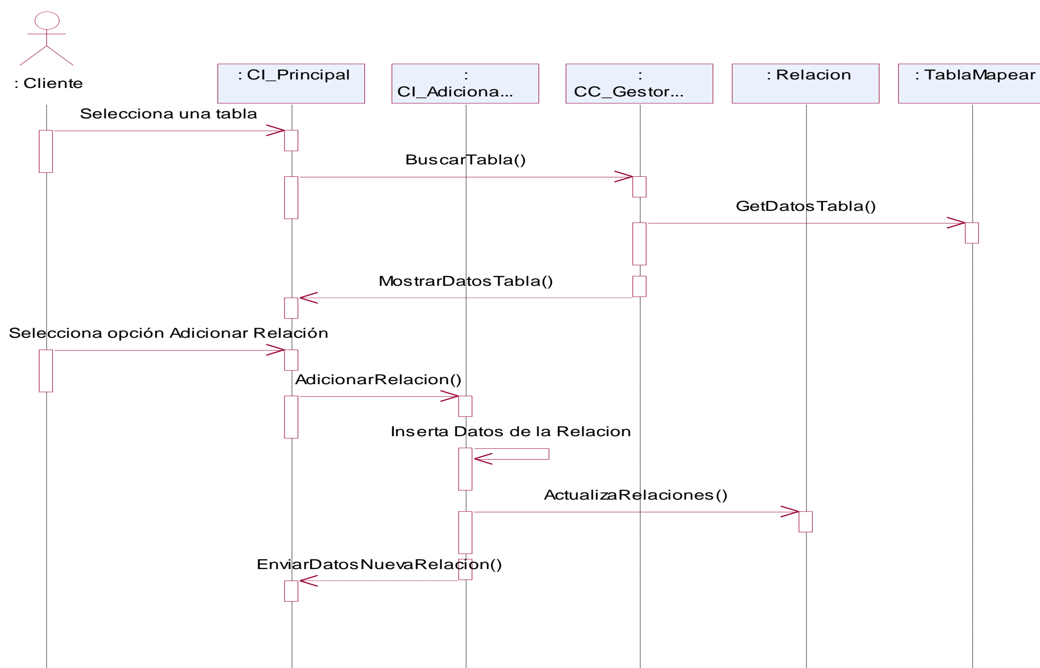


Figura 35 Diagrama de Secuencia: Caso de Uso Gestionar Relación. Escenario Adicionar Relación

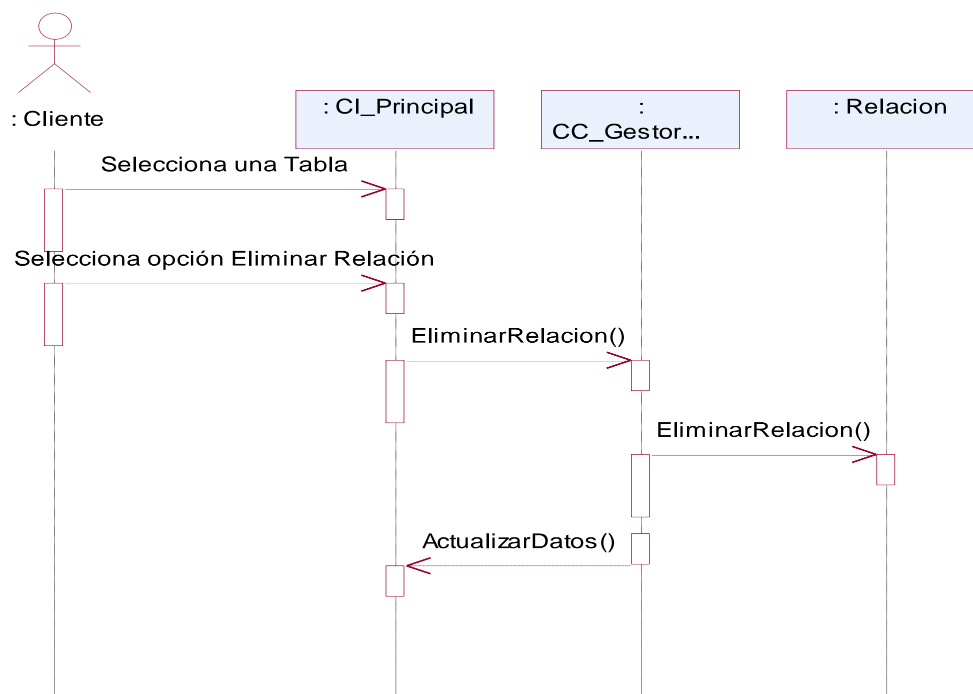


Figura 36 Diagrama de Secuencia: Caso de Uso Gestionar Relación. Escenario Eliminar Relación

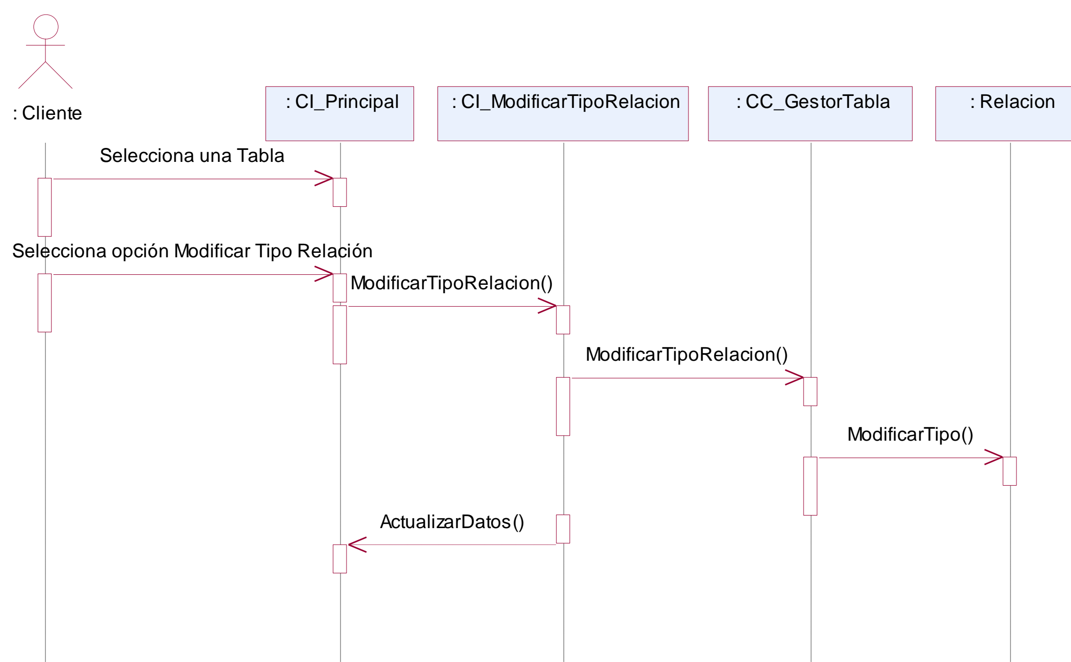


Figura 37 Diagrama de Secuencia: Caso de Uso Gestionar Relación. Escenario Modificar Tipo Relación

Anexo VIII: Diagramas de Colaboración y Secuencia para el Caso de Uso Gestionar Fichero de Esquemas.

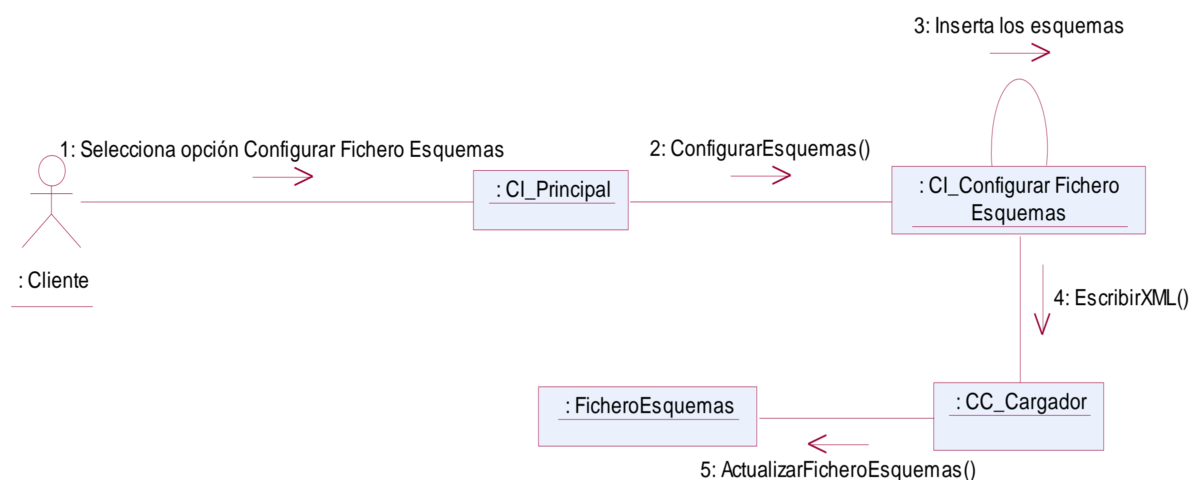


Figura 38 Diagrama de Colaboración: Caso de Uso Gestionar Fichero de Esquemas. Escenario Adicionar Esquema

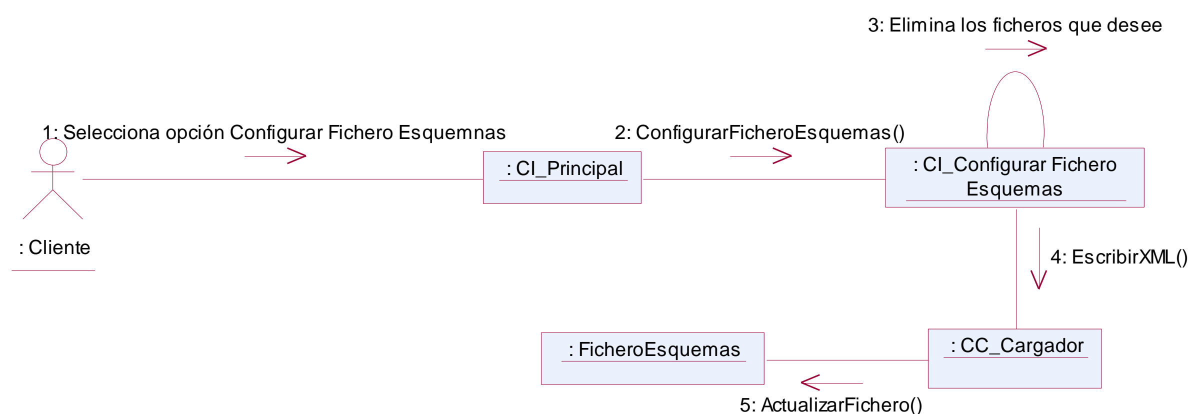


Figura 39 Diagrama de Colaboración: Caso de Uso Gestionar Fichero de Esquemas. Escenario Eliminar Esquema

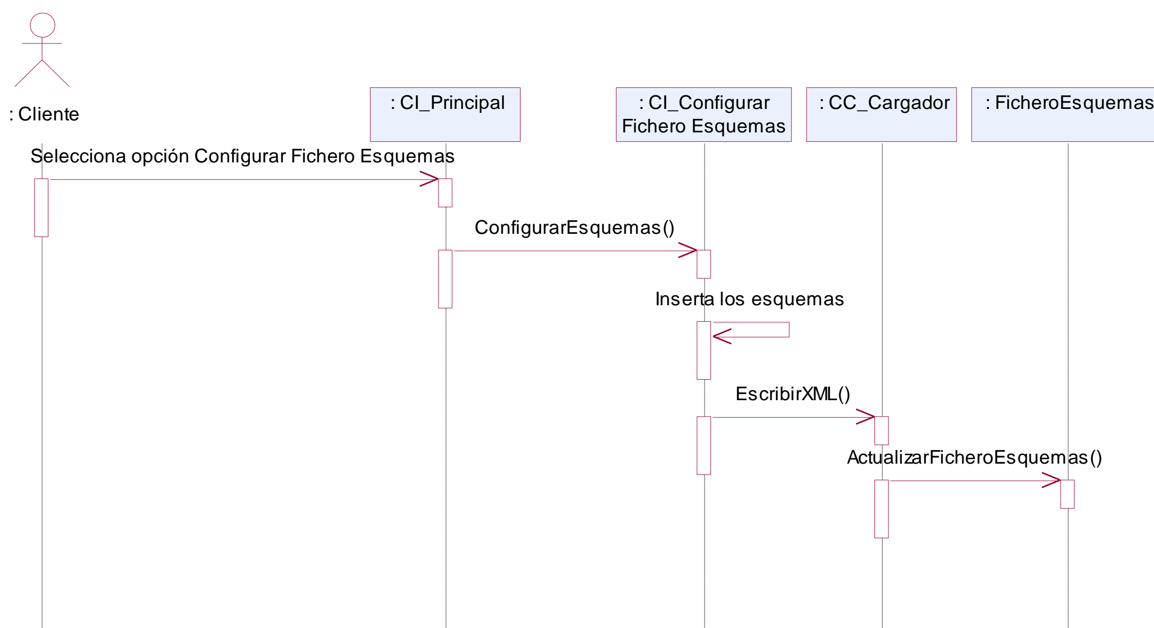


Figura 40 Diagrama de Secuencia: Caso de Uso Gestionar Fichero de Esquemas. Escenario Adicionar Esquema

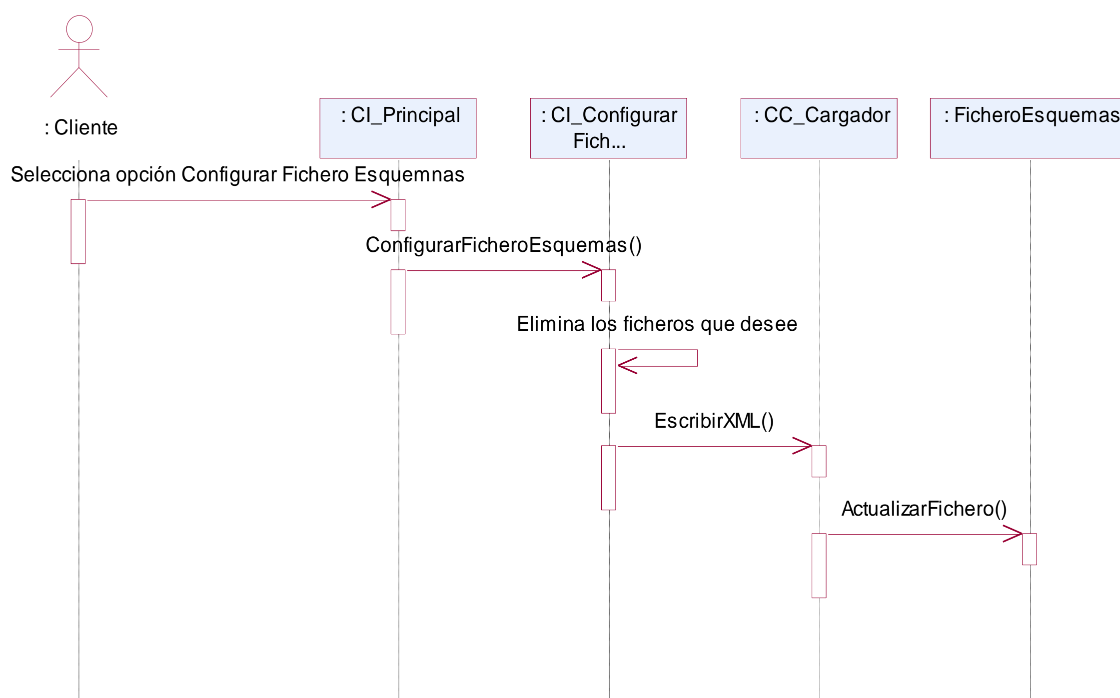


Figura 41 Diagrama de Secuencia: Caso de Uso Gestionar Fichero de Esquemas. Escenario Eliminar Esquema

Anexo IX: Ejemplo de código generado.

```
<?php
class DatActivofijotangible extends BaseDatActivofijotangible
 {
   public function setUp()
     {   parent::setUp();
      $this->hasOne('DatModuloaft', array('local'=>'idactivofijotangible','foreign'=>'idactivofijotangible'));
      $this->hasMany('DatMovinventarioaft', array('local'=>'nroinventario','foreign'=>'nroinventario'));
      $this->hasOne('DatModuloaft', array('local'=>'idactivofijotangible','foreign'=>'idaftmodulo'));

     }

   public function GetLLave()
     {
       $query = new Doctrine_Query ();
       $result = $query ->select('MAX('idactivofijotangible')')->from('DatActivofijotangible')->execute();
       $arr = $result->toArray();
       return (is_array($arr) ? $arr[0]['MAX'] + 1 : 1);

     }

   public function Buscar($idactivofijotangible)
     {
       $temp = $this->conn->getTable('DatActivofijotangible')->find($idactivofijotangible);
        return $temp->toArray();

     }

   public function GetTodos()
     {
       $query = new Doctrine_Query ();
       $result = $query->from('DatActivofijotangible')->execute ();
       return $result->toArray();

     }
```

Anexo X: Ejemplo de clase generada.

```
<?php
abstract class BaseNomMarcamodelo extends Doctrine_Record
 {
   public function setTableDefinition()
     {
       $this->setTableName('nom_marcamodelo');
       $this->hasColumn('idmarca', 'numeric', null, array('notnull' => true, 'primary' => true,
                                 'sequence' => 'mod_activofijotangible.sec_modelomarca_seq'));
       $this->hasColumn('descripcion', 'character varying', 60 , array('notnull' => false, 'primary' => false));
       $this->hasColumn('idpadre', 'numeric', null, array('notnull' => false, 'primary' => false));
       $this->hasColumn('lft', 'numeric', null, array('notnull' => false, 'primary' => false));
       $this->hasColumn('rgt', 'numeric', null, array('notnull' => false, 'primary' => false));
     }

   public function Setup()
     {
       parent::setUp();
     }
 }//fin clase
?>
```

Anexo XI: Umbrales para los parámetros de calidad evaluados con la métrica Número de Descendientes y rangos de valores para evaluar estos parámetros.

	Categoría	Criterio
Reutilización	Baja	<= Prom
	Media	Entre Prom y 2 * Prom
	Alta	> 2 * Prom

Tabla 48 Umbrales para el parámetro de calidad Reutilización

	Categoría	Criterio
Abstracción	Indefinida	> 5
	Afectada	Entre 2 y 5
	Definida	<= 2

Tabla 49 Umbrales para el parámetro de calidad Abstracción

	Categoría	Criterio
Cohesión	Baja	> 5
	Media	Entre 2 y 5
	Alta	<= 2

Tabla 50 Umbrales para el parámetro de calidad Cohesión

	Categoría	Criterio
Cantidad de Pruebas	Baja	> 5
	Media	Entre 2 y 5
	Alta	<= 2

Tabla 51 Umbrales para el parámetro de calidad Cantidad de Pruebas

Clase	Clase padre	Número de descendientes	Reutilización	Abstracción de la clase base	Cohesión	Cantidad Pruebas
GeneradorDomain	Generador	0	Baja	Definida	Alta	Baja
GeneradorBussines	Generador	0	Baja	Definida	Alta	Baja
GeneradorDomainG	Generador	0	Baja	Definida	Alta	Baja
Generador		3	Alta	Afectada	Media	Media

Tabla 52 Rango de valores de para la evaluación técnica de los atributos de calidad (Reutilización, Abstracción del diseño, Nivel de Cohesión y Cantidad de Pruebas)

Anexo XII: Árbol de herencia para las superclases Generador y Relacion.

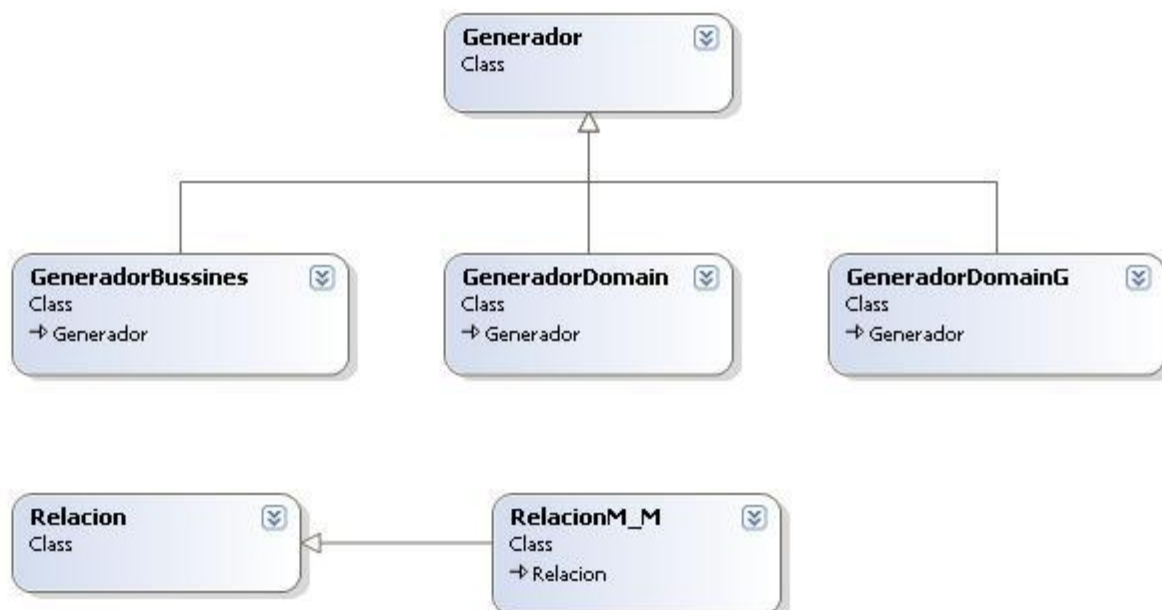


Figura 42 Árbol de herencia para las superclases Generador y Relacion

Anexo XIII: Resultados de la evaluación de la métrica TOC y su influencia en los atributos de calidad (Responsabilidad, Complejidad de Implementación y Reutilización)

No.	Clase	No. Atributos	No. Operaciones	Responsabilidad	Complejidad	Reutilización
1.	Proyecto	5	5	Baja	Baja	Alta
2.	Fichero	3	3	Baja	Baja	Alta
3.	BaseDatos	3	3	Baja	Baja	Alta
4.	Esquema	2	2	Baja	Baja	Alta
5.	Tabla	6	6	Media	Media	Media
6.	CampoTabla	6	6	Media	Media	Media
7.	Llave	2	2	Baja	Baja	Alta
8.	ConsultaDQL	6	6	Media	Media	Media
9.	Relación	6	6	Media	Media	Media
10.	RelaciónM_M	1	1	Baja	Baja	Alta
11.	GestorProyecto	1	8	Media	Media	Media
12.	GestorBaseDatos	1	3	Baja	Baja	Alta
13.	GestorEsquema	1	11	Media	Media	Media
14.	GestorTabla	1	16	Alta	Alta	Baja
15.	Cargador	3	4	Baja	Baja	Alta
16.	ManagerExcepcion	1	3	Baja	Baja	Alta
17.	Generador	1	6	Media	Media	Media
18.	GeneradorDomain	1	24	Alta	Alta	Baja
19.	GeneradorDomainG	1	3	Baja	Baja	Alta
20.	GeneradorBussines	1	5	Baja	Baja	Alta

Tabla 53 Resultados de la evaluación de la métrica TOC y su influencia en los atributos de calidad (Responsabilidad, Complejidad de Implementación y Reutilización)