GENERADOR DE CÓDIGO COMPOSITE ENTITY

MAURICIO CASTRO COLLAZOS

UNIVERSIDAD AUTÓNOMA DE OCCIDENTE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA INFORMACION
PROGRAMA INGENIERIA INFORMATICA
SANTIAGO DE CALI
2007

GENERADOR DE CÓDIGO COMPOSITE ENTITY

MAURICIO CASTRO COLLAZOS

Pasantía para optar por el título de
Ingeniero en Informática

Director
MARY ELIZABETH RAMIREZ CANO
Ingeniera de Sistemas

UNIVERSIDAD AUTÓNOMA DE OCCIDENTE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA INFORMACION
PROGRAMA INGENIERIA INFORMATICA
SANTIAGO DE CALI
2007

Nota de aceptación:

Aprobado por el comité de grado en cumplimiento de los requisitos exigidos por la Universidad Autónoma de Occidente para optar al titulo de Ingeniero en Informática.

Ing. MARY E. RAMIREZ CANO
Director

Santiago de Cali,02 de Febrero de 2007.

AGRADECIMIENTOS

Agradezco a todo el personal de Lucasian Labs Ltda. En especial al jefe del área de desarrollo John Cortes y al Chief Architect Mauricio Naranjo por brindarme la oportunidad de desarrollar a este proyecto.

Un agradecimiento muy especial a la profesora Mary Elizabeth Ramírez, la cual mostró una gran disposición a lo largo del proyecto y apoyo en la revisión de los documentos de entrega y los lineamientos propuestos para la opción de grado pasantia en la Universidad Autónoma de Occidente.

Un especial reconocimiento a mis padres que me apoyaron en los momentos difíciles de esta etapa tan importante de mi vida, me dieron las fuerzas y el amor necesario para afrontar todos los obstáculos que se me presentaron.

CONTENIDO

LISTADO DE FIGURAS

LISTADO DE ANEXOS

GLOSARIO

CE: siglas que corresponden al patrón de diseño Composite Entity (entidad compuesta). [1]

DAO: siglas que corresponden al patrón de Data Access Object(Objetos de acceso a datos). [1]

DTO: siglas que corresponden al patrón de diseño Data Transfer Object (objeto de transferencia de datos). [1]

EJB: siglas que corresponden a Enterprise Java Bean.

FABRICA DE CONEXIONES: es un componente Leaf que permite obtener las conexiones hacia una base de datos, como su nombre lo indica es un fabrica y permite acceder a cualquier repositorio central de datos soportado por Leaf.

FRAMEWORK: representación de una arquitectura de software

GEF: siglas que corresponden a Graphical Editing Framework o Framework de edición grafica

JELLY: tecnología de Scripting basada en XML que a partir de parámetros configurados suministrados el usuario sirve como plantilla de generación de código.

JSF: siglas que corresponden a Java Server Faces.

LEAF: siglas que corresponden al Framework de Lucasian Labs al cual pertenecerá este componente (Lucasian Enterprise Aplplication Framework).

ORQUESTACION: herramienta que permite unir varios servicios Web, donde cada uno de ellos realiza una tarea o actividad. Estas actividades tienen un flujo o W*orkflow* predefinido. Una vez que una actividad termina la ejecución continúa en la siguiente actividad definida.

RELACION SEMANTICA: es una relación entre dos tablas que no se hace por medio de las llaves primarias y foráneas, es decir se relacionan por cualquier campo con la tabla entidad.

---

[1] Core J2EE Patterns [en linea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septirmbre de 2006]. Disponible en internet: *http://java.sun.com/blueprints/corej2eepatterns/Patterns/*

SOA: siglas que corresponden a Services Oriented o Arquitectura Orientada a Servicios.

TABLA ENTIDAD: tabla de base de datos que es la base de una entidad de negocio compuesta. Sobre esta entidad de la base de datos se buscan las relaciones que sirven para construir el diagrama de Composite Entiy.

TABLA DOMINIO: tabla de base de datos que esta compuesta por dos campos (identificador y descripción), sobre la cual se hacen múltiples transacciones por ser una tabla de información.

TABLA HIJA: tabla de base de datos que tiene una relación maestro detalle con la entidad base. La relación entre estas dos tablas es que la llave primaria de la entidad equivale a un campo dentro de la tabla.

TABLA PADRE: tabla de base de datos que tienen una relación maestro detalle hacia la tabla entidad. La relación entre estas tablas es que la llave primaria equivale a un campo de la tabla entidad.

VO: siglas que corresponden al patrón de diseño Value Object (valores de objeto). [1]

WIZARD: formularios navegables de eclipse donde el usuario configura sus acciones**.**

RESUMEN

En el mundo hay una gran tendencia al desarrollo de aplicaciones empresariales por medio de la tecnología JEE orientada a servicios. Lucasian Labs es una empresa Colombiana la cual gracias a su experiencia en más de 30 proyectos en el sector bancario bajo esta tecnología ha logrado proponer una arquitectura bastante robusta, basada en estándares y apoyada sobre Frameworks de productividad.

Actualmente esta lanzando al mercado un nuevo release de Leaf (Lucasian Enterprise Application Framework) el cual tiene un nuevo componente de generación de código basado en el patrón de diseño Composite Entity, tema de este proyecto.

El siguiente documento mostrara las bases teóricas y tecnológicas que forman parte de este proyecto de grado, por ser un proyecto sin precedentes actuales, tiene un gran componente investigativo, el cual finalmente se convierte en el marco teórico de este documento.

Los patrones de diseño son la base conceptual de este proyecto, por esta razón se hace bastante referencia sobre ellos en el documento, principalmente para los que corresponden a la capa de negocio propuesta por Sun MicroSystem y los que son propietarios de Lucasian Labs. El proyecto fundamenta su ejecución en los siguientes patrones de diseño: Composite Entity, Data Transfer Object, Domain Object, Data Access Object y Value Object.

Una parte de la base investigativa del proyecto se centra en el IDE eclipse y todo su ambiente de ejecución, ya que provee plugins fundamentales para la construcción del producto final del proyecto. Las tecnologías mas utilizadas son SWT, GEF y draw2d con las cuales se crean las herramientas con las que interactúa el usuario final.

INTRODUCCIÓN

En la actualidad hay una gran demanda por soluciones tecnológicas que aceleren los procesos empresariales, hay una necesidad latente en los distintos sectores comerciales de migrarse o reforzar sus sistemas por medio de JEE.

Lucasian Labs es una compañía que esta comprometida con esta necesidad y por eso, ha buscado identificar las problemáticas comunes de los procesos críticos de desarrollo de sistemas informáticos. Gracias a su experiencia y reconocimiento en el mercado, busca por medio de la nueva versión de su producto Leaf3i, aumentar la productividad y la velocidad de la implementación de sistemas informáticos empresariales. Leaf (Lucasian Enterprise Application Framework), herramienta que en la actualidad lleva alrededor de dos años en el mercado en su versión 2.0, busca en su nueva versión, incluir componentes que lo hagan más robusto.

Además de su experiencia en el sector empresarial, Lucasian Labs se apoya en los estándares de desarrollo de la tecnología JEE y los patrones de diseño, dado que estos hacen más robustos los productos que se ofrecen a la industria dedicada al desarrollo de software.

La solución generada en este proyecto, apoyará al Desarrollador en la fase de implementación al momento de ejecutar operaciones desde el modelo orientado a objetos a la base de datos relacional, en resumen, como producto final del desarrollo de este proyecto, se entregará un generador de código basado en el patrón de diseño Composite Entity. Esta implementación permitirá a los usuarios de la tecnología JEE hacer un mapeo de las entidades de negocio compuestas, de manera que las operaciones masivas a nivel de objetos sean sencillas.

1. ANTECENDETES

A nivel nacional, no es muy común encontrar generadores de código, ya que pocas empresas han tenido la iniciativa que tuvo en su momento Lucasian Labs de implementar una herramienta de generación masiva de clases.

Actualmente, ya se cuenta con una versión del Leaf en producción, la cual se queda un poco corta en cuanto a los servicios ofrecidos por los productos de las grandes casas de desarrollo de software como Oracle, Sun. Por esta razón se tomó la iniciativa de crear nuevos generadores de código, entre los cuales se encuentra el generador de Composite Entity, los cuales le permitan a la empresa tener una oferta más fuerte ante el mercado.

Inicialmente el generador de código para el patrón Composite Entity era una idea que formaba parte del grupo de investigación de Lucasian Labs, al ver las necesidades del mercado, se decidió integrarlo en la versión 3 de su producto Leaf3i. La implementación de este componente de generación fue asignada al empleado Mauricio Castro Collazos, proponente de este proyecto.

Existen generadores de código para documentación dentro de las clases, generadores de EJB (Enterprise Java Beans) y DAO (Data Transfer Object), pero ninguno que haga un mapeo objeto relacional basado en el patrón Composite Entity.

A nivel mundial, la mayoría de productos que se enfocan en la generación de código, se concentran en un solo patrón de diseño o en una sola tecnología. Las grandes casas de desarrollo ofrecen algunas opciones, las cuales están totalmente atadas a su tecnología y se alejan mucho de ser eficientes en un desarrollo de un proyecto. Algunas veces cuesta mucho tiempo integrar las soluciones que ofrecen los distintos proveedores, debido a los conceptos particulares sobre los que son construidos.

2. OBJETIVOS

2.1. OBJETIVO GENERAL

Desarrollar un componente de generación de código basado en el patrón Composite Entity, para el producto Leaf3i de Lucasian Labs.

2.2. OBJETIVOS ESPECIFICOS

- Definir requerimientos funcionales del generador de código de los Composite Entity.
- Realizar modelo de casos de uso y diagrama de componentes para el generador de código Composite Entity.
- Desarrollar el componente de generación de código para el patrón Composite Entity.
- Desarrollar el componente de generación de código para el patrón DTO (Data Transfer Object).
- Desarrollar el componente de generación de código para el patrón Domain Object
- Desarrollar componente gráfico con el que el usuario interactuará para la generación de código.
- Integrar componente gráfico y componentes de generación.
- Realizar pruebas de funcionamiento del generador de código Composite Entity.

3. JUSTIFICACION

El proyecto presentado en esta propuesta, es una solución al problema que se vive a diario en las empresas en relación a los altos costos de tiempo e inversión de dinero en la fase de implementación del desarrollo de proyectos de software.

Es común encontrar implementaciones de código que se repiten en los proyectos, más cuando se hacen sobre la tecnología JEE. Lucasian Labs ha identificado algunas actividades que demandan un alto consumo de tiempo de desarrollo, entre ellas se encuentra que siempre se atacan las entidades compuestas por separado, provocando que las operaciones masivas básicas como inserciones, búsquedas, actualizaciones y eliminaciones, en mas de una tabla sean complejas y demoradas de implementar.

El componente de generación de código del patrón Composite Entity, se presenta como una solución teórica y tecnológica, la cual reducirá en gran proporción el tiempo empleado por los desarrolladores de software en la implementación de código para la interacción con las entidades compuestas de negocio.

Cabe anotar que en el mercado no hay ninguna herramienta generadora de código para este patrón, ya que es compleja la abstracción de la problemática y pocas empresas se han enfocado en el desarrollo de generadores de código.

Entre los aspectos tangibles que justifican esta solución se resaltan los siguientes aspectos:

- Reducción en tiempo invertido por los desarrolladores y arquitectos de software en el desarrollo de sus proyectos empresariales.
- Reducción de costos de financiamiento de un proyecto empresarial, ya que en la tarea que antes se concentraban 2 o 3 desarrolladores con el uso de la herramienta solo se utilizaría 1.
- Lucasian Labs podrá tener un nuevo componente de generación en su producto, el cual lo hará más llamativo en el mercado.
- El documento generado servirá como apoyo a los estudiantes de la universidad, ya que les dará una visión actual sobres los estándares de desarrollo y de los patrones de diseño que propone JEE.

4.0 METODOLOGIA

La metodología a seguir se listara conforme a las tareas básicas programadas dentro de la empresa para alcanzar los objetivos planteados para el proyecto, es decir, para cada uno de los pasos que se describirán habrá un tratamiento especial por lo específicas que son las tareas. La propuesta es tener un proceso iterativo base de trabajo en el cual están involucrados los siguientes pasos:

- Paso1: Investigación del proceso
- Paso2: Abstracción del concepto y empalme con las directrices de la empresa.
- Paso3: Implementación de solución tecnológica.
- Paso4: Integración (En algunos casos)
- Paso5: Pruebas de la solución
- Paso6: Popularización y discusión de la solución.
- Paso7: Ajustes a la solución.(En algunos Casos)
- Paso8: Entrega de la solución. (En algunos Casos)

Por el tamaño de la aplicación y el plazo propuesto por la empresa se decidió no incluir diagramas de clases y diagramas de secuencia en el proyecto, es decir algunos de los componentes ya tienen unos estándares de desarrollo propuestos y no es de gran relevancia realizar un diseño sobre ellos, ya que retardarían el proyecto.

Cabe anotar que habrá tres actores fijos implicados en este proceso Luis Alfaro Valderrama (Framework Director) y Mauricio Castro Collazos (Proponente) y Mary Ramírez (Directora del proyecto).

A continuación se describirá cada una de los pasos definidos por la empresa, donde se seguirá la metodología propuesta anteriormente:

- ✓ Pruebas de concepto de CE (Composite Entity)

  Esta tarea involucra al paso número 1 y 2 de la metodología, Investigación del proceso y empalme con las directrices de la empresa respectivamente, es decir, por medio de esta prueba de concepto se definirá el alcance final del generador de código. Se describen a continuación los pasos generales que se deben seguir para lograr las pruebas de concepto:

- Leer documentación actual sobre el patrón Composite Entity (Core pattern Desing for J2EE). (Proponente)
- Leer documentación actual sobre el patrón DTO (patrón de diseño propuesto por Lucasian Labs). (Proponente)

- Comprender y documentar la idea del nuevo patrón de diseño propuesto por Lucasian Labs Domain Object. (Proponente)
- Revisión del enfoque que Lucasian Labs quiere darle al patrón Composite Entity para incluirlo en su producto Leaf3i. (Proponente)
- Crear un modelo de base de datos que permita tener un ambiente de pruebas propicio, el cual será utilizado mas adelante como modelo base en las generaciones de código y en las pruebas interacción con las entidades compuestas de la base de datos(Proponente)
- Revisión por parte del director de la empresa asignado. (Framework Director)
- Generación de un documento, que permita tener un balance claro sobre la prueba de concepto para el patrón Composite Entity como componente del producto Leaf3i. Este documento, permitirá definir el alcance del generador de código. (Proponente)

✓ Pruebas de generación

Esta tarea involucra el paso número 3 y 4 de la metodología propuesta, implementación de solución tecnológica y pruebas de la solución, este desarrollo se hará basado en la especificación entregada en la fase anterior.

- Crear un ejemplo de código basado en el modelo de base de datos creado anteriormente, que permitirá dar una idea clara de la funcionalidad que ofrecida por una clase generada masivamente en un futuro. (Proponente).
- Revisión de servicios expuestos en la clase de código implementada, correcciones y sugerencias para hacer más robustos los servicios ofrecidos. (Framework Director).
- Hacer pruebas sobre el código implementado para ver la eficiencia que puede ofrecer la implementación masiva que se quiere lograr con este componente. (Proponente).
- Incluir conclusiones en el documento de pruebas de concepto de los CE (Proponente).

✓ Definición de servicios

Esta tarea involucra el paso número 5 de la metodología propuesta, popularización y discusión de la solución. Se definirán los servicios ofrecidos por las clases generadas, basados en el desarrollo de prueba hecho en la tarea anterior.

- Presentar en limpio una propuesta los miembros de la empresa Lucasian Labs de los servicios a exponer en las futuras clases generadas. (Proponente).

- Presentar documento al interior de la empresa para tener un acta de la investigación realizada.

Nota: Se acaba la primera iteración del proyecto y se entrega como producto final un documento que define el alcance en cuanto a servicios de las clases generadas

✓ Extracción de meta data

Esta tarea involucra los pasos numero 1,2,3,4,5,6,7 de la metodología y seria la segunda iteración del proyecto y entregaría como producto final componente que permita extraer la metadata necesaria para la construcción de las clases.

- Revisión del estándar actual, por medio del cual se extrae la metadata utilizada en los componentes de generación anteriores. (Framework Director -Proponente).
- Propuesta de extracción de metadata necesaria para los Composite Entity basado en el estándar anteriormente propuesto por la empresa. (Proponente).
- Implementación de código que permita extraer de la base de datos la data necesaria para llevar a cabo la generación futura de clases.
- Pruebas sobre las clases implementadas anteriormente.
- Revisión que permita tener clara las limitaciones o bondades presentadas por la base de datos (Oracle 10g) para los propósitos de generación. (Framework Director).

✓ Generación de plantillas Jelly

Esta tarea involucra los pasos numero 1,2,3,5 de la metodología, seria la tercera iteración del proyecto y entregaría como producto final componente que permitirá la generación de archivos físicos ,Java a partir de la metadata extraída en la tarea anterior.

- Revisión del estándar actual por medio del cual se crean las platillas de generación, teniendo como herramienta el lenguaje de scrpting Jelly. (Framework Director -Proponente).
- Leer documentación actual de Jelly para tener claridad sobre su utilización en las futuras plantillas de generación para el patrón Composite Entity, DTO (Data Transfer Object) y Domain Object. (Proponente)
- Creación de plantillas de generación para los patrones de diseño Composite Entity, Data Transfer Object y Domain Object.
- Implementación de clases intermedias Java-Jelly, que permitirán hacer el llamado a las plantillas, cuando se procede a la generación de código masiva. (Proponente)

- Realizar pruebas que permitan verificar la efectividad de los servicios de las clases generadas con base en las plantillas Jelly. (Proponente).
- Revisión de código generado por medio de las plantillas Jelly. (Framework Director)

✓ Pruebas de concepto de entorno gráfico

Esta tarea involucra los pasos número 1 y 2 de la metodología, Investigación del proceso y empalme con las directrices de la empresa respectivamente, es decir por medio de esta prueba de concepto se definirá el entorno grafico de la entidad de negocio a construir. Inicia la cuarta iteración del proyecto.

- Leer documentación y revisar ejemplos disponibles en la red de la API's de java GEF (Graphical Editing Framework) y Draw2d. (Proponente).
- Elaborar propuesta inicial de los alcances que permite tener el ambiente grafico GEF y como integrarlo al producto Leaf3i (plug-in Eclipse). (Framework Director -Proponente).
- Entregar documento que le permita visualizar a Lucasian Labs los servicios entregados por la API's GEF y Draw2d. (Proponente).

✓ Wizard grafico de CE

Esta tarea involucra los pasos número 3,5,5,6,7, de la metodología, los cuales permiten tener la cuarta iteración completa y entregar el componente grafico de los Composite Entity.

- Leer documentación de desarrollo de plug-ins, propuesto por eclipse (Proponente).
- Revisión de los estándares de codificación dentro del plug-in Leaf3i (Framework Director -Proponente).
- Implementación de clases que permitan tener un ambiente gráfico integrado dentro de Leaf3i, basándose en GEF y Draw2d.
- Socialización del Wizard de generación de diagramas de Composite Entity dentro de la empresa Lucasian Labs del ambiente grafico creado (sugerencias, correcciones).
- Corrección de errores y revisión final de ambiente grafico del generador de Composite Entity.

✓ Integración de ambiente gráfico y generador

Esta tarea involucra al paso número 1,2,3,4,5,6,7, de la metodología, los cuales permitirán tener la quinta Iteración del proyecto y entrega como producto final una versión beta.

- Revisión de los estándares de integración de entorno gráfico – generador de código. (Framework Director -Proponente).
- Implementación de código basado en los estándares propuestos, para integración del entorno gráfico y el generador. (Proponente).
- Pruebas de generación de código desde el ambiente gráfico.
- Popularización de la integración final del ambiente grafico y el generador código.

✓ Pruebas de aceptación

Esta última tarea equivale al punto número 8 de la metodología, entrega de la solución y es la última iteración del proyecto, entrega como producto final del proyecto, el generador de código de Composite Entity, completo.

- Pruebas de aceptación por los miembros de la empresa, simulando una situación empresarial real.
- Revisión y corrección de errores encontrados en las pruebas realizadas anteriormente. (Proponente).
- Entrega final del generador de código de los Composite Entity. (Proponente).

6. ALCANCE DE LA APLICACION (SRS)

En este capitulo se presentara el alcance funcional del proyecto delimitado por medio de los requerimientos, además se mostrara un diseño general del proyecto que incluye un diagrama de casos de uso y diagrama de componentes de la aplicación.

La labor de diseño de software se limito en entregar un bosquejo general de aplicación, a partir del cual el desarrollador pudiera guiarse. Por la complejidad, limitaciones de tiempo y por los diseños que propone las diferentes tecnologías se decidió no incluir diagramas de clases y diagramas de secuencia.

6.1 DEFINICION DE REQUERIMIENTOS

Requerimientos Componente de Generación Composite Entity

RQ	Nombre	Descripción	Nota	Tipo
RQ 1	Extracción de metadata	El sistema debe basado en la tabla escogida por el usuario como tabla entidad extraer la metadata que será la base del diagrama generado	La extracción de metadata se hará sobre base de datos Oracle por medio de la utilización de sentencias SQL sobre las tablas del sistema	F
RQ 2	Generación de diagrama a partir de una entidad	El sistema debe generar un diagrama basado en la extracción de metadata a partir de una entidad escogida por el usuario	El diagrama generado debe estar compuesto por las tablas de base de datos que este directamente relacionado con la entidad. Es decir con las tablas padres (Llaves foráneas de la entidad) y por las tablas hijas (llaves primarias de la entidad que son una llave primaria en la tabla hija).	F
RQ 3	Generación de clases a partir de un diagrama	El sistema debe leer el diagrama realizado por el usuario y a partir de este realizar la generación de: Clase CE, DTO y objetos de las tablas dominio	Las Clases generadas deben permitir al usuario realizar búsquedas, actualizaciones, inserciones y borrado masivo de la entidad de negocio compuesta	F

RQ 4	Modificación de Diagrama generado	El usuario debe poder modificar el diagrama generado por el sistema		F
RQ 5	Generación de un Diagrama vacío	El sistema debe proveerle al usuario la opción de hacer el diagrama de su entidad de negocio desde el comienzo	Es necesario que el usuario del sistema pueda crear su propia entidad de negocio personalizada.	F
RQ 6	Inserción una tabla Entidad	El sistema debe proveerle al usuario la opción de poder ingresar una tabla entidad, a partir de la cual se construirá el resto de la entidad de negocio	Esta tabla entidad debe tener relaciones uno a muchos con tablas de base de datos hijas y/o padres,	F
RQ 7	Inserción una tabla Hija	El sistema debe proveerle al usuario la opción de poder ingresar una tabla hija la cual este directamente relacionada con la llave primaria de la entidad	Esta tabla debe tener una llave foránea que a su vez es la llave primaria de la entidad(esta llave foránea puede formar parte de su llave primaria)	F
RQ 8	Inserción de una tabla padre	El sistema debe proveerle al usuario la opción de poder ingresar una tabla padre la cual estará directamente relacionada con la tabla entidad por medio de una llave foránea	Esta tabla debe tener una llave primaria la cual estará dentro de la entidad por medio de una llave foránea (esta llave foránea puede formar parte de su llave primaria)	F
RQ 9	Inserción de una tabla dominio	El sistema debe proveerle al usuario la opción de poder ingresar una tabla dominio(Tabla tipo de base de datos)	Esta tabla debe tener dos campos sobre los cuales se hará el mapeo objeto sobre la data de base de datos	F
RQ 10	trazo de relaciones entre dos tablas	El sistema debe permitir trazar relaciones entre las tablas del diagrama para que se mas claro		F
RQ 11	Borrado de tablas	El sistema le debe permitir al usuario borrar una tabla del diagrama	El sistema debe validar que la tabla que se borre no sea la tabla entidad	F

RQ 13	Wizard de Diagrama de Composite Entity	El sistema debe proveerle un Wizard de eclipse en donde el usuario pueda escoger el tipo de diagrama a crear , ya sea a partir de una entidad o vacío	Dependiendo del tipo de diagrama a realizar por el usuario, el sistema mostrara las paginas del Wizard que sean necesarias	F
RQ 14	Diagrama de representación de la entidad de negocio	El sistema debe permitir al usuario ver dentro de las tablas del diagrama una figura de representación de las tablas	Esta tabla del diagrama debe contener atributos de la tabla y métodos que contendrá la clase	F
RQ 15	Wizard de Generación de Composite Entity	El sistema de proveerle al usuario un Wizard que permita hacer la generación de código basado en el diagrama creado por el usuario	Este Wizard debe incluir una interfaz de usuario que además de poder hacer la generación de las clases CE ,DTO y dominio también de poder exponer métodos de la clase como Web Services	F
RQ 16	Creación de plantilla de generación CE	El sistema debe contener una plantilla Jelly de generación de CE sobre la cual se construirá la clase a generar.	Las plantillas están construidas basadas en el lenguaje de scripting Jelly	F
RQ 17	Creación de plantilla de generación DTO	El sistema debe contener una plantilla Jelly de generación de DTO sobre la cual se construirá la clase de almacenamiento a generar.	Las plantillas están construidas basadas en el lenguaje de scripting Jelly	F
RQ 18	Creación de plantilla de generación Domain Object	El sistema debe contener una plantilla Jelly de generación de Domain Object sobre la cual se construirá la clase de representación de los datos de la tabla	Las plantillas están construidas basadas en el lenguaje de scripting Jelly	F
RQ 19	Generación de código compatible para exposición de	Los servicios entregados por la clase generadas deben ser compatibles con los estándares de exposición de Servicios	El componente de generación de Servicios Web ya se encuentra desarrollado dentro del plugin	F

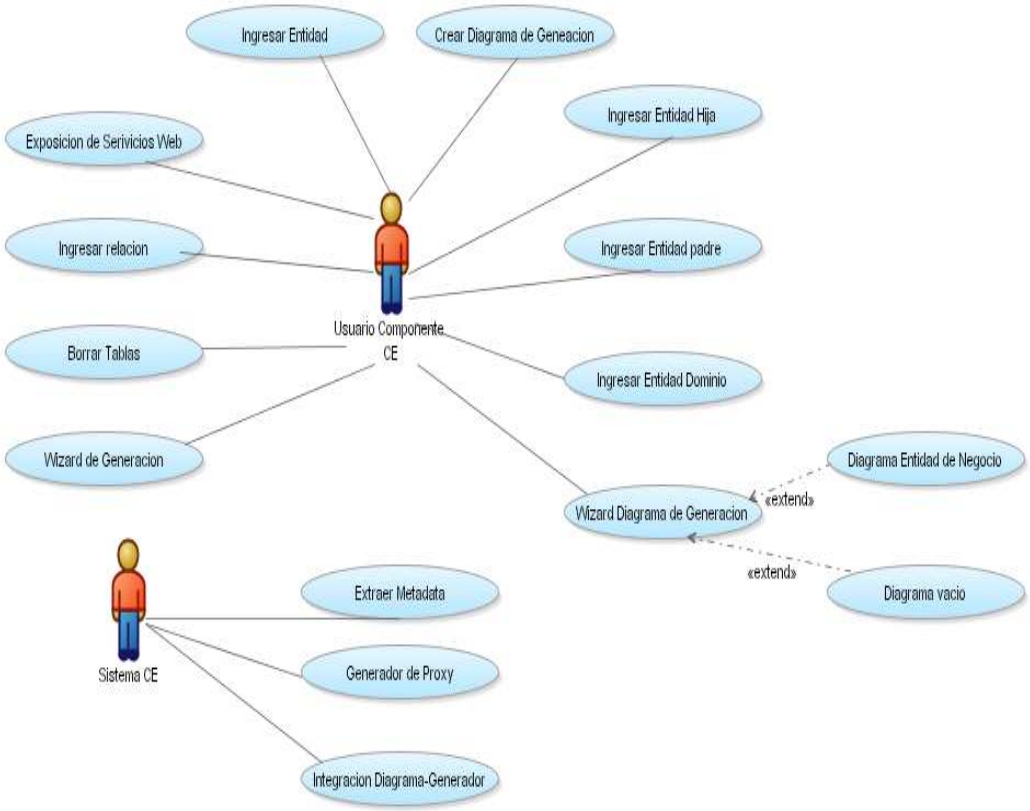
	servicios Web	Web propuesta por Lucasian Labs		
RQ 20	Compatibilidad con las clases DAO generadas	El componente de generación de Composite Entity debe ser compatible con los DAO's (Data Access Object) generados por Lucasian Labs ya que estos serán reutilizados.	El componente de generación de DAO(Data Access Object) ya se encuentra desarrollado dentro del plugin	F
RQ 21	Compatibilidad con la configuración del proyecto	El sistema debe ser compatible con la configuración previamente hecha en el Wizard de configuración del plugin. Es decir deber leer el archivo de propiedades generado por la configuración personalizada del usuario de la herramienta.	El Wizard de configuración de propiedades ya se encuentra implementado dentro del plugin del que formara parte este componente de generación	NF
RQ 22	Compatibilidad la estructura del plugin	El sistema debe estar construido basado en los estándares de desarrollo internos del plugin	Dentro de Lucasian Labs hay estándares internos de desarrollo dentro de los que se encuentran: Estructura de proyectos, Mejores prácticas de programación.	NF
RQ 23	Construcción de componente de generación	El sistema de contener un componente de generación que se invocara dentro del componente de integración	Este componente de generación deberá estar repartido en dos paquetes que forman parte de la estructura del plugin	F
RQ 24	Construcción de componente de integración entre ambiente grafico y generador	El sistema debe contener un integrador el cual haga la lectura del diagrama CE y invoque al componente de generación	Este componente servirá de intermediario entre el diagrama creado por el usuario y la estructura interna de generación de código	F

RQ 25	Construcción del componente de presentación	El sistema debe contener un componente que permita basándose en los parámetros provistos por el usuario construir el diagrama de generación.	F
RQ 26	Compatibilidad con las API's GEF y Draw2d	El ambiente de utilización del plugin debe contener estas dos API's para garantizar el funcionamiento del componente grafico de generación	NF
RQ 27	Operating System	Windows 2000 or Windows XP	T
RQ 28	IDE De desarrollo	Eclipse 3.1.0	T
RQ 29	Maquina virtual Java	Sun Microsystems JRE 1.4	T
RQ 30	Requerimientos de Maquina	Intel® Pentium® III 1.0 512 MB RAM minimum; 1 GB RAM recommended, A mouse or an alternative pointing device	T

6.2 DIAGRAMA DE CASOS DE USO

Los casos de uso presentados en la figura 1, describen la interacción que tendrá el usuario con el sistema. Más adelante se hace una descripción breve de cada uno de los casos de uso, en donde se especifica su funcionalidad, prioridad y en que momento es ejecutado en el sistema.

Figura 1. Diagrama de Casos de Uso



CU Crear Diagrama de Generación

En este caso de uso contempla la opción de que el usuario pueda ingresar los parámetros con los que será construido el diagrama de Composite Entity en Eclipse, es decir aquí se ingresa el nombre del archivo (Diagrama), el proyecto donde va estar ubicado y además con que opción desea generar diagrama. Al usuario se le presentaran dos opciones de generación de diagrama que equivalen dos rutas diferentes en el diagrama de casos de uso, que son *Diagrama Entidad de negocio y Diagrama vacío.*

CU Wizard Diagrama de Generación

Este caso de uso se ejecuta cuando se invoca la acción de Eclipse Diagrama de Generación Composite Entity y permite abrir el Wizard donde se configuran los parámetros de entrada de los casos de uso *Diagrama Entidad de negocio* o *Diagrama Vacío*.

CU Diagrama Entidad de negocio

Este caso de uso contempla las operaciones que debe ejecutar el sistema si el usuario selecciona la opción crear un diagrama a partir de una entidad de la base de datos. Las operaciones que ejecuta este caso de uso es:

- Mostrar al usuario las tablas ha escoger como tabla base de la entidad de negocio. En esta actividad se muestran todas las tablas que hay en el esquema de base de datos escogido por el usuario en la configuración de propiedades de Leaf.

- Basado en la tabla escogida por el usuario, el sistema consume el *CU Extracción de Metadata.*

- A partir de la extracción de metadata el sistema construye un diagrama que permite al usuario saber que tablas tiene la entidad compuesta de negocio.

CU Extracción de metadata

Este caso de uso es ejecutado por el sistema y es llamado si previamente se ha invocado el caso de uso *Diagrama Entidad de negocio.* Consiste en que internamente el sistema debe hacer una lectura de metadata, es decir a partir de la entidad escogida por el usuario se hará una búsqueda directamente a la base de datos y se extraerá todas las relaciones de llaves primarias y llaves foráneas de las tablas involucradas en la entidad de negocio compuesta. Basado en la lectura de relaciones de base de datos se hace una extracción de metadata o de composición relacional de las tablas que componen la entidad compuesta, el orden en que se extrae la información es similar al de recorrer una estructura de árbol, es decir se inicia por las entidades padres, basado en esta información se extrae la información de la tabla base y finalmente cada una de las entidades hijas.

CU Diagrama Vació

Este caso de uso tiene una relación de extends con el caso de uso *Crear Diagrama de Generación* y su función es crear una entidad compuesta manualmente, es decir, se le da la opción de que a partir del conocimiento del sistema personalice su entidad de negocio desde su origen.

CU Ingresar Entidad

Este caso de uso permite ingresar una entidad base al diagrama. Como regla del sistema no puede haber dos tablas entidad. Las tabla entidad del diagrama es la más importante, ya que a partir de ella se hace la búsqueda de relaciones de base de datos y la generación de código.

CU Ingresar Entidad Padre

Este caso de uso permite ingresar una nueva entidad padre al diagrama, esta tabla deberá estar relacionada directamente con la tabla entidad, es decir, debe cumplir con el requisito de que la llave primaria debe estar referenciada por un campo de la tabla entidad. Adicionalmente se podrán ingresar entidades por medio de relaciones semánticas, es decir relaciones lógicas a la tabla entidad.

CU Ingresar Entidad Hija

Este caso de uso permite ingresar una nueva entidad hija al diagrama que tenga relación padre – hijo (Llave primaria entidad – llave foránea en la tabla) con la entidad base. El sistema debe soportar la múltiple referenciacion hacia la llave primaria de la entidad base. Adicionalmente se podrán ingresar entidades por medio de relaciones semánticas, es decir relaciones lógicas a la tabla entidad.

CU Ingresar Entidad Dominio

Este caso de uso permite ingresar una tabla dominio a la entidad compuesta, sobre la cual se generara un archivo Java que contendrá la información física de la tabla tipo que referencia, esta debe estar relacionada directamente con la entidad.

CU Ingresar Relación

Este caso de uso permite ingresar una nueva relación entre las tablas del diagrama y solo se ejecuta si hay dos tablas seleccionadas. Lo que permite este caso de uso es ingresar una nueva relación, la cual construye una relación semántica entre la tabla entidad y las tablas que haya dentro del diagrama.

CU Borrar Tabla

Este caso de uso permite eliminar una tabla de la entidad compuesta de negocio representada por el diagrama, validando que esta no sea una tabla base que tenga relaciones dentro del diagrama.

/CU Wizard de generación

Este caso de uso permite invocar la acción que hace el llamado al Wizard donde se configuraran los parámetros de entrada del caso de uso *Generación de Código*. Es ejecutado cuando el usuario pulsa la opción Generación de Código de Composite Entity creada dentro del plugin de Eclipse.

CU Generador de Código

Este caso de uso contempla la opción de generar código a partir del diagrama de Composite Entity que haya configurado el usuario previamente, se valida que no se haga generación si el diagrama se encuentra vacio. Este caso de uso hace el llamado al componente de generación que es invisible para el usuario, pero que es finalmente el que crea los archivos físicos basado en las plantillas de generación.

Como archivos a generar están las clases CE (Clase Composite Entity), DTO (Objeto de almacenamiento), Domain Object (opcional en caso de que haya tablas tipos ingresadas en el diagrama).

CU Integración Diagrama-Generador

Este caso de uso es ejecutado por el sistema y su tarea especifica es integrar la información del diagrama configurado por el usuario y el generador de código que tiene inmerso el plugin para los Composite Entity. Una vez leído el diagrama se hará el llamado al generador, que será el que finalmente creará físicamente las clases basado en las plantillas Jelly.

CU Exposición de servicios Web

Este caso de uso permite al usuario marcar que métodos de las clases generadas se desean incluir en el archivo de despliegue de servicios Web, es decir se ejecuta cuando el usuario a través del Wizard de generación selecciona que servicios de la clase desea exponer como Web Services.

CU Generador de Proxy

Este caso de uso es ejecutado por el sistema y su tarea principal es que basado en la información recolectada en el caso de uso *Exposición de servicios Web,* permita generar una clase la cual sirva de intermediario entre la tecnología de exposición de servicios Web, en nuestro caso Axis[2] (Herramienta de Apache) y la capa de negocio del sistema. Este Proxy sirve para capturar las excepciones que ocurran en el servidor de aplicaciones, con el fin de tener control sobre los errores que no puedan ser detectados dentro de la aplicación.

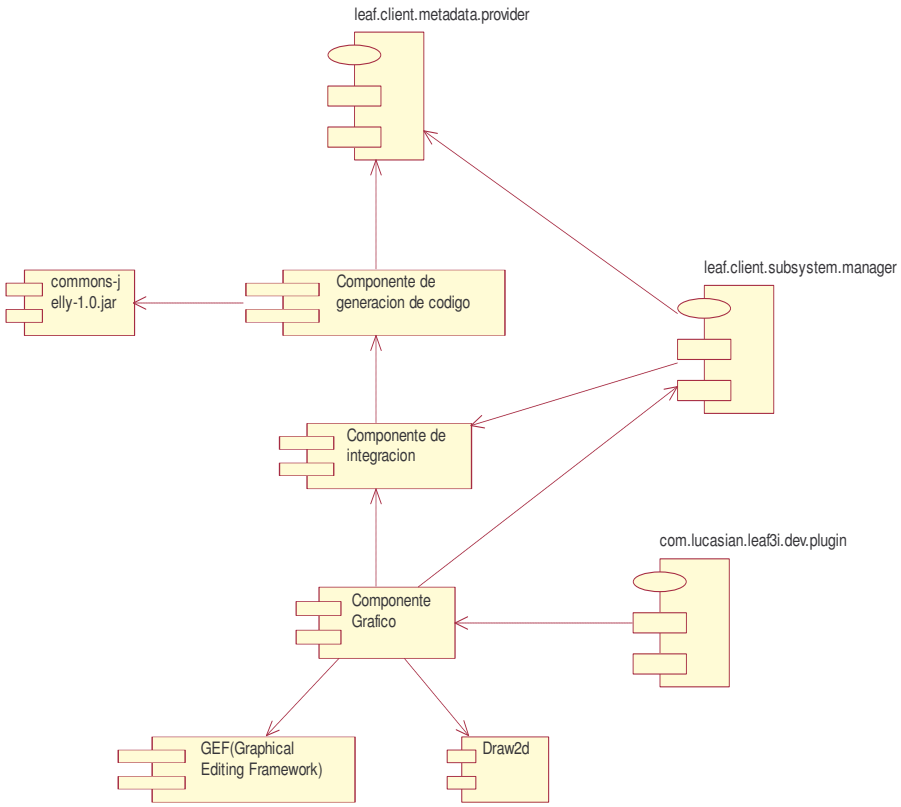
---

[2] Web Services Axis [en línea]. U.S : The Apache Software Foundation, 2006. [consultado 12 de Septiembre de 2006]. Disponible en internet: http://ws.apache.org/axis/

6.3 DIAGRAMA DE COMPONENTES

Este diagrama de componentes explica como es la interacción de los componentes a desarrollar con los demás elementos que se tiene actualmente dentro de Lucasian Labs. En este capitulo se da descripción breve de cada uno de los componentes.

Figura 2. Diagrama de componentes



MetadataProvider (Leaf.client.metadata.provider)

Componente encargado de extraer la descripción de la data, sobre la cual se pretende hacer el mapeo objeto-relacional. Esto lo logra mediante parámetros definidos por el usuario, es decir se extrae la información con la cual están construidas las tablas físicamente en la base de datos (nombre, campos, tamaño de los campos, llaves primarias, llaves foráneas, tipo de dato del campo).

Componente de Generación (leaf.code.pattern.generator)

Este componente generador de código es el que construye físicamente los archivos java generados. Su labor principal es que basado en los parámetros entregados por el usuario en la capa de presentación, se construyen clases java por medio de los valores extraídos de la base de datos; tarea realizada previamente por medio del componente *MetadataProvider*.

Para la creación de los archivos se utilizan parámetros como: directorio de generación, tipo de archivo a generar (DAO, VO, Business Services, DTO, ROE), formateador de código, Copyright y plantilla Jelly de generación.

Commons- Jelly

Herramienta propietaria de Apache que interpreta parámetros de entrada enviados en tiempo de ejecución, sobre los cuales crea los archivos Java físicamente. Esta herramienta, que esta basada en XML, permite tener código Java embebido lo que hace más fácil la generación de código para este proyecto.

Componente de Integración

Este componente sirve para realizar la lectura del diagrama de Composite Entity configurado por el usuario .El objetivo de este componente es que sirva como un intérprete de la información contenida dentro del diagrama para que luego basado en esta información, el sistema genere las clases de CE.

Componente Grafico

Este componente es utilizado directamente por el usuario y es donde después de haber generado un diagrama de Composite Entity, el sistema permite hacer modificaciones (Insertar tablas, relaciones, redimensionar el editor) o simplemente el llamado al componente de generación.

Administrador de Subsistemas (leaf.client.subsytem.manager)

Este componente es de integración, permite guardar las configuraciones del proyecto y conectar los componentes de presentación y los componentes de generación. Es decir permite que el usuario dentro de los Wizard de Eclipse pueda tener la información de la base de datos y que a la vez la interfaz de usuario se pueda comunicar con los demás componentes del sistema.

Plugin (com.lucasian.leaf3i.devp.plugin)

Componente de presentación, que permite tener el ambiente de generación inmerso en el IDE Eclipse, aquí están definidas las acciones que permite realizar el Framework, los Wizard con los que interactúa el usuario, los menús

contextuales de la aplicación. Es decir, este componente es el que permite ingresar los parámetros de generación y el llamado a los generadores.

GEF (Graphical Editing Framework)

Componente de eclipse que permite crear editores que contengan figuras con las que pueda interactuar el usuario. Este componente permite al usuario interactuar con las entidades de negocio (moverlas, borrarlas, modificar textos), crear acciones y eventos sobre el editor.

Draw2d

Componente gráfico de Eclipse que crea las figuras 2d dentro del editor, es decir, este componente es el que permite ver gráficamente las entidades dentro del diagrama de Composite Entity.

7. DESARROLLO DEL PROYECTO

La metodología adoptada en este proyecto es particular por lo específica de la solución, por esta razón no se siguió un estándar de desarrollo como el que propone RUP (Rational UniFied Process). Durante este capitulo se mostrara una a una las herramientas conceptuales y tecnológicas sobre las que se desarrollo el proyecto.

Inicialmente se estudio la propuesta de integrar el Componente de Generación de Composite Entity dentro Leaf, para ver que tan factible era la inclusión del proyecto en la planeacion semestral de la empresa, se decidió que la propuesta era ambiciosa, pero que representaba una diferencia ante productos como Toplink de Oracle, Hibernate de Sun.

El primer paso al iniciar el desarrollo del proyecto fue la investigación del patrón de diseño y la forma de implementación de esté dentro del producto. Las tareas mas relevantes ejecutadas en esta etapa fue la investigación del Framework de patrones adoptado por Lucasian Labs, la definición del alcance productivo de las clases generadas y finalmente como reutilizar lo ya implementado en la versión anterior del producto.

Una vez terminada esta etapa se inicio con la implementación de código, es decir, se empezó con la primera etapa de desarrollo que fue complementar el componente metadata-provider de modo que se pudiera extraer la información de composición de las tablas necesarias para la creación de las clases.

La segunda etapa del desarrollo fue crear el generador de código basado en los estándares de la empresa, se comenzó por investigar la tecnología Jelly y crear las plantillas que permiten la creación de archivos físicos Java y la creación de las plantillas de generación Jelly, se crearon las clases de invocación y paso de parámetros para esta tecnología.

La tercera etapa del desarrollo se centro en las tecnologías que permitieron crear la interfaz de usuario final del componente, es decir, las herramientas con las que el usuario final interactuária con el sistema. Las tecnologías investigadas en esta etapa son fundamentalmente Frameworks que permiten la creación de ambientes gráficos.

La última etapa del desarrollo fue la integración y las pruebas que se le hicieron al componente ya inmerso dentro de eclipse. Se realizo una prueba de concepto sobre el componente de generación y el código generado.

Nota: Siendo consecuentes con la metodología propuesta en capítulo ocho, el desarrollo del proyecto se concentro en hacer un proceso iterativo, en donde el resultado entregado en cada uno de los pasos era el punto de partida para el siguiente.

7.1 PRODUCTO DE REFERENCIA

En este capitulo se dará una breve introducción de Lucasian Labs y su producto Leaf3i, herramienta de la cual hace parte el componente final de este proyecto Composite Entity.

7.1.1 Lucasian Labs. Lucasian Labs es una compañía de capital intelectual, la cual brinda servicios especializados en consultoría, desarrollo y educación, focalizados en arquitectura, diseño, desarrollo y puesta en producción de proyectos basado en la tecnología JEE para empresas del sector financiero, telecomunicaciones y gobierno. La fortaleza de la empresa está en la utilización práctica de técnicas de ingeniería de software y arquitectura para el desarrollo de aplicaciones empresariales de múltiples niveles, construidas con servicios distribuidos haciendo uso de estándares y herramientas ampliamente adoptadas en la industria de software.

7.1.2 Leaf3i. LEAF es una plataforma de servicios empresariales, la cual permite aumentar la productividad y calidad del desarrollo de proyectos JEE. Leaf3i es un Framework Caja-Blanca que implementa la Arquitectura de Referencia SOA para Java Enterprise Edition, soportada en patrones de diseño. Leaf3i se puede ver como una plataforma que esta compuesta por una serie de componentes tanto teóricos como tecnológicos, los cuales permiten tener un producto robusto. A continuación se describirá cada uno de los componentes de Leaf3i:

- **Core J2EE Design Patterns**. Framework estándar de patrones de diseño JEE definido por los servicios de consultoría de Sun Microsystems.

- **Lucasian Design Patterns (LDP)**. Patrones de diseño J2EE de Lucasian Labs Ltda. complementarios a los patrones propuestos por Sun Microsystems.

- **LEAF Business Wizards (LBW).** Los asistentes están implementados como plugins sobre la plataforma Eclipse 3.x, generan código fuente de alto rendimiento basados en la especificación de la arquitectura de referencia para JEE. Estos componentes son portables entre distintos servidores de aplicaciones.

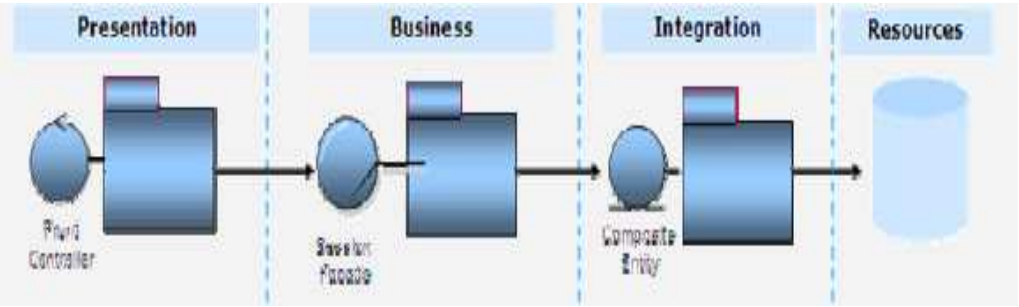
- **LEAF Components**. A través de la implementación de proyectos JEE, se han recopilado componentes reutilizables entre proyectos, los cuales están probados, documentados y que satisfacen requerimientos comunes de infraestructura en Java Enterprise Edition; entre ellos esta el manejo de excepciones y la auditoria de calidad.

7.1.3 Componentes Leaf3i. Leaf3i es una herramienta de productividad orientada a proyectos J2EE, la cual define una arquitectura basada en la experiencia en proyectos. Esta arquitectura esta basada en la teoría de capas , en la cual están definidas estrategias, patrones y generadores de código que permiten ser más ágiles al momento de implementar un proyecto.

Leaf3i es mucho más robusto en cuanto a la plataforma de servicios como se explico en el capitulo 9.2 del documento, pero para mostrar como encaja esté proyecto dentro de la arquitectura propuesta por Lucasian, se mostrara la capa de negocio en donde están involucrados los patrones de diseño y los generadores de código. En el siguiente grafico se muestra el diagrama de componentes que debe tener un proyecto hecho con Leaf3i:

Diagrama de Componentes Leaf3i

Figura 3. Arquitectura por capas propuesta por Lucasian Labs



- **Presentación:** Esta es la capa de usuario final una arquitectura por capas, Lucasian Labs propone un de los Framewors mas utilizados para su implementación en los proyectos, JSF (Java Server Faces).

  JSF [3](Java Server Faces) es un framework de desarrollo basado en el patrón MVC (Modelo Vista Controlador). Este framework pretende normalizar y estandarizar el desarrollo de aplicaciones Web. JSF es un conjunto de API's para representar componentes de una interfaz de usuario y administrar su estado, manejar eventos, validar entradas, definir el esquema de navegación de las páginas y dar soporte para internacionalización y accesibilidad. El resultado final de utilizar Java Server Faces es un conjunto de paginas Web (JSP) las cuales tienen asociadas Back in Beans (Clases controladoras de eventos) y son invocados por reglas de navegación. Como se puede ver en la figura 3, los JSP implementados basándose en JSF son llamados por un controlador frontal dependiendo de la operación escogida por el usuario.

[3] Java Server Faces [en línea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: *http://java.sun.com/javaee/javaserverfaces/*

- **Business:** Esta capa permite definir operaciones de negocio del lado del servidor. La herramienta estándar utilizada en esta capa son los EJBs[4], los cuales proporcionan un modelo de componentes para el lado del servidor. El objetivo de los Enterprise Java Beans es dar al programador un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad), para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes nos permite que éstos sean flexibles y sobre todo reutilizables.

  Los EJB son un conjunto de interfaces y clases Java que son localizadas por el contenedor de aplicaciones y que dependiendo de el evento realizado por el usuario en presentación se llama a la lógica (código) implementada para esa operación. La conexión entre la capa de presentación y la capa de negocio se hace por medio de una de las interfaces expuestas por los EJBs, Sesión Fachada (Fachada).

- **Integración:** En esta capa se ubica la solución final de este proyecto y es básicamente la integración entre los servicios de negocio (EJB) y los componentes que interactúan con la base de datos (DAO, ROE). Los CE pueden ser vistos como una interfaz la cual dependiendo del llamado del servicio de negocio pueden ejecutar búsquedas, inserciones, actualizaciones y eliminaciones masivas invocando los DAOs asociados a la entidad compuesta invocada por el servicio de negocio.

  Los DAOs son clases abstractas las cuales por medio de JDBC(Java Database Connectivity) pueden conectarse con la base de datos y por medio de Prepared Statements guardados realizar operaciones SQL..

  Mas adelante en este documento se explicara más a fondo los patrones de diseño involucrados en esta capa.

- **Resources:** Esta capa se define los componentes físicos de almacenamiento de datos, es decir Bases de datos, LDAP (Lightweight Directory Access Protocol) .

7.1.4 Leaf3i – SOA. La Arquitectura Orientada a Servicios es un concepto de arquitectura de software y define la utilización de servicios para dar soporte a los requerimientos de software del usuario.

SOA proporciona una metodología y un marco de trabajo para documentar las capacidades de negocio, puede dar soporte a las actividades de integración y consolidación.

---

[4] Enterprise Java Beans [en línea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: *http://java.sun.com/products/ejb/*

En un ambiente SOA, los nodos de la red empresarial hacen disponibles sus recursos a otros participantes en la red como servicios independientes a los que tienen acceso de un modo estandarizado. La mayoría de las definiciones de SOA identifican la utilización de Servicios Web en su implementación.

Para Leaf3i es una arquitectura que le permite estar de la mano con la tendencia de desarrollo de software dentro de las empresas y además poderse integrar con cualquier herramienta que pueda consumir sus servicios. Para estar de la mano con SOA Leaf permite exponer los métodos de las clases generadas como servicios Web basados en los estándares [5]actuales.

Para este proyecto las clases generadas podrán ser expuestas como servicios Web los cuales podrán ser consumidos por clientes creados por el desarrollador o simplemente por un orquestador de aplicaciones.

7.2 FRAMEWORK DE PATRONES

Un patrón describe, con algún nivel de abstracción, una solución experta a un problema. Normalmente, un patrón está documentado en forma de una plantilla. Aunque es una práctica estándar documentar los patrones en un formato de plantilla especializado, esto no significa que sea la única forma de hacerlo. Además, hay tantos formatos de plantillas como autores de patrones, esto permite la creatividad en la documentación de patrones.

Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que describen soluciones para todo, desde el análisis hasta el diseño y desde la arquitectura hasta la implementación. Además, los patrones existen en diversas áreas de interés y tecnologías. Por ejemplo, hay un patrón que describe como trabajar con un lenguaje de programación específico o un segmento de la industria específico, como la sanidad.

En este capitulo se presentaran los patrones de diseño involucrados dentro del proyecto y que rol juegan en la arquitectura propuesta por Lucasian Labs.

7.2.1. Composite Entity. EL patrón Composite Entity[6] es propuesto por Sun como una solución a las entidades Compuestas. Lucasian Labs ha estudiado esta propuesta y ha visualizado que encaja dentro de propósitos de realizar un mapeo objeto relacional de un repositorio central de datos.

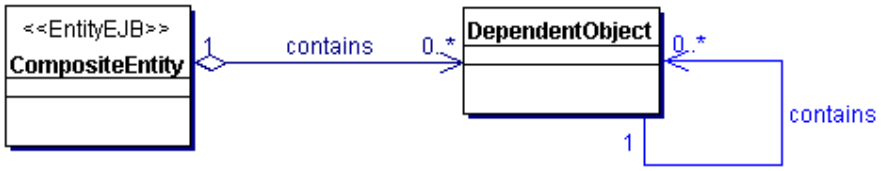
---

[5] Axis Architecture Guide [en línea]. U.S : The Apache Software Foundation, 2006. [consultado 12 de Septiembre de 2006]. Disponible en internet: http://ws.apache.org/axis/java/architecture-guide.html

[6] Core J2EE Patterns – Composite Entity [en linea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: http://java.sun.com/blueprints/corej2eepatterns/Patterns/CompositeEntity.html

La propuesta de Sun esta enfocada hacia la interacción de objetos persistentes, entendiendo como objetos persistentes aquellos que son almacenados. Un objeto persistente puede ser genérico o dependiente. La diferencia entre ambos es que en un objeto dependiente su ciclo de vida depende de un objeto padre o objeto genérico, que a su vez es el que controla sus operaciones. La interacción entre objetos dependientes y genéricos se puede ver como un árbol. La raíz del árbol seria el objeto genérico, que a su vez puede ser un objeto dependiente y las hojas serian los objetos dependientes de este.

Figura 4. Diagrama Composite Entity propuesto por SUN

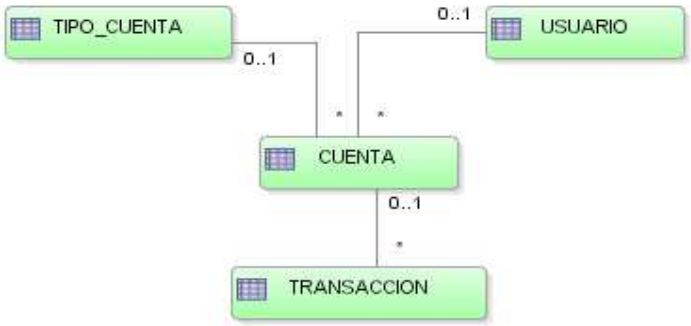


Esta propuesta encaja perfectamente con las intensiones de Lucasian Labs de hacer un mapeo objeto relacional de las entidades de negocio compuestas de la base de datos. Es muy común ver que una entidad de negocio esta distribuida en gran cantidad de tablas y que realizar operaciones (inserciones, búsquedas, borrado, actualización) sobre la data es bastante complicado.

Para aclarar un poco el concepto se puede citar un ejemplo sencillo una cuenta bancaria. *Cuenta* es una entidad compuesta, para verlo mas claro se puede ver la siguiente figura:

- **MER** (Modelo entidad relación)

Figura 5. MER ejemplo cuenta bancaria.



El anterior modelo entidad relación muestra la interacción básica de una cuenta bancaria, se evidencia por tanto que la cuenta tiene asociado un tipo de cuenta y un usuario, además puede ser incluida en una o muchas transacciones.

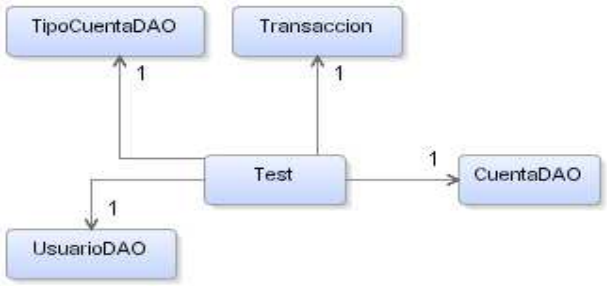
Por ejemplo, si hiciéramos un mapeo por separado, hacer una búsqueda de la información de un usuario que hizo una transacción bancaria, debería ejecutar los siguientes pasos:

- Buscar la transacción
- Buscar la el numero de cuenta
- Buscar la información del usuario
- Buscar la información del tipo de cuenta

Ya visto como un diagrama de clases el usuario tendría la siguiente representación:

- Diagrama de Clases Utilización DAO

Figura 6. Diagrama de clases mapeo objeto relacional DAO por separado



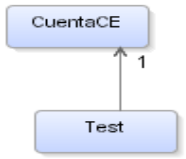
La propuesta que se tiene por medio de la implementación del patrón Composite Entity es hacer un mapeo objeto relacional de toda la entidad, en donde la secuencia de pasos se resumiría a ingresar solamente a:

- Buscar la información de los padres(Tipo cuenta , usuario)
- Buscar la transacción.

Ya visto como un diagrama de clases, el usuario tendría la siguiente interacción:

- Diagrama de utilización Composite Entity

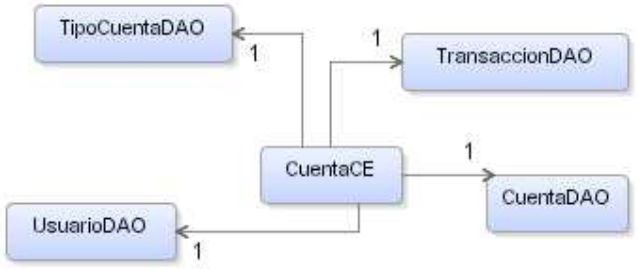
Figura 7. Diagrama de clases CE ejemplo de prueba



Es decir el esfuerzo computacional hecho por parte del usuario es muy poco, ya que si vemos la composición de un CE entregado por el sistema seria así:

- Diagrama de Clases Composición Composite Entity

Figura 8. Composición Composite Entity



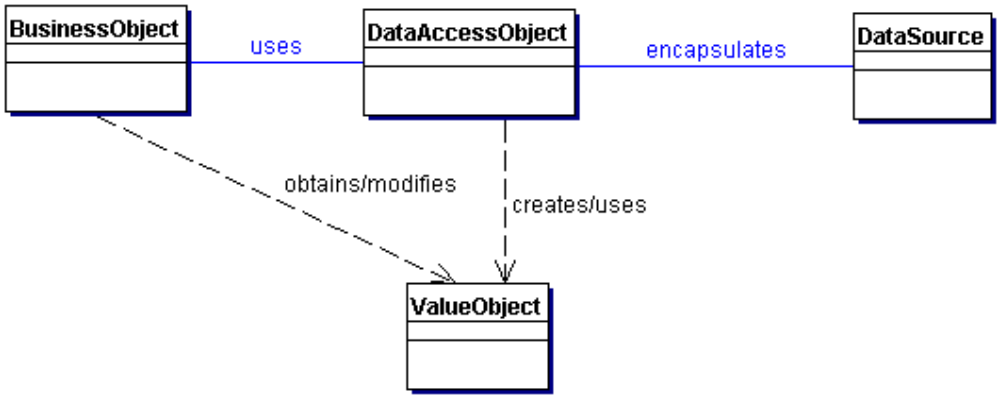
El código generado basado en el patrón Composite Entity, aplicado a los proyectos seria una herramienta, que basada en relaciones de base de datos y entradas de relaciones dadas por el usuario, simplificaría totalmente las operaciones hacia la base de datos. Esto se puede asegurar ya que el patrón de diseño garantiza abstraer totalmente una entidad de negocio con todos sus objetos genéricos y dependientes desde la base de datos y exponer métodos java que involucren a cada tabla de la entidad de negocio en operaciones comunes o individuales.

7.2.2 Patron DAO (Data Access Object). El patrón DAO es propuesto por Sun como el mecanismo de acceso a datos, es decir, es la herramienta que propone J2EE para interactuar directamente con el repositorio central de datos. Esta fuente de datos puede ser un almacenamiento persistente como una RDMBS, un repositorio LDAP.

Los componentes de negocio que tratan con el DAO utilizan una interface simple expuesta por el DAO para sus clientes. El DAO oculta completamente los detalles de implementación de la fuente de datos a sus clientes. La implementación del DAO no cambia según la base de datos, ya que este patrón permite adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio. Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos

La siguiente figura muestra el diagrama de clases que representa las relaciones para el patrón DAO:

Figura 9. Representación del patrón DAO propuesto por SUN

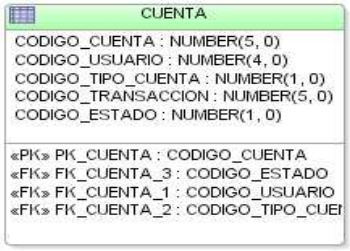


Los DAO's son muy importantes para el desarrollo de este proyecto ya que permitirán que los CE (Bussines Object) puedan operar con la base de datos. Es decir los DAO's ya están implementados dentro del plugin de eclipse y serán reutilizados masivamente en las clases generadas por este proyecto.

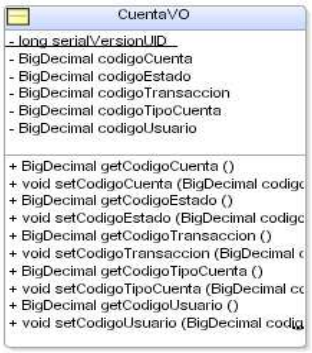
7.2.3 Patrón DTO (Data Transfer Object). El patrón de diseño DTO es un objeto de almacenamiento. Es decir en un objeto DTO puede almacenar objetos VO, que son la representación objeto de la metadata de las tablas. Para este proyecto son de gran importancia ya que se almacenaran los valores de cada una de las tablas padres que compongan una entidad de negocio y además colecciones de VO de objetos de las tablas hijas.

7.2.4 Patrón VO (Value Object). Las clases VO son utilizadas como medio de almacenamiento de la información extraída de la base de datos, es decir por cada campo de una tabla de base de datos se ofrece un método get y un metodo set, los cuales sirven para interactuar con el repositorio central de datos. Estas clases permiten tener una aplicación escalable y de facil mantenimiento. Para ilustrar un poco sobre este patrón se mostrara la representación objeto de la tabla Cuenta versus la representación modelo entidad relación.

- Representación de base de datos tabla Cuenta



- Representación Objeto de la tabla Cuenta



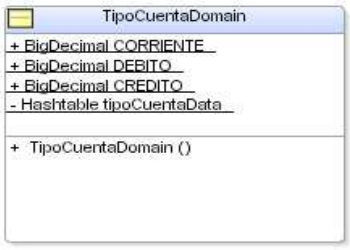
7.2.5 Patrón Domain Object. El patrón Domain Object esta concebido dentro de Lucasian Labs como una representación objeto de los registros físicos de una tabla tipo. Es muy común ver en el modelamiento de esquemas de base de datos, tablas de 2 campos las cuales tienen un número determinado de registros (Ej. Tablas de estados), sobre las cuales se hacen cantidad de consultas que afectan el rendimiento de las transacciones y los cuales casi nunca son modificados, es por esta razón que los ingenieros de Lucasian proponen que estas tablas sean llevadas a objetos haciendo mucho mas rápida la consulta.

Siguiendo con el ejemplo de las cuentas podemos tomar la tabla tipo_cuenta, la cual probablemente siempre tenga el mismo número de registros y siempre que se necesite hacer una operación sobre la cuenta probablemente deberá consultar el tipo de cuenta.

Ej. Visto como registros de base de datos:

Id	Descripción
1	Cuenta de ahorros
2	Cuenta Corriente
3	Crédito

Visto como una clase:



La utilización de este patrón dentro de los proyectos es de gran ayuda, ya que la transacciónalidad sobre las tablas dominio o tablas tipo es bastante alta en

los proyectos y dependiendo del diseño de base de datos puede ser muy costoso el número de consultas hacia este tipo de tablas.

7.3 GENERADOR DE CODIGO

En este capitulo se explicara el ambiente de generación de clases y las herramientas involucradas en la fase de creación de archivos Java de este proyecto.

7.3.1 Jelly[7]. Es un lenguaje de Scriptng basado en XML que sirve como plantilla de generación de código dentro del componente CE. Jelly es una herramienta que permite tener código de Java embebido haciendo más sencilla la creación de las clases, ya que se puede invocar directamente bloques de código para hacer comparaciones, ciclos y demás sentencias básicas de programación.

Jelly define algunos tags XML, los cuales permiten realizar operaciones de programación dentro de las plantillas, a continuación se dará una breve descripción de los más utilizados para crear plantillas de generación:

Choose	Condiciona la evaluación de un bloque de código sobre una misma decisión
forEach	Ciclo que permite iterar cualquier tipo de colección de objetos de Java.
Case	Casos que evalúan los diferentes valores de un mismo atributo
New	Permite crear un nuevo objeto Java
Switch	Se define el atributo que va a ser evaluado dentro de los casos
If	Permite evaluar una expresión
While	Permite iterar un objeto java
Import	Permite hacer el import de un objeto de java que sea utilizado para crear la plantilla

Dentro de Lucasian Labs se ha adoptado una metodología de desarrollo de los generadores de código que esta compuesta por tres pasos:

- Clase de invocación : Esta clase permite que sea invocado el generador recibiendo los parámetros de configurados por el usuario en su interacción con la presentación del sistema.

---

[7] Jelly Executable XML [en línea]. U.S : The Apache Software Foundation, 2006. [consultado 12 de Septiembre de 2006]. Disponible en internet: http://jakarta.apache.org/commons/jelly/

- Plantilla Jelly: Basada en los parámetros pasados por la clase de invocación y los Tags XML escritos por el desarrollador tener la generación física de la clase.

- Clase generada: Salida presentada al usuario la cual no podrá ser modificada por él.

Para tener un poco más claro el proceso de desarrollo de un generador de código basado en Jelly, se mostrara un ejemplo de un generador de un archivo HTML. A continuación se expondra como es el proceso:

- Clase de invocación

```
package com.lucasian.leaf.generator.log;

import com.lucasian.leaf.common.DirectoryUtil;


public class HTMLogGenerator {
    public HTMLogGenerator() {
    }

    public void buildHtmlDoc(String templateName, String logPath,
        ArrayList pcvo, String fileName) {
        try {
            Date fecha = new Date();
            String logDiretory = logPath + "/" + (fecha.getYear() + 1900) +
                "-" + (fecha.getMonth()+1) + "-" + fecha.getDate();
            DirectoryUtil.createDirectoryStructure(logDiretory);
             OutputStream output = new FileOutputStream(logDiretory + "/" +
                    fileName + ".html");
            JellyContext context = new JellyContext();
            context.setVariable("ProgressControlVO", pcvo);

            XMLOutput xmlOutput = XMLOutput.createXMLOutput(output);
            context.runScript(new java.io.File(templateName), xmlOutput);

            xmlOutput.flush();
            xmlOutput.close();
            output.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Plantilla Jelly

```
generationLogTemplate.jelly

<?xml version="1.0" encoding="UTF-8"?>
<j:jelly xmlns:j="jelly:core" xmlns:my="actionFramework">
        <j:whitespace>
                <html>
                <head>
                        <title>Leaf Generation Log</title>
                </head>
                <h3 align="center">Leaf Generation Log</h3>
                <table border="5" >
                        <j:set var="counter" value='1'/>
                                <tr bgcolor = "#AAAAAA">
                                <td width="50%">Date</td>
                                <td width="50%">Task</td>
                                <td width="50%">State</td>
                        </tr>
                        <j:forEach items="${ProgressControlVO}" var="i">
                                <j:choose>
                                    <j:when test="${counter==0}">
                                            <tr  bgcolor = "#CCCCCC">
                                                    <td width="50%">${i.date}</td>
                                                    <td width="50%">${i.className}</t
                                                    <td width="50%">${i.state}</td>
                                                    <j:set var="counter" value='1'/>
                                            </tr>
                                    </j:when>
                                    <j:when test="${counter==1}">
                                            <tr>
                                                    <td width="50%">${i.date}</td>
                                                    <td width="50%">${i.className}</t
                                                    <td width="50%">${i.state}</td>
                                                    <j:set var="counter" value='0'/>
                                            </tr>
                                    </j:when>
                                </j:choose>
                        </j:forEach>
                </table>
                </html>
        </j:whitespace>
</j:jelly>
```

- Archivo Generado

Figura 10. Generación Jelly de archivo HTML

Leaf Generation Log

Date	Task	State
Wed Oct 25 12:02:58 COT 2006	Metadata extraction	started
Wed Oct 25 12:02:59 COT 2006	RecidenciaVO.java	finished
Wed Oct 25 12:02:59 COT 2006	DependeciasVO.java	finished
Wed Oct 25 12:03:00 COT 2006	CargoVO.java	finished
Wed Oct 25 12:03:00 COT 2006	AreasVO.java	finished
Wed Oct 25 12:03:00 COT 2006	EmpleadosVO.java	finished
Wed Oct 25 12:03:00 COT 2006	HijosEmpleadoVO.java	finished

Como se puede observar es una herramienta fácil de usar, la cual dentro de los componentes de generación de Lucasian Labs se ha explotado para generar clases complejas como DAO's, VO y ahora será utilizado para la generación de clases involucradas dentro del patrón Composite Entity.

7.3.2 Componente de generación. Actualmente en la generación de clases de Leaf se ha adoptado una metodología sencilla, la cual basado en parámetros configurados por el usuario, el sistema hace una generación masiva de clases.

El componente de generación para los Composite Entity, CEGenerator permite basado en los siguientes parámetros generar las clases CE, DTO y Domain Object:

- Directorio de plantillas: Directorio físico (C:\Documents and Settings\<User-sesion >\Configuración local\Temp) del computador donde este instalado el Framework donde son descargadas directamente desde el servidor de Lucasian Labs.

- Directorio de generación: Directorio físico del computador donde este instalado el Framework donde se crearan las clases generadas.

- Plantilla de Generación: Archivo Jelly el cual es la base de creación de las clases, este puede ser invocada una o más veces en una sola generación.

- Plantilla de formateo de clases: Archivo Jalopy que permite basado en parámetros configurados por el desarrollador, generar código indeltado y con saltos de línea definidos.

- Objeto de Generación: Objeto que puede ser considerado como un VO el cual trae todos los parámetros configurados por el usuario en la fase de presentación.

7.3.3 Generador CE. Este generador permite crear físicamente el archivo basado en el patrón Composite Entity**.** La plantilla de generación contiene la información necesaria para crear los métodos que permiten al usuario realizar las transacciones masivas sobre la base de datos (inserciones, búsquedas, actualizaciones y eliminaciones). Esta clase generada tiene como principios de construcción la abstracción del patrón de diseño Composite Entity que tiene Lucasian Labs.

Los servicios presentados a continuación fueron discutidos al interior de la empresa y se expondrán en las clases generadas en un futuro .A continuación se presentaran los métodos con sus respectivos nombres y segmento Jelly de construcción:

Servicios Composite Entity

- *Add* (Inserciones masivas)

  Este método permite al usuario hacer inserciones masivas en la entidad de negocio, solamente suministrando un objeto de almacenamiento DTO (Data Transfer Object). El proceso que hace internamente, es realizar una lectura de la entidad de negocio para establecer las prioridades de inserción, es decir para poder hacer la inserción de los objetos hijos previamente debe haberse hecho la inserción dentro de la entidad, respetando la integridad referencial.

  Nota: Las inserciones solo se harán sobre la tabla entidad (base de la entidad de negocio) y sobre sus correspondientes hijos, es decir se excluye la inserción sobre las tablas padres de la entidad.

  Plantilla Jelly

```
public void add( ${PARAM_CLASS.className}DTO
${PARAM_CLASS.attributeName}DTO )
throws ${EXCEPTION} {
                Assert.isNotNull(${PARAM_CLASS.attributeName}DTO ,
                "${PARAM_CLASS.className}DTO  specified is null!!");
                final String METHOD_NAME = "add";
                final long START_TIME =
                System.currentTimeMillis();<j:forEach
                items="${PARAM_CLASS.attributesCEVOSet}" var="i">
                <j:if test="${(!i.isChildren() &amp;&amp;
                !i.isParent() &amp;&amp; !i.isDomain())}">
        try{
                ${i.methodName}DAO ${PARAM_CLASS.attributeName}DAO =
                daoFactory.get${i.methodName}DAO();
                </j:if><j:if test="${i.isChildren()}">
                ${i.methodName}DAO ${i.attributeName}DAO =
                daoFactory.get${i.methodName}DAO();
                </j:if></j:forEach><j:forEach
                items="${PARAM_CLASS.attributesCEVOSet}" var="i">
                <j:if test="${(!i.isChildren() &amp;&amp;
                !i.isParent()  &amp;&amp; !i.isDomain())}">
                ${i.attributeName}
                DAO.insert(${PARAM_CLASS.attributeName}
                DTO.get${i.methodName}VO());
                </j:if><j:if test="${i.isChildren()}">

                if(${PARAM_CLASS.attributeName}DTO.get${i.methodName}
                DTOSet() != null)
                {
                        ${i.attributeName}
                        DAO.insert(${PARAM_CLASS.attributeName}
                        DTO.get${i.methodName}DTOSet(),0);
                }</j:if></j:forEach>
            }finally{
                PerformanceLogger.writeLog(getClass().getName(),
                METHOD_NAME,
                        START_TIME);
            }
```

- *Find*ByPrimary *(búsqueda Masiva sobre los padres y la entidad)*

  Este método permite que a partir de un objeto VO (Value object) de la entidad, se busque todos los campos de la tabla por medio de la llave primaria y además cada uno de los padres con sus respectivos datos. Por medio de la metadata extraída previamente a la generación se expone un método por medio del cual se pueda tener toda la información de la tabla entidad y todas las relaciones foráneas que apunten hacia ella. Cabe anotar que una tabla padre puede ser una tabla Dominio.

  Plantilla Jelly

```
public ${PARAM_CLASS.className}DTO
findByPrimary(${PARAM_CLASS.className}VO object)
throws ${EXCEPTION}
 {

        final String METHOD_NAME = "findByPrimary";
        final long START_TIME = System.currentTimeMillis();
        ${PARAM_CLASS.className}DTO ${PARAM_CLASS.attributeName}
        DTO= new  ${PARAM_CLASS.className}DTO ();

  try{
        ${PARAM_CLASS.className}DAO ${PARAM_CLASS.attributeName}D
        = daoFactory.get${PARAM_CLASS.className}DAO();
        ${PARAM_CLASS.className}VO ${PARAM_CLASS.attributeName}VO
        ${PARAM_CLASS.attributeName}DAO.findByPrimaryKey(object);

        ${PARAM_CLASS.attributeName}DTO.set${PARAM_CLASS.classNam
        VO (${PARAM_CLASS.attributeName}VO);
        <j:forEach items="${PARAM_CLASS.attributesCEVOSet}"
        var="i"><j:if test="${(i.isParent())}">
        ${i.methodName}DAO ${i.attributeName}DAO =
        daoFactory.get${i.methodName}DAO();
        <j:if test="${i.getLengthForeing()>0}"><j:set var="counte
        value='0'/>
        <j:forEach items="${i.foreing.keys()}" var="k">
        <j:if test="${(counter &lt;= i.getLengthForeing())
        &amp;&amp; counter != '0' }">
        ${i.methodName}VO ${i.attributeName}VO${counter}= new
        ${i.methodName}VO();

        ${i.attributeName}VO${counter}.set${i.foreing.get(k)}
        (${PARAM_CLASS.attributeName}VO.get${k}());
        ${PARAM_CLASS.attributeName}DTO.set${i.methodName}
        VO${counter}(${i.attributeName}
        DAO.findByPrimaryKey(${i.attributeName}VO${counter}));
        </j:if><j:if test="${counter == '0'}">
        ${i.methodName}VO ${i.attributeName}VO= new ${i.methodNam
        VO();
        ${i.attributeName}VO.set${i.foreing.get(k)}
        (${PARAM_CLASS.attributeName}VO.get${k}());
        ${PARAM_CLASS.attributeName}DTO.set${i.methodName}
        VO(${i.attributeName}
        DAO.findByPrimaryKey(${i.attributeName}VO));
        </j:if><j:set var="counter" value='${counter+1}'/>
        </j:forEach ></j:if><j:if test="${i.getLengthForeing()==
        0}">
        ${i.methodName}VO ${i.attributeName}VO= new ${i.methodNam
        VO();<j:forEach items="${i.foreing.keys()}" var="k">
        ${i.attributeName}VO.set${i.foreing.get(k)}
        (${PARAM_CLASS.attributeName}VO.get${k}());
        </j:forEach >
        ${PARAM_CLASS.attributeName}DTO.set${i.methodName}
        VO(${i.attributeName}
        DAO.findByPrimaryKey(${i.attributeName}VO));
        </j:if></j:if><j:if test="${(i.isDomain())}">

        ${i.methodName}Domain ${i.attributeName}Domain = new
        ${i.methodName}Domain();<j:forEach
        items="${i.foreing.keys()}" var="j" >

        ${i.attributes.get(j.substring(0,1).toLowerCase().concat(
        substring(1)))} value${i.methodName}=
        ${PARAM_CLASS.attributeName}VO.get${j} ();
        </j:forEach>
        ${PARAM_CLASS.attributeName}DTO.set${i.methodName}
        (${i.attributeName}Domain.find${i.methodName}
```

- *Find*AllChild *(búsqueda masiva sobre la entidad y sus hijos)*

  Este método permite realizar una búsqueda a partir de un valor de llave primaria suministrado por el usuario. Es decir este método permitirá al usuario manejar el concepto de Maestro – Detalle a partir de la tabla entidad hacia todos sus objetos hijos.

  Plantilla Jelly

```
public ${PARAM_CLASS.className}DTO  findAllChild(
        <j:forEach items="${PARAM_CLASS.attributesCEVOSet}" var="i"><j:if
        test="${!i.isParent() &amp;&amp; !i.isChildren() &amp;&amp; !i.isDomain()}"><j:s
        var="counter" value='1'/><j:forEach items="${i.attributes.keys()}" var="j"><j:if
        test="${counter == '1'}">
         String ${j}</j:if><j:if test="${counter > '1'}">,
         String ${j}</j:if><j:set var="counter" value='${counter+1}'/></j:forEach ></j:i
         </j:forEach >
        ) throws ${EXCEPTION} {

        final String METHOD_NAME = "findAllChild";
        final long START_TIME = System.currentTimeMillis();<j:set var="countChild"
        value='0'/><j:forEach items="${PARAM_CLASS.attributesCEVOSet}" var="i">
        <j:if test="${i.isChildren() &amp;&amp; !i.isSemantics()}"><j:set var="countChil
        value='${count+1}'/></j:if></j:forEach ><j:if test="${countChild &gt; 0}">
        Object[] objectArray =null;</j:if>

        ${PARAM_CLASS.className}DTO ${PARAM_CLASS.attributeName}DTO= new
        ${PARAM_CLASS.className}DTO ();
try{

        ${PARAM_CLASS.className}DAO ${PARAM_CLASS.attributeName}DAO =
        daoFactory.get${PARAM_CLASS.className}DAO();
        ${PARAM_CLASS.className}VO ${PARAM_CLASS.attributeName}VO= new
        ${PARAM_CLASS.className}VO ();
        <j:forEach items="${PARAM_CLASS.attributesCEVOSet}" var="i">
        <j:if test="${!i.isParent() &amp;&amp; !i.isChildren() &amp;&amp; !i.isDomain()}
        <j:forEach items="${i.attributes.keys()}" var="j">
        ${PARAM_CLASS.attributeName}
        VO.set${j.substring(0,1).toUpperCase().concat(j.substring(1))}(new
        ${i.attributes.get(j)}(${j}));</j:forEach >

        StringBuffer parameters${PARAM_CLASS.className} = new StringBuffer();<j:set
        var="count" value='0'/>
        <j:forEach items="${i.parameters.keys()}" var="k" ><j:if test="${count == '0'}">
        parameters${PARAM_CLASS.className}.append(
        "${k}=").append(${i.parameters.get(k)});</j:if><j:if test="${  count > '0' }">
        parameters${PARAM_CLASS.className}.append(" AND " ).append(
        "${k}=").append(${i.parameters.get(k)});</j:if><j:set var="count"
        value='${count+1}'/></j:forEach ></j:if></j:forEach ><j:forEach
        items="${PARAM_CLASS.attributesCEVOSet}" var="i"><j:if test="${i.isChildren()
        &amp;&amp; !i.isSemantics()}">

        ${i.methodName}DAO ${i.attributeName}DAO = daoFactory.get${i.methodName}
        DAO();</j:if></j:forEach >
        ${PARAM_CLASS.attributeName}DTO.set${PARAM_CLASS.className}
        VO(${PARAM_CLASS.attributeName}DAO.findByPrimaryKey(${PARAM_CLASS.attributeName}
        VO));
        <j:forEach items="${PARAM_CLASS.attributesCEVOSet}" var="i"><j:if
        test="${i.isChildren()&amp;&amp; !i.isSemantics()}">

        objectArray=  ${i.attributeName}
        DAO.findByAnyWhere(parameters${PARAM_CLASS.className}.toString()).toArray();
        ${i.methodName}VO ${i.attributeName}VOArray [] = new ${i.methodName}VO[
        objectArray.length ];
        for(int i=0; i &lt; objectArray.length; i++){
                ${i.attributeName}VOArray[i] = (${i.methodName}VO) objectArray[ i ];
        }
        ${PARAM_CLASS.attributeName}DTO.set${i.methodName}DTOSet(${i.attributeName}
        VOArray$);</j:if></j:forEach >

    }finally{
                PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
                        START_TIME);
    }

 return ${PARAM_CLASS.attributeName}DTO;
}
```

- *FindBy*TablaPadre *(Búsqueda de la tabla entidad con uno de sus padres)*

  Este método permite al usuario tener dentro de un objeto de almacenamiento DTO (Data Transfer Object) solamente los valores de la tabla entidad y uno de sus padres. Esta búsqueda se hacer por medio de de la llave primaria de la tabla padre de la entidad.

  Plantilla Jelly

```
public ${PARAM_CLASS.className}DTO findByForeing${i.methodName}(<j:if
test="${i.isParent()}"><j:set var="counter" value='1'/><j:forEach
items="${i.attributes.keys()}" var="j"><j:if test="${counter == '1'}">
        String ${j}</j:if><j:if test="${counter > '1'}">,
        String ${j}</j:if><j:set var="counter" value='${counter+1}'/></j:forEach >
        </j:if>
        ) throws ${EXCEPTION} {

         final String METHOD_NAME = "findByForeing${i.methodName}";
         final long START_TIME = System.currentTimeMillis();
         ${PARAM_CLASS.className}VO ${PARAM_CLASS.attributeName}VOArray [] = null;
         Object[] objectArray = null;
         ${PARAM_CLASS.className}DTO ${PARAM_CLASS.attributeName}DTO= new
         ${PARAM_CLASS.className}DTO ();
try{

         ${PARAM_CLASS.className}DAO ${PARAM_CLASS.attributeName}DAO =
         daoFactory.get${PARAM_CLASS.className}DAO();
         ${i.methodName}DAO ${i.attributeName}DAO = daoFactory.get${i.methodName}DAO();
         <j:if test="${i.getLengthForeing()>0}"> <j:set var="counter" value='0'/>
         <j:forEach items="${i.foreing.keys()}" var="count" indexVar="index">
         <j:if test="${(counter &lt;= i.getLengthForeing()) &amp;&amp; counter != '0' }">
         ${i.methodName}VO ${i.attributeName}VO${counter}= new ${i.methodName}VO();
         ${i.attributeName}VO${counter}.set${i.foreing.get(count)}(new
         ${i.attributes.get(count.substring(0,1).toLowerCase().concat(count.substring(1)))}
         (${count.substring(0,1).toLowerCase().concat(count.substring(1))}));
         ${PARAM_CLASS.attributeName}DTO.set${i.methodName}VO${counter}(${i.attributeName}
         DAO.findByPrimaryKey(${i.attributeName}VO${counter}));</j:if><j:if test="${counter
         == '0'}">
         ${i.methodName}VO ${i.attributeName}VO= new ${i.methodName}VO();
         ${i.attributeName}VO.set${i.foreing.get(count)}(new
         ${i.attributes.get(count.substring(0,1).toLowerCase().concat(count.substring(1)))}
         (${count.substring(0,1).toLowerCase().concat(count.substring(1))}));
         ${PARAM_CLASS.attributeName}DTO.set${i.methodName}VO(${i.attributeName}
         DAO.findByPrimaryKey(${i.attributeName}VO));</j:if>
         <j:set var="counter" value='${counter+1}'/></j:forEach ></j:if> <j:if
         test="${i.getLengthForeing()==0}">
         ${i.methodName}VO ${i.attributeName}VO= new ${i.methodName}VO();<j:forEach
         items="${i.foreing.keys()}" var="count" indexVar="index">
         ${i.attributeName}VO.set${i.foreing.get(count)}(new
         ${i.attributes.get(count.substring(0,1).toLowerCase().concat(count.substring(1)))}
         (${count.substring(0,1).toLowerCase().concat(count.substring(1))}));</j:forEach >
         ${PARAM_CLASS.attributeName}DTO.set${i.methodName}VO(${i.attributeName}
         DAO.findByPrimaryKey(${i.attributeName}VO));</j:if>
         StringBuffer parameters${i.methodName} = new StringBuffer();<j:set var="counter"
        value='0'/> <j:forEach items="${i.parameters.keys()}" var="k" ><j:if
        test="${counter == '0'}">
         parameters${i.methodName}.append("${k}=").append(${i.parameters.get(k)});</j:if>
         <j:if test="${  counter > '0' }">
         parameters${i.methodName}.append( " AND ").append(
        "${k}=").append(${i.parameters.get(k)});</j:if><j:set var="counter"
        value='${counter+1}'/></j:forEach >
         objectArray = ${PARAM_CLASS.attributeName}
         DAO.findByAnyWhere(parameters${i.methodName}.toString()).toArray();
         ${PARAM_CLASS.attributeName}VOArray = new ${PARAM_CLASS.className}VO[
         objectArray.length ];
         for(int i=0; i &lt; objectArray.length; i++){
            ${PARAM_CLASS.attributeName}VOArray[i] = (${PARAM_CLASS.className}VO)
            objectArray[ i ];
         }
         ${PARAM_CLASS.attributeName}DTO.set${PARAM_CLASS.className}VOSet(
        ${PARAM_CLASS.attributeName}VOArray);
     }finally{
         PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
                START_TIME);
     }
  return ${PARAM_CLASS.attributeName}DTO;
}</j:if></j:forEach>
```

- *BrowseFor*TablaHija (*Búsqueda de la tabla entidad con uno de sus hijos*)

  Este método permite al usuario tener dentro de un objeto de almacenamiento DTO (Data Transfer Object) solamente los valores de la tabla entidad y uno de sus hijos. Esta búsqueda se hace por medio de la llave primaria de la entidad. Este método cambia en su firma, ya que se debe hacer para cada uno de las tablas hijas.

  Plantilla Jelly

```
public ${PARAM_CLASS.className}DTO browseFor${i.methodName}(<j:set var="counter"
value='1'/><j:forEach items="${i.attributes.keys()}" var="j"><j:if test="${counter ==
'1'}">
          String ${j}</j:if><j:if test="${counter > '1'}">,
          String ${j}</j:if><j:set var="counter" value='${counter+1}'/></j:forEach >

        )throws ${EXCEPTION} {

          final String METHOD_NAME = "browseFor${i.methodName}";
          final long START_TIME = System.currentTimeMillis();
          ${i.methodName}VO ${i.attributeName}VOArray [] = null;
          Object[] objectArray = null;
          ${PARAM_CLASS.className}DTO ${PARAM_CLASS.attributeName}DTO= new
          ${PARAM_CLASS.className}DTO ();
try{
         <j:if test="${PARAM_CLASS.lengtEntity &lt;= i.attributes.size()}">
          ${PARAM_CLASS.className}DAO ${PARAM_CLASS.attributeName}DAO =
          daoFactory.get${PARAM_CLASS.className}DAO();
          </j:if>
          ${i.methodName}DAO ${i.attributeName}DAO = daoFactory.get${i.methodName}DAO();
          <j:if test="${PARAM_CLASS.lengtEntity &lt; i.attributes.size()}"><j:forEach
          items="${i.attributes.keys()}" var="j">
          ${PARAM_CLASS.className}VO ${j}${i.methodName}VO= new  ${PARAM_CLASS.className}VO
          ();
          <j:forEach items="${i.foreing.keys()}" var="count" indexVar="index">
          <j:if
          test="${j.substring(0,1).toUpperCase().concat(j.substring(1)).equals(count)}">
          ${j}${i.methodName}VO.set${i.foreing.get(count)}(new
          ${i.attributes.get(count.substring(0,1).toLowerCase().concat(count.substring(1))}
          }(${count.substring(0,1).toLowerCase().concat(count.substring(1))}));</j:if>
          </j:forEach >${PARAM_CLASS.attributeName}
          DTO.set${j.substring(0,1).toUpperCase().concat(j.substring(1))}${i.methodName}
          VO(${PARAM_CLASS.attributeName}DAO.findByPrimaryKey(${j}${i.methodName}
          VO));</j:forEach>
          </j:if><j:if test="${PARAM_CLASS.lengtEntity == i.attributes.size()}">
          ${PARAM_CLASS.className}VO ${PARAM_CLASS.attributeName}VO= new
          ${PARAM_CLASS.className}VO (); <j:forEach items="${i.foreing.keys()}" var="count'
          indexVar="index">
          ${PARAM_CLASS.attributeName}VO.set${i.foreing.get(count)}(new
          ${i.attributes.get(count.substring(0,1).toLowerCase().concat(count.substring(1))}
          }(${count.substring(0,1).toLowerCase().concat(count.substring(1))}));
          </j:forEach >
          ${PARAM_CLASS.attributeName}DTO.set${PARAM_CLASS.className}
        VO(${PARAM_CLASS.attributeName}DAO.findByPrimaryKey(${PARAM_CLASS.attributeName}
        VO));</j:if>
          StringBuffer parameters${i.methodName} = new StringBuffer();<j:set var="counter"
          value='0'/> <j:forEach items="${i.parameters.keys()}" var="k" ><j:if
          test="${counter == '0'}">
          parameters${i.methodName}.append("${k}=").append(${i.parameters.get(k)});</j:if>
          <j:if test="${  counter > '0' }">
          parameters${i.methodName}.append(" AND
        ").append("${k}=").append(${i.parameters.get(k)});</j:if><j:set var="counter"
        value='${counter+1}'/></j:forEach >
          objectArray = ${i.attributeName}
          DAO.findByAnyWhere(parameters${i.methodName}.toString()).toArray();
          ${i.attributeName}VOArray = new ${i.methodName}VO[ objectArray.length ];
          for(int i=0; i &lt; objectArray.length; i++){
             ${i.attributeName}VOArray[i] = (${i.methodName}VO) objectArray[ i ];
          }
          ${PARAM_CLASS.attributeName}DTO.set${i.methodName}DTOSet(${i.attributeName}
          VOArray);

    }finally{
         PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
               START_TIME);
    }

  return ${PARAM_CLASS.attributeName}DTO;

}</j:if></j:forEach>
```

Nota: Dentro de la implementación manual realizada para definir los servicios, se detecto que se puede presentar la situación de múltiples referencias a la llave primaria de la entidad, situación que se debe tener en cuenta para poder entregarle un objeto completo al usuario.

- *Update* (Actualización de la tabla entidad y todos sus hijos)

  Este método permite al usuario que a partir de un Objeto DTO (Data Transfer Object) se realice la actualización de los valores teniendo en cuenta las prioridades (relaciones padre-hijo de base de datos).

  Plantilla Jelly

```
public void deleteByPrimary(<j:forEach items="${PARAM_CLASS.attributesCEVOSet}" var="i">
<j:if test="${!i.isParent() &amp;amp;&amp;amp; !i.isChildren() &amp;amp;&amp;amp; !i.isDomain()}"><j:set
var="counter" value='1'/><j:forEach items="${i.attributes.keys()}" var="j"><j:if
test="${counter == '1'}">
         String ${j}</j:if><j:if test="${counter > '1'}">,
         String ${j}</j:if><j:set var="counter" value='${counter+1}'/></j:forEach ></j:i
         </j:forEach >
        ) throws ${EXCEPTION} {

        final String METHOD_NAME = "deleteByPrimary";
        final long START_TIME = System.currentTimeMillis();
try{
        ${PARAM_CLASS.className}DAO ${PARAM_CLASS.attributeName}DAO =
        daoFactory.get${PARAM_CLASS.className}DAO();

        <j:forEach items="${PARAM_CLASS.attributesCEVOSet}" var="i"><j:if
        test="${!i.isParent() &amp;amp;&amp;amp; !i.isChildren() &amp;amp;&amp;amp; !i.isDomain()}">
        StringBuffer parameters${PARAM_CLASS.className} = new StringBuffer();<j:set
        var="count" value='0'/><j:forEach items="${i.parameters.keys()}" var="k" ><j:if
        test="${count == '0'}">
        parameters${PARAM_CLASS.className}.append(
        "${k}=").append(${i.parameters.get(k)});</j:if><j:if test="${  count > '0' }">
        parameters${PARAM_CLASS.className}.append(" AND " ).append(
        "${k}=").append(${i.parameters.get(k)});</j:if><j:set var="count"
        value='${count+1}'/></j:forEach ></j:if></j:forEach ><j:forEach
        items="${PARAM_CLASS.attributesCEVOSet}" var="i"><j:if test="${i.isChildren()}">
        ${i.methodName}DAO ${i.attributeName}DAO = daoFactory.get${i.methodName}
        DAO();</j:if></j:forEach ><j:forEach items="${PARAM_CLASS.attributesCEVOSet}"
        var="i"><j:if test="${i.isChildren()}">
        ${i.attributeName}
        DAO.deleteByAnyWhere(parameters${PARAM_CLASS.className}.toString());</j:if>
        </j:forEach >
        ${PARAM_CLASS.attributeName}
        DAO.deleteByAnyWhere(parameters${PARAM_CLASS.className}.toString());

    }finally{
         PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
                START_TIME);
    }
}
```

- *Delete* (Eliminación de la tabla entidad y todos sus hijos)

  Este método permite al usuario que a partir de una llave primaria suministrada por el usuario se haga la eliminación primero de los hijos que estén relacionados y luego el registro en tabla entidad.

Plantilla Jelly

```
public void update(${PARAM_CLASS.className}DTO ${PARAM_CLASS.attributeName}DTO,boolean
parcial ) throws ${EXCEPTION} {
              Assert.isNotNull(${PARAM_CLASS.attributeName}DTO , "${PARAM_CLASS.className}
              DTO  specified is null!!");
      final String METHOD_NAME = "update";
      final long START_TIME = System.currentTimeMillis();
try{
      ${PARAM_CLASS.className}DAO ${PARAM_CLASS.attributeName}DAO =
daoFactory.get${PARAM_CLASS.className}DAO();<j:forEach
items="${PARAM_CLASS.attributesCEVOSet}" var="i"><j:if test="${i.isChildren()}">
      ${i.methodName}DAO ${i.attributeName}DAO = daoFactory.get${i.methodName}DAO();
      ${i.methodName}VO[] ${i.attributeName}VO= ${PARAM_CLASS.attributeName}
      DTO.get${i.methodName}DTOSet();
      for(int i=0;i&lt;${i.attributeName}VO.length;i++)
      {
               ${i.attributeName}DAO.update(${i.attributeName}VO[i],parcial);
      }</j:if></j:forEach >
      ${PARAM_CLASS.attributeName}DAO.update(${PARAM_CLASS.attributeName}
      DTO.get${PARAM_CLASS.className}VO(),parcial);
   }finally{
         PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
               START_TIME);
   }
}
```

7.3.4 Generador DTO. Este generador permite crear físicamente el archivo basado en el patrón Data Transfer Object. La clase generada sirve como medio de almacenamiento, ya que por los principios teóricos del patrón permite encapsular objetos complejos, que para este proyecto son de gran importancia ya que cada objeto complejo representara una tabla de la entidad de negocio a construir.

Esta clase es de almacenamiento, por esta razón esta compuesta en su gran mayoría por métodos get y set; adicional hay un método clone que consiste retornar un objeto tipo Object el cual podrá ser referenciado por medio de un casting de Java. Debido a su tamaño la plantilla de generación no se presentara en este documento, Se adjuntara como un archivo aparte (templateDTOCE) en la entrega del documento.

7.3.5 Generador Domain Object. Este generador permite crear físicamente el archivo basado en el patrón Domain Object La plantilla de generación contendrá la información necesaria para crear los métodos que permiten acceder a los registros de las tablas dominios como objetos.

Los servicios presentados a continuación fueron discutidos al interior de la empresa y hacen parte de los servicios entregados por estas clases .A continuación se presentaran los métodos con sus respectivos nombres y segmento Jelly de construcción:

- *GetAll* (Devuelve todos los objetos de la tabla dominio)

  Este método permitirá al usuario tener todos los valores (objetos) que equivalen a cada registro (Data) de la base de datos de una tabla tipo.

- *Find* (Busca la descripción de un solo registro)
  Busca la descripción de un solo objeto de la clase que equivale a un registro físico de la tabla tipo

- *FindSet* (Busca la descripción de uno o mas objetos)

  Busca la descripción de uno o mas objetos a partir de un vector suministrado por el usuario.

Plantilla Jelly

```
<?xml version="1.0" encoding="UTF-8"?>
<j:jelly xmlns:j="jelly:core" xmlns:my="actionFramework">
<j:whitespace>


package ${DOMAIN_PACKAGE};

import java.math.BigDecimal;


import java.util.Hashtable;


/**
 *
 * Implementa el patron Domain Entity el cual extrae los registros
 * propios de la data
 *
 */


public class ${PARAM_CLASS.methodName}Domain
{
        <j:forEach items="${PARAM_CLASS.attributes.keys()}" var="i" >
        <j:if test="${!PARAM_CLASS.isNativo()}">
         public static ${PARAM_CLASS.attributeType} ${PARAM_CLASS.attributes.get(i)} =new
         ${PARAM_CLASS.attributeType}("${i}") ;
        </j:if >
        <j:if test="${PARAM_CLASS.isNativo()}">
         public static ${PARAM_CLASS.attributeType} ${PARAM_CLASS.attributes.get(i)} = ${i} ;
        </j:if >
        </j:forEach >
        private static Hashtable ${PARAM_CLASS.attributeName};

        public  ${PARAM_CLASS.methodName}Domain()
        {
           ${PARAM_CLASS.attributeName} = new Hashtable();
           <j:forEach items="${PARAM_CLASS.attributes.keys()}" var="i" >

                        ${PARAM_CLASS.attributeName}.put(${PARAM_CLASS.attributes.get(i)},"${PAR
                        CLASS.attributes.get(i)}");

           </j:forEach >
        }
        public String find${PARAM_CLASS.methodName}(${PARAM_CLASS.attributeType}  value)
        {

                return (String)${PARAM_CLASS.attributeName}.get(value);

        }

        public String[] find${PARAM_CLASS.methodName}Set(${PARAM_CLASS.attributeType}[] value)
        {
                        String retorno[] = new String [ value.length];
                        for(int i=0;i &lt; retorno.length; i++)
                        {
                                retorno[i]=(String)${PARAM_CLASS.attributeName}.get(value[i]);
                        }
                        return retorno;
        }

        public Hashtable getAll()
        {
                return ${PARAM_CLASS.attributeName};
        }
}


</j:whitespace>
</j:jelly>
```

7.4 TECNOLOGÍAS DE DESARROLLO

En este capitulo se presentan las herramientas tecnológicas utilizadas para construir la interfaz grafica de usuario final del componente de generación de código.

Este proyecto tiene varios componentes con los que interactúa el usuario, los cuales son desarrollados dentro del IDE Eclipse. Entre estos se encuentran dos muy importantes en el manejo de la aplicación que son los Wizard de la aplicación y el diagrama que permite visualizar la entidad de negocio construida por el sistema, basado en el patrón Composite Entity.

7.4.1 Eclipse. Eclipse es un IDE multiplataforma, de licencia libre para crear aplicaciones clientes de cualquier tipo. La primera y más importante aplicación que ha sido realizada con este entorno, es el afamado IDE Java llamado *Java Development Toolkit* (JDT) y el compilador incluido en Eclipse, que se usaron para desarrollar el propio Eclipse.

El proyecto Eclipse se divide en tres subproyectos:

- El Core de la aplicación, que incluye el subsistema de ayuda, la plataforma para trabajo colaborativo, el Workbench (construido sobre SWT y Jface) y el Workspace para gestionar proyectos.

- *Plugin Development Environment* (PDE), que proporciona las herramientas para el desarrollo de nuevos módulos.

Eclipse fue creado originalmente por IBM. Ahora lo desarrolla la fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

El Entorno integrado de desarrollo (IDE) de Eclipse emplea módulos (en inglés *plug-in*) para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. El mecanismo de módulos permite que el entorno de desarrollo soporte otros lenguajes además de Java.

Los componentes gráficos (widget) de Eclipse están basados en un juego de herramientas de tercera generación para Java de IBM llamado SWT, que mejora los de primera y segunda generación de Sun (AWT y Swing, respectivamente). La interfaz de usuario de Eclipse cuenta con una capa intermedia de interfaz gráfica (GUI) llamada Jface, lo que simplifica la creación de aplicaciones basadas en SWT.

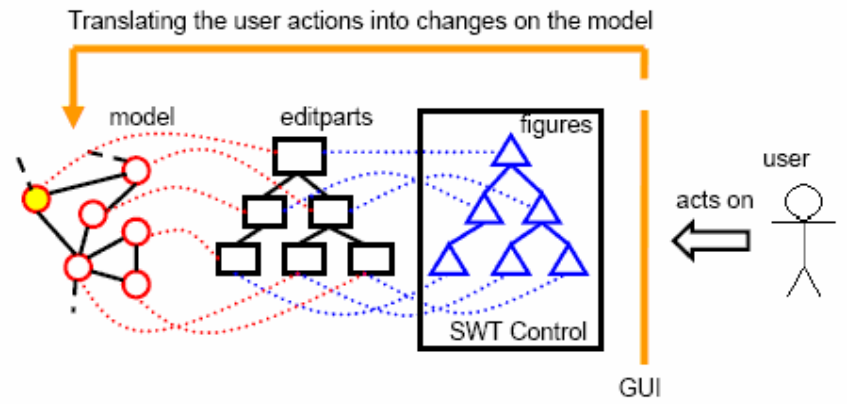
En cuanto a las aplicaciones clientes, Eclipse provee Frameworks muy ricos para el desarrollo de aplicaciones gráficas, definición y manipulación de

modelos de software, aplicaciones web, etc. En este proyecto es de gran importancia su utilización, ya que permite tener una aplicación en un ambiente amigable y de fácil utilización.

Para el generador de los Composite Entity Eclipse es el IDE de desarrollo y finalmente también es el ambiente de ejecución, por ser multiplataforma y libre permite que se desarrollen nuevas herramientas sobre el, sin pagos de licenciamiento y portables.

7.4.2 GEF. GEF (Graphical Editing Framework) permite tener un ambiente grafico dentro de un editor dentro de Eclipse. Este plugin de Eclipse es de licencia GNU(software libre).

Figura 11. Diagrama MVC GEF



Este plugin esta basado en un MVC (Modelo Vista Control) en donde su estructura esta dividida así:

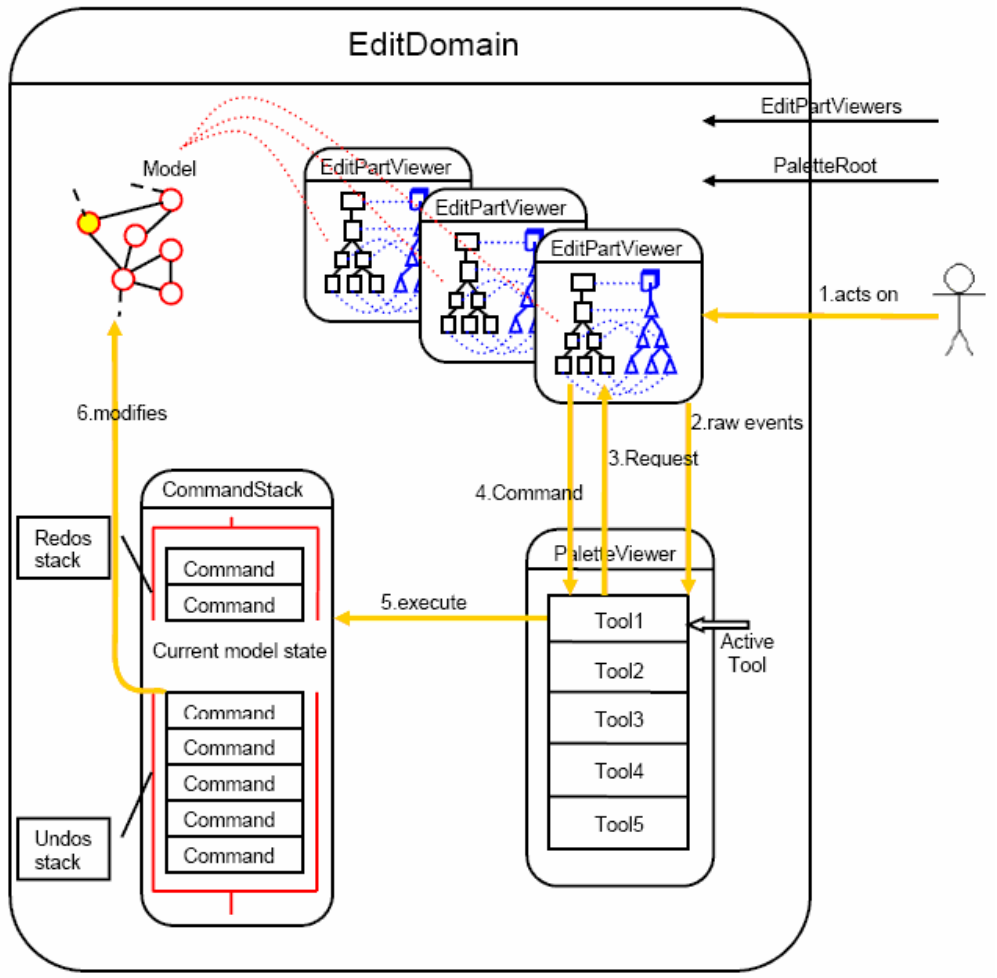
- **Figures**: Es básicamente la interfaz de usuario, en este caso son las figuras que representan a cada tabla de la base de datos dentro del diagrama.

- **EditParts**: Esta capa puede ser llamada dentro de un MVC como el controlador, ya que responde a las acciones del usuario e invoca cambios en el modelo. Es decir en esta capa es donde el sistema escucha los eventos que realice el usuario dentro del editor y los envía a procesar dependiendo de la acción que haya echo el usuario. En esta capa esta inmersas las clases de edición.

- **Model**: Esta es la representación específica del dominio de la información sobre la cual funciona la aplicación. Este modelo dentro del proyecto permite definir la jerarquía existente entre los diferentes objetos, es decir aquí se podrá definir que el usuario pueda crear un nuevo esquema, al cual se agregaran los diferentes tipos de entidad (tabla padre, tabla hija, tabla domino, tabla entidad), los cuales se pueden unir mediante relaciones.

GEF provee varias herramientas adicionales a las figuras, como lo son la paleta para agregar componentes y el control de menús contextuales. En el siguiente diagrama podemos ver una descripción completa de todos los componentes con los cuales cuenta este plugin:

Figura 12. Diagrama de interacción GEF



GEF define algunos principios muy útiles para el manejo de de interfaces de usuario. Para este proyecto estos principios son de gran importancia, ya que definen muy bien los procesos para manejar las acciones del usuario con el sistema y del sistema con el usuario, es decir una interacción bidireccional. A continuación se describirá cada uno de estos principios:

- El usuario actúa en el diagrama con el ratón y el teclado.

- Las acciones del usuario en el diagrama son instancias EditPartViewer o capturadotes de acciones dentro del diagrama. El EditPartViewer transmite todos los eventos al controlador.

- El controlador interpreta estos eventos para construir procesos. Los procesos son objetos usados por GEF para especificar qué operaciones tienen que ser ejecutadas en el modelo.
- El controlador envía los procesos a clases de edición del modelo con tres objetivos principales: uno para conseguir los comandos que ponen los procesos en ejecución y dos métodos secundarios que sirven para retroalimentar al objeto principal. Los comandos son objetos que permiten ingresar las modificaciones en el modelo. Las políticas de edición o EditPolicies son objetos capaces de construir comandos de retorno de los procesos.
- Después de obtener un comando de una clase de edición, el controlador puede ejecutarlo como un CommandStack.
- El CommandStack ejecuta el comando y modifica el modelo, luego se actualiza la GUI.

A continuación se mostraran algunos términos muy útiles del Framework GEF:

PalleteViewer: Esta herramienta permite agregar nuevos elementos al diagrama. Para este proyecto es muy útil ya que permitirá ingresar los diferentes tipos de entidades y además ingresar una nueva relación entre las tablas.

Command Stack: Son los eventos que están definidos en la capa de control, los cuales dependiendo de la acción ejecutada en presentación, definen que operación realizar sobre el modelo de datos.

Command: Son objetos implementados en el modelo para que este sea editado cuando ocurra un evento en presentación. Los métodos definidos para estos objetos son:

- void execute (): Es el método que se llama cuando ocurre un evento dentro del diagrama.
- void undo (): Este método es llamado cuando el usuario quiere devolverse un instante antes a la modificación realizada en el diagrama.

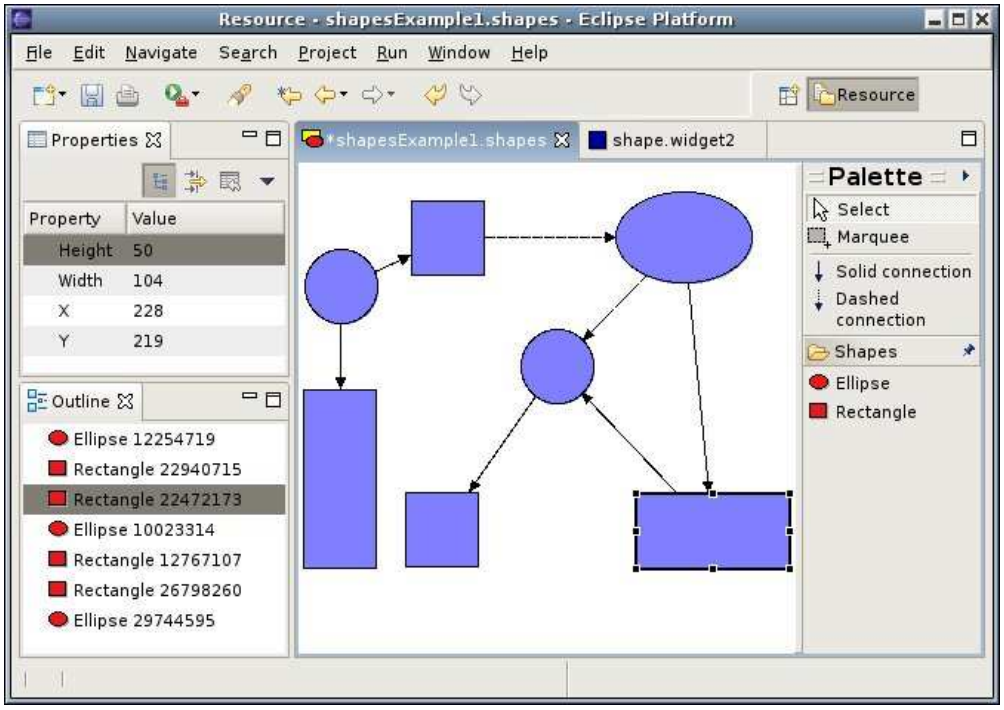
- void redo (): Este método es llamado cuando el usuario quiere ir una instancia adelante en el diagrama. Por ejemplo cuando el usuario borra una tabla del diagrama el podría ir adelante y la tabla aparecería otra vez en el diagrama.
- boolean canExecute (): Este método es llamado antes de la ejecución y define si un comando puede ser ejecutado o no.

EditPolices: Son interfaces que proveen métodos globales que cada editor debe tener, es decir es una abstracción de las operaciones permitidas dentro del diagrama.

El resultado final una vez se utilice GEF dentro de un proyecto es un editor dentro del cual el usuario puede incluir figuras en 2 dimensiones , en el cual se

pueden realizar modificaciones que afectaran directamente el modelo de datos y un archivo físico el cual solo podrá visualizarse si se tienen este plugin en el ambiente de ejecución . Un ejemplo de esto se puede ver en la siguiente figura:

Figura 13. Ejemplo aplicación construida con GEF



Como podemos ver cada figura de las que aparecen en ese diagrama puede ser una representación de una tabla, la cual tiene unas propiedades específicas (Definidas en el modelo) y una paleta de herramientas la cual permite ingresar un nuevo componente al diagrama. Por esta razón GEF es la herramienta escogida para implementar el diagrama de Composite Entity.

GEF permite crear modelos gráficos muy fácilmente, para este proyecto su utilización es de suma importancia ya que permitirá crear un diagrama que represente una entidad de negocio compuesta a partir de los parámetros (esquema de base de datos, nombre de la tabla entidad) suministrados por el usuario.

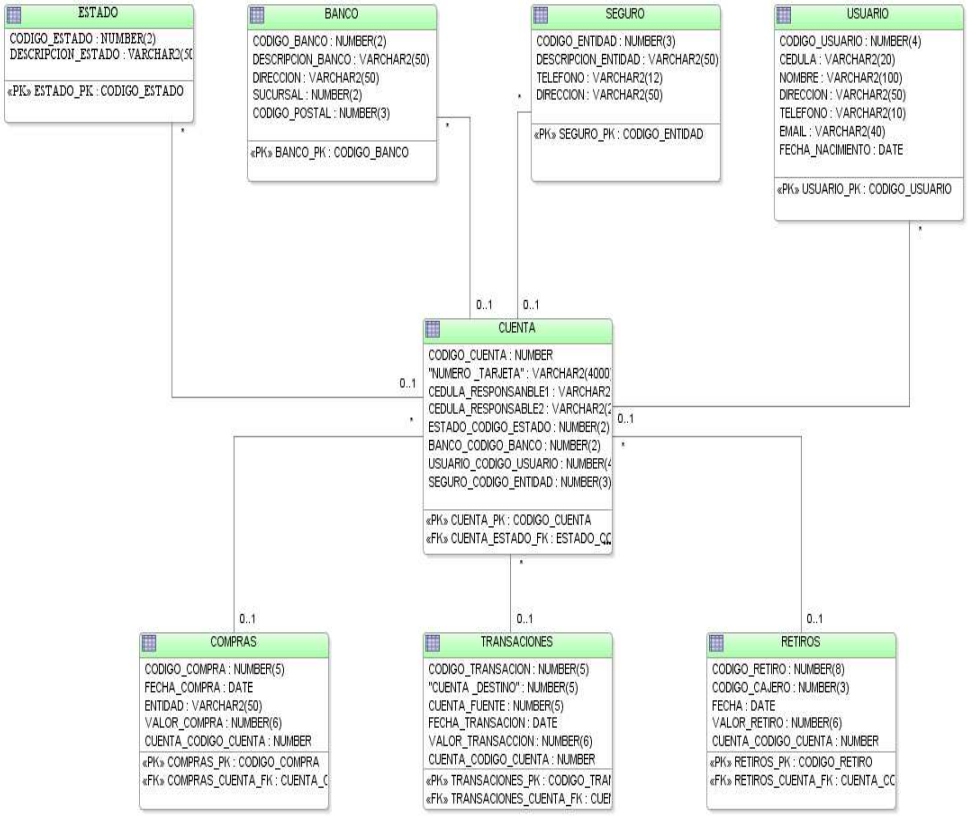
8. PRUEBA DE CONCEPTO GENERADOR

En este capitulo se mostrara el funcionamiento del producto final de este proyecto, incluyendo interacción con los Wizard y el código generado. Se seguirá el proceso paso a paso para la generación de clases java, basado en el patrón Composite Entity y todo su ambiente de ejecución.

8.1. MODELO DE DATOS DE PRUEBA

El Componente de Generación de Composite Entity debe ser probado con un modelo de datos que permita ver su efectividad, es decir una entidad de negocio que este compuesta por una serie de tablas que sean objetos padres u objetos entidad de una tabla del modelo.

Figura 14. MER prueba de concepto componente de generación Composite Entity



Nota: El script de creación - anexo 1.0

El objetivo principal de la creación de este modelo de datos es mostrar una generación de código de una entidad de negocio que involucre tablas relacionadas entre si, ya sea como tablas padres, tablas dominio, tablas hijas o tablas entidad.

8.2. COMPONENTE DE GENERACION DE DIAGRAMA

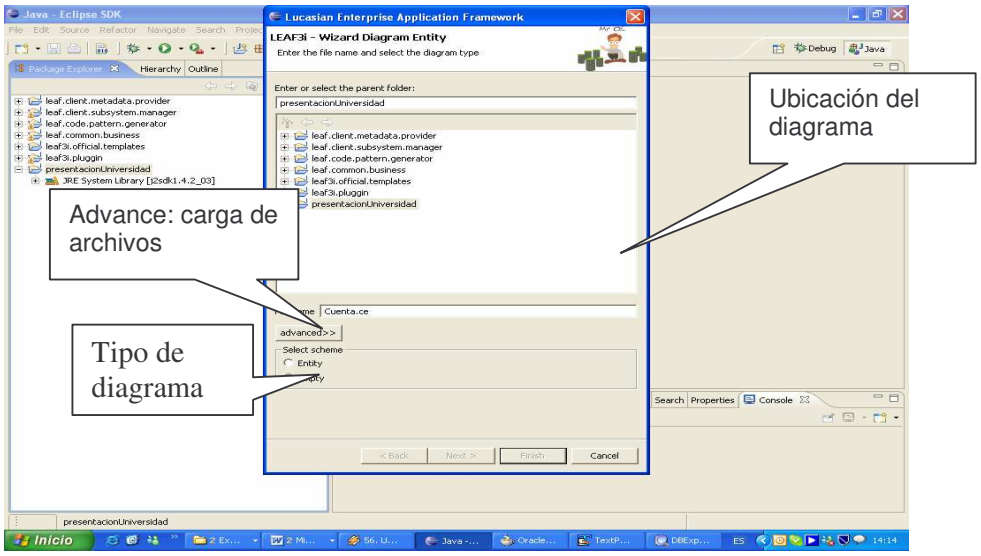
El componente de generación de diagrama de Composite Entity es un Wizard, el cual permite al usuario la creación de un diagrama que representa una entidad de negocio. Esta entidad de negocio representada en el diagrama es la que permite hacer la generación de código.

8.2.1. Generación de diagrama. La creación del diagrama puede hacerse de dos formas diferentes una es a partir de una entidad seleccionada por el usuario o la otra es crear el diagrama desde cero escogiendo la opción de diagrama vació.

En la siguiente figura podemos identificar 3 diferentes componentes que deben ser tenidos en cuenta al momento de configurar el ambiente de generación del diagrama.

Este paso comprende el caso de uso CU *Crear Diagrama de Generación*

Figura 15. Paso 1 generación de diagrama Composite Entity



- Componente de Ubicación física del Diagrama: Este componente permite ubicar el archivo en uno de los proyectos que están en el Workspace activo de eclipse.

- Advance: Este componente permite cargar un archivo de tipo ce desde una ruta física del sistema y copiarlo dentro del proyecto escogido en el Componente de Ubicación del Diagrama.

- Select Schema: Este componente permite seleccionar si, el diagrama se crea a partir de una entidad de la base de datos o si se crea vació, dependiendo de la opción escogida por el usuario se habilita o deshabilita el botón finísh.

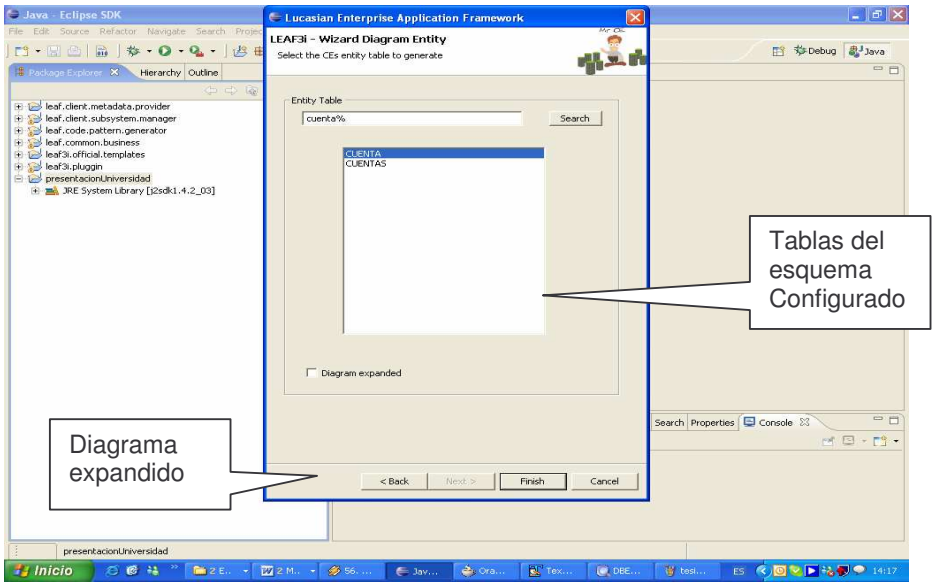
8.2.2. Generación de diagrama a partir de una entidad. En este paso se debe escoger la tabla de base de datos sobre la cual se va a realizar la generación de código basada en el patrón Composite Entity, en caso de que la opción escogida en el paso 1 haya sido entidad. La ejecución de este paso consume caso de uso *CU Diagrama Entidad de negocio.*

Sobre esta página del Wizard vale la pena destacar, que sobre la lista de selección se cargaran las tablas del esquema de base de datos seleccionado por el usuario en el Wizard de configuración de Leaf3i. A partir de esta tabla se realizara un proceso de búsqueda de tablas padres y tablas hijas para esa entidad, esta búsqueda se hace por medio de llaves primarias y foráneas.

Adicional a esto el diagrama generado se puede crear en modo expandido o comprimido, por defecto esta en modo comprimido.

En el momento en que se pulse el botón finísh, se ejecuta el CU Extracción de metadata y se crea el diagrama de generación.

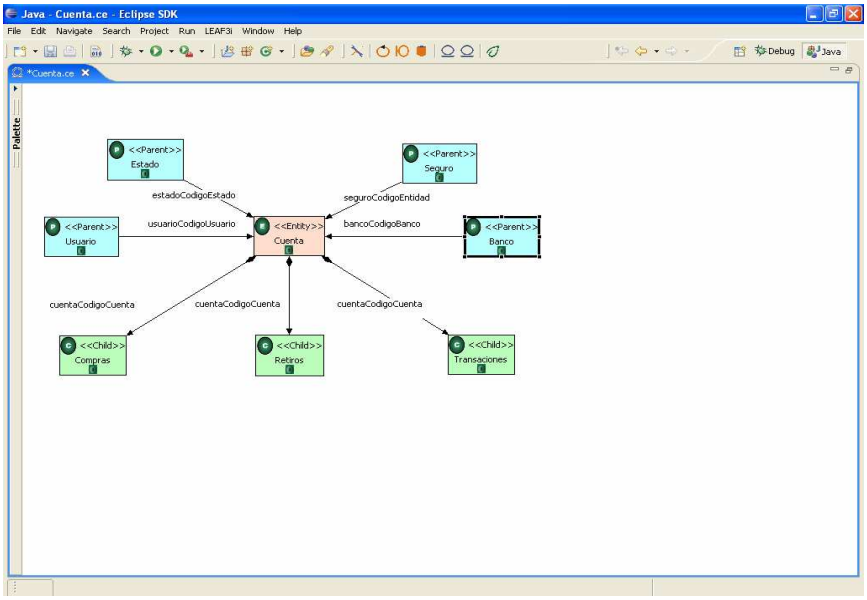
Figura 16. Paso 2 generación de diagrama Composite Entity



El diagrama mostrado en la figura 17 es el resultado final de la generación de un diagrama de Composite Entity.

- Diagrama en modo comprimido (default)

Figura 17. Diagrama de Composite Entity modo comprimido



- Diagrama en modo Expandido

Figura 18. Diagrama de Composite Entity modo expandido, entidad base y tablas padres

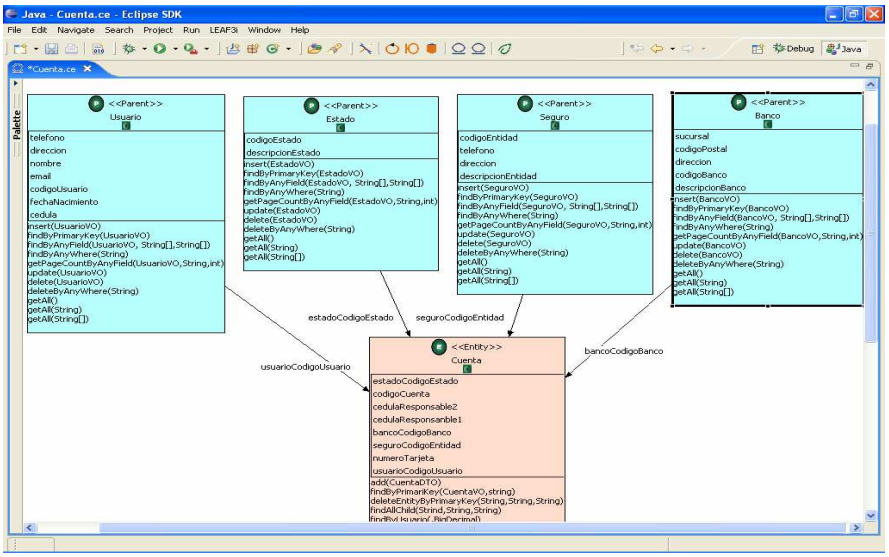
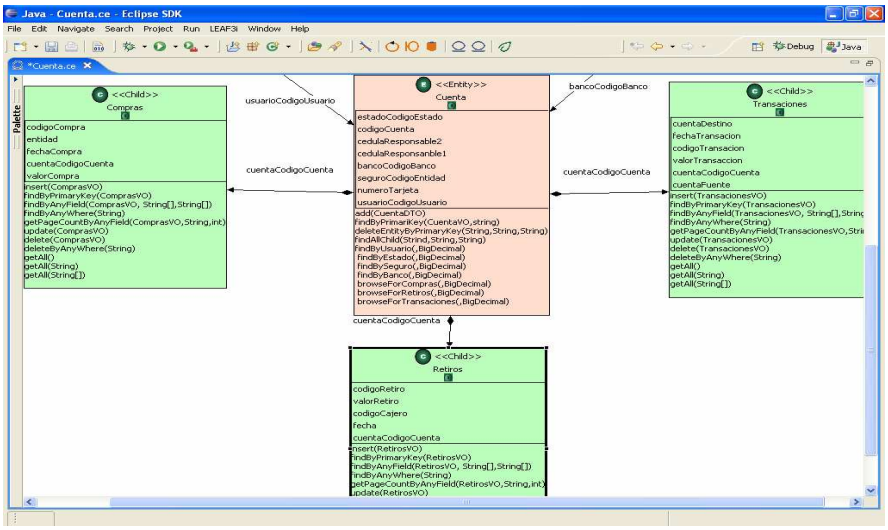


Figura 19. Diagrama de Composite Entity modo expandido, entidad base y tablas hijas



Como podemos ver en las figuras 18 y 19 las tablas generadas dentro del diagrama son de diferentes colores y tienen etiquetas diferentes. Adicional a las tablas, se muestran unas líneas que relacionan entre si las entidades, mostrando sobre que campos es interacción de las tablas. Para aclarar un poco las convenciones del diagrama se puede ver la siguiente descripción:

- Tabla Padre: Tabla de base de datos que tienen una relación maestro detalle hacia la tabla entidad. La relación entre estas tablas es que la llave primaria equivale a un campo de la tabla entidad. El equivalente a objeto luego de hacer el mapeo objeto relacional es una clase DAO. Etiqueta: <<Parent>>.

- Tabla hija: Tabla de base de datos que tienen una relación maestro detalle a partir de la tabla entidad. La relación entre estas dos tablas es que la llave primaria de la entidad equivale a un campo dentro de la tabla. El equivalente a objeto luego de hacer el mapeo objeto relacional es una clase DAO. Etiqueta: <<Child>>.

- Tabla Entidad: Tabla de base de datos que es la base de una entidad de negocio compuesta. Sobre esta entidad de la base de datos se buscan las relaciones que sirven para construir el diagrama. El equivalente a objeto luego de hacer el mapeo objeto relacional es una clase CE. Etiqueta: <<Entidad>>.

- Tabla Dominio: Tabla de base de datos que esta compuesta por dos campos (identificador y descripción), sobre la cual se hacen múltiples transacciones por ser una tabla de información. El equivalente a objeto luego de hacer el mapeo objeto relacional es una clase plana de Java. Etiqueta: <<dominio >>.

8.2.3. Personalización del Diagrama. El diagrama de Composite Entity esta orientado a que el usuario pueda realizar las modificaciones necesarias a la entidad de negocio sobre la cual se va a realizar la generación de código.

Una vez generado el diagrama se le pueden ingresar o eliminar componentes. Los componentes que se pueden agregar al diagrama son:

- Relations: Esta opción permite agregar el componente que permite relacionar dos entidades del diagrama entre si, mostrando por que atributo se realiza la relación.

- New Entity: Al escoger esta opción el usuario puede ingresar una tabla entidad, cabe anotar que n un diagrama solo se podrá tener una tabla entidad. Está opción ejecuta el caso de Uso *CU Ingresar entidad.*

- New Parent Entity: Esta opción le permite al usuario ingresar una tabla que sea padre de la entidad, la cual debe estar relacionada lógica o semántica. . Está opción ejecuta el caso de Uso *CU Ingresar entidad Padre.*

- New Child Entity: Esta opción le permite al usuario ingresar una tabla que sea hija de la entidad, la cual debe estar relacionada lógica o semántica. . Está opción ejecuta el caso de Uso *CU Ingresar entidad hija.*

- New Domain Entity: Esta opción le permite al usuario ingresar una tabla que sea hija de la entidad, la cual debe estar relacionada semánticamente con la entidad. Está opción ejecuta el caso de Uso *CU Ingresar entidad dominio.*

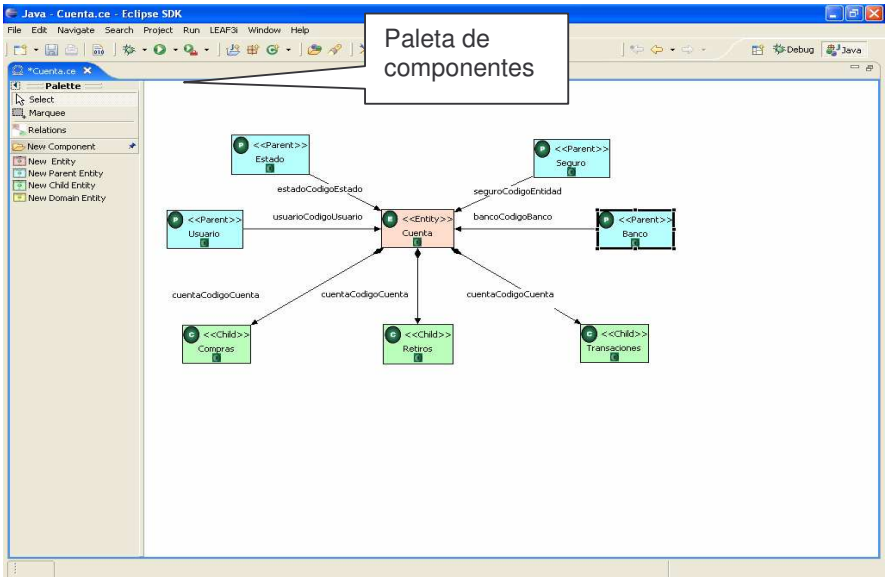
Adicional a estos componentes en esta paleta se ofrecen opciones para interactuar con el diagrama, estas son:

- Select Esta opción permite seleccionar un componente en caso de que se quiera arrastrar o eliminar, es decir realiza las operaciones del puntero de Windows.

- Marquee: Este Opción permite seleccionar uno o mas componentes en caso de quiera arrastrar o eliminar.

En la siguiente figura en la parte superior izquierda se puede visualizar la paleta que tiene asociado el diagrama generado.

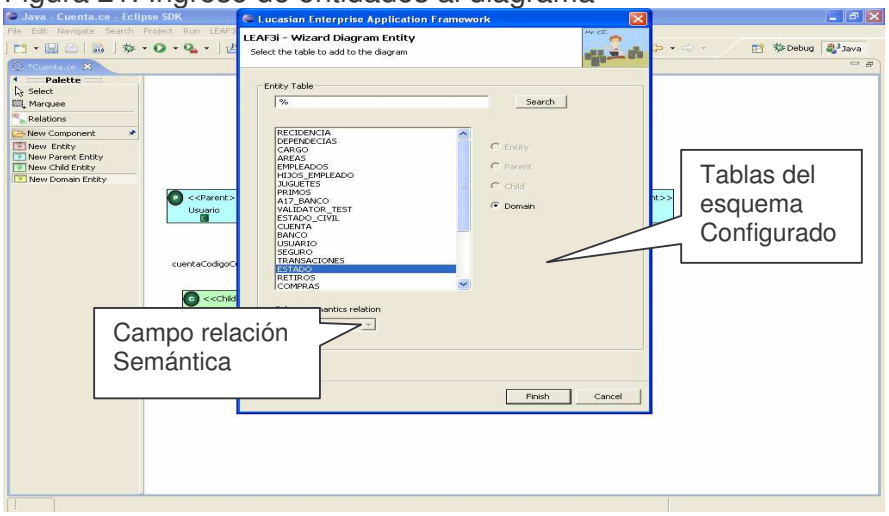
Figura 20, Paleta de componentes



- **Ingresar Entidad**

Cuando se selecciona agregar cualquiera de los componentes de entidad, es decir tabla padre, tabla hija, tabla entidad o tabla dominio el sistema le mostrara el Wizard que se ve en la siguiente figura:

Figura 21. Ingreso de entidades al diagrama



Este Wizard permite al usuario ingresar una tabla del esquema de base de datos que se tenga configurado, dependiendo del tipo de componente que

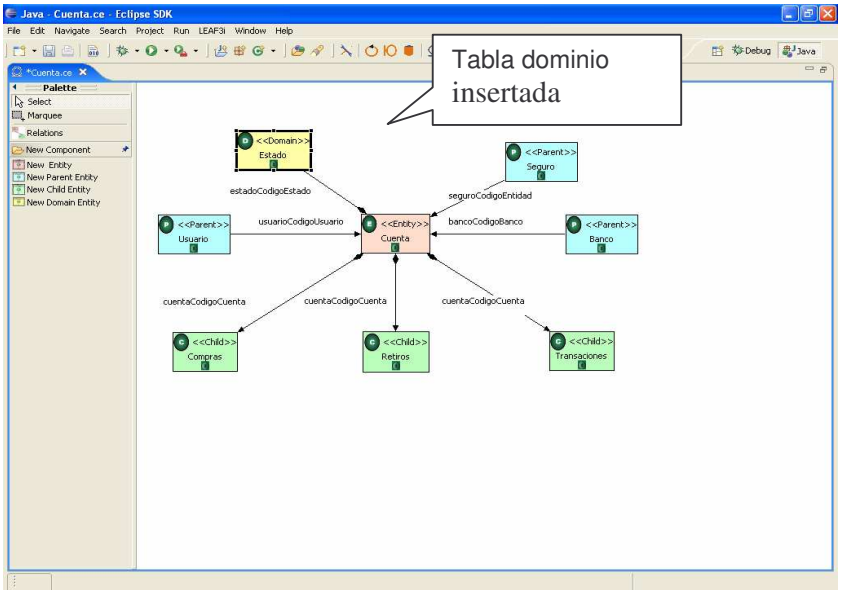
escoja el usuario el sistema internamente ejecutara *CU Ingresar Entidad Padre, CU Ingresar Entidad Hija, CU Ingresar Entidad Dominio, CU Ingresar Relación.*

El componente insertado se relaciona semánticamente con la tabla entidad por la columna escogida en el combo que esta en la parte inferior izquierda. La relación será semántica, ya que todas las relaciones lógicas son detectadas cuando se hace la búsqueda en la base de datos.

Nota: En caso de que el usuario cree el diagrama desde cero, el sistema validara si se puede ingresar una tabla entidad y si la tabla escogida esta relacionada físicamente con la entidad.

Como se puede ver en la siguiente figura, la tabla que se selecciono en la figura 21, se relaciona con la entidad por el campo estadoCodigoEstado escogido en el paso anterior.

Figura 22. Tabla dominio insertada al diagrama

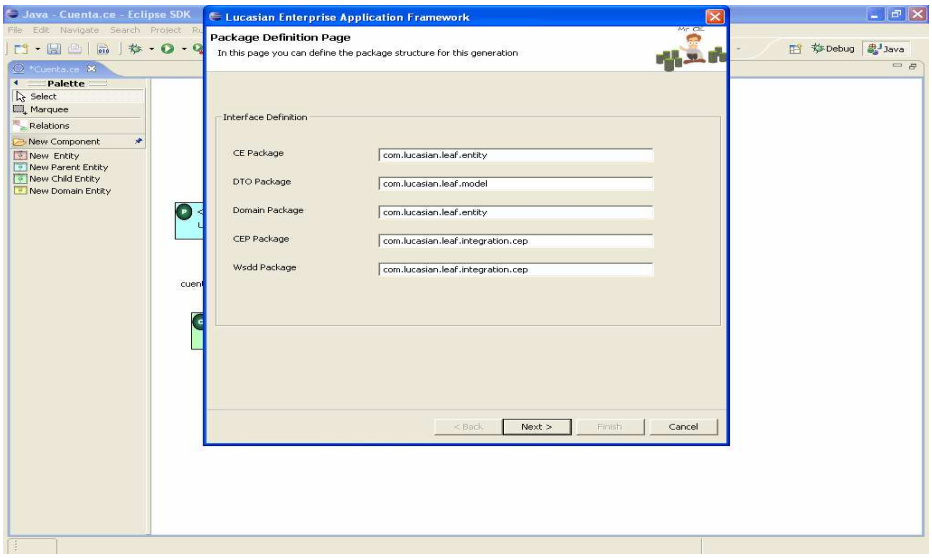


Como se puede ver hasta aquí se ha llevado un proceso sencillo, en el cual se ha mostrado la generación de un diagrama de Composite Entity a partir de una tabla entidad. La utilización de este componente es sencilla y permite visualizar la misma entidad de negocio representada en el MER (Modelo Entidad Relación) inicial, pero ya vista en un diagrama de objetos. Las relaciones entre los objetos es de composición.

8.3 GENERACION DE CÓDIGO DE COMPOSITE ENTITY

Esta es la segunda parte del ambiente grafico de los Composite Entity y consiste en que a partir del diagrama creado se haga una generación de código. En la siguiente figura se muestra el primer paso del Wizard :

Figura 23. Pasó 1 Wizard de Generación Composite Entity



En este primer paso se da la opción al usuario de hacer la configuración de distribución de paquetes de las clases generadas. A continuación se describirá cada una de las opciones:

- CEPackage: Permite configurar en que paquete del proyecto de eclipse utilizado, ubicar la clase de Composite Entity; Por defecto se crea en el paquete base mas la extensión .entity. Debe tenerse en cuenta de que esta clase debe ir dentro del proyecto de eclipse de negocio por lineamientos de arquitectura.

- DTOPackage: Permite configurar la ubicación de la clase Data Transfer Object que es instanciada y utilizada dentro de la clase Composite Entity. Por defecto la ubicación es en el paquete base mas la extensión .model, ya que hace parte del modelo genérico de datos.

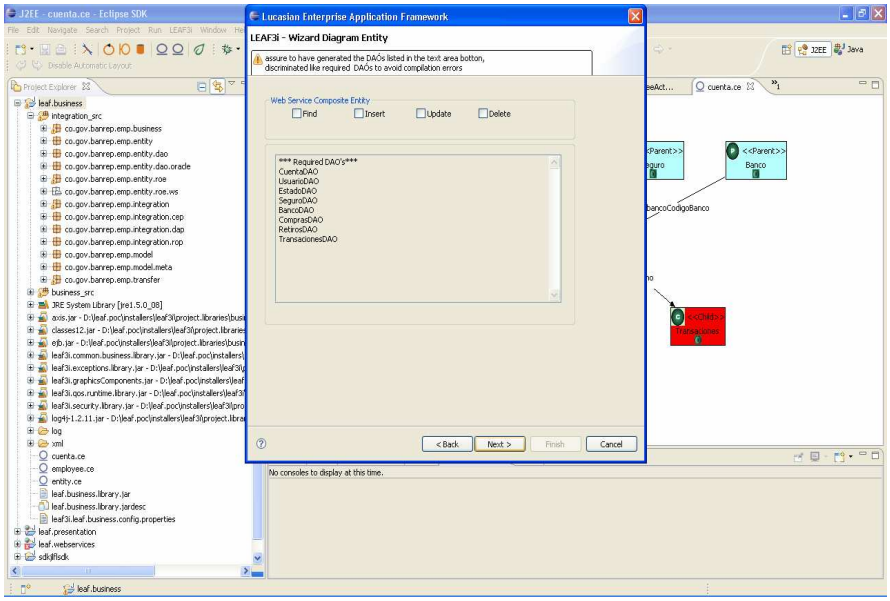
- DomainPackage: Permite configurar la ubicación de las clases Domain Object en caso de que existan, cabe resaltar que estas también serán instanciadas y utilizadas dentro de la clase de Composite Entity. Por defecto esta clase queda ubicada en el mismo paquete .entity.

- CEP Package: Permite configurar la ubicación de clase Composite Entity Proxy, que será utilizada para la integración de los componentes de negocio con la exposición de los métodos de la clase generada de Composite Entity como Web Service. Por defecto queda ubicada en el paquete base mas la extensión integration.cep.

- WSDD Package: Permite configurar la ubicación del descriptor de despliegue de los métodos expuestos como Web Services, es decir estos archivos son los que finalmente permiten que el contenedor de la aplicación ubique los servicios.

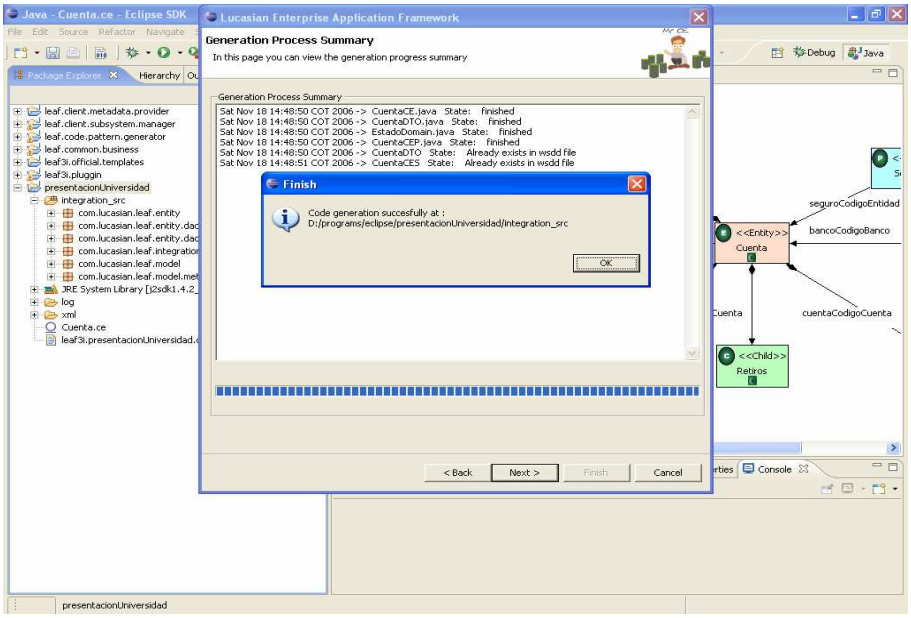
El siguiente paso del Wizard de generación de código de Composite Entity permite configurar que métodos de la clase generada, se desea exponer como Web Services. Como se puede ver en la figura 24 se permite al usuario seleccionar el método buscar, insertar, actualizar Eliminar. Además se pueden ver que DAO's deben haberse generado previamente para que no se presenten problemas de compilación.

Figura 24. Pasó 2 Wizard de generación de código Composite Entity



Finalmente se muestra ha el usuario la barra de progreso que informa del estado en tiempo real de la generación de código.

Figura 25. Pasó 3 Wizard de generación de código Composite Entity



8.4 CLASE GENERADA DE COMPOSITE ENTITY

La clase generada por medio del Generador de Composite Entity permite al usuario tener un mapeo objeto- relacional de una entidad compuesta de base de datos basado en la premisa cero errores de compilación, cero warnings.

La clase Composite Entity generada por medio de este componente pide como requisito que se hayan generado previamente los objetos de integración con la base datos, es decir los DAO's sugeridos en el paso 2 del generador. Ademas también debe generarse la representación objeto de la metadata para cada tabla, los VO.

A continuación se mostrara cada uno de los métodos generados y su funcionamiento:

- Constructor de la clase

  Este constructor de acceso público recibe la conexión de una fábrica de conexiones, y lo que permite es que se puedan hacer operaciones de la base de datos en los métodos de la clase una vez sea esta instanciada. La factoría de conexiones hace parte de Leaf y ya esta construida por esta razón no será explicada en este documento.

```
/**
  * Constructor de la clase CuentaCE
  *
  */
public CuentaCE(Connection connection) {
    final String METHOD_NAME = "Constructor";

    try {
        this.daoFactory = DAOFactory.getDAOFactory(connection);
    } catch (Exception e) {
        LogWriter.writeErrorLog(LogWriter.ERROR, e.getMessage().trim(),
            this.getClass().getName(), METHOD_NAME);
    }
}
```

- Método add

  Este método es de acceso público y recibe como parámetro un DTO construido por el usuario. Para el ejemplo del modelo de datos Cuenta, como se puede ver en la anterior figura de la clase generada, la inserción se hace basado en la prioridad establecida por las reglas de integridad referencial, es decir, primero se ingresa el objeto completo de la tabla entidad, Cuenta, y luego se hace la validación de nulabilidad por cada uno de los objetos hijos y se insertan.

```
    /**
    * Inserta un nuevo registros en la tabla Cuenta con los valores
    * de las propiedades del objeto  <code>Cuenta DTO</code>. y sus
    * y sus respectivos hijos
    *
    * Cuando el log de QoS es activado se puede referir a este metodo
    * mediante la llave <i>add</i>.
    *
    * @param CuentaDTO        Instancia con los valores almacenar.
    *
    */
    public void add(CuentaDTO cuentaDTO)
        throws com.lucasian.leaf.common.SystemException {
        Assert.isNotNull(cuentaDTO, "CuentaDTO  specified is null!!");

        final String METHOD_NAME = "add";
        final long START_TIME = System.currentTimeMillis();

        try {
            CuentaDAO cuentaDAO = daoFactory.getCuentaDAO();
            ComprasDAO comprasDAO = daoFactory.getComprasDAO();
            RetirosDAO retirosDAO = daoFactory.getRetirosDAO();
            TransacionesDAO transacionesDAO = daoFactory.getTransacionesDAO();
            cuentaDAO.insert(cuentaDTO.getCuentaVO());

            if (cuentaDTO.getComprasDTOSet() != null) {
                comprasDAO.insert(cuentaDTO.getComprasDTOSet(), 0);
            }

            if (cuentaDTO.getRetirosDTOSet() != null) {
                retirosDAO.insert(cuentaDTO.getRetirosDTOSet(), 0);
            }

            if (cuentaDTO.getTransacionesDTOSet() != null) {
                transacionesDAO.insert(cuentaDTO.getTransacionesDTOSet(), 0);
            }
        } finally {
            PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
                START_TIME);
        }
    }
```

- Metodo FindByPrimaryKey

  Este método permite realizar una búsqueda sobre todas las tablas padres de la tabla base. Como se puede ver en el código dentro de los valores que trae el VO que representa la tabla base del esquema, Cuenta, se encuentra un valor de referencia hacia cada una de las tablas padres, entre las que se encuentran Usuario, Estado, Seguro y Banco.

  El VO pasado como parámetro debe contener los valores de llave primaria o de llave semántica para cada una de las tablas que son referenciadas como padres, es decir, con estos valores es que finalmente se envía a hacer la consulta en la base de datos a través del DAO. Una vez se hace la consulta en la base de datos por medio del método invocado en los DAO's, se retorna un VO del mismo tipo que el enviado pero que contiene todos los campos llenos, este finalmente se guarda en el objeto de almacenamiento DTO asociado al Composite Entity Cuenta.

  El DTO construido a lo largo del método es retornado con los valores correspondientes a la tabla base Cuenta y de cada uno de sus padres Usuario, Estado, Seguro y Banco.

```
/**
* Reliza la busqueda de las(los)  Cuenta con los valores
* de las propiedades del objeto  <code>CuentaVO</code>. y
* los correspondientes valores en los padres
*
* Cuando el log de QoS es activado se puede referir a este metodo
* mediante la llave <i>findByPrimary</i>.
*
* @param CuentaVO        Instancia con los valores de la busqueda.
* @return                Instancia de la clase <code>CuentaDTO</code>
*
*/
public CuentaDTO findByPrimary(CuentaVO object)
    throws com.lucasian.leaf.common.SystemException {
    final String METHOD_NAME = "findByPrimary";
    final long START_TIME = System.currentTimeMillis();
    CuentaDTO cuentaDTO = new CuentaDTO();

    try {
        CuentaDAO cuentaDAO = daoFactory.getCuentaDAO();
        CuentaVO cuentaVO = cuentaDAO.findByPrimaryKey(object);

        cuentaDTO.setCuentaVO(cuentaVO);

        UsuarioDAO usuarioDAO = daoFactory.getUsuarioDAO();
        UsuarioVO usuarioVO = new UsuarioVO();
        usuarioVO.setCodigoUsuario(cuentaVO.getUsuarioCodigoUsuario());
        cuentaDTO.setUsuarioVO(usuarioDAO.findByPrimaryKey(usuarioVO));

        EstadoDAO estadoDAO = daoFactory.getEstadoDAO();
        EstadoVO estadoVO = new EstadoVO();
        estadoVO.setCodigoEstado(cuentaVO.getEstadoCodigoEstado());
        cuentaDTO.setEstadoVO(estadoDAO.findByPrimaryKey(estadoVO));

        SeguroDAO seguroDAO = daoFactory.getSeguroDAO();
        SeguroVO seguroVO = new SeguroVO();
        seguroVO.setCodigoEntidad(cuentaVO.getSeguroCodigoEntidad());
        cuentaDTO.setSeguroVO(seguroDAO.findByPrimaryKey(seguroVO));

        BancoDAO bancoDAO = daoFactory.getBancoDAO();
        BancoVO bancoVO = new BancoVO();
        bancoVO.setCodigoBanco(cuentaVO.getBancoCodigoBanco());
        cuentaDTO.setBancoVO(bancoDAO.findByPrimaryKey(bancoVO));
    } finally {
        PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
            START_TIME);
    }

    return cuentaDTO;
}
```

- Método findBypadre

  El método findBypadre permite que por cada entidad padre que haya dentro del modelo se exponga un método de búsqueda. Este método recibe como parámetro la llave primaria de la tabla padre, permite tener en un mismo DTO el valor de un objeto genérico en el caso ejemplo usuario y de la entidad dependiente Cuenta. La información recolectada se retorna en un objeto Cuenta DTO, que representa la composición de objetos de la entidad compuesta de base de datos

```
     * Cuando el log de Qo8 es activado se puede referir a este metodo
     * mediante la llave <i>findByForeingUsuario</i>.

     *  @param          usuarioCodigoUsuario  Parametro que conforma la llave primari
de la tabla USUARIO
     *  @return               Instancia de la clase <code>CuentaDTO</code>

    */
    public CuentaDTO findByForeingUsuario(String usuarioCodigoUsuario)
        throws com.lucasian.leaf.common.SystemException {
        final String METHOD_NAME = "findByForeingUsuario";
        final long START_TIME = System.currentTimeMillis();
        CuentaVO[] cuentaVOArray = null;
        Object[] objectArray = null;
        CuentaDTO cuentaDTO = new CuentaDTO();

        try {
            CuentaDAO cuentaDAO = daoFactory.getCuentaDAO();
            UsuarioDAO usuarioDAO = daoFactory.getUsuarioDAO();

            UsuarioVO usuarioVO = new UsuarioVO();

            usuarioVO.setCodigoUsuario(new BigDecimal(usuarioCodigoUsuario));

            cuentaDTO.setUsuarioVO(usuarioDAO.findByPrimaryKey(usuarioVO));

            StringBuffer parametersUsuario = new StringBuffer();
            parametersUsuario.append("USUARIO_CODIGO_USUARIO=")
                             .append(usuarioCodigoUsuario);

            objectArray = cuentaDAO.findByAnyWhere(parametersUsuario.toString())
                                   .toArray();
            cuentaVOArray = new CuentaVO[objectArray.length];

            for (int i = 0; i < objectArray.length; i++) {
                cuentaVOArray[i] = (CuentaVO) objectArray[i];
            }

            cuentaDTO.setCuentaVOSet(cuentaVOArray);
        } finally {
            PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
                START_TIME);
        }

        return cuentaDTO;
    }
```

- Metodo browseByChild

  Este método estará expuesto para cada uno de los objetos hijos que se tenga dentro del modelo, es decir se expondrá un método o la por cada objeto dependiente. Este método recibe como parámetro la llave primaria de la entidad, en este caso el objeto genérico. Para el ejemplo se podría tener en un mismo objeto de almacenamiento la cuenta y las compras que tenga asociadas.

```
/** Busca todos los hijos que cumplen con la llave primaria de la tabla COMPRAS
  * pasada por parametro y el correspondiente valor  en la tabla entidad Cuenta
  *
  * Cuando el log de QoS es activado se puede referir a este metodo
  * mediante la llave <i>browseForCompras</i>.

  *  @param          cuentaCodigoCuenta  Parametro que conforma la llave primari
de la tabla COMPRAS
  *  @return               Instancia de la clase <code>CuentaDTO</code>
 */
public CuentaDTO browseForCompras(String cuentaCodigoCuenta)
    throws com.lucasian.leaf.common.SystemException {
    final String METHOD_NAME = "browseForCompras";
    final long START TIME = System.currentTimeMillis();
    ComprasVO[] comprasVOArray = null;
    Object[] objectArray = null;
    CuentaDTO cuentaDTO = new CuentaDTO();

    try {
        CuentaDAO cuentaDAO = daoFactory.getCuentaDAO();

        ComprasDAO comprasDAO = daoFactory.getComprasDAO();
        CuentaVO cuentaVO = new CuentaVO();
        cuentaVO.setCodigoCuenta(new BigDecimal(cuentaCodigoCuenta));
        cuentaDTO.setCuentaVO(cuentaDAO.findByPrimaryKey(cuentaVO));

        StringBuffer parametersCompras = new StringBuffer();
        parametersCompras.append("CUENTA CODIGO CUENTA=")
                         .append(cuentaCodigoCuenta);

        objectArray = comprasDAO.findByAnyWhere(parametersCompras.toString())
                                .toArray();
        comprasVOArray = new ComprasVO[objectArray.length];

        for (int i = 0; i < objectArray.length; i++) {
            comprasVOArray[i] = (ComprasVO) objectArray[i];
        }

        cuentaDTO.setComprasDTOSet(comprasVOArray);
    } finally {
        PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
            START TIME);
    }

    return cuentaDTO;
}
```

- Metodo findAllChild

  Este método permite realizar la búsqueda de la tabla entidad y cada uno de sus hijos, es decir este método encapsula un objeto genérico y todos sus objetos dependientes. Este método recibe como parámetro un objeto de tipo String que debe alojar el valor de llave primaria a partir del cual se va realizar la búsqueda, tanto en ella misma como en todos sus hijos. Para el ejemplo que se esta siguiendo, se recibiría como parámetro el valor de código cuenta a partir del cual se buscaran todos los valores que dentro de la tabla Cuenta cumplan y además todas las compras, retiros y transacciones relacionadas con este valor.

  El proceso que sigue, es que basado en el valor enviado como parámetro, se crea un Where que es enviado a cada uno de los DAO's para realizar la consulta a través del método findByAnywhere.

El método findByAnyWhere de los DAO's consiste en que basado en una sentencia Where construida por el usuario, se envía a consultar y se construyen arreglos de objetos que finalmente son retornados.

Para el ejemplo de la cuenta se tendrían colecciones de objetos para la tabla compras, retiros y transacciones.

```
    /**
     * Reliza la busqueda de todos los hijos  que cumplan con los valores pasados
     * por parametro y los correspondientes valores en la tabla entidad Cuenta
     *
     * Cuando el log de QoS es activado se puede referir a este metodo
     * mediante la llave <i>findAllChild</i>.

     *  @param          codigoCuenta  Parametro que conforma la llave primaria de la
  tabla CUENTA
     *  @return                 Instancia de la clase <code>CuentaDTO</code>

     */
    public CuentaDTO findAllChild(String codigoCuenta)
        throws com.lucasian.leaf.common.SystemException {
        final String METHOD_NAME = "findAllChild";
        final long START_TIME = System.currentTimeMillis();
        Object[] objectArray = null;

        CuentaDTO cuentaDTO = new CuentaDTO();

        try {
            CuentaDAO cuentaDAO = daoFactory.getCuentaDAO();
            CuentaVO cuentaVO = new CuentaVO();
            cuentaVO.setCodigoCuenta(new BigDecimal(codigoCuenta));

            StringBuffer parametersCuenta = new StringBuffer();
            parametersCuenta.append("CODIGO CUENTA=").append(codigoCuenta);

            ComprasDAO comprasDAO = daoFactory.getComprasDAO();

            RetirosDAO retirosDAO = daoFactory.getRetirosDAO();

            TransaccionesDAO transaccionesDAO = daoFactory.getTransaccionesDAO();
            cuentaDTO.setCuentaVO(cuentaDAO.findByPrimaryKey(cuentaVO));

            objectArray = comprasDAO.findByAnyWhere(parametersCuenta.toString())
                                    .toArray();

            ComprasVO[] comprasVOArray = new ComprasVO[objectArray.length];

            for (int i = 0; i < objectArray.length; i++) {
                comprasVOArray[i] = (ComprasVO) objectArray[i];
            }

            cuentaDTO.setComprasDTOSet(comprasVOArray);

            objectArray = retirosDAO.findByAnyWhere(parametersCuenta.toString())
                                    .toArray();

            RetirosVO[] retirosVOArray = new RetirosVO[objectArray.length];

            for (int i = 0; i < objectArray.length; i++) {
                retirosVOArray[i] = (RetirosVO) objectArray[i];
            }

            cuentaDTO.setRetirosDTOSet(retirosVOArray);

            objectArray = transaccionesDAO.findByAnyWhere(parametersCuenta.toString())
                                          .toArray();

            TransaccionesVO[] transaccionesVOArray = new
ransaccionesVO[objectArray.length];

            for (int i = 0; i < objectArray.length; i++) {
                transaccionesVOArray[i] = (TransaccionesVO) objectArray[i];
            }

            cuentaDTO.setTransaccionesDTOSet(transaccionesVOArray);
        } finally {
            PerformanceLogger.writeLog(getClass().getName(), METHOD_NAME,
                START_TIME);
        }

        return cuentaDTO;
    }
```

8.5 CLASE GENERADA DTO

La clase Data Transfer Object como se ha hablado a lo largo del documento sirve como medio de almacenamiento. La clase esta compuesta por accesores a cada uno de los atributos de la clase. Estos atributos son Value Object o colecciones de Value Object. Adicional a los get y a los set hay un método que permite clonar el objeto. A continuación se explicara un poco la composición de esta clase

- Atributos y Constructor

```
 * Source Code Generated by: Lucasian Enterprise Application Framework
package com.presentacion.universidad.model;

import com.presentacion.universidad.model.BancoVO;


/**
 * DTO
 *
 * @author leaf3i [leaf.support@lucasian.com]
 * @version 1.0.0
 */
public class CuentaDTO implements Serializable, Cloneable {
    private static final long serialVersionUID = 1L;
    private CuentaVO cuentaVO;
    private CuentaVO[] cuentaVOSet;
    private UsuarioVO usuarioVO;
    private EstadoVO estadoVO;
    private SeguroVO seguroVO;
    private BancoVO bancoVO;
    private ComprasVO[] comprasDTOSet;
    private RetirosVO[] retirosDTOSet;
    private TransacionesVO[] transacionesDTOSet;
    private CuentaVO cuentaCodigoCuentaTransacionesVO;
    private CuentaVO cuentaCodigoCuentaDestinoTransacionesVO;

    public CuentaDTO() {
    }

    public CuentaDTO(UsuarioVO usuarioVO, EstadoVO estadoVO, SeguroVO seguroVO,
        BancoVO bancoVO, ComprasVO[] comprasDTOSet, RetirosVO[] retirosDTOSet,
        TransacionesVO[] transacionesDTOSet,
        CuentaVO cuentaCodigoCuentaTransacionesVO,
        CuentaVO cuentaCodigoCuentaDestinoTransacionesVO, CuentaVO cuentaVO,
        CuentaVO[] cuentaVOSet) {
        this.cuentaVO = cuentaVO;
        this.cuentaVOSet = cuentaVOSet;
        this.usuarioVO = usuarioVO;
        this.estadoVO = estadoVO;
        this.seguroVO = seguroVO;
        this.bancoVO = bancoVO;
        this.comprasDTOSet = comprasDTOSet;
        this.retirosDTOSet = retirosDTOSet;
        this.transacionesDTOSet = transacionesDTOSet;
        this.cuentaCodigoCuentaTransacionesVO = cuentaCodigoCuentaTransacionesVO;
        this.cuentaCodigoCuentaDestinoTransacionesVO =
 cuentaCodigoCuentaDestinoTransacionesVO;
    }
```

Siguiendo el ejemplo de la entidad compuesta Cuenta, para las tablas padres como estado, usuario, seguro y banco, hay un objeto Value Object. Para las tablas hijas se tiene una colección de objetos representada por un arreglo de Value Objects.

Cabe resaltar que para la relación entre Cuenta y Transacción hay una asociación múltiple, ya que para poder tener la figura de cuenta fuente y cuenta

destino se debe tener dos llaves foráneas de la tabla transacción que apunten a la llave primaria de la tabla Cuenta. Esto dentro del objeto DTO se representa en que hay un atributo por cada llave foránea.

Se tienen dos clases de constructores uno vacio en caso de que se quiera crear el objeto por medio de los accesores o uno que recibe uno a uno los parámetros.

9. CONCLUSIONES

- El patrón Composite Entity es una propuesta bastante fuerte para el tratamiento de las entidades compuestas de base de datos en Java. Este patrón permite hacer un mapeo objeto relacional basado en las estructuras de árbol convencionales, en donde hay objetos genéricos y objetos dependientes. La propuesta fundamental de este patrón es poder interactuar con uno o más objetos, instan ciando solo uno de ellos, es decir el objeto genérico. Para este proyecto se abstrajo el concepto y se presenta como una interfaz que encapsula las operaciones de búsqueda, inserción, actualización y eliminación sobre una entidad compuesta de base de datos.

- Las entidades compuestas de base de datos son muy comunes cuando se realiza la composición de almacenamiento de un sistema, ya que este tipo de modelos representan el core del negocio en un proyecto. Cuando se realiza una implementación JEE se suelen cometer muchos errores sobre el mapeo objeto relacional de este tipo de entidades, por la complejidad en cuanto a cantidades de tablas, validaciones de tipos de datos, nulabilidad, prioridades de inserción y actualización , encaspsulamiento de datos y lo mas critico relaciones entre las tablas de la entidad compuesta. La opción presentada como producto final de este proyecto, el generador de código basado en el patrón Composite Entity, proporciona una herramienta en la que el desarrollador no se tenga que preocupar por ninguno de los detalles antes mencionados, ya que las validaciones son hechas por el sistema antes de la generación de código.

- La definición de metodología propuesta en este proyecto estuvo basada en un desarrollo iterativo, el cual permitiera medir los resultados en cada una de sus fases, dando espacio para investigación, implementación y pruebas. Se decidió adoptar una metodología propia en donde no se hizo mucho énfasis en los diagramas UML, debido al tiempo estipulado para la finalización del proyecto y además por que algunas de las tecnologías utilizadas para el desarrollo ya tienen definidos diagramas de interacción y lineamientos basados en mejores practicas de desarrollo de software, que permiten disminuir el tiempo de implementación en un proyecto.

- La definición de arquitectura representa uno de los aspectos más importantes a seguir en un proyecto de ingeniería Informática, debido a que crea la ruta base sobre la que el sistema se va a comportar en un ambiente de despliegue técnico. JEE propone una arquitectura la cual esta divida por capas, en donde se define muy bien el propósito que tiene cada una dentro del sistema. El patrón Composite Entity, base teórica de este proyecto hace parte de la especificación de arquitectura propuesta por JEE, y su tarea principal es integrar la capa de servicios de negocio y la herramienta

utilizada para interactuar con la base de datos. Cabe anotar que un proyecto empresarial que se inicie sin definir una arquitectura tiende mucho a retrasarse, ya que suele perderse el rumbo del proyecto.

- Los Frameworks son una herramienta de gran ayuda en los proyectos, ya que permiten minimizar el trabajo de los actores implicados en proyecto de desarrollo de software. Actualmente hay pocos en el mercado, debido al esfuerzo que lleva realizar una herramienta la cual sea genérica y de buena calidad. Leaf3i producto al cual pertenece el componente desarrollado en este proyecto, es un Framework tanto teórico como practico, el cual propone como herramienta de productividad en los proyectos generadores de código que son creados a partir de patrones de diseño.

- Eclipse es un IDE que tiene un gran reconocimiento a nivel mundial debido a las múltiples funcionalidades que ofrece, actualmente hay una gran cantidad de plugins los cuales son desarrollados por la comunidad de programadores de Java o por empresas que implementan sus productos sobre esta tecnología. Para este proyecto se utilizaron API's fundamentales de licencia GNU como GEF, JFace, SWT y Draw2d, que permitieron implementar la interfaz con la cual interactúa el usuario.

- El desarrollo del proyecto tuvo como objetivo principal, brindar al usuario una herramienta de alta productividad, que resuelva fácilmente los errores comunes de implementación del mapeo objeto-relacional de las entidades compuestas de base de datos. La versión desarrollada en este proyecto esta disponible solamente para el motor de base de datos Oracle, debido a que es el mas comercial y utilizado en las empresas en Colombia. A futuro se piensa replicar la solución para SQL Sever, DB2 para que hagan del generador de código Composite Entity una herramienta genérica.

BIBLIOGRAFÍA

Catálogo de Patrones de Diseño J2EE. Y II: Capas de Negocio y de Integración[en línea]. España Ciudad Real: Programación en Castellano S.L, 1998. [consultado 23 de Agosto de 2006]. Disponible en Internet: *http://www.programacion.com/tutorial/patrones2/7/*

Core J2EE Patterns [en línea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: *http://java.sun.com/blueprints/corej2eepatterns/Patterns/*

Core J2EE Patterns – Composite Entity [en línea]. U.S : Sun Microsystems, 2004. [Consultado 19 Septiembre de 2006]. Disponible en internet: http://java.sun.com/blueprints/corej2eepatterns/Patterns/CompositeEntity.html

Core J2EE Patterns - Data Access Object [en linea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: *http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html*

Enterprise Java Beans [en línea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: *http://java.sun.com/products/ejb/*

GAMMA E; BECK, K. Contributing to Eclipse: Principles, Patterns, and Plug-Ins. U.S : Addison Wesley, 2003. 381 p.

GefDescription2 [en línea]. U.S : Philippe Ombredanne, 2006. [consultado en 15 Septiembre de 2006]. Disponible en Internet: *http://eclipsewiki.editme.com/GefDescription2?show-menu=false*

Java Tecnology Especifications [en linea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septirmbre de 2006]. Disponible en internet: *http://www.jcp.org/en/jsr/detail?id=127*

Java Server Faces [en línea]. U.S : Sun Microsystems, 2004. [Consultado 23 Septiembre de 2006]. Disponible en internet: *http://java.sun.com/javaee/javaserverfaces/*

Jelly Executable XML [en línea]. U.S : The Apache Software Foundation, 2006. [consultado 12 de Septiembre de 2006]. Disponible en internet: http://jakarta.apache.org/commons/jelly/

Web Services Axis [en línea]. U.S : The Apache Software Foundation, 2006. [consultado 12 de Septiembre de 2006]. Disponible en internet: http://ws.apache.org/axis/

Axis Architecture Guide [en línea]. U.S : The Apache Software Foundation, 2006. [consultado 12 de Septiembre de 2006]. Disponible en internet: http://ws.apache.org/axis/java/architecture-guide.html

SWT: el nuevo sistema de desarrollo de GUIs en Java [en linea]. Madrid: Miro Internacional, 2000. [consultado 1 Septiembre de 2006]. Disponible en Internet : *http://www.gui.uva.es/~laertes/nuke/index.php?option=com_content&task=view&id=54&Itemid=41*

WILLIAM, Crawford; JONATHAN, Kaplan. J2EE Desing Patterns. U.S: Editores O'Reilly & Associates, 1998. 343 p.

ANEXOS

Anexo A. Script Esquema de prueba

```
CREATE TABLE CUENTA
(
CODIGO_CUENTA NUMBER NOT NULL,
"NUMERO _TARJETA" VARCHAR2(4000),
CEDULA_RESPONSANBLE1 VARCHAR2(20),
CEDULA_RESPONSABLE2 VARCHAR2(20),
ESTADO_CODIGO_ESTADO NUMBER(2),
BANCO_CODIGO_BANCO NUMBER(2),
USUARIO_CODIGO_USUARIO NUMBER(4),
SEGURO_CODIGO_ENTIDAD NUMBER(3)
)
;
CREATE TABLE BANCO
(
CODIGO_BANCO NUMBER(2) NOT NULL,
DESCRIPCION_BANCO VARCHAR2(50),
DIRECCION VARCHAR2(50),
SUCURSAL NUMBER(2),
CODIGO_POSTAL NUMBER(3)
)
;
CREATE TABLE USUARIO
(
CODIGO_USUARIO NUMBER(4) NOT NULL,
CEDULA VARCHAR2(20),
NOMBRE VARCHAR2(100),
DIRECCION VARCHAR2(50),
TELEFONO VARCHAR2(10),
EMAIL VARCHAR2(40),
FECHA_NACIMIENTO DATE
)
;
CREATE TABLE SEGURO
(
CODIGO_ENTIDAD NUMBER(3) NOT NULL,
DESCRIPCION_ENTIDAD VARCHAR2(50),
TELEFONO VARCHAR2(12),
DIRECCION VARCHAR2(50)
)
;
CREATE TABLE TRANSACIONES
(
CODIGO_TRANSACION NUMBER(5) NOT NULL,
"CUENTA _DESTINO" NUMBER(5),
CUENTA_FUENTE NUMBER(5),
FECHA_TRANSACION DATE,
VALOR_TRANSACCION NUMBER(6),
CUENTA_CODIGO_CUENTA NUMBER,
```

```
CUENTA_CODIGO_CUENTA_DESTINO NUMBER
)
;
CREATE TABLE ESTADO
(
CODIGO_ESTADO NUMBER(2) NOT NULL,
DESCRIPCION_ESTADO VARCHAR2(50)
)
;
CREATE TABLE RETIROS
(
CODIGO_RETIRO NUMBER(8) NOT NULL,
CODIGO_CAJERO NUMBER(3),
FECHA DATE,
VALOR_RETIRO NUMBER(6),
CUENTA_CODIGO_CUENTA NUMBER
)
;
CREATE TABLE COMPRAS
(
CODIGO_COMPRA NUMBER(5) NOT NULL,
FECHA_COMPRA DATE,
ENTIDAD VARCHAR2(50),
VALOR_COMPRA NUMBER(6),
CUENTA_CODIGO_CUENTA NUMBER
)
;
ALTER TABLE CUENTA
ADD CONSTRAINT CUENTA_PK PRIMARY KEY
(
CODIGO_CUENTA
)
 ENABLE
;
ALTER TABLE BANCO
ADD CONSTRAINT BANCO_PK PRIMARY KEY
(
CODIGO_BANCO
)
 ENABLE
;
ALTER TABLE USUARIO
ADD CONSTRAINT USUARIO_PK PRIMARY KEY
(
CODIGO_USUARIO
)
 ENABLE
;
ALTER TABLE SEGURO
ADD CONSTRAINT SEGURO_PK PRIMARY KEY
(
CODIGO_ENTIDAD
)
 ENABLE
;
```

```
ALTER TABLE TRANSACIONES
ADD CONSTRAINT TRANSACIONES_PK PRIMARY KEY
(
CODIGO_TRANSACION
)
 ENABLE
;
ALTER TABLE ESTADO
ADD CONSTRAINT ESTADO_PK PRIMARY KEY
(
CODIGO_ESTADO
)
 ENABLE
;
ALTER TABLE RETIROS
ADD CONSTRAINT RETIROS_PK PRIMARY KEY
(
CODIGO_RETIRO
)
 ENABLE
;

ALTER TABLE COMPRAS
ADD CONSTRAINT COMPRAS_PK PRIMARY KEY
(
CODIGO_COMPRA
)
 ENABLE
;
ALTER TABLE CUENTA
ADD CONSTRAINT CUENTA_ESTADO_FK FOREIGN KEY
(
ESTADO_CODIGO_ESTADO
)
REFERENCES ESTADO
(
CODIGO_ESTADO
) ENABLE
;
ALTER TABLE CUENTA
ADD CONSTRAINT CUENTA_BANCO_FK FOREIGN KEY
(
BANCO_CODIGO_BANCO
)
REFERENCES BANCO
(
CODIGO_BANCO
) ENABLE
;
ALTER TABLE CUENTA
ADD CONSTRAINT CUENTA_USUARIO_FK FOREIGN KEY
(
USUARIO_CODIGO_USUARIO
)
REFERENCES USUARIO
```

```
(
CODIGO_USUARIO
) ENABLE
;

ALTER TABLE CUENTA
ADD CONSTRAINT CUENTA_SEGURO_FK FOREIGN KEY
(
SEGURO_CODIGO_ENTIDAD
)
REFERENCES SEGURO
(
CODIGO_ENTIDAD
) ENABLE
;

ALTER TABLE TRANSACIONES
ADD CONSTRAINT TRANSACIONES_CUENTA_FK FOREIGN KEY
(
CUENTA_CODIGO_CUENTA
)
REFERENCES CUENTA
(
CODIGO_CUENTA
) ENABLE
;

ALTER TABLE TRANSACIONES
ADD CONSTRAINT TRANSACIONES_CUENTA_FK1 FOREIGN KEY
(
CUENTA_CODIGO_CUENTA_DESTINO
)
REFERENCES CUENTA
(
CODIGO_CUENTA
) ENABLE
;

ALTER TABLE RETIROS
ADD CONSTRAINT RETIROS_CUENTA_FK FOREIGN KEY
(
CUENTA_CODIGO_CUENTA
)
REFERENCES CUENTA
(
CODIGO_CUENTA
) ENABLE

ALTER TABLE COMPRAS
ADD CONSTRAINT COMPRAS_CUENTA_FK FOREIGN KEY
(
CUENTA_CODIGO_CUENTA
)
REFERENCES CUENTA
(
```

```
CODIGO_CUENTA
) ENABLE
;
```