Universidad de las Ciencias Informáticas
Facultad 6

"Extensión de *Visual Paradigm for UML* para el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación".

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Manuel Velázquez Labrada

Tutores: Ing. Beatriz Lara Osorio

Ing. Armando Robert Lobo

La Habana, Junio, 2012. Cuba

"Año 54 de la Revolución"

"…este país vivirá de la inteligencia y de las producciones intelectuales"

Declaración de autoría

Declaro que soy el único autor de este trabajo y autorizo al Centro de Tecnologías de Gestión de Datos, DATEC de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firman la presente a los ____ días del mes de _____ del año 2012.

_____

Autor: Manuel Velázquez Labrada

_____ _____

Tutora: Ing. Beatriz Lara Osorio Tutor: Ing. Armando Robert Lobo

Datos de Contactos

Autor:

Manuel Velázquez Labrada

Universidad de las Ciencias Informáticas, La Habana, Cuba.

Email: mvlabrada@estudiantes.uci.cu

Tutora:

Ing. Beatriz Lara Osorio

Universidad de las Ciencias Informáticas, La Habana, Cuba.

Email: blara@uci.cu

Tutor:

Ing. Armando Robert Lobo

Universidad de las Ciencias Informáticas, La Habana, Cuba.

Email: arobert@uci.cu

AGRADECIMIENTOS

A DIOS

Por permitirme terminar mi carrera profesional, y culminar esta etapa de mi vida. Por darme unos padres maravillosos que me han brindado la oportunidad de ser una persona con profesión. Por la vida que

A MI FAMILIA

Por el apoyo incondicional que me ofrecieron durante mi carrera profesional y alentarme con ánimos de superación. Por brindarme su comprensión, confianza, amor y amistad, porque sin su apoyo no hubiera sido posible la culminación de mi carrera profesional. En especial a mis padres Giselda y Coello, abuelos Delia y Miguel y a mi hermano Yunior por creer en mí y apoyarme en todo momento, por los esfuerzos y sacrificios que hizo para lograr que fuera un profesional, por lo que se ganaron en mi una gran admiración.

Por encomendarme siempre a dios a lo largo de mi carrera, con la esperanza de que superara las dificultades que se me presentaron en mi etapa estudiantil. Por sus consejos para motivarme a seguir adelante y así concluir mi carrera.

A MI NOVIA

A mi novia, por ser tan especial y maravillosa, por su paciencia y por soportar todos estos momentos de alegría y tristeza. Por quererme mucho y brindarme su familia como parte de la mía. Por acogerme en su seno familiar, a mis suegros Ester y Eduardo y demás miembros, muchos agradecimientos.

A MIS AMIGOS

Por demostrarme día a día que la importancia de los hechos es relativa, que lo más importante es levantarse siempre y continuar caminando un paso a la vez.

Por compartir cada alegría, tristeza, victoria y derrota como si fuesen mis familiares. Los quiero mucho a todos.

A MIS MAESTROS

A mis tutores Beatriz y Lobo por guiarme en todo momento, con paciencia y sabiduría. Por hacer posible la elaboración de este artefacto y compartir generosamente su sabiduría y conocimiento.

A todos los profesores que me enseñaron todo lo que sé.

EN GENERAL AGRADEZCO

A todas las personas que de una forma u otra colaboraron en el desarrollo de esta investigación y en mi preparación como futuro ingeniero en Ciencias Informáticas.

DEDICATORIA

A DIOS

Por permitirme gozar de este momento. La culminación de mi carrera profesional.

A MI FAMILIA

Por ser personas de luz en mi vida.

Por su apoyo incondicional.

RESUMEN

En la Universidad de las Ciencias Informáticas se desarrolla un número significativo de proyectos en centros de desarrollo de software vinculados a las facultades. El Centro de Tecnología de Gestión de Datos es uno de ellos, el mismo elabora sus productos utilizando herramientas como *"Visual Paradigm for UML"* y *Ext JS* durante el proceso de desarrollo de software. Esta investigación surge producto de que la herramienta *"Visual Paradigm for UML"* de no provee la funcionalidad de automatizar el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación. El presente trabajo tiene como objetivo desarrollar una extensión de la herramienta *"Visual Paradigm for UML"* para el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación. Con esta extensión se pretende reducir el tiempo de desarrollo en la construcción de aplicaciones y garantizar un alto nivel en la calidad de las mismas.

Palabras claves: *Ext JS*, *Visual Paradigm for UML, plug-in*

ÍNDICE

INTRODUCCIÓN

La Universidad de la Ciencias Informáticas (UCI) junto con otras entidades es el motor impulsor en el país del desarrollo de software. La misma cuenta con varios centros de desarrollo de software, entre los que se encuentra DATEC (Centro de Tecnologías de Gestión de Datos). DATEC mantiene estrecha colaboración con importantes empresas, centros de investigación y desarrollo de alto nivel, así como con comunidades internacionales de desarrollo. Este centro tiene como misión proveer soluciones integrales y consultorías relacionadas con tecnologías de bases de datos y análisis de información. Además de crear nuevas tecnologías de bases de datos, de procesamiento y representación de la información a partir del desarrollo de proyectos, y contribuir con su trabajo al cumplimiento de las misiones fundamentales de la Universidad: la formación y la producción de software.

El desarrollo de aplicaciones web en DATEC con *Ext JS* genera una gran cantidad de componentes para cada solución que se produce. La carga de estos componentes en un navegador puede impactar en el rendimiento de las aplicaciones; una forma de evitar esto es a través de la compresión, ofuscación y mezcla, reduciendo el tamaño y disminuyendo el tiempo total para el inicio de la aplicación. Sin embargo, este paso requiere un enfoque disciplinado centrado en la arquitectura, puesto que una aplicación puede tener cientos de componentes y tornar muy complicada la escritura del Manifiesto de Ofuscación a partir del modelo de implementación. En la actualidad la elaboración de dicho manifiesto requiere de un esfuerzo importante y acarrea errores que demoran la entrega de los componentes empaquetados para las liberaciones de los productos, además de demandar el concurso de especialistas que traduzcan manualmente el modelo de implementación.

Teniendo en cuenta lo expuesto anteriormente se plantea el siguiente **problema a resolver:** ¿Cómo facilitar el empaquetado de componentes *Ext JS*? Por lo que consecuentemente se define como **objeto de estudio:** Ingeniería Dirigida por Modelos, siendo el **campo de acción:** Desarrollo de extensiones a la herramienta de modelado *"Visual Paradigm for UML"*.

Para darle cumplimiento al problema científico planteado se define como **objetivo general:** Desarrollar una extensión de la herramienta *"Visual Paradigm for UML"* para el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación.

Los **objetivos específicos** para llevar a cabo la investigación son**:**

- Elaborar un perfil de UML para reflejar los elementos de empaquetado de componentes.
- Realizar el análisis de la extensión de *Visual Paradigm for UML*.
- Realizar el diseño de la extensión de *Visual Paradigm for UML*.
- Realizar la implementación de la extensión de *Visual Paradigm for UML*.

- Realizar pruebas funcionales a la extensión implementada.

Para dar cumplimiento a los objetivos esbozados se han trazado las siguientes **tareas de la investigación:**

- Revisión bibliográfica de las herramientas existentes y de los enfoques teóricos del Desarrollo Dirigido por Modelo.
- Especificación de un perfil UML a aplicar en el modelado.
- Identificación de los requisitos funcionales y no funcionales.
- Realización del análisis.
- Realización del diseño.
- Implementación de la extensión.
- Realización y documentación de las pruebas funcionales.

El documento estará estructurado de la siguiente forma:

Capítulo 1. Fundamento Teórico: En el presente capítulo se realiza un estudio del arte de los sistemas de empaquetado. Se detallan características y elementos que lo componen y se hace alusión a las herramientas y tecnologías propuestas en el ambiente de desarrollo definido. Además se aborda sobre los conceptos básicos asociados al dominio del problema y que son necesarios para entender el desarrollo de la investigación.

Capítulo 2. Análisis y diseño del sistema: En el presente capítulo se definen los requerimientos, tanto funcionales como no funcionales y se obtiene el diagrama de casos de uso del sistema. Se ofrece una descripción de la arquitectura del sistema y de los principales objetivos del diseño. Se definen y diseñan diagramas de clases e interacción para el diseño seleccionado. Se muestra el despliegue de la aplicación a través del diagrama correspondiente.

Capítulo 3. Implementación y prueba: En este capítulo se realiza el modelo de implementación a partir de los resultados del flujo de trabajo anteriormente descrito. Se obtienen todos los algoritmos necesarios para la implementación de las nuevas funcionalidades. Además se describen las pruebas realizadas al Sistema.

CAPÍTULO 1: FUNDAMENTO TEÓRICO

En este capítulo se aborda brevemente los conocimientos básicos y necesarios para el desarrollo de la extensión aplicando el Desarrollo Dirigido por Modelos. El Desarrollo Dirigido por Modelos refleja todo un proceso vinculado al Lenguaje Unificado de Modelado y proporciona beneficios para la creación de software utilizando modelos como guías en el proceso de desarrollo. Además se documenta la configuración para extensiones (*plug-in*) en la herramienta "*Visual Paradigm for UML"*. Se definen los elementos tecnológicos para el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación aplicando un perfil UML. Por último se seleccionan las herramientas y tecnologías para implementar el *plug-in*.

1.1 Enfoques teóricos del Desarrollo Dirigido por Modelo.

1.1.1 Modelos

Los modelos se encuentran inmersos en todas las ramas y campos de la ciencia a través de los modelos científicos los cuales no son más que una representación abstracta, conceptual, gráfica o visual, física, matemática, de fenómenos, sistemas o procesos a fin de analizar, describir, explicar, simular, en general, explorar, controlar y predecir esos fenómenos o procesos. Un modelo permite determinar un resultado final a partir de datos de entrada. Se considera que la creación de un modelo es una parte esencial de toda actividad científica. (1)

Cada modelo es una descripción de un proceso que se presenta desde una perspectiva particular, donde se describen una sucesión de fases y un encadenamiento entre ellas. Un modelo es una descripción simplificada de un proceso del software que presenta una visión de ese proceso. Estos modelos pueden incluir actividades que son parte de los procesos y productos de software y el papel de las personas involucradas en la ingeniería del software. (2)

1.1.2 Desarrollo Dirigido por Modelos

Actualmente en un importante patrón de la Ingeniería de Software se ha convertido el Desarrollo Dirigido por Modelos (MDD) con el propósito de manipular amplias complejidades y requerimientos de sistemas con gran cantidad de software, basándose principalmente en la sustitución del código de lenguajes de programación por modelos. De esta forma y en el contexto de este paradigma, los modelos son considerados como entidades de primera clase, permitiendo nuevas posibilidades de crear, analizar y manipular sistemas a través de diversos tipos de herramientas y de lenguajes.

Cada modelo trata generalmente un aspecto, y las transformaciones entre modelos proporcionan un vínculo que permite la implementación automatizada de un sistema a partir de sus correspondientes

modelos, generando también modelos, por lo tanto constituyen una parte integral de esta propuesta basada en modelos. Estas transformaciones requieren soporte especializado en varios aspectos para que: modeladores de sistemas, desarrolladores de transformaciones y desarrolladores de herramientas puedan aplicarla en su máximo potencial. (3)

MDD se sintetiza en la combinación de tres ideas complementarias:

1. **La representación directa**, ya que el foco del desarrollo de una aplicación se desplaza del dominio de la tecnología hacia las ideas y conceptos del dominio del problema, de este modo se reduce la distancia semántica entre el dominio del problema y su representación.
2. **La automatización**, porque promueve el uso de herramientas automatizadas en aquellos procesos que no dependan del ingenio humano, de este modo se incrementa la velocidad de desarrollo del software y se reducen los errores debidos a causas humanas.
3. **Los estándares abiertos**, debido a que éstos no solo ayudan a eliminar la diversidad innecesaria sino que, además, alientan a producir, a menor costo, herramientas tanto de propósito general como especializadas. (4)

Fases para el proceso MDD:

Primera fase: se construye un Modelo Independiente de Plataforma (PIM por sus siglas en inglés), que representa el modelo de más alto nivel del sistema, se desarrolla independientemente de cualquier tecnología y no contienen detalles de la plataforma concreta en que la solución va ser implementada. Estos modelos surgen como resultado del análisis y diseño.

Segunda fase: se transforma a uno o varios Modelos Específico de Plataforma (PSM por sus siglas en inglés), derivados de la categoría anterior, que contienen detalles de la plataforma o tecnología con que se implementará la solución. Esto no es más que la combinación de los diferentes PIM que especifican el sistema con los detalles de cómo se utiliza un tipo de plataforma concreta. (5)

Tercera fase: se genera un Modelo de implementación (IM por sus siglas en inglés), lo cual se traduce a código fuente a partir de cada PSM. (4)

MDD utiliza el término “plataforma” para referenciar detalles tanto tecnológicos como de ingeniería que no son relevantes para la funcionalidad esencial del sistema. La plataforma se define como la especificación de un entorno de ejecución para un conjunto de modelos.

El enfoque MDD permite, luego de haber desarrollado cada PIM derivar automáticamente el resto de los modelos aplicando las correspondientes transformaciones.

1.1.3 Lenguajes de modelado

Un lenguaje de modelado es un lenguaje artificial compuesto por el cúmulo de símbolos estandarizados utilizados para diseñar *software*. Estos lenguajes han ido evolucionado y transcendiendo con el transcurso de los años, con el objetivo de abrir diversas oportunidades para el desarrollo de software hasta llegar al estándar más conocido a nivel mundial, el Lenguaje Unificado de Modelado.

1.1.3.1 Lenguaje Unificado de Modelado

El Lenguaje Unificado de Modelado (UML) es un lenguaje de propósito general que pueden usar todos los modeladores. Está basado en el acuerdo de gran parte de la comunidad informática. Este lenguaje pretende unificar las experiencias acumuladas sobre técnicas de modelado e incorporar las mejores prácticas en un acercamiento estándar.

UML permite la creación de los diferentes modelos que ofrecen las vistas necesarias para la construcción de un software de calidad y permite la compresión del sistema que se quiere realizar tanto por parte de los usuarios finales, como de los desarrolladores que implementarán la solución.

A través de UML se puede desarrollar un diseño sólido y a la vez flexible de la aplicación. Incluye diversos conceptos que se consideran necesarios para utilizar un proceso moderno iterativo, basado en construir una sólida arquitectura para resolver requisitos dirigidos por casos de uso, logrando que el proceso de creación del software sea eficiente y organizado. (6)

UML, como lenguaje para modelar un sistema de software se ha convertido en un estándar con las siguientes características:

- Permite modelar sistemas utilizando técnicas orientadas a objetos (OO).
- Permite especificar las decisiones de análisis y diseño, construyéndose modelos precisos y completos.
- Está compuesto por diversos elementos gráficos que se combinan para conformar diagramas, además cuenta con reglas para combinar dichos elementos.
- Es independiente del lenguaje de programación y de las características de los proyectos, ya que fue diseñado para modelar cualquier tipo de proyecto.
- Integra las mejores prácticas de los lenguajes de modelación existentes.
- A pesar de ser un lenguaje potente, es fácil de aprender y de usar.
- Permite documentar los artefactos de un proceso de desarrollo.
- Capaz de modelar toda la gama de sistemas que se necesite construir.

1.1.4 Lenguaje Unificado de Modelado en el Desarrollo Dirigido por Modelo

El MDD plantea una nueva forma de entender el desarrollo y mantenimiento de sistemas de software con el uso de modelos como principales artefactos en el proceso. Además propone una aproximación para el desarrollo de sistemas de software basado en la separación entre la especificación de la estructura, funcionalidades esenciales del sistema y la implementación final, usando plataformas de implementación específicas. Debido la relevancia e importancia que tiene el MDD aparecen nuevas propuestas frecuentemente basadas en este esquema de desarrollo enfocadas a los más diversos dominios de aplicación, por lo que contar con un lenguaje de modelado adecuado es primordial para la correcta aplicación de las distintas propuestas de MDD, puesto que permite a los grupos de trabajo hacer mayor énfasis en la realización de modelos y las posibles transformaciones aplicadas a los mismos, aprovechando la gran variedad de tecnología UML existente.

1.1.5 Perfiles UML

A pesar de que el lenguaje de modelado UML sea el estándar más utilizado, definido por la OMG (*Object Management Group*) para especificar y documentar cualquier sistema de forma precisa, a veces, se hace necesario contar con algún lenguaje más específico para modelar y representar los conceptos de ciertos dominios particulares. Para ello UML proporciona mecanismos capaces de extender su sintaxis y semántica, entre los que se encuentran los estereotipos, restricciones, y valores etiquetados.

Un perfil en un paquete UML se estereotipa como «*profile*», el cual permitirá extender a un metamodelo o a otro perfil, por lo que en el paquete UML 2.0 se definen algunas razones para extenderlo.

Actualmente existen definidos varios perfiles UML, algunos de los cuales han sido incluso estandarizados por la OMG y otros producidos por organizaciones y empresas fabricantes de lenguajes de programación y herramientas, que aunque no son estándares oficiales, están disponibles de forma pública y son comúnmente utilizados. Cada uno de estos perfiles define una forma concreta de usar UML en un entorno particular.

1.1.5.1 Elaboración del perfil UML

Pasos para la elaboración del perfil UML:

1- Disponer de la definición del dominio de aplicación o metamodelo de la plataforma a modelar con el perfil. Se incluye la definición de las entidades propias del dominio, las relaciones entre ellas, así como las restricciones que limitan su uso y sus relaciones.
2- Se agrega un estereotipo dentro del paquete *«profile»* por cada uno de los elementos del metamodelo solicitado en el perfil. Estos estereotipos tendrán el mismo nombre que los elementos del metamodelo, permitiéndose así una relación entre el metamodelo y el perfil. Es decir, cualquier elemento que sea necesario para definir el metamodelo puede ser etiquetado luego con un estereotipo.
3- Conocer a cuáles elementos del metamodelo de UML que se extiende es posible aplicar un estereotipo. Con esto se logra que cada estereotipo sea aplicado a la metaclase de UML que se utilizó en el metamodelo del dominio para definir un concepto o una relación.
4- Se etiquetan los elementos del perfil, definiendo los atributos que aparecen en el metamodelo, sus tipos y posibles valores iniciales.
5- A partir de las restricciones del dominio se seleccionan las que forman parte del perfil UML, como pueden ser las reglas del negocio y las multiplicidades entre asociaciones. Las restricciones son fáciles de chequear al finalizar el modelo de un sistema según el perfil, sobre todo aquellas asociaciones que han sido estereotipadas con los estereotipos de dicho perfil. De esta forma, los perfiles UML van a describir siempre modelos “bien formados” de un sistema en un dominio de aplicación, es decir, que respetan los límites tanto sintácticos como semánticos que impone cada uno de los perfiles UML que se han utilizado.
Luego de definirse el perfil mediante una relación de dependencia (estereotipada «*apply*») entre el paquete UML que contiene la aplicación, y los paquetes que definen los perfiles UML que se desean utilizar, permite ser representado en la aplicación a utilizar.

1.2 Extensiones a Visual Paradigm for UML

1.2.1 Implementación de la extensiones para la herramienta *Visual Paradigm for UML*

Como consecuencia de haber aplicaciones que no logran brindar a los usuarios las funciones requeridas es objetivo la implementación de extensiones o *plugs-in* que permitan la incorporación de estas carencias. Un *plug-in* es un fragmento de software que interactúa con el núcleo de la aplicación para proporcionar algunas funcionalidades que en la mayoría de los casos suelen ser muy específicas. *“Visual Paradigm for UML”* es una de las herramientas *CASE* (*Computer Aided Software Engineering*) que goza de gran prestigio para el modelado de software, pero presenta inconvenientes para la generación del Manifiesto de Ofuscación a partir del modelo de implementación. No obstante cuenta con los medios para extender funcionalidades, dando soporte a las extensiones de aplicación. La

aplicación provee de forma libre una API (*Application Programming Interface*) la cual permite a los desarrolladores implementar y reutilizar clases e interfaces, desarrollando funciones agregadas que son útiles para el desarrollo de software.

1.2.1.1 Procedimientos para la implementación del *plug-in* para *Visual Paradigm for UML*

Visual Paradigm for UML (VP-UML) posee una librería llamada "openapi.jar" que está ubicada en el paquete de instalación de la herramienta, dentro del paquete "lib/openapi.jar" la cual posibilita el mecanismo de extensión. Una vez conocida la librería de construcción del *plug-in* es importante tener en cuenta la estructura que conforma el *plug-in* para su desarrollo. Un *plug-in* presenta la estructura correspondiente a un paquete que lleva el nombre de *plug-in,* el cual contiene tres paquetes asociados, el primero, a la configuración del *plug-in*, segundo a las acciones correspondientes y tercero a los formularios o diálogos de la implementación. A continuación se desglosa para mayor comprensión la estructura de los paquetes:

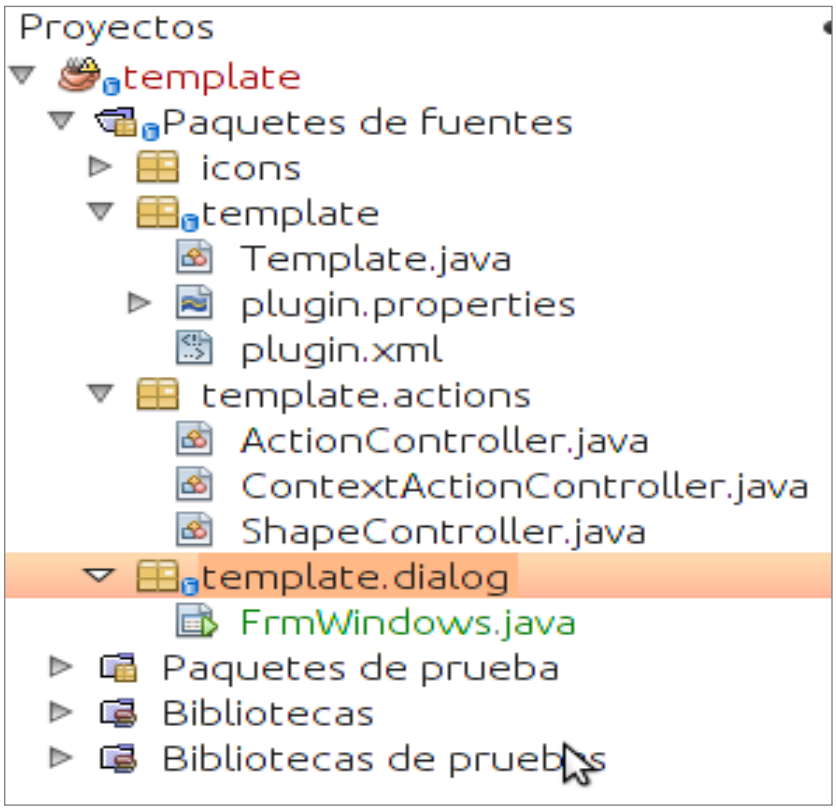


Fig.1: Estructura de un *plug-in* en el entorno de desarrollo *NetBeans* para *VP-UML*.

Primer paquete: está conformado por un conjunto de clases (Plugin.xml y Plugin.java) que permiten la carga y configuración del *plug-in.*

La clase Plugin.xml, base del *plug-in* permite mediante un script XML su configuración para ser cargado por la clase Plugin.java a través de la implementación de la interfaz VPPlugin. Además posibilita el enlace con las librerías asociadas a la implementación, así como la configuración de las acciones tanto a nivel de herramienta como a nivel de contexto, cargando las clases correspondiente a dicha acción. Plugin.java implementa los métodos load y unload utilizados para habilitar la carga y descarga de la extensión.

Ejemplo de la implementación del archivo Plugin.xml:

```
<plugin
id="vpextmi.Vpextmi"
name="Ejemplo de plugin"
description="Éste es mi primer plugin de prueba..."
provider="Visual Paradigm"
class="vpextmi.Vpextmi">
   <runtime>
      <library path="lib/vpextmi.jar" relativePath ="true"> </library>
      <library path="lib/JSBuilder2.jar" relativePath ="true"> </library>
      <library path="lib/libreria.jar" relativePath ="true"> </library>
      <library path="lib/openapi.jar" relativePath ="true"> </library>
   </runtime>
   <actionSets>
      <!--prueba JavaSript-->
      <actionSet id="otroIdTonto">
         <action
            id="vpextmi.action.Action1"
            actionType="generalAction"
            label="Generar Manifiesto de ofuscación"
            tooltip="Esta opción es para la realización de casos de pruebas"
            icon="icons/imagen.png"
            style="normal"
            menuPath="Tools/Report"
            toolbarPath="vpextmi.actions.Toolbar1/#">
            <actionController class="vpextmi.actions.vpextmiAction" />
         </action>
      </actionSet>
```

```
        <!-- prueba ofuscar principal ok
    <actionSet id="otroIdTonto1">
        <action
          id="vpextmi.action.Action2"
          actionType="generalAction"
          label="Ofuscar"
          tooltip="Esta opción es para la realización de casos de pruebas"
          icon="icons/imagen.png"
          style="normal"
          menuPath="Tools/Report"
          toolbarPath="vpextmi.actions.Toolbar1/#">
          <actionController class="vpextmi.actions.GenerateOfuscationManifest" />
        </action>
</actionSet>
      -->
 <!--prueba de menu-->
     <actionSet id="odr.ActionSet">
       <menu
          id="odr.MenuWithLogo"
          label="dd"
          toolbarPath="Menu/Tools/#/Report"
          icon=""
          mnemonic="O"
          tooltip="tooltip">
       </menu>
</actionSet>
      <!--prueba empaq principal ok-->
 <actionSet id="otroIdTonto2">
        <action
          id="vpextmi.action.Action3"
          actionType="generalAction"
          label="Empaquetar componentes"
          tooltip="Esta opción es para la realización de casos de pruebas"
          icon="icons/imagen.png"
```

```
            style="normal"
            menuPath="Tools/Report"
            toolbarPath="vpextmi.actions.Toolbar1/#">
            <actionController class="vpextmi.actions.generatePackagedComponents" />
          </action>
      </actionSet>
      <!--Menu Contextual-->
      <!--prueba ofuscar contextual ok-->
    <contextSensitiveActionSet id="vpextmi.actions.actions.ActionSet">
                        <contextTypes all="true">
                        </contextTypes>
                        <action
                                id="vpextmi.actions.actions.Action"
                                label="Generar Manifiesto de ofuscación"
                                icon="icons/add.png"
                                menuPath="OpenSpecification">
                        <actionController class="vpextmi.actions.GenerateContextOfuscationManifest"/>
                        </action>
       </contextSensitiveActionSet>
    </actionSets>
</plugin>
```

Ejemplo implementación de la clase Plugin.java:

```
public class Plugin implements com.vp.plugin.VPPlugin{
public void loaded(VPPluginInfovppi) {
}
publicvoidunloaded() {
}
}
```

Segundo paquete: es el contenedor de las acciones del *plug-in* definidas a nivel de herramientas y de contexto. Dichas clases, en dependencia de la acción implementan interfaces asociadas a las acciones VPActionController y VPContextAction. Ambas clases definen el performAction, el cual permite ejecutar acciones referentes al evento onClick de la acción.

Ejemplo de la implementación de la clase VPActionController:

```
package sample.plugin.actions;
import java.awt.event.ActionEvent;
public class ContextActionController implements com.vp.plugin.action.VPContextActionController
{
public void performAction(
com.vp.plugin.action.VPAction action,
com.vp.plugin.action.VPContext context,
ActionEvent e
) {
// called when the button is clicked
}
public void updated(
com.vp.plugin.action.VPAction action,
com.vp.plugin.action.VPContext context)
}
}
```

Tercer paquete: contiene los diálogos que se desean mostrar. Los diálogos se muestran mediante el método performAction asociado a la clase de acción correspondiente al diálogo. Ejemplo de implementación:

```
public class ContextActionController implements com.vp.plugin.action.VPContextActionController
{
ViewManagerviewManager = ApplicationManager.instance().getViewManager();
public void performAction(VPAction action, VPContext vpc, ActionEvent ae)
{
FrmWindowsfrm = new FrmWindows();
viewManager.showDialog(frm);
 }
public void update(VPAction action, VPContextvpc)
{
throw new UnsupportedOperationException("Not supported yet.");
}
}
```

Luego de tener la estructura de la implementación del *plug-in* se hace necesario su reconocimiento por la herramienta VP-UML. Para la integración de la extensión, *"Visual Paradigm for UML"* propone la

siguiente organización de desarrollo del *plug-in*. Se crea una carpeta con el nombre de *plug-in* dentro de la carpeta de instalación de "*Visual Paradigm for UML"* la cual contiene la siguiente estructura de despliegue:

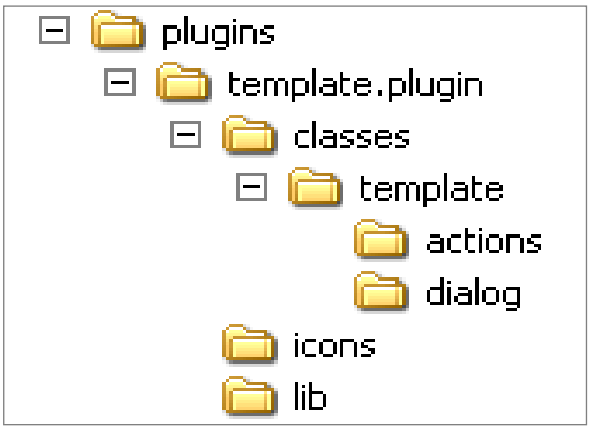


Fig.2: Estructura del despliegue en VP-UML del *plug-in.*

Debido a que la estructura de implementación del *plug-in* es distinta a la de integración con la herramienta se necesita utilizar la herramienta de despliegue del *plug-in* "Deployer", especificando los valores iniciales: nombre del *plug-in*, dirección de proyecto *plug-in* y dirección donde será desplegado.

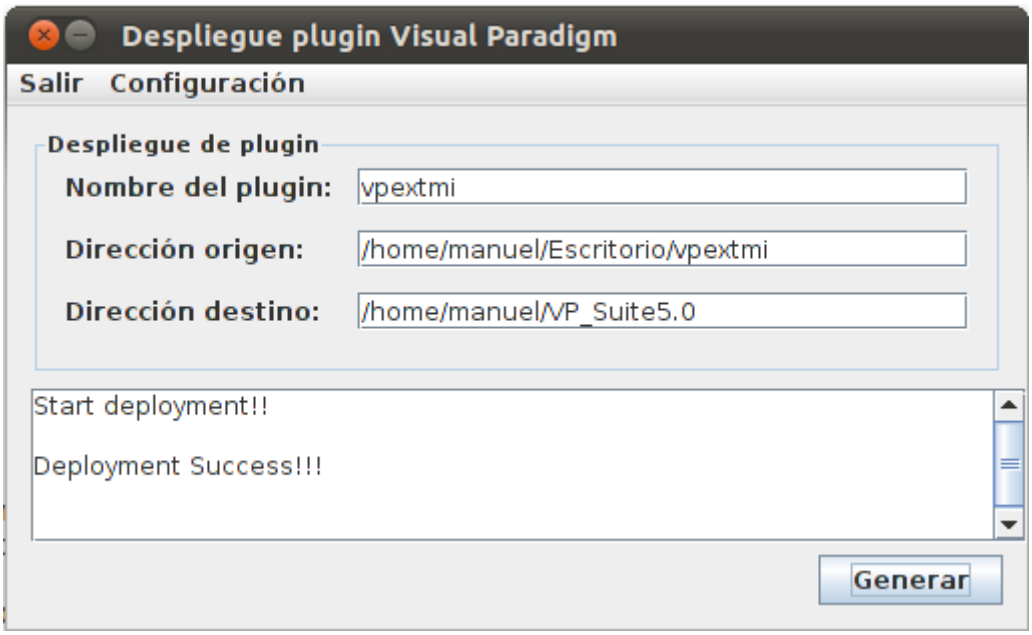


Fig.3: Herramienta de despliegue para VP-UML del *plug-in.*

1.3 Elementos tecnológicos de *Ext JS*.

Ext JS se puede considerar como un motor que permite crear *Rich Internet Applications* (RIA) utilizando componentes predefinidos y evitando que el proceso de validación por los diversos navegadores se vuelva tedioso.

Ext JS es una librería JavaScript que permite construir aplicaciones complejas para Internet. Esta librería incluye:

- Componentes *UI* del alto performance y personalizables.
- Modelo de componentes extensibles.
- Un API fácil de usar.
- Licencias *Open source* y comerciales**.** (7)

1.3.1 Herramienta de ofuscación y compresión de código

Con el propósito de conseguir archivos cada vez más pequeños se suele disponer de dos técnicas esenciales: la minificación o compresión y la ofuscación. La minificación es el proceso de reducir un código a su mínima expresión mediante la eliminación de comentarios, saltos de línea, así como espacios en blanco innecesarios. Como artefacto final se obtiene el código introducido originalmente pero compactado con un volumen de reducción por lo general del 45-50%. La ofuscación, como su nombre lo dice consiste en confundir, es decir, alterar literalmente el código mediante procesos de parseo y sustitución, resultando no sólo un código más compacto, sino diferente en sintaxis al original. El objetivo de la ofuscación es impedir o hacer más difícil los intentos de ingeniería inversa y desamblado que tienen la intensión de obtener una forma muy cercana a la original.

- ***JSBuilder 2.0***

JSBuilder es la herramienta distribuida por los desarrolladores de *Ext JS* para la compresión y ofuscación de código JavaScript y CSS. Se trata de una herramienta multiplataforma, implementada en Java, que utiliza la librería *YUI Compressor*.

1.4 Metodología de desarrollo

1.4.1 OpenUp

OpenUp es una metodología del Proceso Unificado que aplica el desarrollo iterativo e incremental dentro de la estructura del ciclo de vida, puesto que al ser ligero es utilizable en varios tipos de proyectos de software permitiendo disminuir las probabilidades de fracaso en los proyectos pequeños e incrementar las probabilidades de éxito. Además disminuir desde edades tempranas el número de

errores por las características de su desarrollo y centra su enfoque al cliente por ser una metodología ágil. (8)

1.4.1.2 Proceso de desarrollo con *OpenUP*

OpenUP es un proceso iterativo para el desarrollo de software que se caracteriza por ser mínimo, completo, y extensible. El proceso es mínimo ya que solo incluye el contenido del proceso fundamental; completo en que puede ser manifestado como proceso entero para construir un sistema; extensible en que puede ser utilizado como base para agregar o para adaptar más procesos. (9)

1.5 Lenguaje de programación

Un lenguaje de programación se encuentra conformado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones, describiendo el conjunto de acciones que se deben ejecutar. Los lenguajes orientados a objetos son leguajes de programación que hacen introducción a la Programación Orientada a Objetos (POO).

La programación orientada a objetos es una tecnología de empaquetado. Permite a los proveedores embalar unidades de funcionalidad genérica, de tal forma que los consumidores puedan aplicarlas a su aplicación concreta. (10)

El desarrollo de software orientado a objetos requiere, además del uso del lenguaje orientado a objetos de un análisis y diseño orientado a objetos, puesto que el modelamiento visual es la clave para realizar el análisis OO. Actualmente, en la industria del desarrollo de software para el modelamiento de sistemas OO se utiliza el UML como estándar, ya que éste tiene como fin modelar cualquier tipo de sistema usando los conceptos de la orientación a objetos.

La Sun Microsystems en los años 90 desarrolló un lenguaje de programación orientado a objetos, lo que implica que su concepción sea muy próxima a la forma de pensar humana, denominado Java.

Características de Java

- Es un lenguaje que es compilado, generando ficheros de clases compilados, pero estas clases compiladas, son en realidad interpretadas por la máquina virtual de java. Siendo la máquina virtual de java la que mantiene el control sobre las clases que se estén ejecutando.
- Es un lenguaje multiplataforma: El mismo código java que funciona en un sistema operativo, funcionará en cualquier otro sistema operativo que tenga instalada la máquina virtual java.
- Es un lenguaje seguro: La máquina virtual, al ejecutar el código java, realiza comprobaciones de seguridad, además el propio lenguaje carece de características inseguras, como por ejemplo

los punteros.

- Gracias al API de java podemos ampliar el lenguaje para que sea capaz de, por ejemplo, comunicarse con equipos mediante red, acceder a bases de datos, crear páginas HTML dinámicas, crear aplicaciones visuales al estilo Windows .(11)

1.6 Herramientas a utilizar

1.6.1 Herramientas CASE

Las herramientas *CASE* son aplicaciones informáticas, que tienen como objetivo brindar la posibilidad de realizar cálculos de costos, generar código fuente automáticamente de un diseño previamente dado y desempeñar un papel importante en la detección de errores de un producto informático.

1.6.1.1 Visual Paradigm for UML

Visual Paradigm for UML es una herramienta *CASE* que utiliza UML como lenguaje de modelado, es libre de costo y soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. Permite dibujar todos los tipos de diagramas de clases, generar código desde diagramas y documentación. Es una herramienta colaborativa, es decir, soporta a múltiples usuarios trabajando sobre el mismo proyecto; genera la documentación automáticamente en varios formatos Web o PDF, y permite control de versiones. Con el uso de esta herramienta la productividad en el desarrollo de software aumenta, ya que se reduce el costo en términos de tiempo y de dinero, ayudando en todos los aspectos del ciclo de vida de desarrollo del componente en tareas como realizar el diseño de este, documentar o detectar errores. VP-UML posibilita integrarse a varios entornos de desarrollo como son *Visual Studio, Eclipse* y *Netbeans*.

1.6.2 *NetBeans* 7.0

El IDE (Integrated Development Environment) NetBeans es un entorno integrado de desarrollo galardonado disponible para Windows, Mac, Linux y Solaris. *NetBeans* consiste en un IDE de código abierto y una plataforma de aplicaciones que permiten a los desarrolladores crear rápidamente aplicaciones web, escritorio y aplicaciones móviles utilizando la plataforma Java, así como JavaFX, PHP, JavaScript y Ajax, Ruby y Ruby onRails, Groovy y Grails, y C/C++. (12)

1.7 Conclusiones parciales

Después de realizar un estudio sobre la evolución de UML y su vinculación con el Desarrollo Dirigido por Modelos, enfocado al proceso de desarrollo de software, el análisis de las metodologías y tecnologías necesarias para el desarrollo de la herramienta, y teniendo en cuenta principalmente las exigencias y necesidades del cliente, se define: *OpenUP* como metodología de desarrollo de software, ya que es una metodología ágil para proyectos de corta duración diseñada para pequeños equipos de trabajo. Se utiliza como lenguaje de modelado UML adoptado por la Universidad como estándar para el desarrollo de software. Se emplea Java como lenguaje de programación, definido en la API de desarrollo de la herramienta *"Visual Paradigm for UML*". Como herramienta de modelado "*Visual Paradigm for UML*" en su versión 8.0, y como entorno de desarrollo *NetBeans* en su versión 7.0, como herramienta de compresión y ofuscación JSBuilder2.

CAPÍTULO 2: ANÁLISIS Y DISEÑO DEL SISTEMA

En el presente capítulo se describen los conceptos más importantes del entorno donde estará el Sistema. Se muestran los requerimientos funcionales y no funcionales a tener en cuenta para la implementación del *plug-in,* así como la confección del perfil UML necesario. Además, se identifican actores, casos de usos y las relaciones existentes entre ellos. Se aborda acerca de cómo debe funcionar el Sistema y se muestran además los diagramas de clases del diseño, así como los diagramas de secuencia.

2.1 Perfil de UML para reflejar los elementos de empaquetado de componentes

Los perfiles UML se han definido con el propósito de proporcionar un mecanismo de extensión ligera para UML. En UML 1.1, los estereotipos y valores etiquetados se usan como extensiones basadas en cadenas que se podría unir a los elementos del modelo UML de una manera flexible. En las versiones posteriores de UML, la noción de un perfil fue definida con el fin de proporcionar más estructura y precisión a la definición de los valores y estereotipos etiquetados. La infraestructura y las especificaciones de la superestructura de UML 2.0 lo han definido como una técnica específica de metamodelado donde los estereotipos son metaclases especificadas, los valores etiquetados son metaatributos estándares, y los perfiles son determinados tipos de paquetes.

La especificación de la infraestructura de UML es utilizada en varios metaniveles de las diversas especificaciones que la OMG tiene como tarea. Por ejemplo, la superestructura de UML la utiliza para crear el modelo UML, mientras que MOF (Meta-Object Facility) la utiliza para proporcionar capacidad a metamodelos.

Un metamodelado consiste en la idea de reedificar las entidades que forman cierto tipo de modelo y describir las propiedades comunes del mismo en forma de un modelo de objetos. Cuando se ve la clase como un objeto, la clase es una instancia de otra clase. En el metamodelado las entidades del modelo (clases) juegan dos papeles: el papel de plantilla (cuando se ve como clase) y el papel de instancia (cuando se ve como objeto). El metamodelado además, define la estructura semántica y restricciones para una familia de modelos.

Requisitos de la semántica del perfil:

- Para la especialización de un metamodelo de referencia (como un conjunto de paquetes de UML), debe el perfil proporcionar mecanismos de modo que la semántica especializada no esté en contradicción con la semántica del metamodelo de referencia. Es decir, las limitaciones de

perfil típicamente pueden definir reglas de buena formación que sean más restrictivas que las especificadas por el metamodelo de referencia.

- Un perfil debe ser capaz de hacer referencia a las bibliotecas UML específicas del dominio, donde se predefinen ciertos elementos del modelo.
- Debe permitir especificar qué perfiles se aplican a un determinado paquete (o cualquier especialización de ese concepto), con el objetivo de que en el intercambio de modelos para la importación de un entorno se pueda interpretar un modelo correctamente.
- Para que exista mayor claridad de definición y mayor facilidad de intercambio debe ser posible definir una extensión de UML que combine perfiles y librerías de modelos (incluyendo bibliotecas de plantillas) en una única unidad lógica.
- Un perfil debe lograr especializar la semántica de los elementos estándares del metamodelo UML. Por ejemplo, en un modelo con el perfil "modelo de Java," la generalización de las clases debe ser capaz de limitarse a la herencia simple sin tener que asignar explícitamente un estereotipo «clase de Java» a cada instancia de la clase.
- Debe brindar una convención de notación para las definiciones de estereotipos gráficos como parte de un perfil.
- Con la meta de satisfacer el primer requisito, los perfiles UML deben formar un mecanismo de extensión de metamodelo con determinadas restricciones a la forma en que el metamodelo UML puede ser modificado. El metamodelo de referencia se extiende sin cambios en los perfiles, puesto que no se pueden insertar nuevas metaclases en la jerarquía de metaclases de UML o modificar el estándar UML. Estas restricciones no se aplican en el contexto de MOF donde, en principio, puede ser cualquier metamodelo reelaborado en cualquier dirección.

Pese a que definir un nuevo lenguaje a través de MOF, permite mayor grado de expresividad y correspondencia con los conceptos del dominio de una aplicación en particular, se decidió extender el lenguaje utilizando perfiles UML. Los perfiles UML proporcionan una solución al problema de esta investigación sin necesidad de modificar la semántica de UML, sino sólo particularizar algunos de sus conceptos. Determinar un nuevo lenguaje utilizando MOF alargaría el proceso, extendiendo el tiempo de realización de este trabajo al redefinir metaclases, metaatributos y estereotipos, puesto que define por defecto el lenguaje UML. Cuando un lenguaje se describe con MOF, no respeta el metamodelo estándar de UML y esto dificulta que las herramientas UML existentes puedan manejar sus conceptos de una forma natural. Por otro lado, el paquete UMLProfiles 2.0 define una serie de mecanismos para

extender y adaptar las metaclases de un metamodelo cualquiera a las necesidades concretas de un dominio de aplicación.

2.1.1 Perfil UML a aplicar en el modelado

El objetivo de este perfil es identificar los elementos físicos del empaquetado de componentes con el fin de que puedan comprenderse y gestionarse mejor, además explica cómo organizar e integrar los componentes a implementar basándose en las especificaciones de diseño.

2.1.2 Modelo de dominio para el empaquetado de componentes:

En el diagrama se muestran las estructuras que conforman el Modelo de Dominio para el empaquetado de componentes, así como las relaciones entre ellas, permitiendo conocer la arquitectura y organización de los elementos dentro del proyecto.

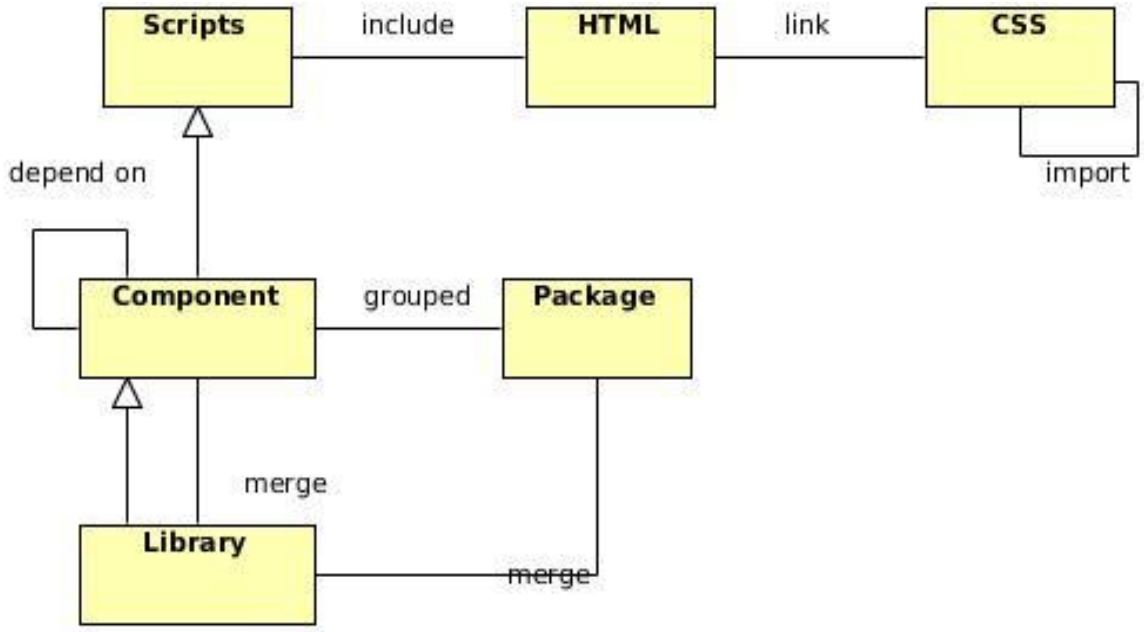


Fig.4: Modelo de dominio del perfil UML.

En este Modelo de Dominio se definen las entidades HTML, Scripts, CSS, Component, Package, Library. La entidad HTML representa cualquier archivo con código fuente en JavaScript, la cual está compuesta por la asociación de uno o varios scripts donde un script puede o no tener una relación de inclusión o mezcla. El componente HTML además puede o no tener varios CSS los cuales permiten la oportunidad de importar otros archivos CSS. La entidad Library permite la mezcla de paquetes y componentes a pesar de ser un *Component* el cual puede tener una o muchas relaciones de dependencias entre sí posibilitando a estos componentes ser agrupados en *Package*. A continuación

se muestra un concepto más amplio de cada una de las entidades pertenecientes a este Modelo de Dominio.

2.1.3 Descripción de las entidades:

CSS: Las CSS (por sus siglas en Inglés Cascading Style Sheets) permiten controlar la presentación de los documentos en la web, posibilitando la especificación de muchos atributos de los elementos que conforman una página web:(color del texto, márgenes, el tipo de letras, tamaño y color, posicionar elementos). Las hojas de estilos en cascada funcionan a base de reglas. En su composición las CSS pueden contar con una o más reglas siendo formadas por dos partes: un selector y la declaración.

HTML: HTML (por sus siglas en inglés HyperText Markup Language) es un lenguaje de marcado predominante para la elaboración de páginas web. Es utilizado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes. Además de poder describir, hasta un cierto punto la apariencia de un documento, posibilita incluir scripts así como CSS, los cuales pueden afectar el comportamiento de navegadores web y otros procesadores de HTML.

Scripts: Los scripts ofrecen a los autores poder extender los documentos HTML de maneras activas e interactivas. Los scripts pueden realizar diversas tareas como suele ser combinar componentes, interactuar con el sistema operativo o con el usuario, además de acompañar a un formulario para procesar los datos a medida que éstos se introducen.

Component: Representa cualquier archivo con código fuente en JavaScript. Es la parte del sistema sustituible, casi independiente e importante que desempeña una función clara en el contexto de una arquitectura bien definida. Un componente cumple los requisitos de la realización de un conjunto de interfaces y proporciona, además, dicha realización.

Package: Se utiliza con el fin de empaquetar de manera lógica y organizada cada uno de los componentes desarrollados.

Library: Es un componente que desempeña funciones específicas dentro del sistema, además de las propias como componente.

2.1.4 Especificación del perfil:

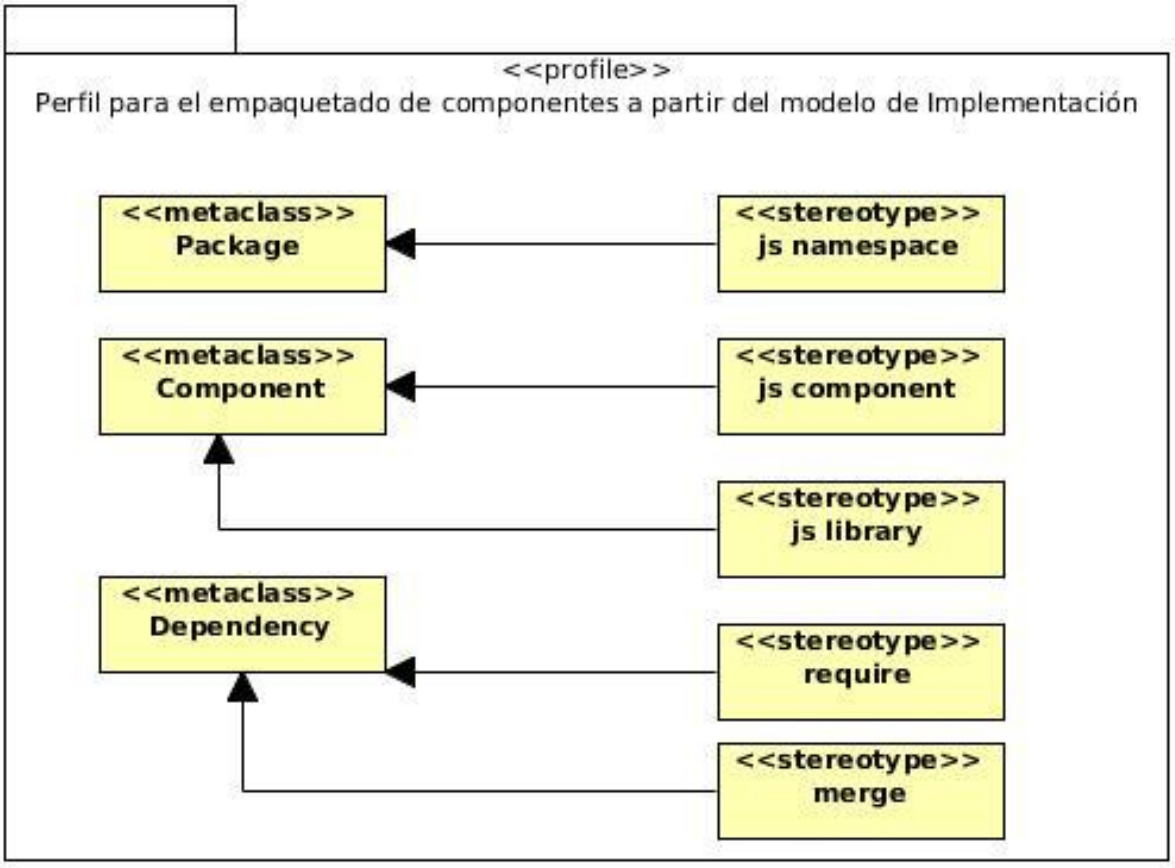


Fig.5: Especificación del perfil.

En la especificación del perfil se define un grupo de estereotipos a partir del Modelo de Dominio, los cuales extienden a metaclases formalizando el lenguaje de modelado. Cada metaclase a la que extiendan los estereotipos significa que cuando se usa un elemento del tipo del estereotipo especificado en los modelos UML, se asegura que ese elemento sigue las mismas reglas semánticas que un elemento del tipo de la metaclase a la que extiende, ya que se ha ampliado el metamodelo, pero en ningún caso se ha modificado su semántica. Los estereotipos *require* y *merge* extienden a la metaclase *Dependency*, *js namespace* a la metaclase *Package,* mientras que *js component* y *js library* extienden a la metaclase *Component*.

2.1.5 Especificación de estereotipos:

Estereotipo: <<js component>>

Metaclase UML a la que amplía: Component

Semántica del estereotipo: Representa componentes los cuales pueden depender de otros componentes.

Estereotipo: <<js library>>

Metaclase UML a la que amplía: Component

Semántica del estereotipo: Representa una especialización de los componentes con características particulares de *merge* con componentes y paquetes.

Estereotipo: <<merge>>

Metaclase UML a la que amplía: Dependency

Semántica del estereotipo: Representa una relación de mezcla entre varios componentes Java Script.

Nota: El orden de la mezcla está en correspondencia con las dependencias topológicamente ordenadas de los js component.

Estereotipo: <<require>>

Metaclase UML a la que amplía: Dependency

Semántica del estereotipo: Representa una relación de necesidad entre varios componentes Java Script.

Estereotipo: <<js namespace>>

Metaclase UML a la que amplía: Package

Semántica del estereotipo: Este estereotipo representa paquetes disponibles bajo un determinado nombre de dominio y que comparten el mismo modelo de objetos teniendo la funcionalidad de agrupar componentes.

2.1.6 Ejemplo de uso en un modelo

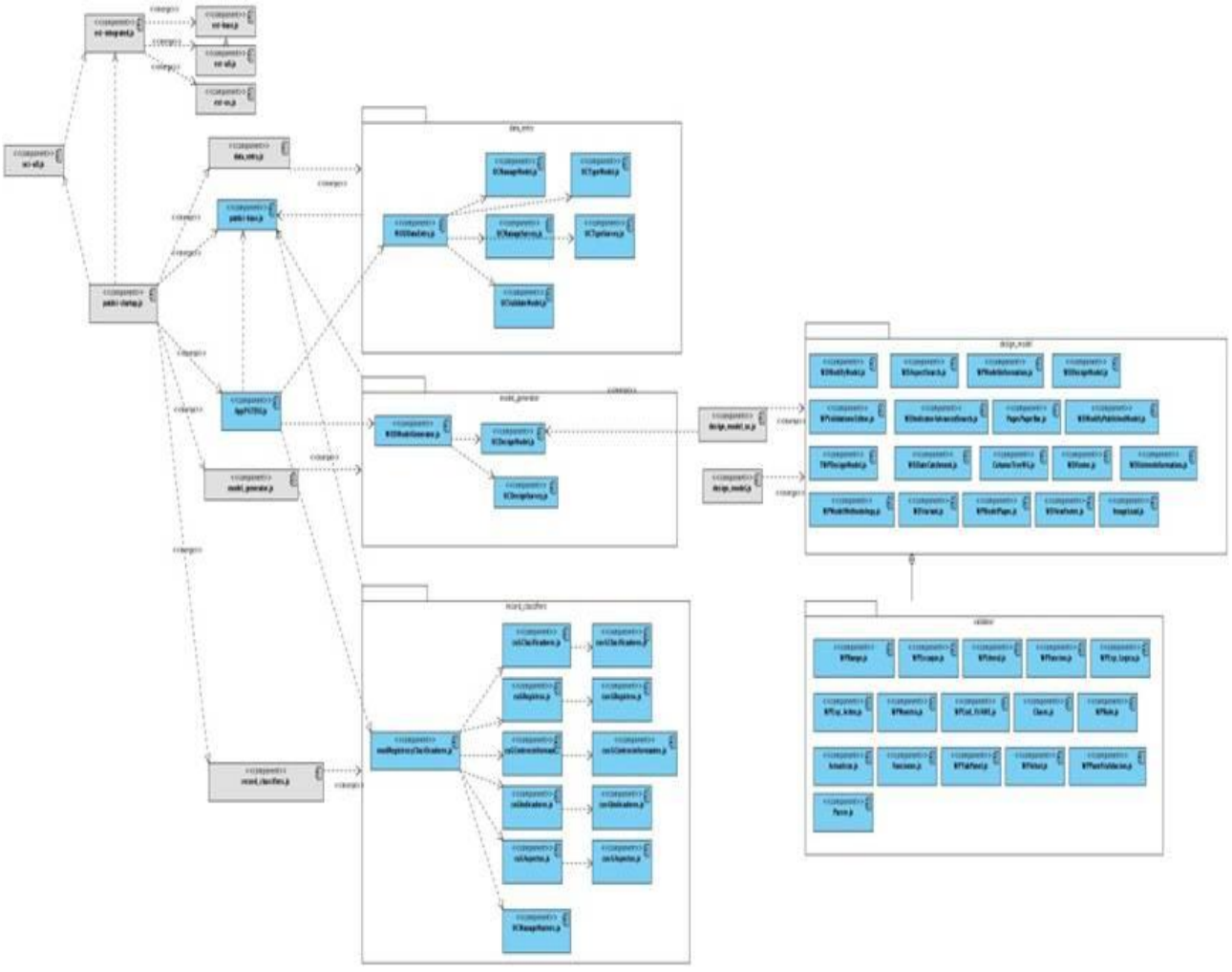


Fig.6: Ejemplo de un modelo de componentes.

2.2 Modelo de dominio de la extensión

Un modelo del dominio captura los tipos más importantes de objetos en el contexto del sistema. Los objetos del dominio representan las “cosas” que existen o los eventos que suceden en el entorno en el que trabaja el sistema. El modelo de dominio se representa fundamentalmente por diagramas de clases en UML. El objetivo del modelado del dominio es comprender y describir las clases más importantes dentro del contexto del sistema. (13)

Diagrama conceptual del dominio

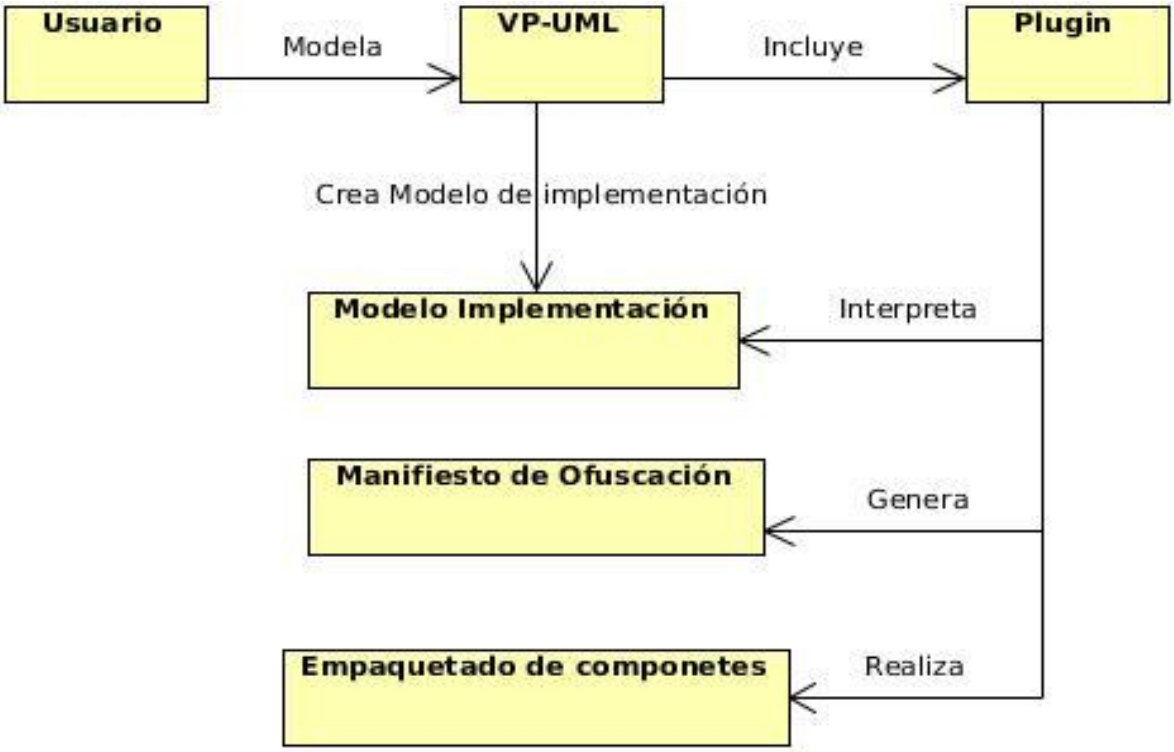


Fig.7: Modelo de dominio de la extensión.

Definición de conceptos del Modelo de dominio

- **Usuario:** es toda aquella persona que va a modelar en la herramienta *"Visual Paradigm for UML".*
- **VP-UML:** es la herramienta de modelado, la cual posibilita crear diversos modelos entre los que se encuentra el modelo de implementación.
- **Plugin:** representa el *plug-in* Vpextmi; contiene todas las funcionalidades que permiten a partir de un modelo de implementación generar el Manifiesto de Ofuscación.
- **Modelo de Implementación:** Este modelo es comprendido por un conjunto de componentes y subsistemas que constituyen la composición física de la implementación del sistema. Fundamentalmente, se describe la relación que existe desde los paquetes y clases del modelo de diseño a subsistemas y componentes físicos. (14) Representa el punto de partida necesario para las posteriores operaciones de generación del manifiesto.
- **Manifiesto de Ofuscación:** Es el código obtenido como resultado de la transformación del modelo de implementación a través del *plug-in.*

- **Empaquetado de componentes:** Es la acción llevada a cabo por el *plug-in* para disminuir el tamaño del manifiesto y en cierto sentido ofuscarlo.

2.3 Especificación de los requisitos del Sistema

2.3.1 Requisitos Funcionales

Los requisitos funcionales (RF) son capacidades o condiciones que el sistema debe cumplir. Los requisitos funcionales definen las funciones que el sistema será capaz de realizar. Expresan la naturaleza del funcionamiento del sistema cómo interacciona el sistema con su entorno y cuáles van a ser su estado y funcionamiento. (15)

Se identificaron los siguientes RF:

RF1. Incluir métodos remotos (load, save).

Descripción: Se incluyen los métodos remotos (load, save) para la carga o descarga del Manifiesto de Ofuscación.

Entrada: Métodos remotos (load, save).

Salida: Métodos remotos incluidos.

RF2. Obtener relación entre componentes.

Descripción: Se obtiene la relación entre dos componentes.

Entrada: Modelo de implementación.

Salida: Relación entre componentes.

RF3. Generar a partir del modelo de implementación el Manifiesto de Ofuscación.

Descripción: Se genera el Manifiesto de Ofuscación a partir del empaquetado del modelo de implementación.

Entrada: Componente empaquetado.

Salida: Se obtiene el Manifiesto de Ofuscación.

RF4. Generar el empaquetado de componentes.

Descripción: se obtiene el código fuente para Ext JS a partir del modelo de implementación y se genera el componente empaquetado.

Entrada: Código fuente

Salida: Componente empaquetado.

2.3.2 Requisitos No Funcionales

Los requisitos no funcionales (RNF) son propiedades o cualidades que el producto debe tener. Estas propiedades o cualidades se refieren a las características que hacen al producto atractivo, usable, rápido o confiable. Por lo general los requisitos no funcionales son fundamentales en el éxito del producto; normalmente están vinculados a los requisitos funcionales, es decir, una vez que se conoce lo que el sistema debe hacer se puede determinar cómo ha de comportarse, qué cualidades o propiedades debe tener. (15)

Se definen los RNF siguientes asumiendo los particulares de la herramienta *"Visual Paradigm for UML",* ya que por sí sola la extensión no cumple funcionalidad*:*

Requisitos de Software

- Se debe tener la herramienta "Visual Paradigm for UML" instalada.
- Se debe tener la máquina virtual de Java OpenJDK 6 ó 7 instalada.

Requisitos de Hardware

- Microprocesador Intel Pentium III con 1, 0 GHz o superior.
- Mínimo 512 MB de RAM (Random Access Memory, por sus siglas en inglés), pero se recomienda 1,0 GB.
- Un mínimo de 400 MB de espacio en disco.

Restricciones del diseño y la implementación

- Herramienta *"Visual Paradigm for UML"* en su versión 8.0 y IDE NetBeans 7.0.
- El lenguaje Java como lenguaje de programación para la implementación, el cual sigue el paradigma de la Programación Orientada a Objetos.
- El sistema operativo a utilizar en el entorno de desarrollo deberá ser: GNU/Linux.

Requisitos de Usabilidad

- Facilidad de uso por parte de los usuarios: La interfaz debe ser lo más descriptiva y amigable posible, permitiendo la fácil interacción con el misma, así como que las operaciones a realizar por los usuarios estén bien descritas, de manera que se puedan entender claramente.
- La interfaz debe tener mensajes contextuales asociados a los objetos.
- Especificación de la terminología utilizada: el sistema debe adaptarse al lenguaje y términos

utilizados por los clientes en la rama abordada con miras a una mayor comprensión por parte del cliente de la herramienta de trabajo.

Requisitos de Soporte

- Grupo de soporte y asesoría: el sistema contará con un grupo de soporte y asesoría al cliente del producto destinado a brindar asesoría y soporte técnico al mismo.

Requisitos de Portabilidad

- El *plug-in* podrá ser instalado y disponer del mismo en diferentes sistemas operativos una vez integrado a la herramienta VP-UML.

2.4 Modelo de Casos de Uso del Sistema (MCUS)

El modelo de casos de uso describe la funcionalidad propuesta del nuevo sistema. Un caso de uso representa una unidad discreta de interacción entre un usuario (humano o máquina) y el sistema. Un caso de uso es una unidad simple de trabajo significativo. Cada caso de uso tiene una descripción que describe la funcionalidad que se construirá en el sistema propuesto. Un caso de uso puede "incluir" la funcionalidad de otro caso de uso o "extender" a otro caso de uso con su propio comportamiento. (16)

Los casos de uso definidos para el sistema son los siguientes:

- **CU1** Generar el Manifiesto de Ofuscación.
- **CU2** Generar el empaquetado de componentes.

2.4.1 Actores del sistema

Un actor es un usuario del sistema. Incluye usuarios humanos y otros sistemas computarizados. Un actor usa un caso de uso para desempeñar alguna porción de trabajo que es de valor para el negocio. El conjunto de casos de uso al que un actor tiene acceso define su rol global en el sistema y el alcance de su acción (16). Además de ser los responsables de realizar actividades que serán automatizadas en el futuro sistema.

Tabla.1: Actores del Sistema

Nombre del Actor	Descripción
Desarrollador	El desarrollador es el actor encargado de realizar el modelo de implementación para generar de forma automática el Manifiesto de Ofuscación.
Visual Paradigm for UML	Es la herramienta de modelado, la cual incluye el *plug-in*.
JSbuilder2	Es la librería interna que posibilita la ofuscación.

2.4.2 Diagrama de Casos de Usos del Sistema.

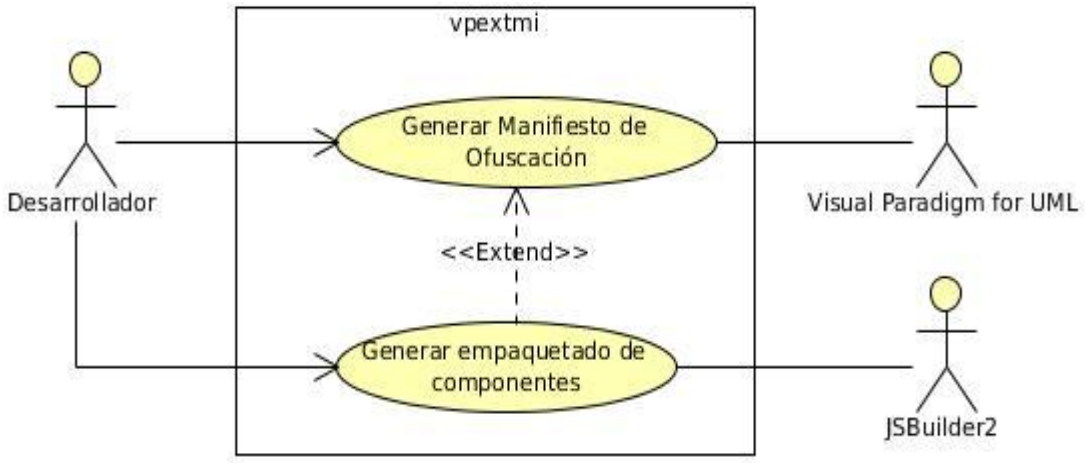


Fig.8: Diagrama de Casos de Usos del Sistema.

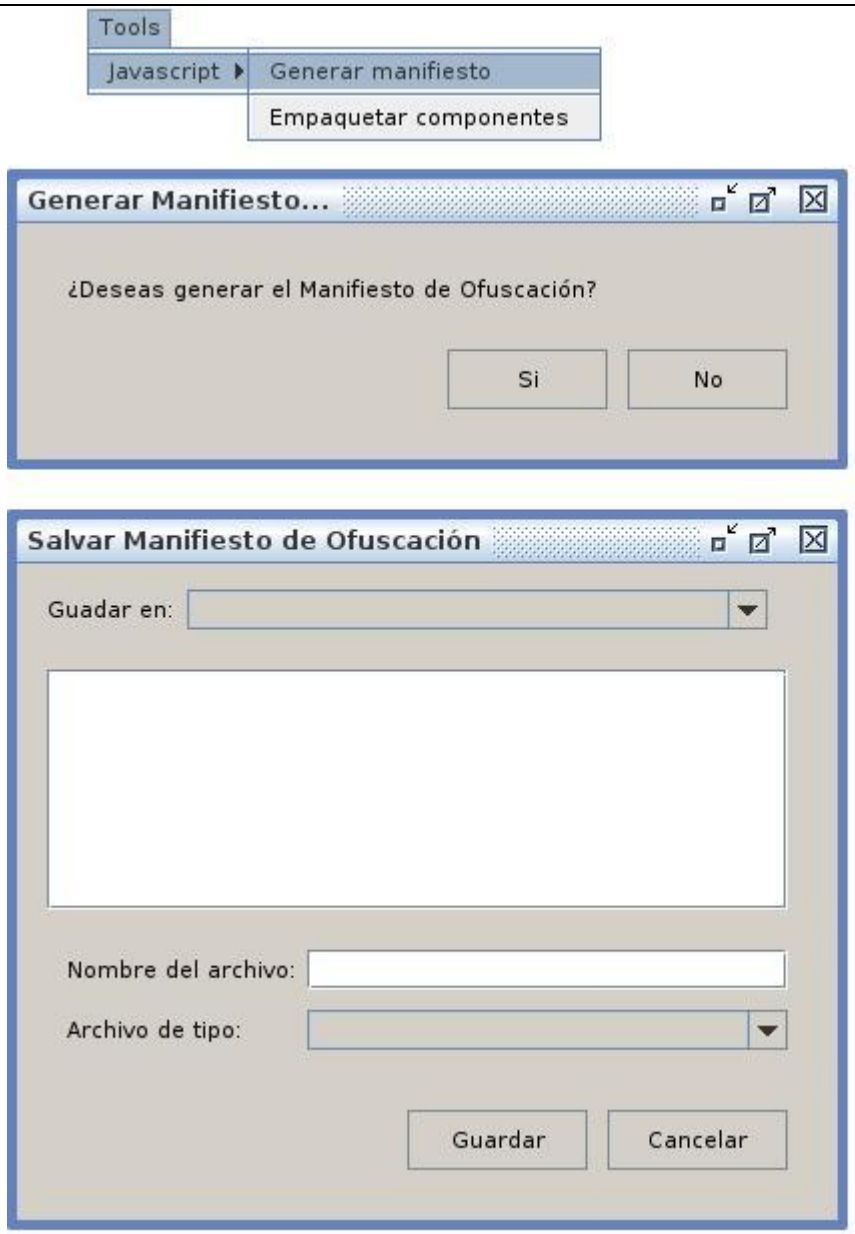
2.4.3 Especificación de los Casos de Usos del Sistema

Tabla.2 Descripción del CU "Generar Manifiesto de Ofuscación"

Caso de Uso:	Generar Manifiesto de Ofuscación
Actores:	Desarrollador
Resumen:	El caso de uso se inicia cuando el desarrollador define los componentes a utilizar para generar el modelo de implementación. El Sistema una vez realizado el modelo de implementación, debe permitir al desarrollador desde el contexto de trabajo obtener el

	Manifiesto de Ofuscación, finalizando así el caso de uso.
Precondiciones:	Se debe haber seleccionado al menos un componente.
Referencias	**RF2**, **RF3**
Prioridad	Crítico

Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El desarrollador realiza el diagrama del modelo de implementación. 2. El desarrollador selecciona los elementos a ofuscar y luego la opción “Generar Manifiesto de Ofuscación” en la barra de herramientas o en el menú contextual.	3. El Sistema transforma el modelo de implementación y muestra una ventana de diálogo con el mensaje “Deseas salvar”. En caso de ser afirmativo el resultado se muestra la interfaz “Guardar”.
4. El desarrollador escoge el destino para salvar y procede a la opción “Salvar”.	5. El Sistema permitirá salvar el manifiesto una vez escogida la dirección destino. 6. El Sistema realiza la opción y termina el caso de uso.
Flujos Alternos	
Acción del Actor	**Respuesta del Sistema**
Si presiona el botón "Cancelar" se deja de realizar automáticamente la acción y finaliza el caso de uso.	



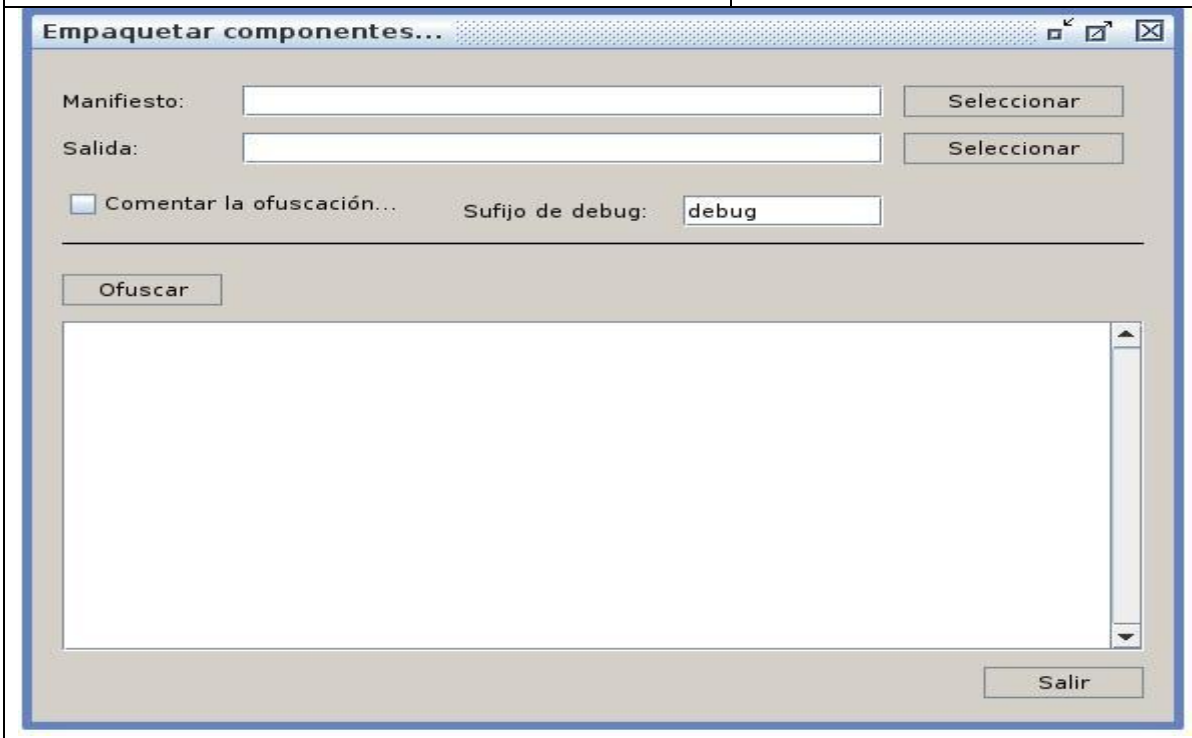
Poscondiciones	Quedó generado el Manifiesto de Ofuscación a partir del modelo de implementación.

Tabla.3: Descripción Textual del CU "Generar empaquetado de componentes"

Caso de Uso:	Generar empaquetado de componentes
Actores:	Desarrollador
Resumen:	El caso de uso se inicia cuando el desarrollador selecciona el Manifiesto de Ofuscación para generar el empaquetado de componentes. El Sistema una vez realizada la carga, debe permitir al desarrollador desde el contexto de trabajo, luego de llenar los datos necesarios generar el empaquetado, finalizando así el caso de uso.
Precondiciones:	Debe existir al menos un componente a empaquetar
Referencias	**RF3, RF4**
Prioridad	Crítico

Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El desarrollador escoge la opción de "Empaquetar Componentes" en la barra de herramientas del menú principal.	2. El Sistema muestra una interfaz con datos referentes al empaquetado para llenar.
3. El desarrollador llena los datos y selecciona la opción "Ofuscar"	4. El Sistema le permitirá al desarrollador, a través de un área de texto incluida en la interfaz ir visualizando el proceso de ofuscación. Concluida la ofuscación se le muestra un mensaje en la misma área de satisfacción.
5. El desarrollador selecciona "Salir" de la interfaz.	6. El Sistema realiza la opción y termina el caso de uso.

Flujos Alternos	
Acción del Actor	**Respuesta del Sistema**
Si presiona el botón "Salir" se deja de realizar automáticamente la acción y finaliza el caso de uso.	



Poscondiciones	Quedó empaquetado el Manifiesto de Ofuscación.

2.4.4 Matriz de Trazabilidad

La matriz de trazabilidad es una técnica que se utiliza para saber cuales requisitos quedan cubiertos por casos de uso ya que permite relacionar los requisitos funcionales a los diferentes elementos del desarrollo.

Tabla.4: Matriz de trazabilidad de los Casos de Usos del Sistema

	RF1	**RF2**	**RF3**	**RF4**
CU1		x	x	
CU2	x			x

2.5 Arquitectura de software

La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución (17). Por lo que un sistema que desee ser extensible y reutilizable, debe tener una arquitectura diseñada con base en ello. Con el propósito y buen uso de la misma se utilizan los patrones que constituyen esquemas genéricos probados para solucionar problemas particulares, el cual es recurrente dentro de un cierto contexto.

2.5.1 Patrones arquitectónicos

Estos patrones de software se encargan de definir la estructura de un sistema, estos a su vez se componen de subsistemas con sus responsabilidades, también tienen una serie de directivas para organizar los componentes del mismo sistema, con el objetivo de facilitar la tarea del diseño de tal sistema. Un patrón arquitectónico se enfoca a dar solución a un problema en específico, de un atributo de calidad, y abarca solo parte de la arquitectura. (18)

2.5.2 Patrones de diseño GOF (Gang Of Four)

Los patrones de diseño son prácticamente modelos que podemos utilizar para exitosamente resolver un determinado problema; puesto que son respuestas que se presentan una y otra vez en nuestras tareas cotidianas, convirtiéndose en soluciones probadas repetidamente.

Los Patrones de Diseño GOF en el campo del Diseño Orientado a Objetos son los más conocidos y usados en la actualidad. Se clasifican en 3 grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento; y respecto a su ámbito en clases y objetos.

Respecto a su propósito

Creacionales: Resuelven problemas relativos a la creación de objetos.

Estructurales: Resuelven problemas relativos a la composición de objetos.

Comportamiento: Resuelven problemas relativos a la interacción entre objetos.

Respecto a su ámbito

Clases: Relaciones estáticas entre clases.

Objetos: Relaciones dinámicas entre objetos. (19)

- **Factory Method (Método de Fabricación)**

El método de fabricación, es del tipo creación. Permite que una clase delegue en sus subclases la creación de objetos sin dejar de saber qué hacer con la instancia creada. Con este método se gana en

flexibilidad, puesto que se facilita, en cuanto a que se hace natural la conexión entre jerarquías de clases paralelas. Las jerarquías de clases paralelas son aquellas que se generan cuando una clase delega algunas de sus responsabilidades en una clase aparte. El patrón método de fabricación establece la relación entre parejas de subclases.

En el desarrollo del *plug-in* este patrón es usado directamente en la implementación, permitiendo la instanciación de los objetos utilizando métodos creacionales, para crear componentes y garantizar a través de estos, que el *plug-in* funcione de forma correcta. En la figura que a continuación se muestra se ejemplifica la aplicación de este método.

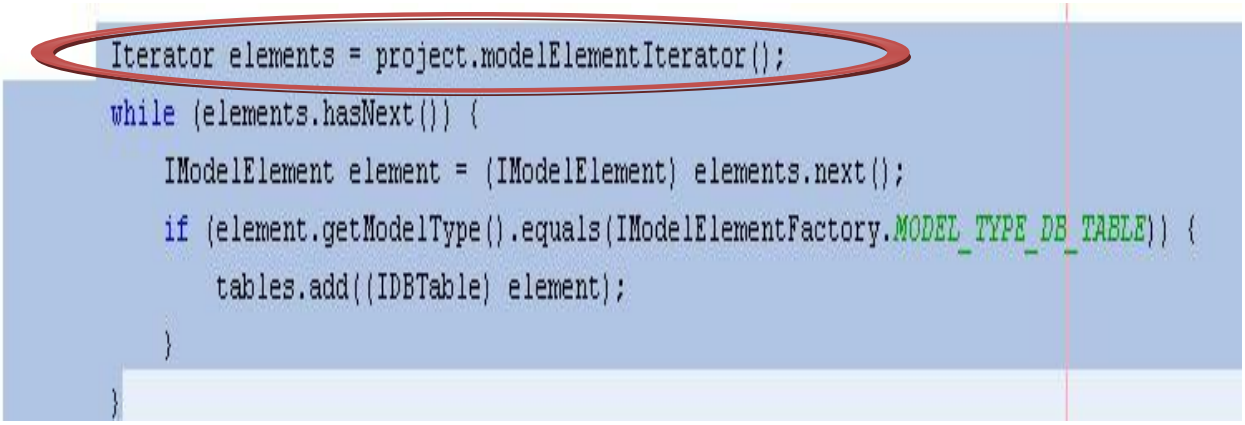
```
public IOperation createOperationGet(String name) {
    IOperation operation = IModelElementFactory.instance().createOperation();
    String operationName = name.substring(0, 1).toUpperCase().concat(name.substring(1));
    operation.setName("get".concat(operationName));
    return operation;
}
```

Fig.9: Ejemplo del método de fabricación.

- **Iterator (Iterador)**

Este patrón es del tipo comportamiento. A nivel de objetos permite acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna. Además posibilita realizar recorridos sobre objetos compuestos independientemente de la implementación de estos. Iterator incrementa la flexibilidad, debido a que brinda nuevas formas de recorrer una estructura con solo modificar el iterador en uso. Permite el paralelismo y la concurrencia, ya que como cada iterador tiene conocimiento del estado en que se encuentra cada momento, le es posible a varios iteradores recorrer simultánea o solapadamente una misma estructura.

Para la elaboración de la extensión este es utilizado directamente en la implantación, para iterar sobre el proyecto, obteniendo una colección de elementos y poder captar de esta forma información a través de operadores secuenciales. A continuación se muestra un ejemplo de este patrón.



```
Iterator elements = project.modelElementIterator();
while (elements.hasNext()) {
    IModelElement element = (IModelElement) elements.next();
    if (element.getModelType().equals(IModelElementFactory.MODEL_TYPE_DB_TABLE)) {
        tables.add((IDBTable) element);
    }
}
```

Fig.10: Ejemplo del método Iterator.

2.6 Modelo del Diseño

El Modelo de Diseño es un modelo de objetos que describe la realización de casos de uso, sirviendo como abstracción del modelo de aplicación con su código fuente. Representa los componentes de aplicación y determina su colocación adecuada y el uso dentro de la arquitectura en general del sistema. Tiene como principales objetivos crear un punto de partida para las futuras actividades de implementación comprendiendo detalladamente los RF y RNF, sistemas operativos, tecnologías de distribución y restricciones relacionadas con el lenguaje de programación.

2.6.1 Modelos de clases del diseño

Diagrama de Clases del Diseño del CU "Generar Manifiesto de Ofuscación".

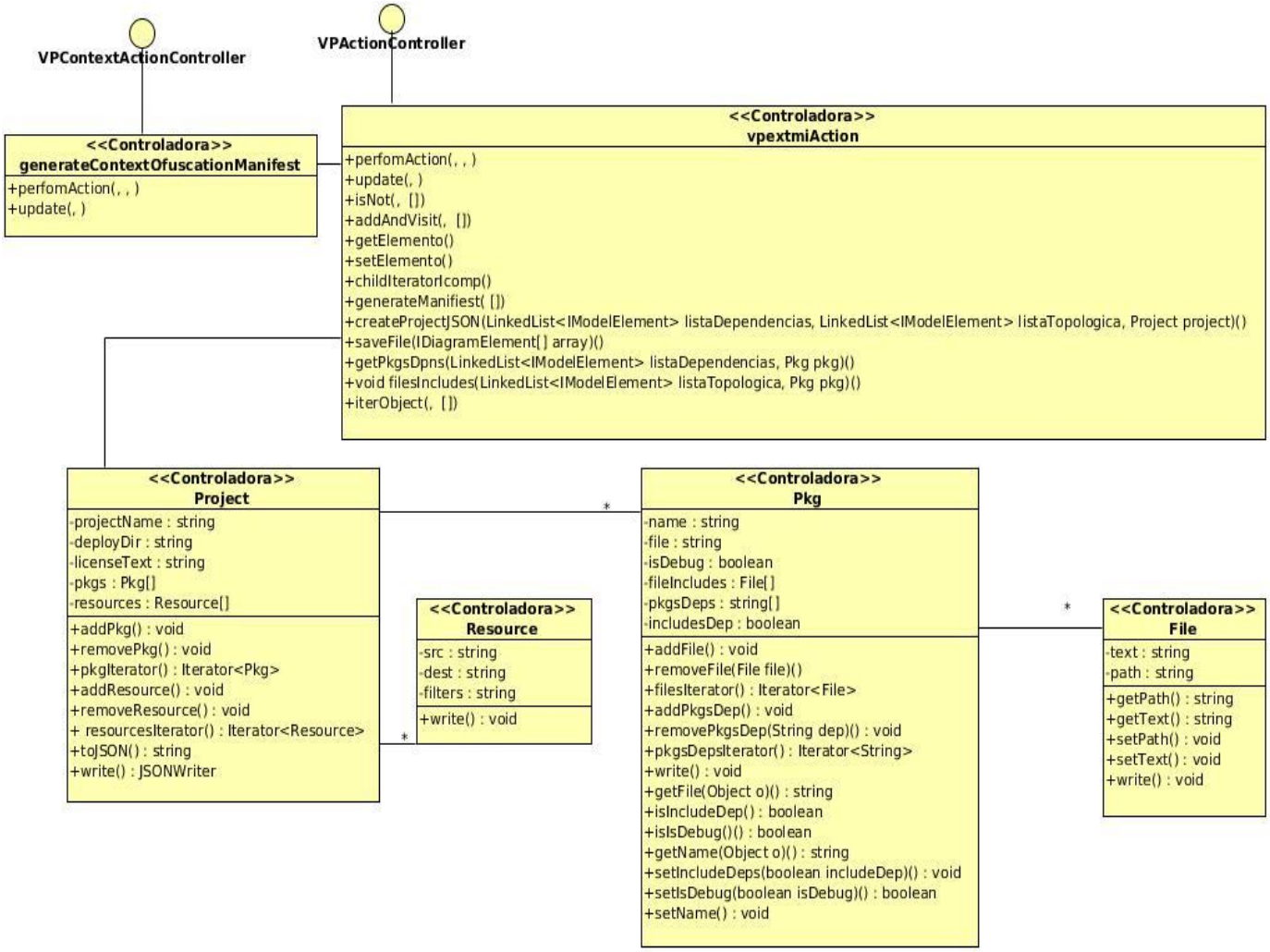


Fig.11: Diagrama de Clases del Diseño del CU "Generar Manifiesto de Ofuscación".

Diagrama de Clases del Diseño del CU "Generar empaquetado de componentes".

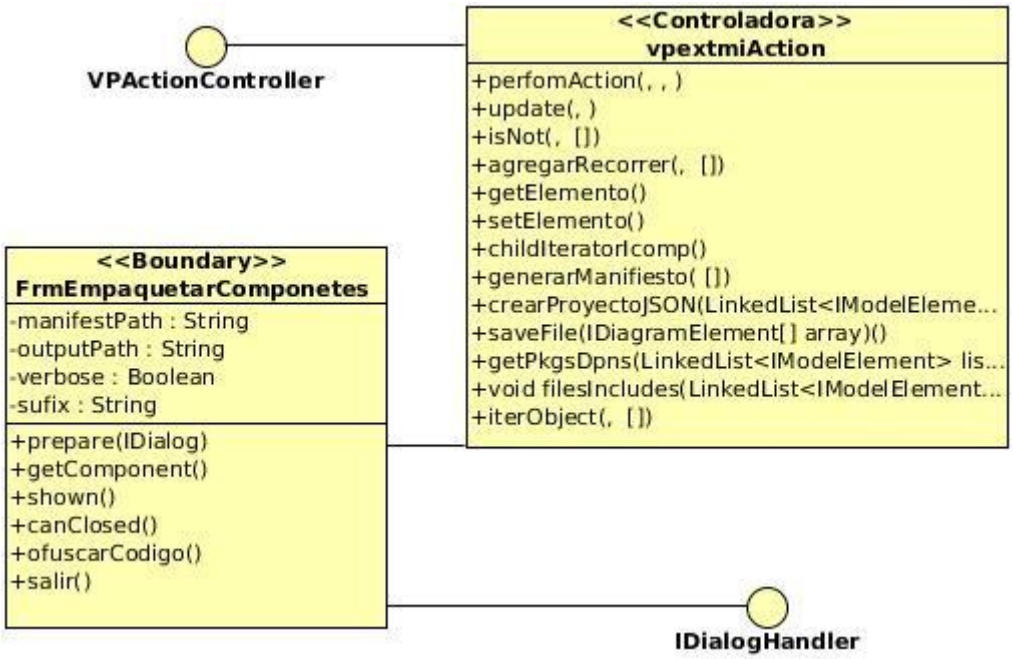


Fig.12: Diagrama de Clases del Diseño del CU "Generar empaquetado de componentes".

Descripción de clases relevantes del diseño.

- **Diagrama de Clases del Diseño para el CU "Generar Manifiesto de Ofuscación".**

Tabla .5 : Descripción de las clases relevantes del diseño para el CU "Generar Manifiesto de Ofuscación".

#	Clase	Tipo de clase	Descripción	Relación con otra clase	Dependencia
01	VPActionController	Interface	Es la clase que brinda el OpenAPI por defecto, la que permite actualizar cada acción realizada a través de la herramienta, así como ejecutar las acciones.	-----	-----

02	VPContextActionController	Interface	Es la clase que brinda el OpenAPI por defecto, la que permite actualizar cada acción realizada a través del contexto, así como ejecutar las acciones en el propio contexto.	-----	-----
03	vpextmiAction	Controladora	Es la clase que permite la generación del Manifiesto de Ofuscación e implementa la principal acción de generar Manifiesto de Ofuscación del *plug-in.*	01 02	Herencia
				05 06 07 08	Dependencia
04	generateContextOfuscationManifest	Controladora	Es la clase que permite la generación del Manifiesto de Ofuscación e implementa la acción contextual de generar Manifiesto de Ofuscación del *plug-in.*	03	Dependencia

05	Project	Controladora	Constituyen en su conjunto la estructura base para la generación de un manifiesto en JSON, el cual luego se aplica al Manifiesto de Ofuscación.	-----	-----
06	File				
07	PKG				
08	Resource				

- **Diagrama de Clases del Diseño para el CU "Generar empaquetado de componentes".**

Tabla .6 : Descripción de las clases relevantes del diseño para el CU "Generar empaquetado de componentes".

#	Clase	Tipo de clase	Descripción	Relación con otra clase	Dependencia
01	VPActionController	Interface	Es la clase que brinda el OpenAPI por defecto, la que permite actualizar cada acción realizada a través de la herramienta, así como ejecutar las acciones.	-----	-----
02	IDialogHandler	Interface	Es la clase que brinda la API de Java por defecto, la que permite manejar las propiedades de	-----	-----

			los diálogos a través de los paneles.		
03	vpextmiAction	Controladora	Es la clase que permite la generación del Manifiesto de Ofuscación e implementa la principal acción de generar Manifiesto de Ofuscación del *plug-in.*	01	Herencia
04	FrmEmpaquetarComponentes	Controladora	Es la clase que permite realizar el empaquetado de componentes.	02	Herencia
				03	Dependencia

2.7 Diagramas de Interacción

Los diagramas de interacción describen el comportamiento de un sistema mostrando la interacción entre usuarios, sistemas y subsistemas mediante la secuencia de mensajes. Entre estos diagramas se encuentran los de secuencia y colaboración. Los diagramas de secuencia, para el diseño del sistema propuesto muestran la secuencia de mensajes entre objetos durante un escenario concreto. Mientras que los diagramas de colaboración son útiles en la fase exploratoria en la identificación de objetos y la interacción entre estos.

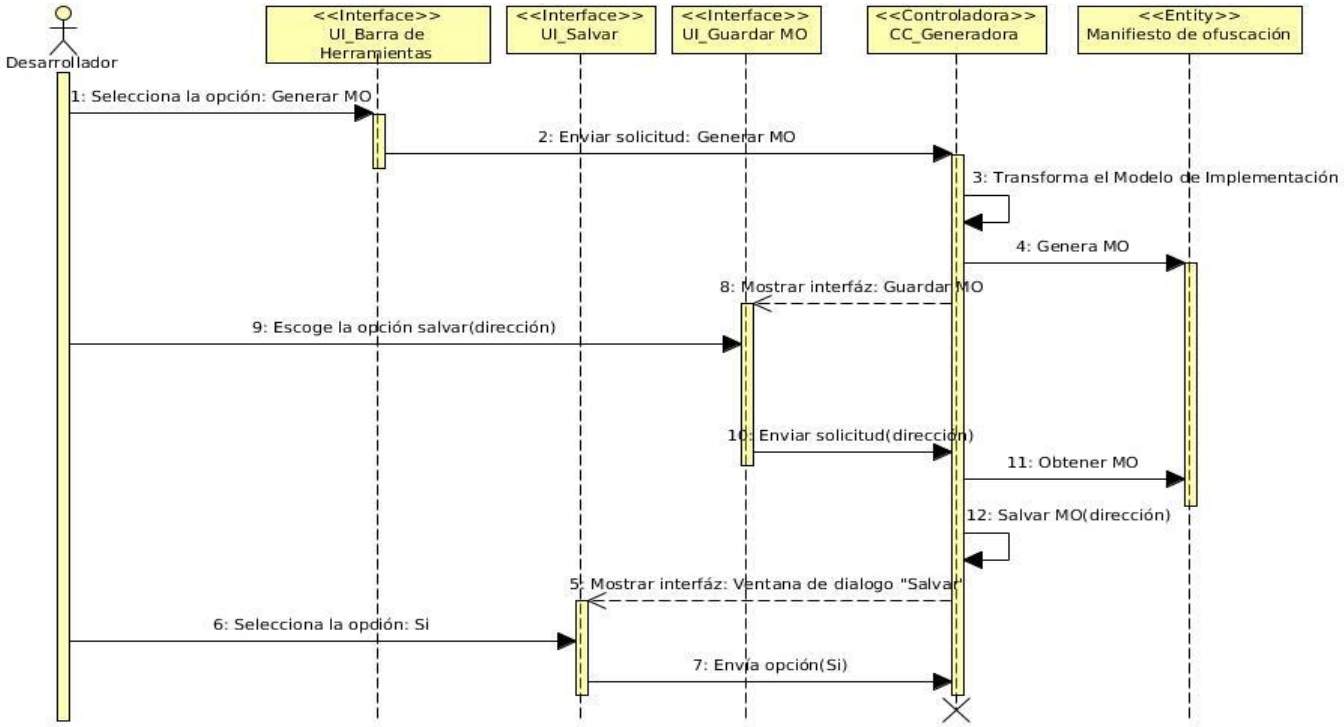


Fig.13: Diagrama de Secuencia para el CU “Generar Manifiesto de Ofuscación”.

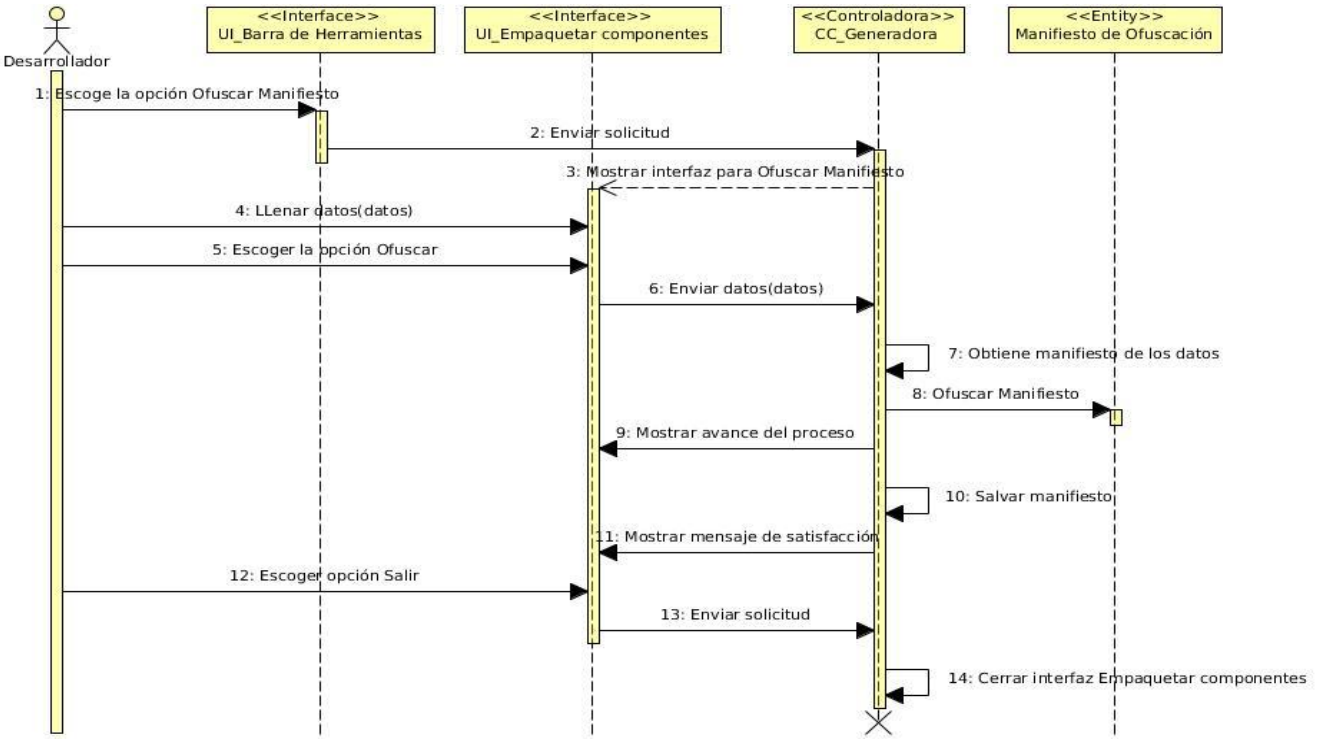


Fig.14: Diagrama de Secuencia para el CU “Generar empaquetado de componentes”.

2.8 Descripción del Proceso de Transformación de los Modelos en *"Visual Paradigm for UML"*.

El proceso de transformación de los modelos en la herramienta *VP-UML* se realiza a partir de la obtención de los componentes presentes en los diagramas.

Tabla.7: Descripción de los componentes, para el proceso de transformación.

Diagramas	
Componentes	**Descripción**
IDBTable	Interfaz que define una tabla o entidad en el diagrama entidad relación.
IRelationShip	Interfaz que define una relación entre cualquier objeto IModelElement.
IDepedency	Interfaz que define una relación de dependencia entre cualquier objeto IModelElement.
IClass	Interfaz que define una clase en un diagrama de clases.
IAttribute	Interfaz que define los atributos de una clase.
IOpreration	Interfaz que define las operaciones asociadas a la clase.
IAssociation	Interfaz que define la relación entre clases de asociación.
IDiagramUIModel	Interfaz que permite visualizar los diagramas.
IDiagramElement	Interfaz que define los elementos de un diagrama.
ApplicationManager	Interfaz que permite el control de la aplicación.

2.9 Conclusiones parciales.

En el presente capítulo se definieron los conceptos más importantes para el desarrollo de la aplicación, y se especificaron los requisitos funcionales y no funcionales para el correcto funcionamiento de esta. Se identificaron, actores, casos de uso y la relación existente entre ellos reflejada en el diagrama de casos de uso del sistema. Se describieron detalladamente los casos de uso del sistema para obtener así un mayor entendimiento de los requisitos funcionales previamente establecidos. Se especificó la estructura del sistema, a través de los diagramas de clases del diseño y los diagramas de secuencia. Se describieron además los patrones de diseño empleados para el proceso de desarrollo del *plug-in*.

CAPÍTULO 3 IMPLEMENTACIÓN Y PRUEBA

En el presente capítulo se realiza una extensión de la herramienta "*Visual Paradigm for UML*", mediante un *plug-in* que permita automatizar el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación. El mismo tendrá aplicado un perfil de UML definido por medio de estereotipos, que expresan la semántica del perfil, así como, la generación de código fuente partiendo de la interpretación de los modelos de UML llevados al lenguaje Java Script.

3.1 Modelo de implementación

El modelo de implementación compuesto por los diagramas de despliegues y componentes es una descripción de cómo los elementos de diseño son implementados en componentes. Detalla los componentes a construir y su organización en nodos físicos para el funcionamiento del sistema.

3.1.1 Diagrama de componentes

Los diagramas de componentes son utilizados para modelar la vista estática de un sistema estructurando el modelo de implementación que muestra la organización y las dependencias existentes entre conjuntos de componentes. Estos describen los elementos físicos del sistema y sus relaciones, además de que muestran las opciones de realización incluyendo código fuente, binario y ejecutable. Los componentes representan cualquier elemento físico que forma parte del sistema, permitiéndoles ser representados por nodos y sus operaciones solo se pueden alcanzar a través de interfaces. Los componentes pueden ser simples archivos, paquetes, bibliotecas cargadas dinámicamente, tablas.

El presente modelo de implementación describe cada uno de los componentes asociados al diseño de clases propuesto para la construcción del *plug-in* para la herramienta de modelado *VP-UML*, así como la relación de dependencia entre los componentes que la integran.

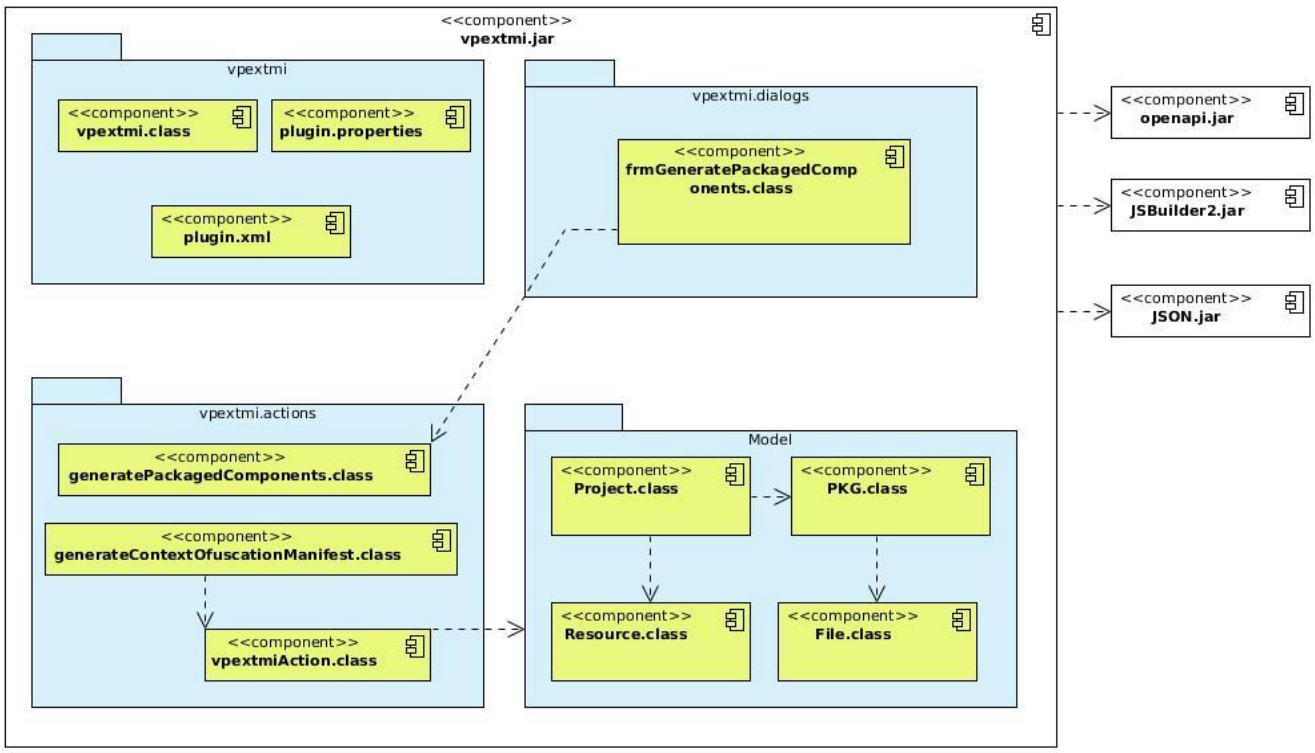


Fig.15: Diagrama de Componentes.

Descripción de los componentes más relevantes:

Nombre del plugin: "*vpextmi"* representa la aplicación el modelado en *"Visual Paradigm for UML"* para el marco *Ext JS* con la aplicación del Desarrollo Dirigido por Modelos.

Paquete vpextmi: Agrupa a las clases asociadas a la configuración del *plug-in* como son: plugin.xml, vpextmi.java y plugin.properties definidas en la estructura que propone la herramienta para la extensión.

Paquete vpextmi.actions: Agrupa las clases referentes a las acciones principales y contextuales como son: GenerateContextOfuscationManifest.java, vpextmiAction.java y GenerateEComponent.java.

Paquete vpextmi.dialog: Agrupa los formularios presentes en la aplicación como es el caso de FrmEmpaquetarComponentes.java el cual funciona a través de la biblioteca JSBuilder2.jar.

Paquete JSON: Constituye la base para la generación del Manifiesto de Ofuscación a través de la estructura del manifiesto en JSON, el cual describe cómo se van a generar los paquetes de salida a través del código fuente. La estructura del Manifiesto se encuentra dada por las clases: Project.java, PKG.java, File.java y Resource.java.

Plugin.xml: Su función consiste en la configuración del *plug-in.* Define el nombre del *plug-in*, descripción, proveedor, librerías a utilizar y las acciones a ejecutar.

Vpextmi.java: Su función está dirigida a la carga y descarga del *plug-in* mediante la clase VPPluginInfo que provee el openapi.jar, capturando la información definida en el archivo **plugin.xml.** vpextmi.java implementa la interfaz VPPlugin definida en el openapi.jar la misma implementa los método **loaded ()** y **unloaded ()** para su función.

Plugin.properties: Su función es definir propiedades, asociadas a los eventos y acciones del *plug-in*.

GenerateContextOfuscationManifest.java: Es la clase referente al control de las acciones de contexto la cual implementa la interfaz VPContextActionController definida por el openapi.jar para el manejo de las acciones a nivel de contexto.

GenerateEComponent.java: Es la clase referente al control de las acciones a nivel de herramientas, la misma implementa la interfaz VPActionController definida por el openapi.jar para el manejo de acciones de herramientas.

vpextmiAction.java: Es la clase referente al control de las acciones a nivel de herramientas. La misma implementa la interfaz VPActionController definida por el openapi.jar para el manejo de acciones de herramientas. Además, esta clase tiene la funcionalidad de generar el Manifiesto de Ofuscación.

FrmEmpaquetarComponentes.java: Representa el formulario asociado a la generación del empaquetamiento de componentes mediante la implementación de la interfaz IDialogHandler.java.

Project.java: Su función está dirigida a la estructura para la creación de un proyecto en el Manifiesto de Ofuscación haciendo uso del patrón Visitor.

File.java: Su función está dirigida a la estructura para la creación de un fichero para un proyecto en el manifiesto de JSON haciendo uso del patrón Visitor.

PKG.java: Su función está dirigida a la estructura para la creación de un paquete para un proyecto en el manifiesto de JSON haciendo uso del patrón Visitor.

Resource.java: Su función está dirigida a la estructura para la creación de un recurso para un proyecto en el manifiesto de JSON haciendo uso del patrón Visitor.

JSBuilder2.jar: Librería que permite procesar un manifiesto en JSON y generar el empaquetamiento de este.

openapi.jar: Librería que permite al plug-in trabajar con los recursos de VP-UML para su posterior integración.

JSON.jar: Librería que permite crear un manifiesto en JSON.

3.2 Modelo de pruebas del software

Las pruebas constituyen elementos críticos para medir la calidad del software. El objetivo de esta etapa es determinar la calidad del producto de software desarrollado; para ello se deben efectuar medidas que permitan comprobar el grado de cumplimiento de las especificaciones iniciales del sistema. La prueba de software constituye un proceso enfocado a evaluar la lógica interna del software y sus funciones externas, además, es un proceso de ejecución de un programa con la intención de descubrir con éxito un error hasta el momento no detectado.

Nivel de Prueba

Para comprobar y verificar la funcionalidad correcta del *plug-in*, se realizarán **Pruebas de Desarrollador** puesto que es el primer nivel de prueba y estas son diseñadas e implementadas por el equipo de desarrollo.

Técnica de prueba

La técnica de prueba utilizada es la técnica de **Caja Negra** también conocidas como prueba de comportamiento, las cual se lleva a cabo sobre la interfaz del software. Estas pruebas se centran en la especificación del componente o aplicación a ser probada para elaborar los casos de prueba. El objetivo de estas pruebas consiste en demostrar que las funciones del software son operativas, es decir, que el conjunto de condiciones de entrada ejercite todos los requisitos funcionales del programa y sea aceptado de forma adecuada en la producción de resultados correctos.

Para el diseño de los casos de prueba de Caja Negra se utilizó la **Técnica de la Partición de Equivalencia** la cual divide el campo de entrada de un programa en variables de equivalencia con juegos de datos de entrada y salida. Las variables de equivalencia contituyen un conjunto de estados válidos y no válidos para las condiciones de entrada de un programa. Se definen dos tipos de variables de equivalencia, las *válidas*, que representan entradas válidas al programa, y las *no válidas*, que representan valores de entrada erróneos, aunque pueden existir valores no relevantes a los que no sea necesario proporcionar un valor real de dato.

Tipo de prueba

Se utilizará las **Pruebas Funcionales,** con el fin de asegurar el trabajo apropiado de los requisitos funcionales, así como entrada de datos, procesamiento y obtención de resultados. Estas pruebas se centran en los requisitos funcionales y casos de uso a través de la técnica de Caja Negra.

Las pruebas funcionales se encuentran compuestas por las **Pruebas de Funcionalidad**. Estas pruebas se realizan fijando su atención en la validación de las funciones, métodos, servicios y Casos de Uso. Además permitió analizar cada funcionalidad implementada para verificar el cumplimiento de todos los requisitos establecidos y de esta manera poder satisfacer las necesidades existentes.

Casos de prueba

Las pruebas de Caja Negra son aplicadas mediante los casos de prueba que constituyen un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para cumplir un objetivo en particular o una función esperada.

A continuación se presentan los casos de pruebas, para los casos de uso “Generar Manifiesto de Ofuscación” y “Generar empaquetado de componentes” con el objetivo de comprobar que el *plug-in* funcione de forma correcta, donde:

V: indica válido, I: indica inválido, NA: que no es necesario proporcionar un valor del dato en este caso, ya que es irrelevante.

Caso de prueba: Generar Manifiesto de Ofuscación.

Tabla.8: Descripción de las variables para el CU “Generar Manifiesto de Ofuscación”.

No	Nombre de campo	Clasificación	Valor Nulo	Descripción
1	nombre de archivo	Campo de texto	no	La variable debe ser cualquier combinación de letras y números.

Tabla.9: Matriz de datos para el CU “Generar Manifiesto de Ofuscación”.

Escenario	Descripción	Variable1	Respuesta del sistema	Flujo central
EC 1.1 Guardar Manifiesto con campos vacío.	El usuario no especifica el texto de identificación.	V(campo vacío)	El sistema no procede con la ejecución del CU.	1-El usuario selecciona en la barra de herramientas el menú Reporte. 2-Luego selecciona la
EC 1. Guardar Manifiesto con datos inválidos.	El usuario deberá especificar datos de entrada.	I(,” @.; <*)	El sistema no procede con la ejecución del CU, puesto que no reconoce caracteres	

			extraños al comienzo del dato.	opción generar Manifiesto de Ofuscación dentro del menú funciones de JavaScript ubicado en Reporte.
EC 1.3 Guardar Manifiesto datos válidos.	El usuario deberá especificar datos de entrada.	V(k1k , kk,12)	El sistema guarda el Manifiesto en el destino seleccionado con el dato de identificación.	

Caso de prueba: Generar Manifiesto de Ofuscación.

Tabla.10: Descripción de las variables para el CU "Generar empaquetado de componentes".

No	Nombre de campo	Clasificación	Valor Nulo	Descripción
1	Manifiesto	text field	no	Recoge la dirección origen del manifiesto a ofuscar.
2	Salida	text field	no	Recoge la dirección destino del manifiesto a ofuscar.
3	Comentario	checkbox	si	Informa si desea realizar comentario en el proceso del empaquetamiento.
4	Debug	text field	si	Se introduce el sufijo de debug.

Tabla.11: Matriz de datos para el CU "Generar empaquetado de componentes".

Escenario	Descripción	Variable 1	Variable 2	Variable 3	Variable 4	Respuesta del sistema	Flujo central
EC 1.1 Empaquetar component	El usuario deberá dejar campo vacío.	V()	V(home/ dir)	V(seleccionada)	V(1)	Se muestra un mensaje	1-El usuario seleccion a en la

es con campos vacíos.	El usuario deberá dejar campo vacío.	V(home/dir)	V()	V(seleccionada)	V(1)	indicando llenar campo obligatorio.	barra de herramientas el menú Reporte. 2-Luego seleccion a la opción generar empaquetado de componentes dentro del menú funciones de JavaScript ubicada en Reporte.
	El usuario deberá dejar campo vacío.	V()	V()	V(seleccionada)	V(1)		
	El usuario deberá dejar campo sin seleccionar.	V(home/dir)	V(home/dir)	V(sin seleccionar)	V(1)	Se procede a ejecutar el CU según los datos.	
	El usuario deberá dejar campos sin seleccionar.	V(home/dir)	V(home/dir)	V(seleccionada)	V()		
	El usuario deberá dejar campos sin seleccionar.	V(home/dir)	V(home/dir)	V(sin seleccionar)	V()		
EC 1.2 Empaquetar componentes con campos llenos.	El usuario deberá llenar todos los campos.	V(home/dir)	V(home/dir)	V(sin seleccionar)	V(1)	Se procede a ejecutar el CU según los datos.	
EC 1.3 Empaquetar componentes con campos inválidos.	El usuario deberá entrar datos inválidos.	V(home/dir)	V(home/dir)	V(sin seleccionar)	V(cadena)	Se muestra un mensaje indicando llenar campo obligatorio.	

Después de realizar las pruebas de Caja Negra mediante los casos de prueba asociados a cada caso de uso, se comprobó el correcto funcionamiento del *plug-in* y la correcta validación de los campos, verificando que solo se acepten los caracteres válidos para los mismos. Cada dificultad detectada en el desarrollo del *plug-in* y resueltas a raíz del trabajo continuo de los desarrolladores, fueron recogidas en

la planilla de No Conformidades (NC). En la siguiente tabla se muestra un resumen de las dificultades encontradas:

PD: Pendiente **RA:** Resuelta

Tabla .12: Resumen de las No Conformidades como resultado de las pruebas aplicadas.

Fecha	Versión	Caso de Prueba	Cantidad de NC	Cantidad De NC PD	Cantidad de NC RA
20-5-2012	1.0	CU-Generar Manifiesto de Ofuscación.	1	1	-
20-5-2012	1.0	CU-Generar Empaquetado de componentes.	2	2	-
25-5-2012	1.1	CU-Generar Manifiesto de Ofuscación.	1	-	1
25-5-2012	1.1	CU-Generar Empaquetado de componentes.	2	-	2

Para más información referente a las dificultades encontradas en el proceso de desarrollo del plugin: ver **(Anexo 1).**

3.3 Conclusiones parciales

En este capítulo se realizó el modelo de implementación con el propósito de mostrar los componentes del sistema y sus relaciones, a través del diagrama de componentes. Además, se realizaron pruebas para validar que los requisitos fueron implementados correctamente a través del método de Caja Negra, aplicando la técnica de partición de equivalencia.

CONCLUSIONES

Como resultado del presente trabajo de diploma y para dar cumplimiento a los objetivos propuestos se puede concluir que se obtuvo una extensión de la herramienta CASE *Visual Paradigm for UML* para automatizar el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación. Tal extensión beneficia al proceso de desarrollo de software en DATEC, particularmente al departamento de Integración de Soluciones, dado que a partir del Modelo de Implementación se generarán los componentes empaquetados, cuando antes debería ser escritos por especialistas experimentados en un trabajo tedioso y repetitivo no exento de errores y que en muchos casos retrasaba el cumplimiento de los cronogramas previstos. Lo anterior puede resumirse concretamente en:

- Se elaboró un perfil de UML para reflejar los elementos de empaquetado de componentes.
- Se realizó el análisis, diseño e implementación de la extensión de *Visual Paradigm* capaz de automatizar el empaquetado de componentes *Ext JS* a partir del Modelo de Implementación.
- Se probó el correcto funcionamiento de la herramienta, a través de las pruebas de caja negra realizadas mediante los casos de prueba.

RECOMENDACIÓN

Este trabajo trajo consigo la elaboración de una extensión de la herramienta *Visual Paradigm for UML* para el empaquetado de componentes de *Ext JS* a partir del Modelo de Implementación, desarrollándose el cumplimiento de los objetivos del mismo. A pesar de ello nuevas ideas en su perfeccionamiento han ido surgiendo por lo que se recomienda:

- Incorporar como característica a la extensión la generación del Diagrama de Componentes a partir de un manifiesto de *JSBuilder.*

REFERENCIAS BIBLIOGRÁFICAS

1. **González, Abdiel E. Cáceres.** Programación orientada a objetos. *Cinvestav.* [Online] [Cited: 11 20, 2011.] http://computacion.cs.cinvestav.mx/~acaceres/courses/udo/poo/files/slides/POO-02.pdf.

2. **L.Nahuel, L.Ocaranza, M.Pinasco.** Herramientas de soporte al proceso de desarrollo dirigido por modelos. *JIDIS.* [Online] [Cited: 11 4, 2011.] http://www.jidis.frc.utn.edu.ar/papers/45308ce78aafdf2092719a59d01c.pdf.

3. La-Diversidad-de-Los-Objetos-1-1-Caracteristicas. *Scribd.* [Online] [Cited: 11 2, 2011.] http://es.scribd.com/doc/25171397/1-La-Diversidad-de-Los-Objetos-1-1-Caracteristicas.

4. Arquitectura de Software Dirigida por Arquitectura de Software Dirigida por Modelos. *IE.* [Online] [Cited: 11 4, 2011.] www.ie.inf.uc3m.es/grupo/docencia/reglada/ASDM/Presentacion.pdf.

5. ModelosDeProcesoDeSoftware. *Tecnologico.* [Online] [Cited: 11 5, 2011.] http://www.mitecnologico.com/Main/ModelosDeProcesoDeSoftware..

6. **Quintero, Juan Bernardo y Anaya, Raquel.** MDA y el papel de los modelos en el proceso de desarrolo de software. *SCIELO.* [Online] [Cited: 11 16, 2011.] http://www.scielo.org.co/scielo.php?pid=S1794-12372007000200011&script=sci_arttext.

7. **Lorente, Abel Ernesto.** Conceptos UML. *Diseño UML.* [Online] [Cited: 11 17, 2011.] http://diseouml2009.blogspot.com/2009/04/que-es-conceptos-uml.html.

8. lo bueno lo malo y lo feo. *Antartec Blogs.* [Online] [Cited: 11 17, 2011.] http://blogs.antartec.com/desarrolloweb/2008/10/extjs-lo-bueno-lo-malo-y-lo-feo/.

9. **Flores, Carmina Lizeth Torres.** Establecimiento de una Metodología de Desarrollo de Software. *Fit.* [Online] [Cited: 11 19, 2011.] http://fit.um.edu.mx/CIDET/publicaciones/COMP-004-2008%20Establecimiento%20de%20una%20Metodolog%C3%ADa%20de%20Desarrollo%20de%20Software%20para%20la%20Universidad%20de%20Navojoa%20Usando%20OpenUP.pdf.

10. OpenUP como alternativa metodológica para proyectos pequeños de software. *OpenUp.* [Online] [Cited: 11 20, 2011.] http://epf.eclipse.org/wikis/openup/.

11. Características del lenguaje. *Tikal.* [Online] [Cited: 11 20, 2011.] http://tikal.cifn.unam.mx/~jsegura/LCGII/java3.htm.

12. Bienvenido a NetBeans y www.netbeans.org. *NETBEANS.* [Online] [Cited: 11 21, 2011.] http://netbeans.org/index_es.html.

13. Modelo de Dominio. *Scribd.* [Online] [Cited: 11 22, 2011.] http://es.scribd.com/doc/21296187/40/Modelo-del-Dominio.

14. Modelo de Implementación. *MeRinde.* [Online] [Cited: 11 23, 2011.]

http://merinde.net/index.php?option=com_content&task=view&id=495&Itemid=291.

15. Especificación de Requisitos. *Elvex.* [Online] [Cited: 12 1, 2011.] http://elvex.ugr.es/idbis/db/docs/design/2-requirements.pdf.

16. El Modelo de Caso de Uso. *SPARXSYSTEMS.* [Online] [Cited: 12 1, 2011.] http://www.sparxsystems.com.ar/resources/tutorial/use_case_model.html.

17. **Reynoso, Carlos Billy.** Introducción a la Arquitectura de Software. *WillyDev.* [Online] [Cited: 12 2, 2011.] http://www.willydev.net/descargas/prev/IntroArq.pdf.

18. Patrones Arquitectonicos. *buenas tareas.* [Online] [Cited: 12 6, 2011.] http://www.buenastareas.com/ensayos/Patrones-Arquitectonicos/1069013.html.

19. Patrones de diseño. *Departamento de Informática.* [Online] [Cited: 12 7, 2011.] http://www.infor.uva.es/~felix/datos/priii/tr_patrones-2x4.pdf.

20. Patrones de"Gang of Four". *Is.* [Online] [Cited: 12 7, 2011.] http://is.ls.fi.upm.es/docencia/proyecto/docs/patrones_gof.pdf.

BIBLIOGRAFÍAS

1. Ejecutar Shell Scripts desde Java . *Java.Lang.NullPointer.* [Online] [Cited: 5 7, 2012.] http://javalangnullpointer.wordpress.com/.

2. Netbeans: Un IDE de los más completos de la actualidad. *RINCONTECNO.* [Online] [Cited: 12 12, 2011.] http://php.rincontecno.com.

3. Convert a JSON string to object in Java? *Stackoverflow.* [Online] [Cited: 5 5, 2012.] http://stackoverflow.com/questions/1395551/convert-a-json-string-to-object-in-java.

4. Visual Paradigm Discussion Forum. *Visual Paradigm.* [Online] http://forums.visual-paradigm.com/jforum.html?module=search&action=search&clean=1&search_terms=all&search_forum=&search_cat=&sort_by=p.post_time&sort_dir=DESC&search_keywords=menu.

5. UML Y Patrones by Craig Larman. *Scribd.* [Online] [Cited: 6 8, 2012.] http://es.scribd.com/doc/457980/UML-y-Patrones-by-Craig-Larman.

6. Desarrollo ágil de Softwarecon Arquitecturas Dirigidas por Modelos. *Scribd.* [Online] [Cited: 6 8, 2012.] http://es.scribd.com/doc/71854769/372/BIBLIOGRAFIA-Y-REFERENCIAS-WEB.

7. **Díez, Jorge Manrubia.** Desarrollo de Software Dirigido por. *jorgemanrubia.* [Online] [Cited: 6 8, 2012.] jorgemanrubia.net/blog/wp-content/plugins/.../download.php?id=2.

8. Modelos De Desarrollo De Software. *PDF IN.* [Online] [Cited: 6 8, 2012.] http://www.pdfin.com/pdf_modelos_de_desarrollo_de_software.html.

9. Conferencia #7.Disciplina de Prueba. Ingeniería de Software I. *EVA.* [Online] [Cited: 6 7, 2012.] http://eva.uci.cu.

10. Prueba Caso De Prueba Defecto Falla Error Verificacion Validacion. *Tecnologico.* [Online] [Cited: 6 7, 2012.] http://www.mitecnologico.com/Main/PruebaCasoDePruebaDefectoFallaErrorVerificacionValidacion.

11. Pruebas_Funcionales. *Carolina.* [Online] [Cited: 6 6, 2012.] http://carolina.terna.net/ingsw3/datos/Pruebas_Funcionales.pdf.

12. TiposPruebasSoftware. *CETIC.* [Online] [Cited: 6 6, 2012.] http://www.cetic.guerrero.gob.mx/pics/art/articles/113/file.TiposPruebasSoftware.pdf.

13. Patrones de"Gang of Four". *Is.* [Online] [Cited: 12 7, 2011.] http:// is.ls.fi.upm.es/docencia/proyecto/docs/patrones_gof.pdf.

14. Patrones de diseño. *Departamento de Informática.* [Online] [Cited: 12 7, 2011.] http://www.infor.uva.es/~felix/datos/priii/tr_patrones-2x4.pdf.

15. Patrones Arquitectonicos. *buenas tareas.* [Online] [Cited: 12 6, 2011.] http://www.buenastareas.com/ensayos/Patrones-Arquitectonicos/1069013.html.

16. **Reynoso, Carlos Billy.** Introducción a la Arquitectura de Software. *WillyDev.* [Online] [Cited: 12 2, 2011.] http://www.willydev.net/descargas/prev/IntroArq.pdf.

17. El Modelo de Caso de Uso. *SPARXSYSTEMS.* [Online] [Cited: 12 1, 2011.] http://www.sparxsystems.com.ar/resources/tutorial/use_case_model.html.

18. Especificación de Requisitos. *Elvex.* [Online] [Cited: 12 1, 2011.] http://elvex.ugr.es/idbis/db/docs/design/2-requirements.pdf.

19. Modelo de Implementación. *MeRinde.* [Online] [Cited: 11 23, 2011.] http://merinde.net/index.php?option=com_content&task=view&id=495&Itemid=291.

20. Modelo de Dominio. *Scribd.* [Online] [Cited: 11 22, 2011.] http://es.scribd.com/doc/21296187/40/Modelo-del-Dominio.

21. Bienvenido a NetBeans y www.netbeans.org. *NETBEANS.* [Online] [Cited: 11 21, 2011.] http://netbeans.org/index_es.html.

22. Características del lenguaje. *Tikal.* [Online] [Cited: 11 20, 2011.] http://tikal.cifn.unam.mx/~jsegura/LCGII/java3.htm.

23. **González, Abdiel E. Cáceres.** Programación orientada a objetos. *Cinvestav.* [Online] [Cited: 11 20, 2011.] http://computacion.cs.cinvestav.mx/~acaceres/courses/udo/poo/files/slides/POO-02.pdf.

24. OpenUP como alternativa metodológica para proyectos pequeños de software. *OpenUp.* [Online] [Cited: 11 20, 2011.] http://epf.eclipse.org/wikis/openup/.

25. **Flores, Carmina Lizeth Torres.** Establecimiento de una Metodología de Desarrollo de Software. *Fit.* [Online] [Cited: 11 19, 2011.] http://fit.um.edu.mx/CIDET/publicaciones/COMP-004-2008%20Establecimiento%20de%20una%20Metodolog%C3%ADa%20de%20Desarrollo%20de%20Software%20para%20la%20Universidad%20de%20Navojoa%20Usando%20OpenUP.pdf.

26. lo bueno lo malo y lo feo. *Antartec Blogs.* [Online] [Cited: 11 17, 2011.] http://blogs.antartec.com/desarrolloweb/2008/10/extjs-lo-bueno-lo-malo-y-lo-feo/.

27. **Lorente, Abel Ernesto.** Conceptos UML. *Diseño UML.* [Online] [Cited: 11 17, 2011.] http://diseouml2009.blogspot.com/2009/04/que-es-conceptos-uml.html.

28. **Quintero, Juan Bernardo y Anaya, Raquel.** MDA y el papel de los modelos en el proceso de desarrolo de software. *SCIELO.* [Online] [Cited: 11 16, 2011.] http://www.scielo.org.co/scielo.php?pid=S1794-12372007000200011&script=sci_arttext.

29. ModelosDeProcesoDeSoftware. *Tecnologico.* [Online] [Cited: 11 5, 2011.] http://www.mitecnologico.com/Main/ModelosDeProcesoDeSoftware.

30. Arquitectura de Software Dirigida por Arquitectura de Software Dirigida por Modelos. *IE.* [Online] [Cited: 11 4, 2011.] www.ie.inf.uc3m.es/grupo/docencia/reglada/ASDM/Presentacion.pdf.

31. **L.Nahuel, L.Ocaranza, M.Pinasco.** Herramientas de soporte al proceso de desarrollo dirigido por modelos. *JIDIS.* [Online] [Cited: 11 4, 2011.] www.jidis.frc.utn.edu.ar/papers/45308ce78aafdf2092719a59d01c.pdf.

32. La-Diversidad-de-Los-Objetos-1-1-Caracteristicas. *Scribd.* [Online] [Cited: 11 2, 2011.] http://es.scribd.com/doc/25171397/1-La-Diversidad-de-Los-Objetos-1-1-Caracteristicas.

ANEXOS

Anexo 1: Registro de defectos y dificultades detectados.

Tabla .13: Plantilla de No Conformidades

Elemento	No	No conformidad	Aspecto correspondiente	Etapa de detección	Clasificación	Estado NC	Respuesta del Equipo Desarrollo
Nombre de archivo	1	A la hora de guardar el Manifiesto de Ofuscación acepta el punto. Ejemplo k.k	Nombre del archivo en el caso de uso generar Manifiesto de Ofuscación.	Prueba	S	PD 20-5-2012 RA 25-5-2012	NC corregida.
Manifiesto	2	A la hora de ofuscar el manifiesto se dejan campos obligatorios vacíos y se procede sin mostrar mensaje.	Origen del manifiesto en el caso de uso generar empaquetado de componentes.	Prueba	S	PD 20-5-2012 RA 25-5-2012	NC corregida.
Manifiesto	3	Cuando se ofusca el manifiesto, se permite editar el área de texto.	En el caso de uso generar empaquetado de componentes.	Prueba	R	PD 20-5-2012 RA 25-5-2012	NC corregida.

Anexo2: Imagen de la aplicación para empaquetar componentes.

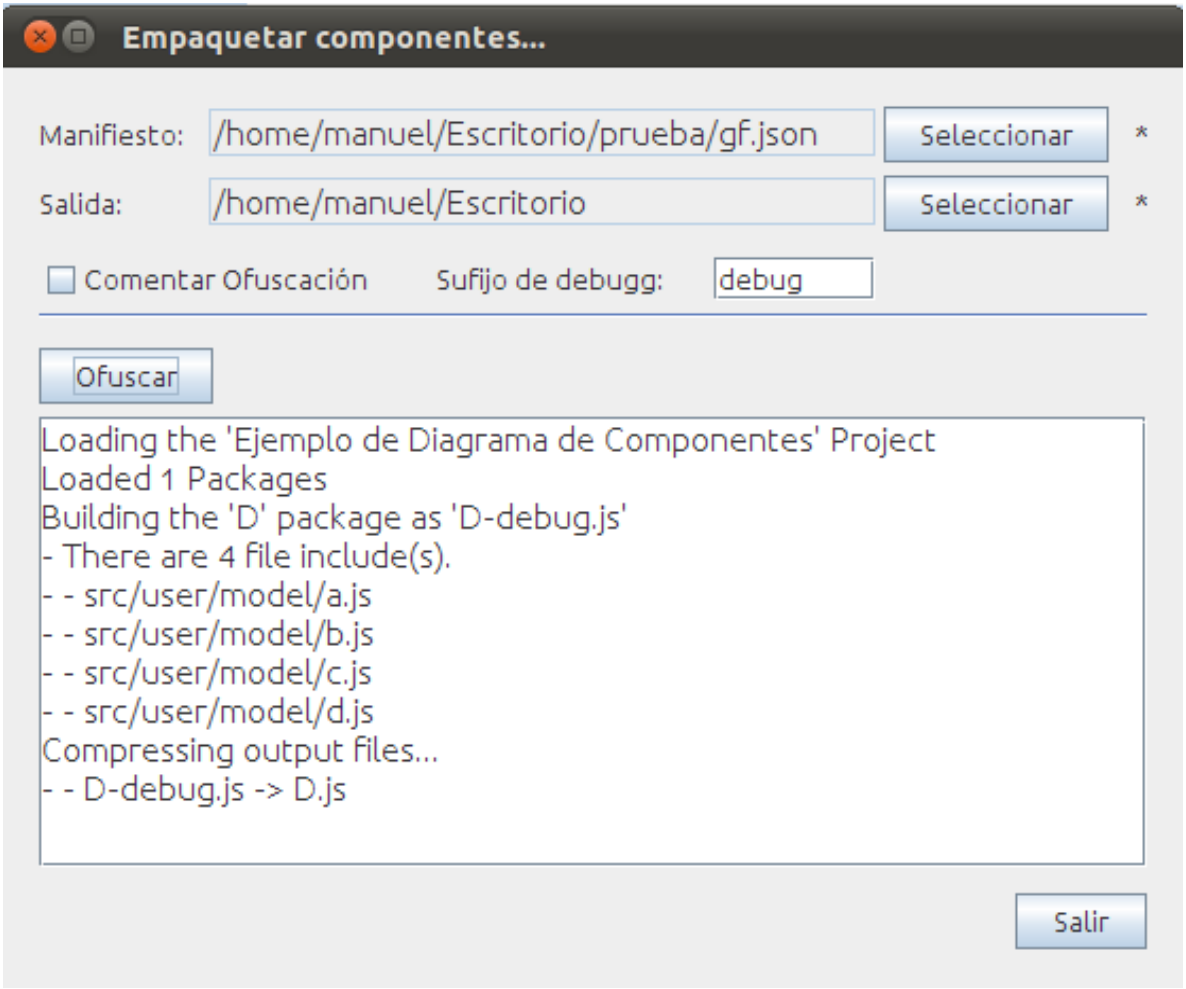


Fig.16: Interfaz del CU “Generar empaquetado de componentes”.

Anexo3: Imagen del manifiesto generado a través de la funcionalidad Generar Manifiesto de Ofuscación.

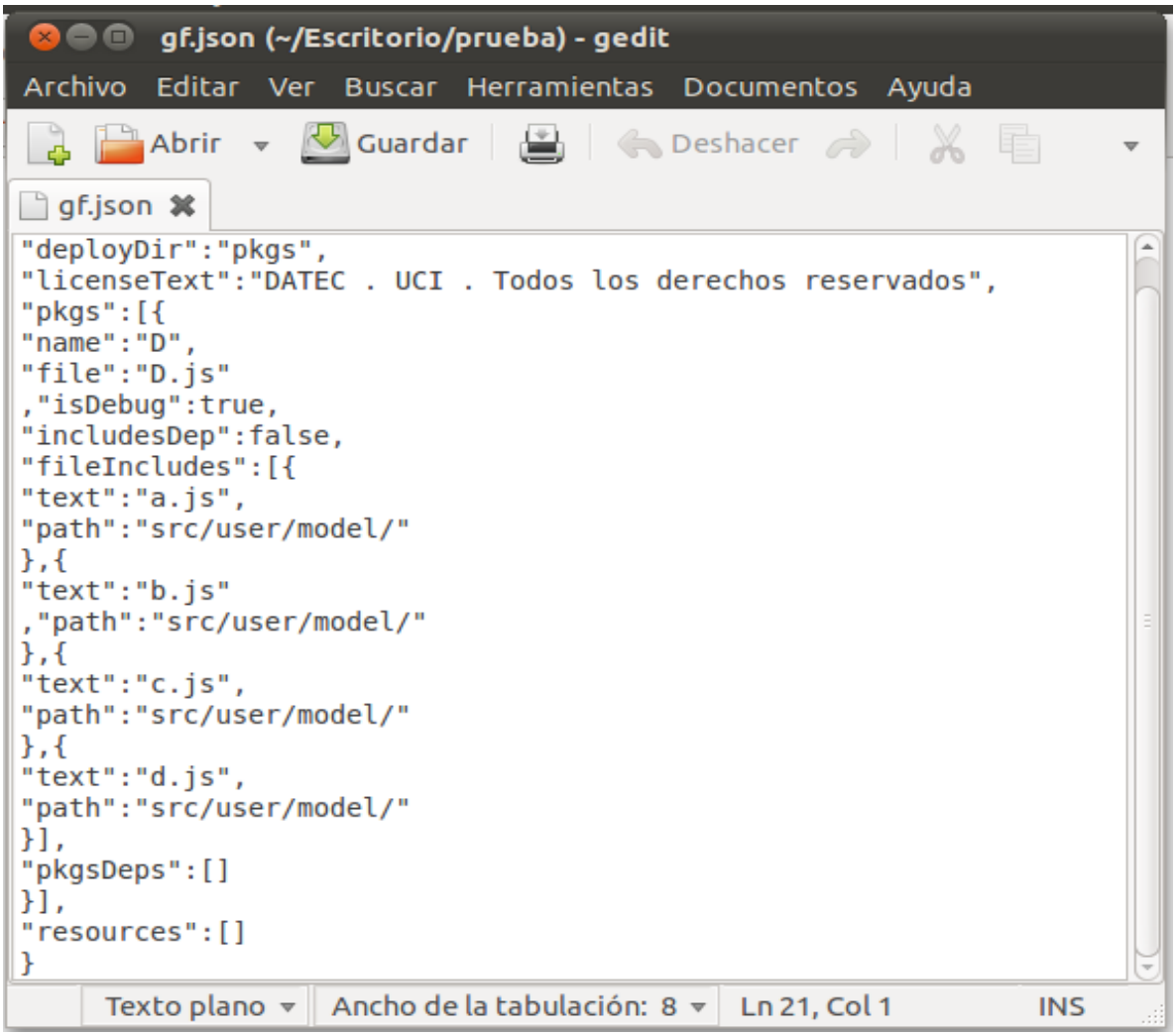


Fig.17: Interfaz del CU “Generar Manifiesto de Ofuscación”.

GLOSARIO DE TÉRMINOS

Component: Representa componentes de modelado en la herramienta *Visual Paradigm for UML*.

Metamodelo: Un metamodelo es un modelo que define el lenguaje para expresar un modelo.

Metodologías: Se refiere a los métodos de investigación en una ciencia. Se entiende como la parte del proceso de investigación que permite sistematizar los métodos y las técnicas para llevarla a cabo.

Metodología Ágil: Constituyen un nuevo enfoque en el desarrollo de software, mejor aceptado por los desarrolladores de proyectos que las metodologías convencionales, debido a la simplicidad de sus reglas y prácticas.

MOF: Meta-Object Facility un lenguaje para describir lenguajes de modelado constituido por un conjunto de interfaces estándar que se pueden utilizar para definir y manipular una serie de meta-modelos interoperables y sus modelos correspondientes. Captura la diversidad de estándares de modelamiento para integrar diferentes tipos de modelos y metadatos e intercambiarlos entre diferentes herramientas.

OMG: Object Management Group es una asociación abierta fundada en 1989 que produce y mantiene especificaciones de la industria del software para la portátil de aplicaciones reutilizables en entornos distribuidos y heterogéneos. Algunos de los estándares definidos por la OMG son el UML y el CORBA, que ofrece interoperabilidad multiplataforma a nivel objetos de negocios.

Package: Representa el metamodelo que ofrece la herramienta Visual Paradigm for UML para representar paquetes. Un paquete puede contener varios paquetes y componentes.

Pruebas: Son un elemento importante para garantizar la calidad de los productos. Su principal objetivo es encontrar la mayor cantidad de errores en el software. No aseguran la ausencia de defectos en el software, sólo demuestran la presencia de errores en el mismo.

Software: Todo programa o aplicación para realizar tareas específicas.