INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN
SECCIÓN DE GRADUADOS

HERRAMIENTA PARA LA OBTENCIÓN DEL DIAGRAMA DE INTERACCIÓN DE CLASES DE COMPONENTES SWING: ODICC-SWING

TESIS DE MAESTRIA EN CIENCIAS
QUE PARA OBTENER EL GRADO DE:
MAESTRO EN CIENCIAS EN COMPUTACIÓN
P R E S E N T A
L. I. ADOLFO ROMERO ALVARIO

DIRECTOR: DR. BÁRBARO JORGE FERRO CASTRO

MÉXICO, D.F. OCTUBRE 2002

TABLA DE CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.

Pág.

Resumen

Tradicionalmente las aplicaciones de escritorio han sido diseñadas con una interfaz de usuario construida dentro de la aplicación. El código para la interfaz de usuario de la aplicación es usualmente acoplada al código que contiene la funcionalidad de la aplicación. El lograr un entendimiento claro del código fuente con el objetivo de realizar una transformación de código escrito en la forma tradicional mencionada anteriormente, a un código orientado a la visualización remota de aplicaciones, requiere de técnicas que ayuden a la separación de objetos propios de la interfaz gráfica de los objetos que implementan la lógica de la aplicación. El lograr una clara separación de los diferentes objetos que intervienen en una aplicación, tales como objetos del dominio de la aplicación de los de manejo de interfaz, es una actividad que ayuda al entendimiento del código. En la presente tesis se logra separar los objetos de la interfaz gráfica de los objetos que contienen la lógica del negocio, así como obtener el diagrama de secuencia a partir del análisis de código fuente, se presenta un modelo para la obtención de dicho diagrama y se construye una herramienta basada en este modelo.

Abstract

Usually the desktop applications have been designed with an Interface of user constructed within the application. The code for the Interface of user of the application usually is connected to the code that contains the functionality of the application. Obtaining a clear understanding of the source code with a object to making a transformation of code writing in traditional form, one code oriented to the remote visualization of applications, requires of techniques that help the separation of own objects of the graphical interface of the objects that implement the logic of the application. Obtaining a clear separation of the different objects that take part in an application, such as objects of the dominion of the application of those of interface handling, is an activity that helps the understanding of the code. In the present thesis it is managed to separate the objects of the graphical interface of the objects that contain the logic of the business, as well as to obtain the diagram of sequence from the source code analysis, a model for the obtaining of this diagram appears and a tool based on this model is constructed.

Capítulo

1

Introducción

El estado actual de las arquitecturas distribuidas y los sistemas de código móvil, hacen posible esquemas que convierten aplicaciones diseñadas para visualizarse en la misma computadora donde reside el código, a aplicaciones que interaccionan con el usuario de manera remota, en la que se separa la visualización de la lógica de la aplicación y de cualquier otro componente que esa lógica necesite.

Las aplicaciones orientadas a consolas (con una interfaz de usuario de secuencias de líneas de comando) pueden accederse de manera remota sin mucha dificultad, mediante técnicas de terminales remotas conocidas desde hace más de 20 años. Sin embargo, esa tarea no resulta tan simple cuando la aplicación se diseña con una interfaz gráfica de usuario, basada en alguno de los sistemas de ventanas existentes y con un marco de trabajo específico.

Las técnicas para diseñar y codificar aplicaciones con visualización remota pueden agruparse en dos clases generales: aquellas que establecen principios de diseño que se siguen para construir desde cero aplicaciones de este tipo, y técnicas de reingeniería para convertir aplicaciones diseñadas bajo un esquema de visualización local.

El segundo grupo de técnicas tiene en común esquemas para el razonamiento del código fuente de la aplicación, técnicas de ingeniería inversa y otras que persiguen el objetivo de lograr un entendimiento del código fuente.

El reto para el desarrollo de técnicas de ingeniería inversa, que persiguen separar la lógica de presentación del resto de la aplicación es grande. A pesar de que existen metodologías de diseño con modelos de referencias que contribuyen a esa separación, lo cierto es que la mayoría de las aplicaciones de escritorio (que usan una interfaz gráfica) se diseñan y construyen de manera monolítica, mezclando las diferentes partes del sistema con acoplamientos muy fuertes. El razonamiento de estos programas es complicado.

Por otra parte, las herramientas actuales llevan al diseñador a usar de manera desmedida muchos componentes gráficos que embellecen la interfaz de la aplicación, y que no se consideran como una parte importante que debe documentarse y diseñarse, siguiendo principios arquitectónicos de reuso. La forma de proceder con el diseño y la codificación obvia la complejidad de la interfaz de usuario y del procesamiento de eventos que conlleva este estilo de programación. La manera de diseñar se supedita a la consideración de que “existen muchos wizards en la herramienta que éstamos usando, por lo que no debemos preocuparnos por el diseño de la interfaz de usuario”. El resultado final es un código que mezcla aspectos de manera innecesaria y que no permite un buen mantenimiento correctivo y preventivo.

Para lograr un entendimiento claro del código fuente con miras a realizar una transformación de código escrito en la forma tradicional (la forma mencionada en el párrafo anterior), a un código orientado a la visualización remota de aplicaciones, se necesitan técnicas que ayuden a la separación de objetos propios de la interfaz gráfica de los objetos que implementan la lógica de la aplicación. Denotamos con el término “objetos interfaces” a los componentes de la aplicación que tiene que ver con la visualización y el manejo de eventos producidos por las acciones de los usuarios.

La presente tesis aborda uno de los problemas a resolver con vistas a una arquitectura de sistemas de visualización remota: el de extraer del código fuente, un conjunto de diagramas de secuencia de objetos visuales que serán utilizados para modelar el procesamiento distribuido de eventos y la visualización remota de los objetos interfaces.

Con este objetivo, se desarrolla un modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente. Para validar la conveniencia de

este modelo, se construye una herramienta para la construcción y visualización del diagrama de secuencia de objetos visuales, realizando un análisis de los objetos que intervienen en la construcción de la interfaz gráfica de usuario y construyendo un diagrama de secuencia descrito con UML [OMG]. Se diseña una estructura persistente que representa el modelo y que podrá usarse para diseñar las técnicas de visualización remota.

1.1 DESCRIPCIÓN DEL PROBLEMA

La carencia de técnicas que ayudan a transformar un código escrito en la forma tradicional, a un código orientado a la visualización remota de aplicaciones, es la principal motivación de este trabajo de tesis. Una de las actividades necesarias en ese proceso de conversión es la obtención de los modelos de interacción de los objetos interfaces que intervienen en la construcción de la interfaz gráfica de usuario (GUI). La representación de esos modelos mediante diagramas de secuencia (ver Apéndice A) resulta útil para el entendimiento de la aplicación.

El proceso de entendimiento del código fuente de una aplicación es vital en la fase de mantenimiento del mismo, comienza con el análisis de la documentación generada durante la primera fase del ciclo de vida del software. Pero, en muchos de los sistemas ya desarrollados, no existe o no es suficiente tal documentación para facilitar el mantenimiento del mismo.

1.1.1 Justificación del estudio

En la fase de mantenimiento de software, el proceso de entendimiento del código fuente suele consumir más de la mitad del esfuerzo de programación [RAJ00].

La esencia del entendimiento de un programa es identificar los artefactos y comprender sus relaciones; este proceso es esencialmente realizado en varios niveles de abstracción. En el nivel más alto de abstracción encontramos los documentos de análisis y diseño de sistemas, en el mejor de los casos cuando estos existen. Sin embargo, dichos documentos no reflejan los artefactos internos de la aplicación, como lo son los objetos visuales. Dentro de los objetos que componen una aplicación es importante identificar y realizar una clasificación de los objetos pertenecientes a la interfaz gráfica y a los objetos pertenecientes al dominio de aplicación, en aras de lograr un mejor entendimiento del código.

Para comprender el código fuente de una aplicación es necesario realizar diversas actividades tales como, revisión de los diagramas de interacción entre objetos, casos de uso, diagramas de colaboración, etcétera, y todo lo demás que se encuentre a la mano. Sin embargo ésta documentación pocas veces refleja la interacción que se realiza entre objetos interfaces, que son necesarios en la construcción de la interfaz gráfica (GUI).

Las interfaces gráficas de usuario se diseñan siguiendo el estilo de programación por eventos. El modelo de eventos es una característica propia del marco de trabajo que se use para el desarrollo de la interfaz gráfica. Se trata de un estilo arquitectónico de invocación implícita, que sigue esquemas similares a las técnicas de publicación-suscripción, y que se mezcla con el estilo arquitectónico propio que usa el diseñador para construir la aplicación global.

La mezcla de estilos impone una estructura compleja a los programas. Esto puede ser atribuido a que el diseñador de la aplicación se concentra más en el desarrollo de la solución que en los objetos de la interfaz gráfica a utilizar. El resultado es que no se cuenta con un modelo claro que ayude al entendimiento de la aplicación en lo concerniente a la interfaz gráfica.

En esta tesis se parte de la hipótesis que es necesario contar con el modelo de interacción entre los objetos interfaces y el resto de la aplicación, que manifieste de manera clara el procesamiento de eventos en la aplicación. Se asume también que es correcto la representación de ese modelo mediante diagramas de interacción expresados en UML (ver Apéndice A).

La utilidad de un modelo de este tipo aumenta si disponemos de una herramienta prototipo que demuestre lo efectivo de la técnica, por lo que parte importante de este trabajo es el diseño y construcción de esa herramienta.

1.1.2 Beneficios esperados

Al realizar el presente trabajo de investigación se pretenden alcanzar los siguientes beneficios:

- Lograr una separación de los objetos gráficos que intervienen en la construcción de la interfaz gráfica de usuario, de los objetos propios del dominio de la aplicación y los objetos encargados del manejo de eventos.
- Formulación de un modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente.
- Construcción de un diagrama extra (diagrama de interacción de objetos visuales), como complemento de la documentación del sistema analizado.
- Analizar la interacción de los objetos gráficos que intervienen en la construcción de la interfaz gráfica de usuario (GUI) para estudiar la reacción de la aplicación al disparo de eventos.
- Contar con una herramienta para mejorar el proceso de entendimiento y mantenimiento de un programa.
- Generar un repositorio de información, el cual podrá ser utilizado por otra aplicación para realizar la transformación del código, a un código que permita la visualización remota de aplicaciones.

- Construir a una herramienta capaz de realizar la construcción y visualización gráfica del diagrama de secuencia de objetos.
- Eliminar el problema de compatibilidad entre plataformas de hardware y software, ya que la herramienta se desarrollará en lenguaje Java.
- La tesis representa un punto de apoyo para investigaciones futuras en el área de ingeniería en reversa.

1.2 PROPUÉSTA DE SOLUCIÓN

Se propone la formulación de un modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente y la construcción de un sistema que utilice este modelo, para que a partir de código escrito en lenguaje Java y que utilice Swing, realice la construcción y posterior visualización de un diagrama de secuencia de objetos, construyendo así una herramienta que permita al ingeniero de software realizar el análisis de eventos de la interfaz gráfica de una aplicación.

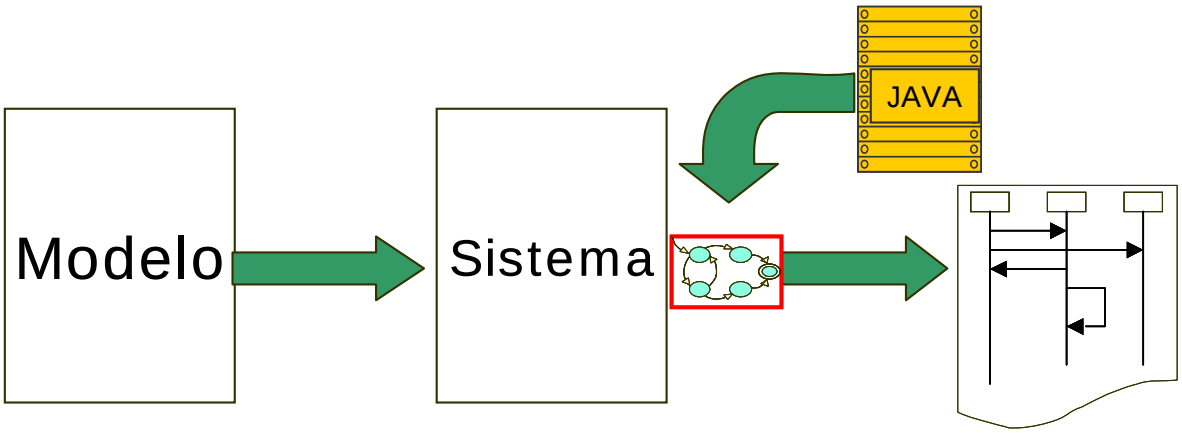


Figura 1.1 Gráfica ilustrativa de la propuesta de solución.

De una forma gráfica se puede observar en la figura 1.1 el esquema global del prresente trabajo, el cuadro de la izquierda, titulado modelo, representa el primer paso que es la formulación del modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente. El siguiente cuadro denominado sistema, simboliza el desarrollo de la aplicación basada en el modelo y quien como entrada tiene archivos con código Java y realiza el analisis de código fuente para lograr asi construir el diagrama de interacción de objetos.

El modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente comprende lo siguiente:

1. Algoritmo de clasificación de objetos, consiste de una serie de pasos para lograr la identificación y tipificación de los objetos que intervienen en la

aplicación. El resultado es la taxonomía de los objetos de la aplicación. Además identifica las posibles interacciones entre las clases del código fuente, almacenándolas en el repositorio descrito en el apéndice C.

2. Forma canónica para la representación de la interacción entre objetos. Está descrita por una gramática libre al contexto, y construye la forma canónica que ayuda en la generación visual del diagrama de interacción.
3. La función de transformación de la forma canónica a la representación visual de interacción entre objetos. Es la encargada de construir la representación visual en notación UML del diagrama de secuencia a apartir de la forma canónica generada en el paso anterior.

1.3 OBJETIVOS

Formular un modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente

Construcción de una herramienta que permita extraer de un código fuente escrito en Java y utilizando Swing para la construcción de la interfaz gráfica de usuario, el diagrama de secuencia de objetos.

Contribuir al proyecto de investigación “Arquitectura distribuida de software para la visualización y control remoto de objetos interfaces” el cual se lleva a cabo en el Laboratorio de Tecnología de Software del Centro de Investigación en Computación del IPN.

Contribuir a las investigaciones en el área de ingeniería en reversa en el Centro de Investigación en Computación.

1.4 METODOLOGÍA

La metodología que se siguió en este proyecto se describe a continuación:

Paso 1

Para lograr la construcción del diagrama de secuencia, es necesario realizar un análisis detallado de código fuente, encontrando de ésta manera las relaciones estáticas entre los objetos de una aplicación.

Se diseñó y construyó un repositorio de información que sirve para almacenar el nombre de los paquetes, clases e interfaces que forman parte de Swing, con la finalidad de poder identificar tales objetos dentro de la aplicación analizada. Además este repositorio sirve para almacenar las interacciones entre las clases de la aplicación analizada.

Con la información almacenada en el repositorio se analiza la firma de la clases y es posible obtener una clasificación de los objetos que intervienen en la aplicación para identificar cuales de ellos son:

A) De la interfaz gráfica. Se identifican si estos extienden o heredan de alguna clase o interfaz declarada dentro de Swing.

B) Pertenecientes al manejo de eventos. Si estos implementan alguna interfaz necesaria para el manejo de eventos, definidas por Swing.

C) Propios del dominio de la aplicación. En el caso de que no pertenezcan a alguna de las anteriores.

Paso 2

El paso siguiente consiste en la formulación de la forma canónica para la representación de la interacción de objetos. Para esto se hizo uso de la teoría de gramáticas libres del contexto, además de las características y funcionalidad del lenguaje Java, en la cual nos concentraremos en el modelo de eventos Swing.

Una vez realizado lo anterior se diseñó e implementó un módulo de transformación de código Java a la forma canónica. Se utiliza el paradigma de programación orientada a objetos, relaciones entre clases, así como las características del lenguaje de programación Java en su parte correspondiente a la construcción y manejo de interfaz gráfica Swing. La construcción de las formas canónicas se construye con base en las interacciones almacenadas en el repositorio.

Paso 3

Por último se diseñó e implementó un módulo de construcción y visualización del diagrama de interacción entre objetos visuales. En este módulo se utiliza el paradigma de programación orientada a objetos, gramáticas libres del contexto, así como del lenguaje de programación Java, la forma canónica para la interacción de objetos visuales, y la función de transformación de la forma canónica a la representación visual de interacción entre objetos.

Se eligió para la codificación de la herramienta el lenguaje de programación Java, ya que éste proporciona la ventaja de ser independiente de la plataforma de hardware y sistema operativo utilizado para ejecutar el sistema.

La estructura general (información de entrada y salida) de este proyecto de tesis se muestra en la figura 1.2.

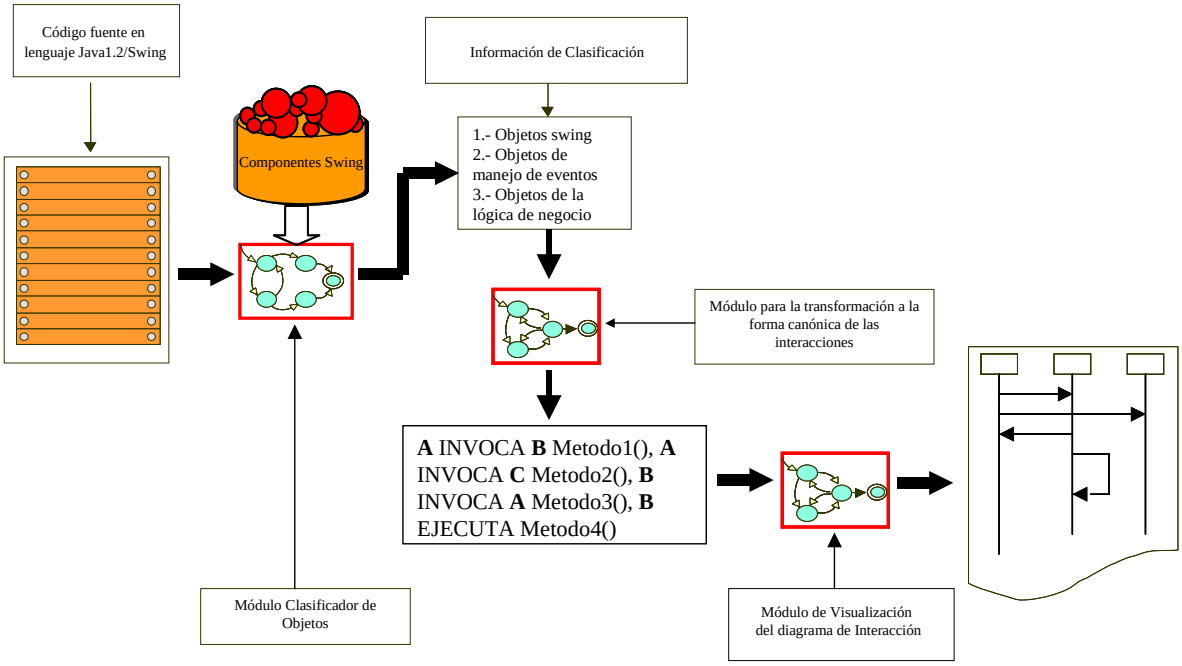


Figura 1.2 Estructura general de entradas y salidas de información del producto generado por este proyecto de tesis.

1.5 ALCANCE Y LIMITACIONES

Alcance

Se formula el modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente.

Se desarrolla una herramienta capaz de realizar una identificación y clasificación de los objetos participantes en una aplicación. Dicha clasificación se encuentra orientada a permitir un mejor análisis del manejo de eventos en la interfaz gráfica. La clasificación de objetos es la siguiente:

Objetos pertenecientes a la interfaz gráfica: Componentes Swing. Los Componentes Swing, son aquellos que están definidos por Swing 1.1, y se encuentran en el repositorio de información ver Apéndice C.

Objetos de manejo de eventos: Los objetos de manejo de eventos, son aquellos que proporcionan código ejecutable para dar tratamiento a los eventos.

Objetos propios del dominio de la aplicación. Son los objetos que son parte de la lógica del negocio.

La herramienta es capaz de realizar la construcción y su posterior visualización en un ambiente gráfico del diagrama de secuencia de los objetos que intervienen en un código escrito en lenguaje Java. Permitiendo al Ingeniero de Software analizar el manejo de eventos de la interfaz gráfica, así como las clases de la aplicación.

Limitaciones

Se analizará únicamente código en lenguaje Java 1.2, de aquel código fuente que utilice para la construcción de la interfaz gráfica Swing 1.1.

En la herramienta no se verifican errores en el código analizado, por lo tanto el mismo no debe tener errores de compilación, ni ejecución.

Para la identificación de los comentarios en las clases por analizar, estos deben de éstar de acuerdo con las normas de javadoc [LAM01].

La herramienta no reconoce jerarquía de herencia entre las clases analizadas, ya que se trata de analizar el comportamiento de las clases y no de la jerarquía de clases.

1.6 ORGANIZACIÓN DEL DOCUMENTO

En el capitulo 2, se realiza una investigación documental y se presenta el estado del arte de diferentes tópicos relacionados con la presente tesis. En el capitulo 3, se presenta el marco teórico que da sustento a nuestra investigación. En el capitulo 4, se describe la solución adoptada para resolver el problema planteado, en donde se muestra el modelo para la obtención del diagrama de secuencia a partir del análisis de código fuente, así como la interfaz de la herramienta desarrollada ODICC-Swing. El capitulo 5 muestra algunos ejemplos de corridas dentro de la herramienta desarrollada, así como el diagrama obtenido en el análisis de la aplicación demostrativa. Para el capitulo 6, se muestran nuestras conclusiones y los trabajos futuros sugeridos para enriquecer el trabajo de tesis.

Se presentan cuatro apéndices dentro del documento, el apéndice A, nos muestra la definición del diagrama de secuencia. El apéndice B, muestra algunas definiciones de paquetes e interfaces que son propiedad de Swing (Java Foundation Class), y en el apéndice C, se muestra la descripción del repositorio que sirve para almacenar las interacciones entre clases, así como las definiciones de Swing. El apéndice D muestra el diagrama de clases de la herramienta ODICC-Swing, asi como un breve comentario acerca de su funcionalidad.

Capítulo

2

Estado del Arte

En este capítulo se describe el estado de la práctica actual en torno al tema de ingeniería en reversa, así como los trabajos relacionados con la herramienta generada en este proyecto de tesis, ODICC-Swing, analizando sus características, ventajas y desventajas. Además se menciona cual es la diferencia de ésta tesis con tales trabajos haciendo énfasis en la aportación de nuestro trabajo.

2.1 INTRODUCCIÓN

En los últimos años surgió la *ingeniería en reversa* y desde entonces ha sido motivo de intensa investigación, y prueba de ello son los múltiples artículos publicados en diferentes congresos internacionales relacionados con éste importante tema; entre los tópicos más importantes que han captado la atención de los investigadores están:

- Mejorar la calidad del software, típicamente los sistemas de software son de baja calidad, las modificaciones se tienen que realizar a "mano". La reingeniería está intentando mejorar la calidad de software y producir documentación actual.

- Migración, los sistemas de software antiguo trabajan bien para los usuarios que los vieron nacer, pero están hechos sobre lenguajes obsoletos, plataformas de hardware y sistemas operativos antiguos, y por lo tanto estos sistemas necesitan ser rediseñados para trasportar el software hacia nuevas plataformas o lenguajes.

- Redocumentación, la mayoría de los sistemas de software sufren modificaciones con el tiempo que ya no son documentadas, lo cual provoca que la documentación existente de los sistemas no se encuentre actualizada.

El tópico que más nos interesa por su similitud con el presente trabajo es el relacionado con la r*edocumentación*, la cual es la materia prima para lograr un buen mantenimiento de los sistemas ya probados y que se encuentran en operación. Por ésta razón nos enfocamos al análisis de herramientas y trabajos de investigación relacionados con el mantenimiento de código, en relación con extracción de diagramas, o artefactos de diseño.

2.2 TRABAJOS RELACIONADOS

2.2.1 Java y UML

En este articulo se presentan algunos aspectos generales sobre la historia de UML y su estrecha relación con el lenguaje de programación Java, entre las grandes ventajas de utilizar Java al momento de realizar un modelado con UML, es el hecho de que se puede ir de ida y vuelta en las diferentes etapas del desarrollo de software, y por lo tanto esto ha hecho que las herramientas que utilizan UML, puedan generar código completo para los Applets y aplicaciones Java.

En el artículo se hace referencia a la guía de usuario de Booch, Jacobson, y Rumbaugh, en la cual se indica una fuerte predisposición en usar la generación automática de código y la simulación para obtener el máximo beneficio de las herramientas de modelado, por ejemplo se habla de ingeniería de ida y vuelta para los nueve diagramas de UML.

Sin embargo, el software de Rational [RAT00] de los creadores de UML, proporciona un ambiente de ingeniería de ida y vuelta (Round Trip Engineering, RTE), solamente para los

diagramas de clases y componentes, sin embargo, otros vendedores como Softera, Together Software, Popkin y Tendril tienen capacidades de ingeniería en reversa para los diagramas de secuencia y colaboración en diferentes estados de finalización [JAC00]

De acuerdo con el párrafo anterior, se puede deducir que el software de Rational [RAT00], no posee la característica de poder construir el diagrama de secuencia a partir del análisis de código fuente de una aplicación escrita en lenguaje Java, lo cual es uno de los objetivos de la presente tesis.

2.2.2 Creando Especificaciones desde el código: técnicas de ingeniería en reversa [PTB00]

En este artículo se presentan algunas técnicas de ingeniería en reversa, las cuales producen abstracciones orientadas a objetos así como anotaciones funcionales, las cuales son utilizadas para la comprensión de la estructura esencial y operaciones de la aplicación, además provee un diseño formal el cual puede hacer el código mucho más manejable en la fase de mantenimiento del mismo.

Este artículo se centra en las aplicaciones de procesamiento de datos, utilizando el lenguaje de programación COBOL, el cual ya no es actual, los pasos que se mencionan de la ingeniería en reversa son:

1. Organización de datos y abstracción. En este paso el código contiene simples instancias (objetos) de una clase, en lugar de amplios grupos separados de instrucciones desconectadas aplicadas a datos aislados. Hay varios tipos de datos abstractos definidos, tales como la definida para el tipo de dato abstracto "file" en COBOL, y son buscadas en el código para lograr las abstracciones de clases.
2. Reorganización del código. El siguiente paso es separar el programa dentro de procedimientos distintos (instancias de proceso) los cuales ejecutan una simple función de entrada y salida, posiblemente sobre varios archivos o estructuras de datos internas, y se identifica ésta funcionalidad exactamente.

La desventaja de este trabajo, es que queda limitado a aplicaciones de procesamiento de datos, como lo son las desarrolladas en COBOL, además de que no alcanza a presentar diagramas de diseño en UML.

2.2.3 Recuperación de diseño mediante búsqueda automática de patrones de diseño estructurales en software orientado a objetos [KRA96]

En este artículo se muestra una herramienta denominada *PAT*, en ésta se extrae información de diseño directamente de los archivos de cabecera de C++ y se almacenan en un repositorio. Los patrones son expresados como reglas Prolog y la información de diseño es también trasladada; una simple consulta de Prolog es usada para la búsqueda de todos los

patrones. En ésta herramienta únicamente se reconocen instancias de algunos patrones de diseño estructurales mencionados por Erich Gamma [GAM95], los cuales son: Adapter, Bridge, Composite, Decorator, y Proxy.

Los pasos que sigue PAT para la búsqueda de patrones de diseño son:

1. Cada patrón es representado como un diagrama OMT estático, estos diagramas constituyen el repositorio P (de patrones).

2. Un programa es usado para convertir P en una regla de Prolog para cada patrón de diseño.

3. Se utiliza la herramienta ooCASE para realizar el mecanismo de análisis estructural, ooCASE extrae información de diseño de los archivos de cabecera de C++ y la representa en el repositorio en la notación OMT (Object Modelling Technique). La parte resultante del repositorio es llamada D (de diseño). La información relevante que es extraída de los archivos de cabecera de C++ es: nombre de clases, nombre de atributos, nombre de métodos, relaciones de herencia, y relaciones de agregación y asociación.

4. Otro programa es usado para convertir D en la representación Prolog.

5. Una consulta Q de Prolog detecta todas las instancias de patrones de diseño de P en el diseño D examinado.

2.2.4 Ingeniería inversa de diseño y detección automática de patrones de diseño en Smalltalk[BRO--]

En este artículo se muestra una herramienta la cual recupera información de diseño desde código *Smalltalk*, y la usa para detectar algunos patrones de diseño como son: Composite, Decorator, y Template Method. Además muestra el diagrama de clases del código analizado, utilizando la herramienta CASE Rational Rose [RAT00], para hacer esto, toma la información de diseño recuperada del código, después genera un archivo en el mismo formato que los archivos de diseño generados por Rational Rose. Este trabajo no logra recuperar diagrama alguno de interacción, ya sea el de secuencia o el de colaboración.

2.2.5 Conclusiones

Una de las limitaciones importantes de las herramientas descritas es que ninguna de las mencionadas logra realizar la construcción y visualización del diagrama de secuencia entre objetos, a partir del análisis de código fuente.

ODICC-Swing se desarrolló en lenguaje Java lo cual le proporciona una ventaja sobre las herramientas mencionadas anteriormente, al permitir ejecutarse en cualquier plataforma y sistema operativo que soporte Java.

Otra ventaja que posee ODICC-Swing es que utiliza un Modelo Canónico para la interacción entre objetos, lo cual le da formalidad a la solución utilizada para la construcción del diagrama de secuencia entre objetos de una aplicación. Además de que ésta aplicación sirve para analizar eventos de la interfaz gráfica.

Capítulo

3

Marco Teórico

En este capítulo se describen los fundamentos teóricos más importantes que dan sustento a la investigación realizada en la tesis, aspectos como: ingeniería en reversa, modelo orientado a objetos, modelo de eventos Swing, gramáticas libres del contexto y el Lenguaje de Modelado Unificado (UML).

3.1 INTRODUCCIÓN

Es indispensable describir algunos de los fundamentos teóricos utilizados en la realización de ésta tesis, así como también la relación de los mismos con este proyecto. A continuación se describirá el contexto en el cual se relacionan tales conceptos con este trabajo.

Con respecto a la *Ingeniería en Reversa*, el contexto de relación es que este proyecto de tesis ayuda de cierta forma[1] a la comprensión de código fuente, al recuperar un documento de diseño como lo es el diagrama de secuencia de clases de componentes Swing.

El contexto de relación del *Modelo Orientado a Objetos* con ésta tesis es que el diagrama de secuencia de clases, obtenido en el presente trabajo, forma parte de la metodología de Objetos. ODICC-Swing analiza código escrito en lenguaje Java, el cual obedece al paradigma Orientado a Objetos. Además de que el lenguaje Java utilizado para la codificación de la herramienta, también es Orientado a Objetos.

Debido a que ODICC-Swing construye el diagrama de secuencia de clases de componentes Swing, los cuales son utilizados en la construcción de la interfaz gráfica de usuario, es necesario analizar el *Modelo de Eventos Swing* del lenguaje Java 1.2 [BUR95].

La relación de las *Gramáticas Libres del Contexto* con el tema de tesis se presenta porque el modelo canónico de interacción entre objetos formulado en este proyecto consta de una gramática libre del contexto[2].

El *Lenguaje de Modelado Unificado* (UML) es el poseedor de la sintaxis y la semántica del diagrama de secuencia, el cual es un producto generado por el presente trabajo de tesis, por lo tanto es necesario ubicarse en el contexto de tal diagrama.

A continuación se describen cada uno de los fundamentos mencionados anteriormente.

3.2 INGENIERÍA EN REVERSA

Existe una amplia variedad de mecanismos para la comprensión de programas. Ellos pueden ser agrupados en tres categorías: visualización de código fuente, análisis con conocimiento y experiencia incorporado y las técnicas asistidas por computadora tal como la Ingeniería en Reversa [PTB00].

[1] Se dice que de cierta forma, porque no analiza exhaustivamente el software en cuanto a su estructura estática y dinámica, sólo se centra en el análisis estático del código fuente referente a la parte de objetos que sirven en la construcción de la interfaz gráfica.

[2] Decimos que consta de una gramática libre del contexto por el tipo de reglas de producción que posee.

La visualización de código fuente es una actividad esencialmente humana, el ingeniero de software manualmente analiza porciones de código fuente en algún visualizador de código, es una forma muy usual de analizar el código fuente, pero no es viable cuando se trata de analizar grandes cantidades de código fuente.

La segunda categoría consiste en un análisis incorporando conocimiento y experiencia. Lo anterior puede ser ejecutado a través de entrevistas informales con personas con suficiente conocimiento acerca del sistema analizado. Ésta aproximación es muy factible si hay suficiente personal involucrado en el desarrollo del proyecto, sin embargo existen muchas desventajas al momento de utilizar ésta técnica, como por ejemplo el hecho de que el producto de software haya sido adquirido a una compañía de terceros.

Ante las situaciones antes presentadas la solución se encuentra en la tercera categoría, la cual se basa en los mecanismos de la Ingeniería en Reversa. Un ambiente de Ingeniería en Reversa puede administrar lo complejo de la comprensión de un código fuente ayudando al ingeniero de software a extraer información de los artefactos de software de alto nivel, a través de un análisis de código fuente. Esto libera a los ingenieros de software de la tediosa, manual y errónea lectura del código fuente.

3.2.1 Tareas de la Ingeniería en Reversa

Existen diversas tareas de Ingeniería en Reversa. A continuación describiremos algunas de las más importantes: análisis de programas, reconocimiento de patrones, asignación de conceptos, redocumentación y recuperación de la arquitectura [TILL96].

Análisis de programas.

Muchos sistemas comerciales se centran en el análisis de código fuente y una simple reestructuración del mismo, usando la técnica más común de la ingeniería en reversa: el análisis de programas. Lo típico en el análisis de programas incluyen el análisis del control de flujo de datos, análisis de flujo de control y gráficas estructuradas.

Reconocimiento de patrones.

Los ingenieros de software usualmente buscan en el código esperando encontrar ciertos patrones dentro de él. Los patrones pueden ser estructurales o de comportamiento, dependiendo si una búsqueda de código tiene una estructura sintáctica especifica, o si algunos componentes comparten un flujo de datos especifico, o tienen una dinámica especifica.

Asignación de Conceptos.

La asignación de conceptos es una tarea que consiste en descubrir los conceptos individuales orientados al humano y asignándolas entonces a sus contrapartes implementadas en el código.

Redocumentación.

La redocumentación es una de las más antiguas formas de la Ingeniería en Reversa, es el proceso de retroactivamente proveer documentación de un sistema de software existente. Si la redocumentación toma forma de modificaciones dentro de los comentarios en el código fuente, ésta toma la forma de reestructuración. Sin embargo puede ser subclasificada como una subárea de la Ingeniería en Reversa por que la reconstrucción de la documentación es típicamente usada para la comprensión de programas.

Recuperación de la Arquitectura.

La comprensión de un código es especialmente problemática para los ingenieros de software y los administradores técnicos quienes son los responsables del mantenimiento de dichos sistemas. La documentación que exista para estos sistemas usualmente describe partes muy pequeñas del sistema, por lo tanto no describe la arquitectura en su conjunto. Usando la Ingeniería en Reversa para reconstruir los aspectos de la arquitectura de software ayuda en gran parte al equipo encargado de realizar mantenimiento de la aplicación.

3.3 MODELO ORIENTADO A OBJETOS

El modelo orientado a objetos es una filosofía para diseñar e implementar sistemas de software y se basa en las propiedades de:

- *Abstracción de datos.* Es un modelo de software que empaqueta una estructura de datos junto con un conjunto de operaciones asociadas a ésta, y es la propiedad que habilita el ocultamiento y encapsulado de datos.

- *Encapsulado de datos.* Un objeto encapsulado es una unidad de software única con fronteras propias y bien definidas que protegen los detalles de su representación interna y con una interfaz de comunicación explícita.

- *Herencia.* Es la propiedad de los lenguajes orientados a objetos de asumir las propiedades (atributos y comportamiento) que caracterizan a una clase de objetos denominada base a otra clase denominada derivada. Ésta propiedad es la que permite la programación por incrementos, extendiendo la funcionalidad de las clases participantes en la solución de un problema.

- *Polimorfismo.* Propiedad del comportamiento que tienen los objetos de responder de diferente manera a mensajes recibidos. El polimorfismo se implementa a través de la sobrecarga de operadores y/o funciones, ya que la forma se asocia en tiempo de ejecución.

Este modelo apoya directamente la reutilización de software ya que proporciona mecanismos que permiten adaptar un componente de código a nuevos usos o aplicaciones sin modificar su definición actual.

Las estructuras de programas orientados a objetos se organizan en clases taxonómicas. A mayor nivel en la jerarquía corresponde una mayor generalidad, a menor nivel en la jerarquía se incrementa la particularidad de las clases.

Éste es el mecanismo que proporciona el medio para hacer programación por incrementos, puesto que a mayor jerarquía las clases pueden ser reutilizadas en más aplicaciones diferentes, extendiendo su funcionalidad mediante clases más específicas.

El elemento clave del modelo de programación orientada a objetos es la habilidad que proporcionan los lenguajes que soportan éste paradigma para facilitar el uso de librerías de componentes reusables.

Varios avances en la Ingeniería de Software hicieron más viable la reusabilidad de software, la programación orientada a objetos (POO) permitió la construcción de sistemas más generales, donde muchas partes del código puedan ser aplicadas a múltiples sistemas.

3.4 MODELO DE EVENTOS SWING

En este modelo de eventos un componente puede iniciar (disparar) un evento. Cada tipo de evento está representado por una clase distinta. Cuando un evento es disparado, éste es recibido por uno o mas “listeners” (escuchadores) sobre los cuales actúa el evento. La fuente del evento y el lugar donde el evento es manejado pueden estar separados.

Los componentes Swing pueden generar muchos tipos de eventos. A continuación se presentan solo algunos de ellos [ECK99].

Acciones realizadas	Tipo Listener
El usuario hace clic sobre un botón, presiona enter sobre un Text Field o selecciona un ítem del menú.	ActionListener
El usuario cierra un Frame (ventana principal)	WindowListener
El usuario presiona un botón del mouse mientras el puntero se encuentra sobre algún componente.	MouseListener
El usuario mueve el mouse sobre un componente	MouseMotionListener
Un componente se hace visible	ComponentListener
El componente obtiene el fuente del teclado	FocusListener
Una tabla o una lista de selección cambia.	ListSelectionListener

Cada Listener de eventos es un objeto de una clase que implementa un particular tipo de interface listener. Así que un programador, todo lo que tiene que hacer es crear un objeto listener y registrarlo en el componente que dispara el evento. El registro se realiza invocando al método **addXXXListener()** del componente que dispara el evento, en el cual **XXX** representa el tipo de evento escuchado.

Cuando se crea una clase Listener, la única restricción es que debe implementar la interfase apropiada.

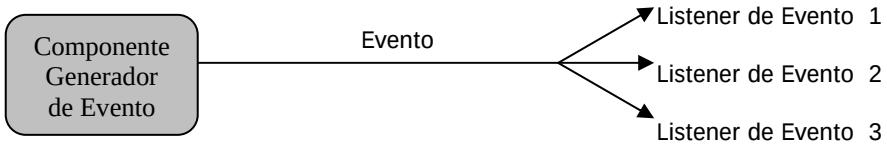


Figura 3.1 Diagrama general de eventos.

La figura 3.1 ilustra la posibilidad de tener registrados a más de un objeto Listener para un evento específico de algún componente.

3.5 GRAMÁTICAS LIBRES DEL CONTEXTO

Una gramática libre del contexto es un conjunto de variables cada una de las cuales representa un lenguaje. Los lenguajes representados por las variables se describen de manera recursiva en términos de las mismas variables, llamadas *no terminales* y de símbolos primitivos llamados *terminales*. Las reglas que relacionan a las variables se conocen como *producciones*[HOP93].

Una gramática libre del contexto tiene cuatro componentes[AHO90]:

1. Un conjunto de símbolos *terminales*. Símbolos básicos con los cuales se forman las cadenas del lenguaje.

2. Un conjunto de símbolos no *terminales*. Son variables que denotan conjuntos de símbolos gramaticales.

3. Un conjunto de *producciones.* Definen reglas gramaticales, están formadas por un *lado izquierdo*, una *flecha*, y un *lado derecho*. El lado izquierdo consta de un símbolo no terminal, el lado derecho puede estar formado por símbolos terminales, símbolos no terminales, o una combinación de ambos.

4. Un *símbolo inicial*. Es un símbolo único y es no terminal.

Una gramática libre del contexto se denota por **G=(V,T,P,S)**, en donde :

V: es el conjunto de símbolos no terminales.
T: es el conjunto de símbolos terminales.
P: es el conjunto de producciones.
S: es el símbolo inicial.

A continuación se muestra un ejemplo de una gramática que define expresiones aritméticas simples.

Las producciones son:

expr → expr op expr
expr → (expr)
expr → -expr
expr → id
op → +
op → -
op → *
op → /
op → ↑

Los símbolos terminales son: id + - * / ↑ ( )
Los símbolos no terminales son: expr y op.
El símbolo inicial es: expr.

3.6 LENGUAJE DE MODELADO UNIFICADO (UNIFIED MODELING LANGUAJE, UML)

El Unified Modeling Language (UML: Lenguaje de Modelado Unificado) es un lenguaje de modelado de propósito general para especificar, visualizar, construir y documentar los artefactos de sistemas de software, así como modelar negocios y otros sistemas que no son de software. El UML tiene un fuerte conjunto de conceptos de lenguaje de modelado de propósito aplicable a diferentes dominios.

La especificación de UML, representa la convergencia de las mejores practicas en la industria de la tecnología orientada a objetos. UML es el sucesor de tres previos lenguajes de modelado de objetos: Booch, OMT y OOSE. UML es la unión de los tres lenguajes y más, ya que UML provee expresividad adicional para manejar los problemas que estos métodos no solucionaban.

Una de las metas primarias de UML, era tomar ventaja del estado de la industria permitiendo a las herramientas de modelado de objetos visuales ínter operar. Sin embargo, para contribuir al intercambio de información entre las herramientas, UML proporciona:

- Definición formal de un análisis y diseño orientado a objetos (OA&D), una meta modelo para representar la semántica de los modelos, el cual incluye

modelos estáticos, modelos de comportamiento, modelos de uso y modelos arquitectónicos.

- Especificaciones IDL (Interface Definition Languaje) para modelar el intercambio entre herramientas de OA&D. Este documento incluye un conjunto de interfaces IDL que soportan construcción dinámica.
- Una notación legible por los humanos para representar los modelos. Este documento define las notaciones de UML, sintaxis de gráficas para expresar consistentemente la rica semántica de UML.

3.6.1 Tipos de diagramas de Interacción

Un patrón de interacción de varias instancias es mostrado como un diagrama de interacción. Los diagramas de interacción se dividen en dos, basados sobre la misma información, especificada por una interacción, pero cada uno hace énfasis en un aspecto en particular. Los dos diagramas son: diagrama de secuencia y diagrama de colaboración.

Los diagramas de secuencia muestran la secuencia explícita de solicitud-respuesta, e indican el tiempo en que se realizan las acciones. Los diagramas de colaboración muestran las relaciones entre las instancias y son mejores para comprender todos los efectos sobre una instancia determinada y para expresar un diseño procedural.

Un diagrama de secuencia muestra una interacción presentada en una secuencia de tiempo. En particular, muestra la participación de las instancias en la interacción en su línea de vida con los estímulos y peticiones que ellas intercambian en la secuencia de tiempo. No muestra las asociaciones entre objetos.

Un diagrama de secuencia presenta una colaboración con una interacción superpuésta. Una colaboración define un conjunto de participantes y las relaciones que son significativas para un propósito en particular. Los participantes definen roles que las instancias representan cuando interactúan unas con otras.

Una interacción es definida en el contexto de una colaboración. Ésta especifica la comunicación entre los roles. Más precisamente, contiene un conjunto de mensajes ordenados parcialmente, cada uno especificando una comunicación, por ejemplo, que señal será enviada o que operación será invocada, así como también los roles que serán tomados por el generador y el receptor, respectivamente.

El diagrama de secuencia viene en diferentes formas orientado a diferentes propósitos, por ejemplo para ilustrar el control de la ejecución, concurrencia, etc. Un diagrama de secuencia puede existir en una forma genérica (describiendo todas las posibles secuencias) y en una forma de instancia (describiendo una secuencia de acuerdo a la forma genérica). Ver apéndice A, para una descripción de la sintaxis y semántica del diagrama de secuencia.

Capítulo

4

Modelo Conceptual

Aquí se describe la solución adoptada para resolver el problema planteado en el primer capítulo, además se mencionan las partes que componen la misma, y su funcionamiento. Dichas partes son: modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente y la herramienta de software construida; explicando su arquitectura, y la interfaz al usuario presentada por la misma.

4.1 INTRODUCCIÓN

De acuerdo con el problema planteado en el primer capítulo, se realizó un análisis y se llegó a la conclusión de que una forma de obtener el comportamiento de los objetos que conforman la interfaz gráfica de usuario es detectarlo a partir de código fuente: por lo tanto hay que basarse en su estructura estática.

El primer paso en el análisis del código, es realizar una identificación y clasificación de los objetos encontrados, de ésta forma se pueden distinguir tres tipos de objetos: objetos Swing, objetos de manejo de eventos y objetos propios del dominio de la aplicación. Otra función del algoritmo es detectar la invocación que se realiza de las funciones miembro entre las clases.

El segundo paso es encontrar en el código fuente interacciones entre clases que son comunes en las interfaces gráficas; tales interacciones son: *invoca y ejecuta*. Donde, invoca es la *solicitud* de la ejecución de un método de un objeto a otro, y *ejecuta* es la solicitud de ejecución de un método del mismo objeto. Debido a que las interacciones que se detectan en *ODICC-Swing* son las que permite representar el diagrama de secuencia de UML (Unified Modeling Languaje), en el Apéndice A, se presenta la descripción completa de los elementos de dicho diagrama. Al momento de detectar las interacciones son almacenadas en el repositorio descrito en el Apéndice C.

Para la identificación de objetos Swing y de manejo de eventos se tomaron en cuenta las definiciones de clases proporcionadas por Java Foundation Classes (JFC-Swing) para la construcción de interfaces gráficas de usuario. La definición de los paquetes y clases que se utilizan en Swing, se presentan en el Apéndice B.

El tercer paso es la definición de una función de transformación, que utilice la forma canónica para transformarla a la representación visual del diagrama de interacción de objetos de acuerdo a la definición de UML.

Tomando como base los pasos descritos en párrafos anteriores, se formula un modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente, el cual consta de tres partes:

1. Algoritmo de clasificación de objetos que intervienen en la construcción de la aplicación
2. Forma canónica para la representación de la interacción entre objetos, y
3. Función de transformación de la forma canónica a su representación visual de interacción entre objetos.

Para demostrar la viabilidad del modelo, se construye una aplicación que utiliza este modelo, para que a partir del análisis de la estructura de clases de código fuente escrito en Java 1.2 y que utilice Swing realice un reconocimiento que permita construir el diagrama

de secuencia de objetos propios de la interfaz gráfica del programa analizado, así como su posterior visualización gráfica del mismo.

A continuación se explica el modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente, posteriormente se describirá la arquitectura de la aplicación y su interfaz al usuario.

4.2 MODELO PARA LA OBTENCIÓN DEL DIAGRAMA DE INTERACCIÓN DE OBJETOS A PARTIR DEL ANÁLISIS DE CÓDIGO FUENTE

El modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente esta comprendido de tres partes:

1. Algoritmo de clasificación de objetos que intervienen en la construcción de la aplicación
2. Forma canónica para la representación de la interacción entre objetos, y
3. Función de transformación de la forma canónica a su representación visual de interacción entre objetos.

4.2.1 Algoritmo de clasificación de objetos que intervienen en la construcción de la aplicación.

Consiste en una serie de pasos para lograr la identificación y tipificación de los objetos que intervienen en la aplicación. El resultado es la taxonomía de los objetos de una aplicación. Además identifica las posibles interacciones entre las clases del código fuente, almacenándolas en el repositorio descrito en el apéndice C.

Para realizar la clasificación de las clases de una aplicación es necesario tener en cuenta los siguientes criterios de identificación:

1.- Para identificar que clases pertenecen a la Interfaz gráfica. Se analiza en la declaración de la clase si ésta extiende o hereda de alguna clase o interfase declarada dentro de Swing. Como por ejemplo, si una clase tiene la siguiente declaración:

```
class MultiListenerPanel extends JPanel
```

Entonces la clase `MultiListenerPanel` se considera que forma parte de la interfase gráfica, ya que la clase `JPanel`, es un componente visual definido en `javax.Swing`.

2.- Para identificar las clases que pertenecen al manejo de eventos. Se analiza en la declaración de la clase si esta implementa alguna interfase definida para el manejo de eventos, presentes en `java.awt.event` como por ejemplo, la siguiente declaración de clase:

```
class Eavesdropper implements ActionListener
```

Entonces la clase `Eavesdropper` se considera de manejo de eventos, ya que `ActionListener` es una interfase para el manejo de eventos.

3.- Para identificar las clases propias del dominio de la aplicación. En el caso de que no pertenezcan a alguna de las anteriores. Es decir que las clases no hereden o implementen de alguna clase o interfase definida para manejo de eventos o de un componente grafico.

4.- Cuando ocurre alguna controversia en los criterios de identificación, por ejemplo, en el caso de la siguiente declaración de clase:

```
class MultiListenerPanel extends JPanel
      implements ActionListener {
```

La clase `MultiListenerPanel s`e considera como componente Swing, ya que un componente Swing, ya es por si solo de manejo de eventos.

Los criterios anteriormente descritos son aplicables única y exclusivamente a las clases que intervienen en una aplicación, es decir las declaraciones de las clases, y no los objetos que son instanciados dentro de los métodos de una clase.

Una vez realizada la identificación y tipificación de las clases participantes en la construcción de una aplicación, se procede a localizar las interacciones de todas las clases, mediante el siguiente algoritmo:

1. Se comienza en un subdirectorio seleccionado por el usuario.
2. Se carga la lista de todos los archivos por analizar.
3. Se recorre toda la lista de archivos y se identifica la firma de todas las clases que intervienen dentro de la construcción de la aplicación actualmente analizada.
4. Ir al principio de la lista analizada.
5. Mientras existan nombres de archivos en la lista, continuar; de lo contrario ir al punto 12.
6. Se elige el archivo con la extensión “.java”
7. Se analiza la firma de la clase aplicando los criterios descritos en párrafos anteriores, para obtener su clasificación.
8. Se extrae la firma de todos sus métodos.
9. Llenar los orígenes.
10. Se extraen las interacciones del origen actual hacia todas las clases dentro de un origen determinado. Las interacciones son las invocaciones que se realizan desde un método origen de una función miembro de una clase.
11. Avanzar en la lista de archivos una posición e ir al punto 5.
12. Ir al principio de la lista de archivos.
13. Mientras existan nombres de archivos en la lista continuar, de lo contrario ir al punto 16.

14. Analizar el código de cada método, para encontrar las relaciones internas a cada clase. Encontrar la relación “this”. Son las invocaciones que se realizan desde un método origen de una función miembro de una misma clase.
15. Avanzar en la lista de archivos una posición e ir al punto 13.
16. Fin.

En este algoritmo se identifican las siguientes dos relaciones entre objetos, debido a que estas dos relaciones serán mapeadas a los objetos gráficos del diagrama de interacción

1.-Solicitud: es la invocación de un método perteneciente a otro objeto, que se realiza desde un método origen a través de una instancia de clase, y
2.- *Ejecuta:* es la invocación de un método miembro del mismo objeto, que se realiza desde un método origen.

Por método origen, se debe entender un método miembro de una clase determinada desde el cual se analiza la aplicación. Esto nos proporciona la capacidad de analizar la aplicación desde un evento determinado, y ver que ocurre en el disparo de eventos de la interfaz gráfica.

En la aplicación este algoritmo se encuentra implantado en la clase llamada `clasificador.`

Código Java/Swing

```
package MultiListener;

import javax.swing.JFrame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;


public class AppMultiListener
{
 JFrame pantalla;
 public AppMultiListener (){
        MultiListenerPanel x = new MultiListenerPanel();
        pantalla = new JFrame("MultiListener");
        pantalla.getContentPane().add(x);
        pantalla.addWindowListener(new WindowAdapter(){
              public void windowClosing(WindowEvent e){
                      System.exit(0);
               }
        });
        pantalla.setSize(350,300);
        pantalla.show();
 }

  public static void main(String args[]) {
    AppMultiListener demo = new AppMultiListener();
  }
}
```

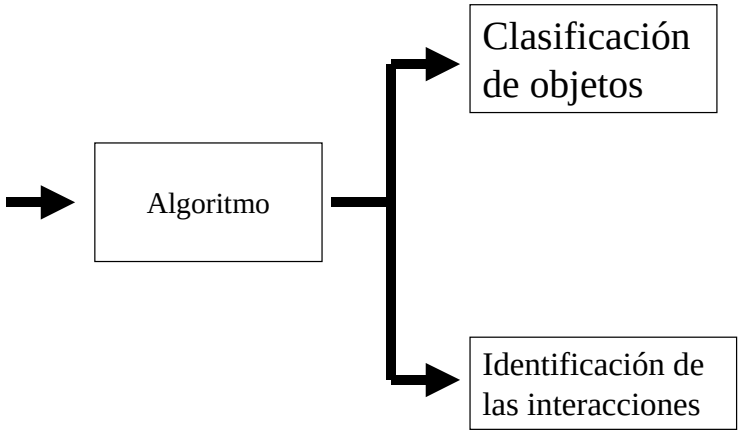


Figura 4.1. Algoritmo de clasificación de objetos.

En la figura 4.1 se muestra la entrada de un código fuente en Java y los productos generados por el algoritmo.

Cabe señalar que durante el proceso de clasificación e identificación de los objetos, la información obtenida se almacena en el repositorio de información a manera de tablas y relaciones, con la finalidad de que en un futuro identificar patrones de diseño.

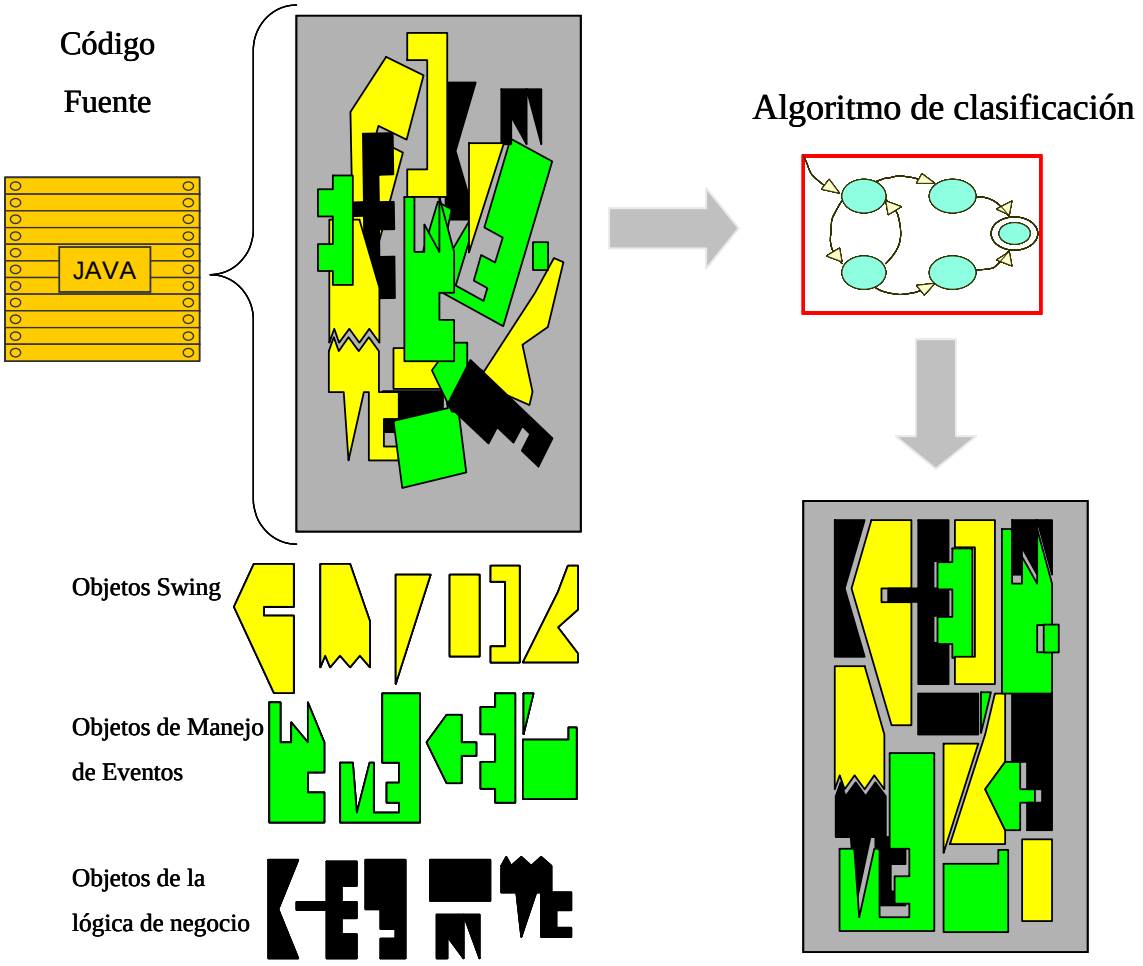


Figura 4.2. Identificación de la taxonomía de objetos

La utilidad del algoritmo de clasificación de objetos se puede apreciar conceptualmente como se ilustra en la figura 4.2, ya que el código fuente puede ser visto como un rompecabezas desordenado, donde las piezas son las clases, y después de la aplicación del algoritmo se puede apreciar perfectamente identificadas las clases de acuerdo a la taxonomía definida dentro del modelo.

4.2.2 Forma canónica para la representación de la interacción entre objetos

La forma canónica para la representación de la interacción entre objetos, sirve para representar la secuencia del envío/recepción de mensajes entre objetos en una secuencia de tiempo. Así mismo, se utilizan los símbolos terminales para representar las relaciones estructurales del código fuente en su forma canónica. Cabe hacer la aclaración que las reglas de producción del modelo no se aplican para la forma canónica del código, sino que se utilizan las reglas del lenguaje nativo, en este caso Java 1.2/Swing.

De manera específica la forma canónica consiste en una gramática libre al contexto[1] en notación BNF (Backus-Naur Form), la cual se enuncia a continuación:

Símbolos no terminales o variables*:*

<interacción>
<acción>
<clase>
<conjunto_def>
<def>
<coma>
<INV>
<EJE>
<fin>
<metodo>

Símbolos terminales*:*

“ ,”
INVOCA
EJECUTA
“ ” (espacio en blanco)
Identificador
“;”

El conjunto de producciones es:

<interacción> := <clase><accion> | <clase>
<clase> := <id><def> | <id>
<INV> := INVOCA
<EJE> := EJECUTA
<def > := <INV><clase><metodo><fin>
<def > := <EJE><metodo><fin>
<coma> := ,
<fin> := “ ” | <coma>
<id> := (“a”-“z” | “A” – “Z” | “_” | “0” – “9”)
<metodo> := <id><PA><argumentos><PC>
<argumentos>:=[<tipo><id>[<coma>]]*
<argumentos>:= [<creacion_objeto>][<coma>]]*
<argumentos>:=<clase_inner>

[1] Como se mencionó, la gramática se considera libre del contexto debido al tipo de reglas de producción que posee.

<tipo>:=<TipoNumerico>|<boolean>
<TipoNumerico>:=<TipoIntegral>|<TipoPuntoFlotante>
<TipoIntegral>:=<byte>|<short>|<int>|<long>|<char>
<TipoPuntoFlotante>:=<float>|<double>

El símbolo inicial es:
<interacción>

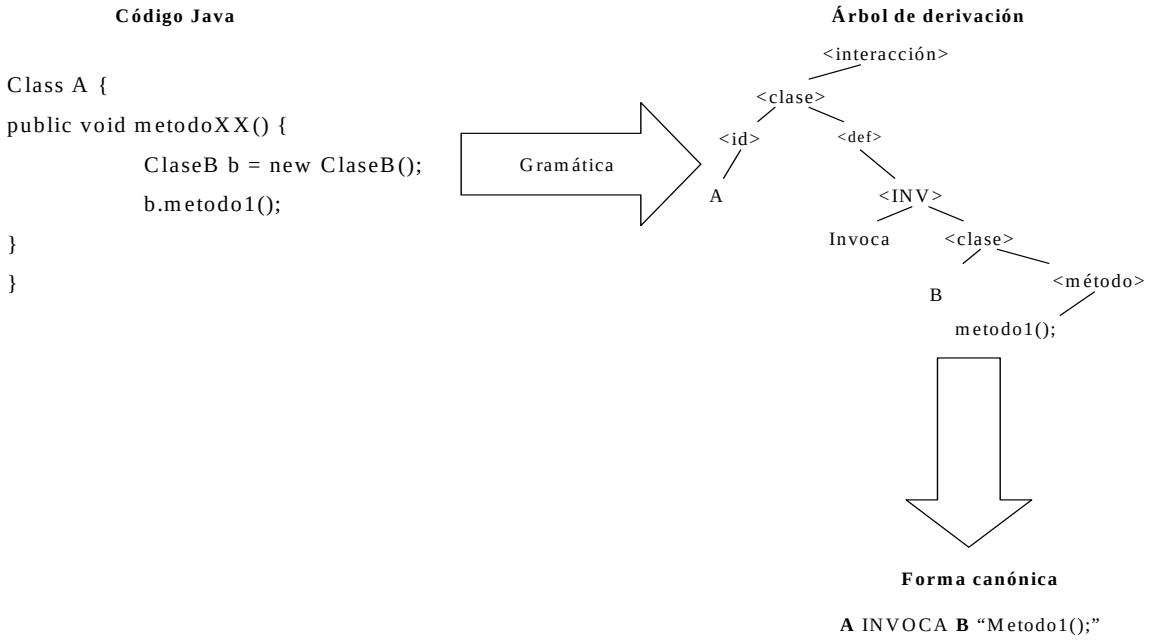


Figura 4.3 Ejemplo de un árbol de derivación, utilizando la forma canónica.

En la figura 4.3 se muestra el árbol de derivación que se obtiene al aplicar las reglas de producción de la gramática definida dentro del modelo.

4.2.3 Función de transformación de la forma canónica a su representación visual de interacción entre objetos

A continuación se describe la función de transformación de la forma canónica generada en el punto 4.2.2 a su forma visual, tomando como base las definiciones realizadas por UML [OMG] para su presentación en el diagrama de secuencia.

La función de transformación está compuesta por los siguientes conjuntos:

C: {Conjunto de Objetos}={A, B, C, ..., OBJ*n*}
I: {Conjunto de Interacciones}={INVOCA, EJECUTA}
M:{Conjunto de métodos}={Metodo1(), Metodo2(), ..., Metodo*n*()}

Cuya relación es la siguiente:

Sea

$$\mathbf{F: C \times I \times C \times M \longrightarrow CV \times IV \times CV}$$

$$\mathbf{FCS = C \times I \times C \times M \; ; \; VD = CV \times IV \times CV}$$

$$\mathbf{F : FCS \longrightarrow VD}$$

Con lo que se tiene que

CV: {Conjunto de Objetos Visuales}={ A , B , C ,..., *N* }

IV: {Conjunto de Interacciones Visuales}={ Metodo1(), Metodo2(), ..., Metodon() ,

Metodo1(), Metodo2(), ..., Metodon()

Metodo1(), Metodo2(), ..., Metodon()

}

FCS
Forma canónica de secuenciación.

VD
Visualización del diagrama.

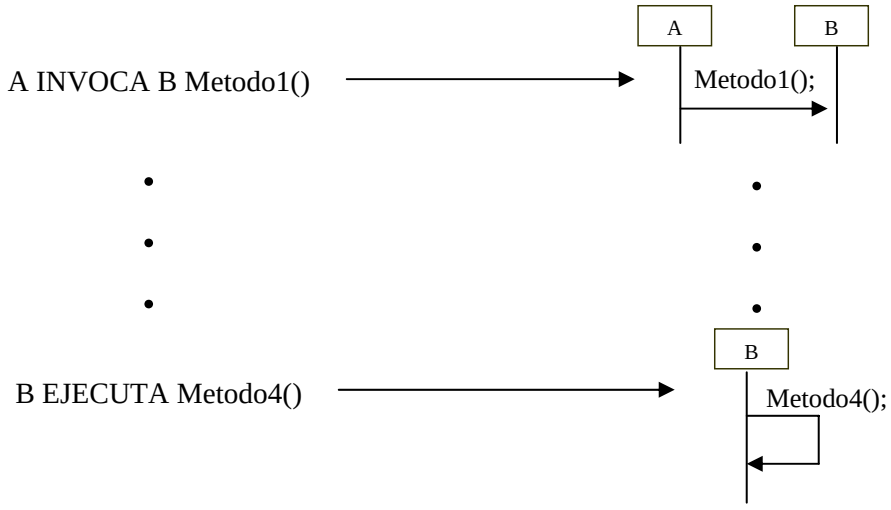


Figura 4.4. Muestra de la función de transformación aplicada a las dos interacciones: INVOCA y EJECUTA.

A continuación se presenta un ejemplo de cómo trabaja la función de transformación.

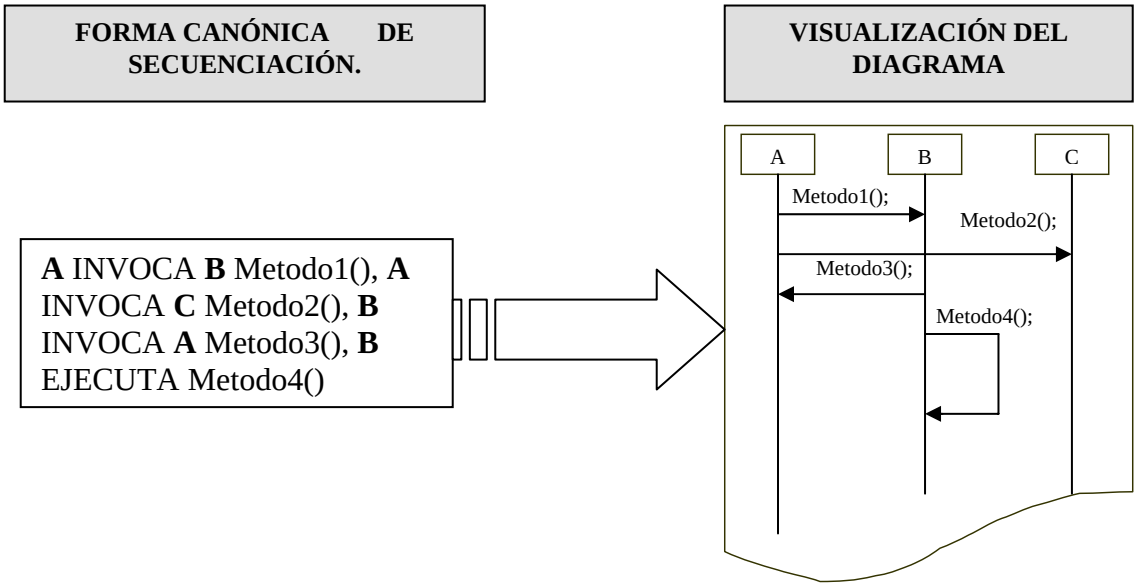


Figura 4.5. Muestra de un modelo canónico de secuenciación y su correspondiente diagrama.

4.3 ARQUITECTURA DE ODICC-Swing

De manera general la herramienta[2] ésta constituida por tres grandes módulos que son:

- Módulo clasificador de objetos,
- Módulo para transformación del código Java-Swing a la forma canónica, y
- Módulo de visualización gráfico del diagrama de secuencia.

Los cuales se muestran en la figura 4.6; la herramienta fue desarrollada en el lenguaje de programación Java. En el apéndice D se muestra el diagrama de clases en notación OMT, así como una breve descripción de su funcionamiento. En los siguientes apartados se explicará el funcionamiento de los módulos que integran la herramienta de software.

[2] A lo largo del documento, se manejará de manera indistinta los términos “herramienta” y “sistema”, para referirse a ODICC-Swing.

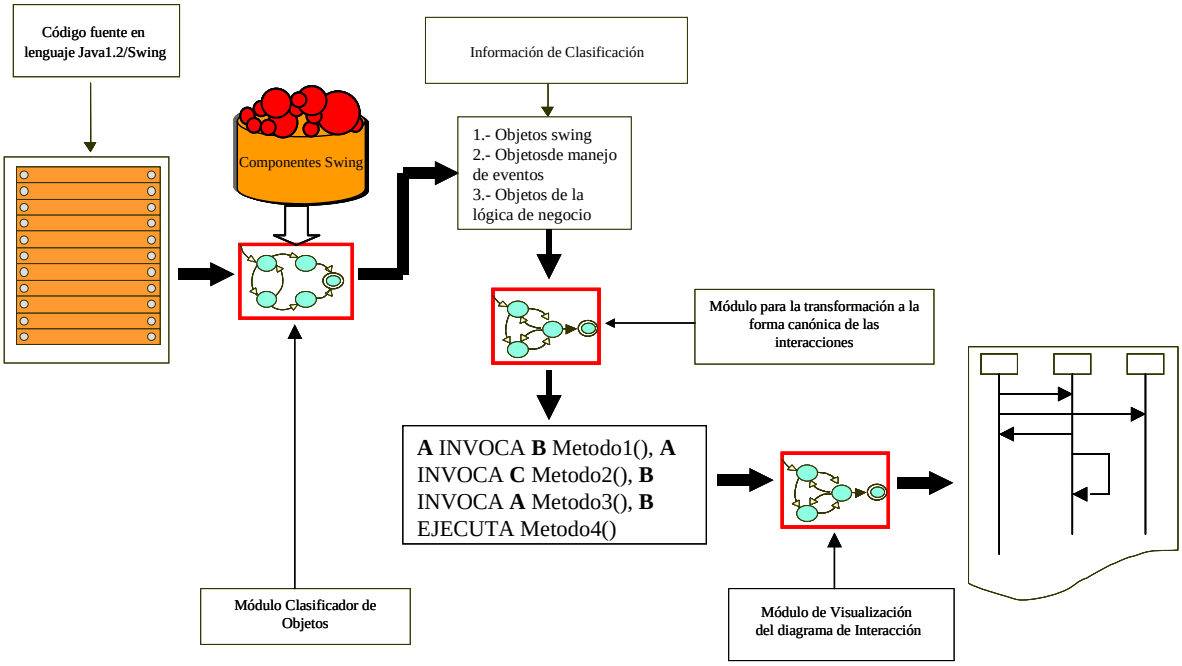


Figura 4.6. Estructura general de la herramienta ODICC-SWING

4.3.1 Módulo clasificador de Objetos

Este módulo analiza en primera instancia todo el código `Java`, el análisis que realiza es a través de la técnica de Ingeniería en Reversa llamada *parsing* [AHO90]; éste módulo realiza tres diferentes tareas, los cuales se describen a continuación:

Tarea 1. Se realiza la lectura de los ficheros de entrada (código escrito en el lenguaje Java 1.2/Swing) con la extensión `.java`. El módulo toma como entrada el conjunto de clases en Java, que pueden estar en uno o varios archivos, la restricción es que deben estar en el mismo subdirectorio.

Tarea 2. Se realiza una comparación de todos los objetos que son creados en el código de la aplicación analizada contra una base de datos previamente creada y almacenada en Access 2000, que contiene información acerca de los componentes que son propietarios de Java/Swing, utilizando el algoritmo descrito en el apartado 4.2.1.

Al momento de realizar el paso anterior, también se obtiene una taxonomía de los objetos que se encuentran declarados en el código fuente.

Para realizar su función, este módulo reconoce en el código analizado los componentes que pertenecen a Swing, ya que son los componentes que se encuentran almacenados en el repositorio de objetos, para cambiar la versión de los componentes por analizar a por ejemplo Swing 1.X, lo que hay que hacer es actualizar la base de datos que contiene las definiciones.

El repositorio de objetos es de gran utilidad ya que proporciona información acerca de la estructura de clases que tienen los componentes gráficos utilizados en la aplicación analizada. Lo anterior ayuda al ingeniero de software a lograr una mejor comprensión acerca de que tipos de objetos son utilizados para la construcción de la interfaz gráfica.

Tarea 3. Al momento de realizar el análisis que detectará las interacciones entre objetos: *invoca, ejecuta;* se genera cierta información (ver Apéndice C) que será almacenada en el repositorio junto a las definiciones de Swing, ésta información es la que sirve de base para la construcción del modelo canónico de interacción.

Las entradas y salidas de este módulo se visualizan en la figura 4.7. Las clases de la herramienta que pertenecen a este módulo son: `main` y `clasificador` del diagrama de clases que se presenta en el apéndice D.

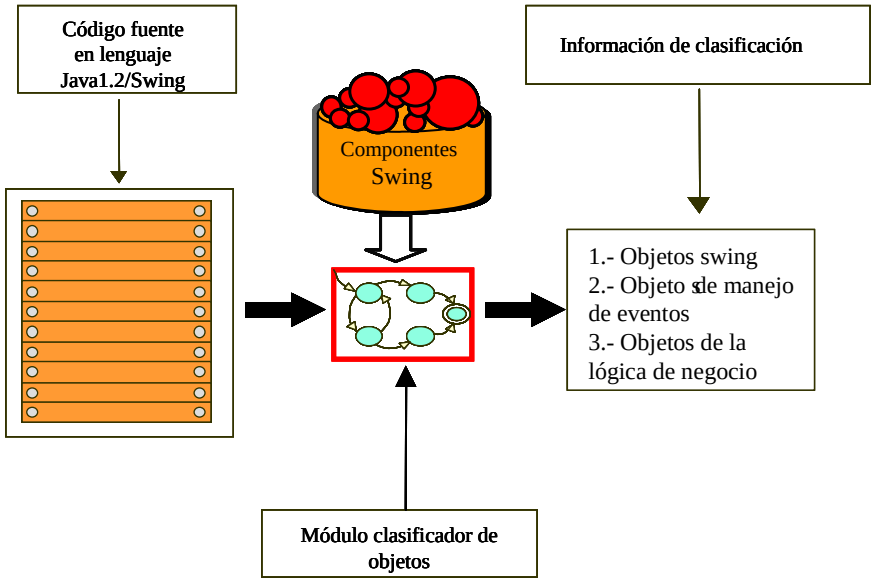


Figura 4.7 Módulo clasificador de objetos.

Las restricciones de este módulo son las siguientes:

1) Se analiza únicamente código en lenguaje Java 1.2 y que además utilice Swing, para la construcción de la interfaz gráfica de usuario.

2) En ésta herramienta no se verifican errores en el código analizado, por lo tanto el mismo no debe tener errores de compilación, ni ejecución.

3) Al momento de ejecutar este módulo, la información que se almacena en el repositorio (Apéndice C) concerniente a la aplicación analizada anteriormente se borrará,

para dar paso a la nueva información del proyecto actual (opción archivo, borrar proyecto de la aplicación). Es decir borra la corrida anterior del programa.

4.3.2 Módulo para transformación del código Java-Swing a la forma canónica para la representación de la interacción entre objetos

Una vez que se han clasificado los objetos del código fuente y se han llenado las tablas correspondientes, se procede a construir la forma canónica de interacción de objetos, aplicando las reglas de producción que se describe en el apartado 4.2.2

Se extrae información del repositorio mediante consultas, las cuales dan como resultado un conjunto de objetos que este módulo se encarga de convertir a una cadena que es la forma canónica de interacción de objetos, y ésta a su vez es pasada como parámetro al siguiente módulo. En la figura 4.8 se muestra las entradas y salidas de este módulo.

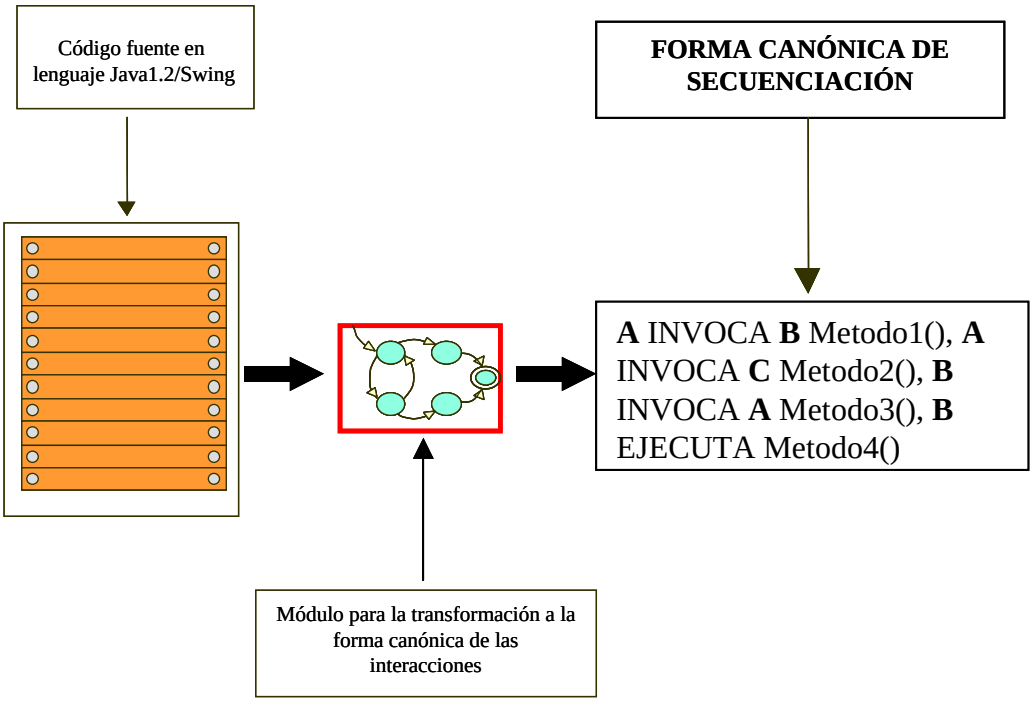


Figura 4.8 Entradas y salidas del módulo de transformación del código Java-Swing a la forma canónica.

Cabe mencionar que el resultado de este módulo es la forma canónica de secuenciación de objetos, la cual es una cadena que contiene la secuencia de peticiones de ejecución de métodos entre los diferentes objetos de la aplicación.

4.3.3 Módulo de visualización gráfica del diagrama de interacción

Este módulo es el encargado de realizar el despliegue visual del diagrama de secuencia construido a partir del modelo canónico.

Este módulo toma como entrada la forma canónica producida por el módulo descrito en el apartado 4.2.2, que es una cadena, y a partir del cual se aplica la función de transformación descrita en el apartado 4.2.3, para construir el diagrama de interacción de clases. La Figura 4.9 muestra las entradas y salidas de este módulo.

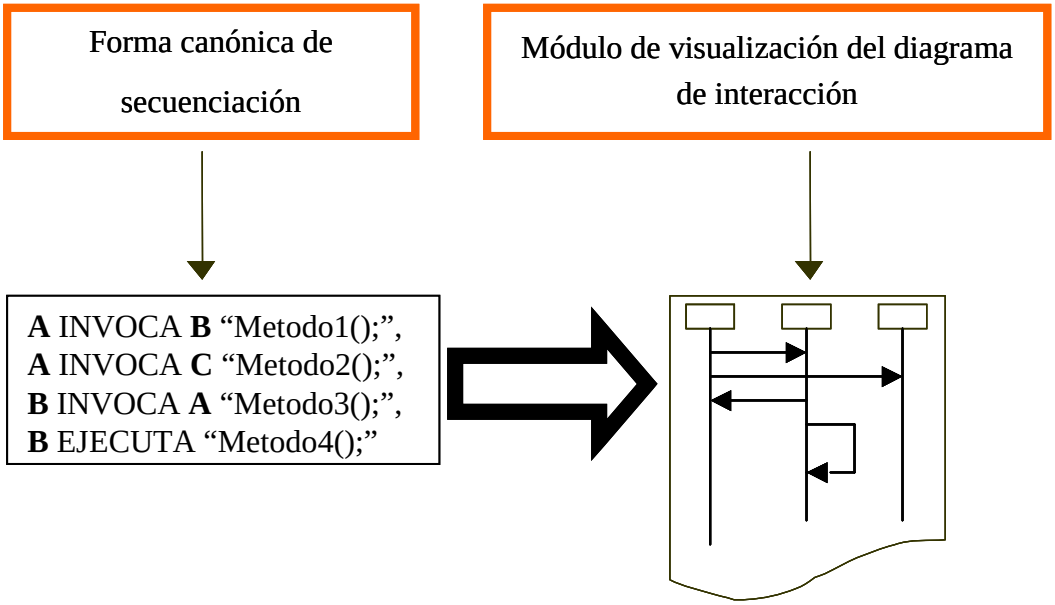


Figura 4.9 Entradas y salidas del módulo de visualización del diagrama de interacción.

4.4 INTERFAZ AL USUARIO

La herramienta proporciona al usuario un entorno gráfico, la pantalla principal contiene una barra con dos menús, en el menú *Archivo*, se selecciona la opción correspondiente; el menú *Ayuda*, contiene una opción, que es acerca de la herramienta. En la figura 4.10 y 4.11 se muestran las opciones de tales menús.

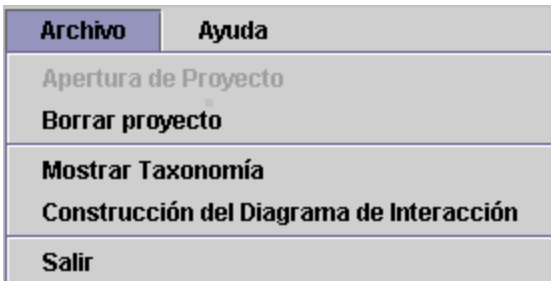


Figura 4.10 Menú Archivo de la herramienta.

En el menú *archivo* se pueden observar tres opciones que se describen en el apartado 4.3.1.

Figura 4.11 Menú Ayuda de la herramienta.

4.4.1 Menú archivo

Dentro del menú *archivo* se encuentran algunas opciones, las cuales se describen a continuación.

La opción de a*pertura de proyectos*, es aquella en la cual el usuario selecciona los archivos con extensión `.java`, que contienen el código fuente por analizar, al seleccionar esta opción se muestra una pantalla como se ilustra en la figura 4.12. Cabe mencionar que el sistema leerá todos los archivos con la extensión `.java` que se encuentran en el subdirectorio al momento de seleccionar el primer archivo de lectura.

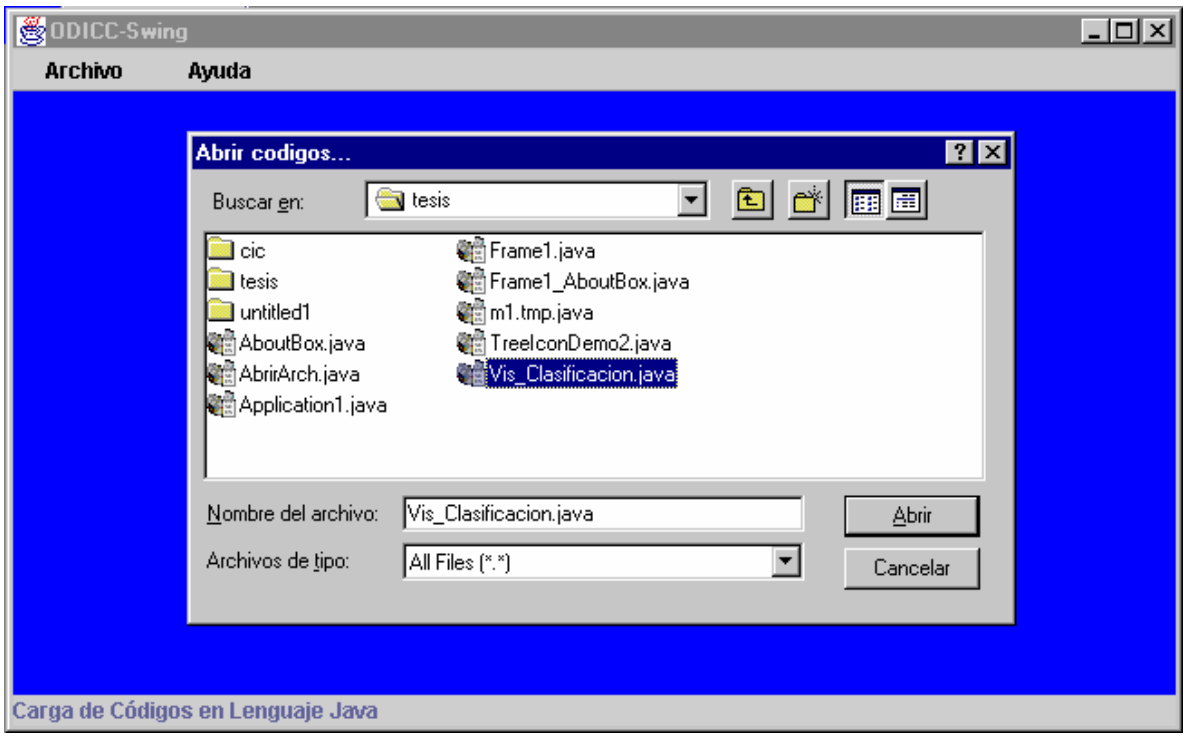


Figura 4.12 Pantalla de la opción *Apertura de proyecto*.

La opción de b*orrar proyecto*, es aquella en la cual el usuario limpiará el repositorio que contiene las interacciones preparando el sistema para realizar una a*pertura de Proyecto*. Al seleccionar ésta opción el sistema únicamente preguntara si ésta realmente seguro de borrar el repositorio, a lo cual el usuario deberá contestar afirmativamente para realizar la ejecución de este módulo (ver figura 4.13).

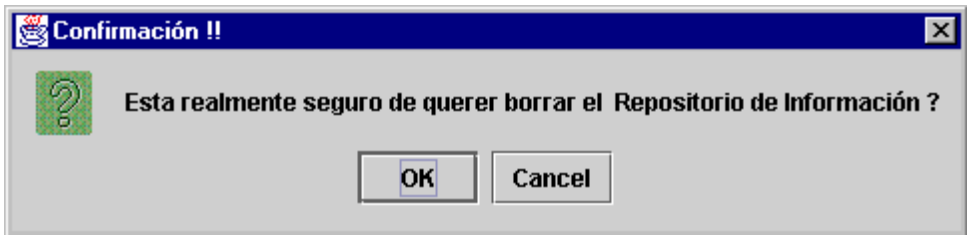


Figura 4.13 Pantalla de confirmación de borrado del repositorio

La opción de *mostrar taxonomía*, es aquella en la que el ingeniero de software, selecciona para visualizar la clasificación obtenida en el análisis del código fuente, al seleccionar ésta opción se presenta una imagen que se muestra en la figura 4.14, en donde aparecen tres espacios de selección que muestran la siguiente información:

En la tableta "*Objetos de manejo de eventos*", se presentan los objetos que la aplicación detectó que contienen código para realizar una acción cuando en la aplicación analizada sea generado un evento.

En la tableta "*Objetos del dominio de aplicación*", se presentan los objetos propios del dominio y que contiene la lógica del negocio.

En la tableta "Componentes Swing", se presentan los objetos que forman parte de la definición de objetos propios de Swing.

En las tres tabletas anteriores se mostrará, la siguiente información de las clases encontradas:

- **Nombre**: Es el nombre encontrado en la definición de la clase.
- **Extends**: Contiene el nombre de la clase que extiende, sirve para identificar las relaciones de herencia entre las definiciones de las clases.
- **Implements**: Contiene el nombre de las interfaces que implementa ésta clase.
- **Comentario**: Se despliega un comentario relacionado con la clase si es que se encontró alguno en la declaración de la misma, al estilo javadoc [LAM01].

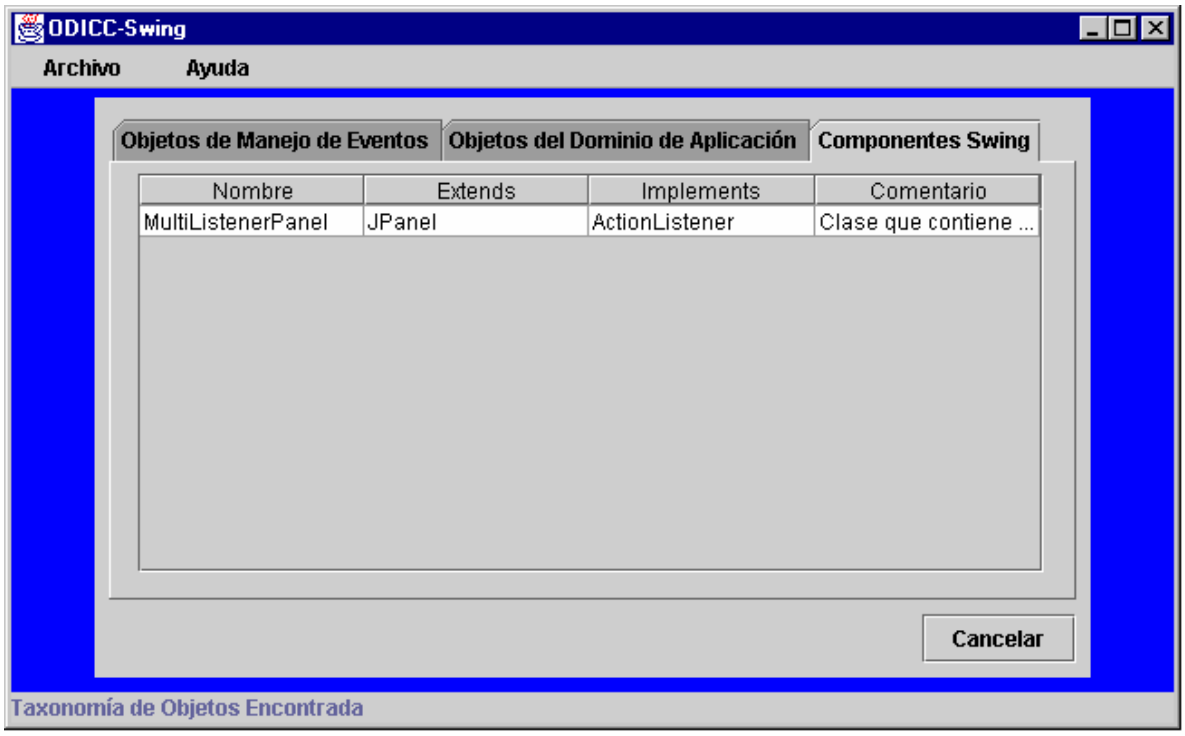


Figura 4.14 Pantalla que muestra la taxonomía realizada por la aplicación de los objetos encontrados en la aplicación analizada.

La opción de *Construcción del diagrama de interacción,* es la más importante de la herramienta, ya que es donde se realiza el análisis de eventos y la construcción del diagrama de secuencia del evento analizado. Cuando se selecciona esta opción se presenta una pantalla como la que se muestra en la figura 4.15, en donde se muestran los clases con sus métodos definidas en el código fuente ingresado en la primera opción. Aquí es donde se elige la clase y método origen por analizar, una vez seleccionado el método origen se pulsa el botón "Construir", para que la herramienta genere el modelo canónico de interacción y a continuación presente el diagrama de secuencia para ese método en particular (Figura 4.16). Ésta pantalla además permite presentar a los objetos participantes en la interacción ya sea por nombre de clase o bien por nombre de variable declaradas en el código.

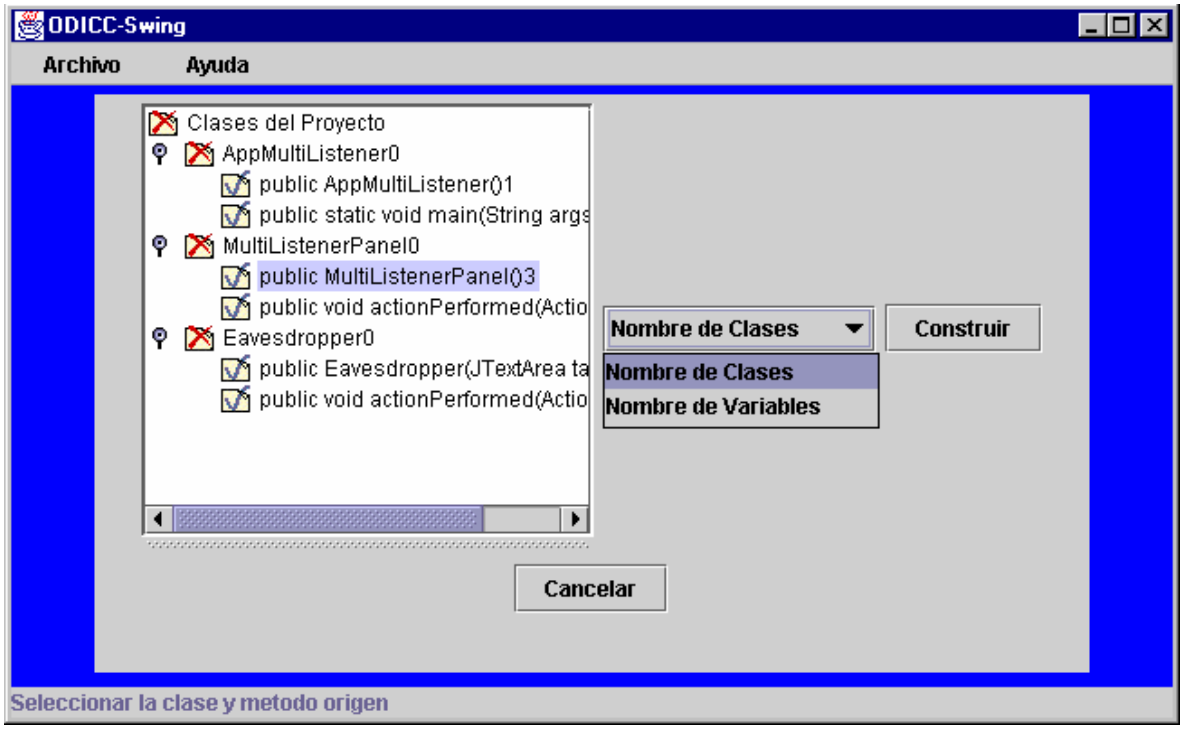


Figura 4.15 Pantalla de selección de la clase y método origen para construir el diagrama de secuencia.

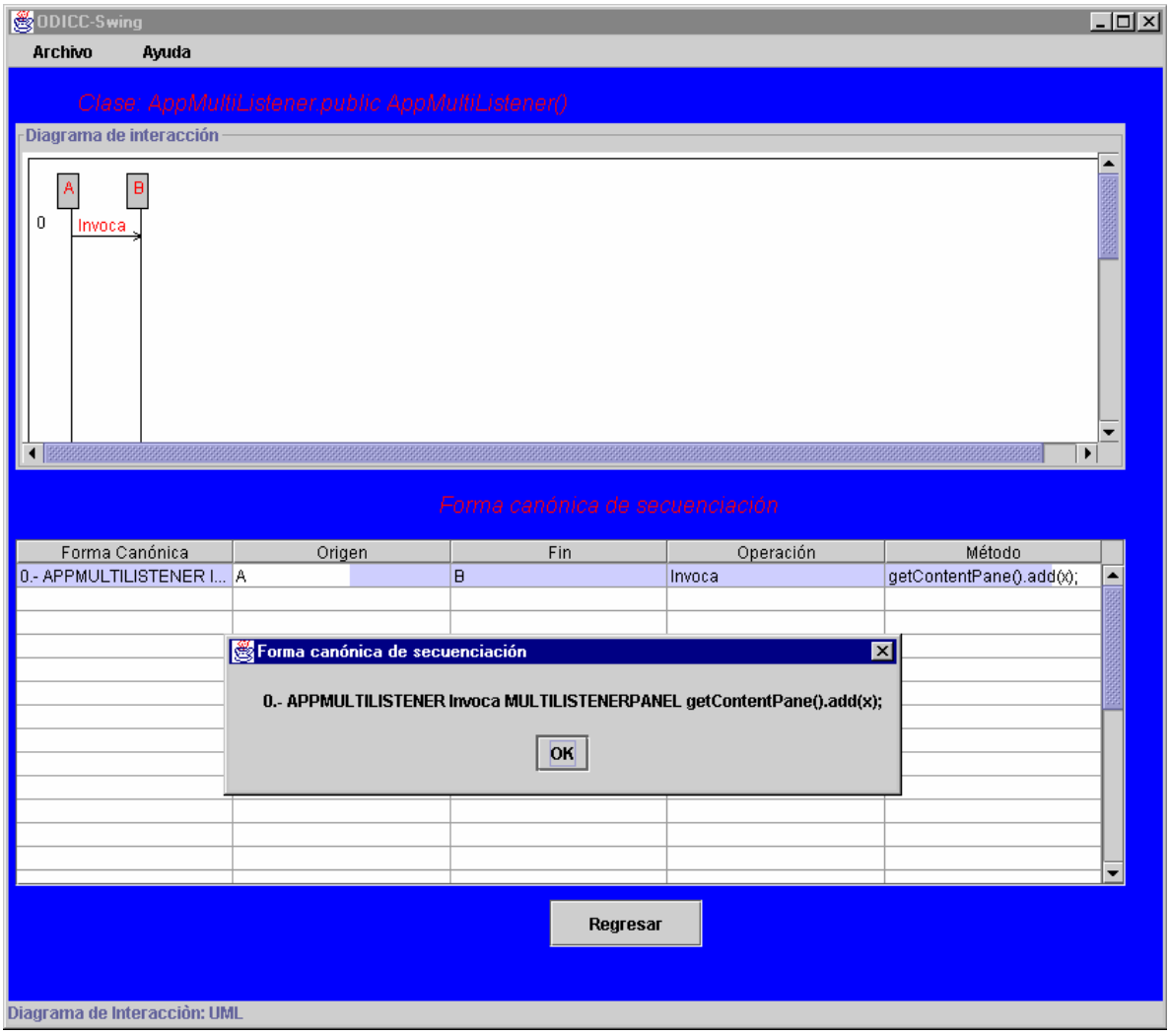


Figura 4.16 Muestra de un diagrama de secuencia.

En la parte superior se muestra el nombre de la clase y método origen, que fue seleccionado en el paso anterior.

Al momento en que la herramienta realiza la construcción del diagrama de secuencia, también genera la información necesaria para la construcción de la forma canónica de secuenciación, dicha información se muestra en la tabla inferior del diagrama construido, esta tabla contiene información de mapeo de la forma canónica al diagrama de secuencia. La tabla contiene las siguientes columnas descritas a continuación:

- **Forma canónica**: Muestra el número secuencial de la interacción representada visualmente, así como la forma canónica completa, con los nombres de los objetos, ya sea en forma de clase o bien con nombre de variable.
- **Origen**: Representa el símbolo de la interacción origen dentro del diagrama de secuencia gráfico.

- **Fin**: Representa el símbolo de la interacción final dentro del diagrama de secuencia gráfico.
- **Operación**: Muestra una de las interacciones detectadas ya sea INVOCA o bien EJECUTA según sea el caso.
- **Método**: Muestra la firma del método con todos sus parámetros necesarios para la interacción.

Cuando se selecciona un renglón en la tabla, el sistema despliega una caja de dialogo que se muestra en la figura 4.17. En ella se indica la forma canónica de la interacción seleccionada. Con nombre de clases o bien de variables, según lo seleccionado en la pantalla de selección de origen, ver figura 4.15.

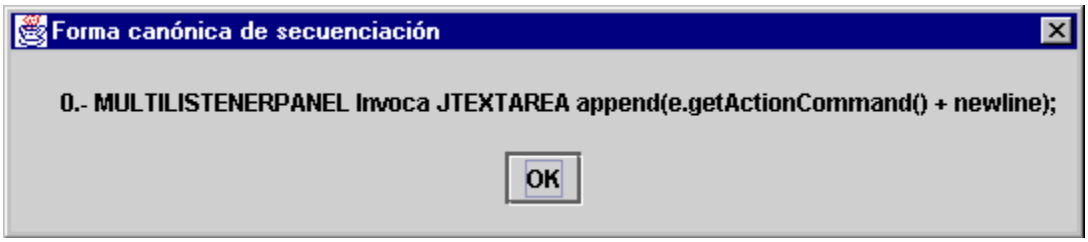


Figura 4.17 Muestra una forma canónica seleccionada

4.4.2 Menú ayuda

En este menú se presenta la información referente a la información de la herramienta, como lo es Versión, Créditos, Contactar y Legal, la cual se presenta en la figura 4.18. En la figura 4.18 se muestra la versión de la herramienta, además del software utilizado en la elaboración del mismo y la fecha de creación.

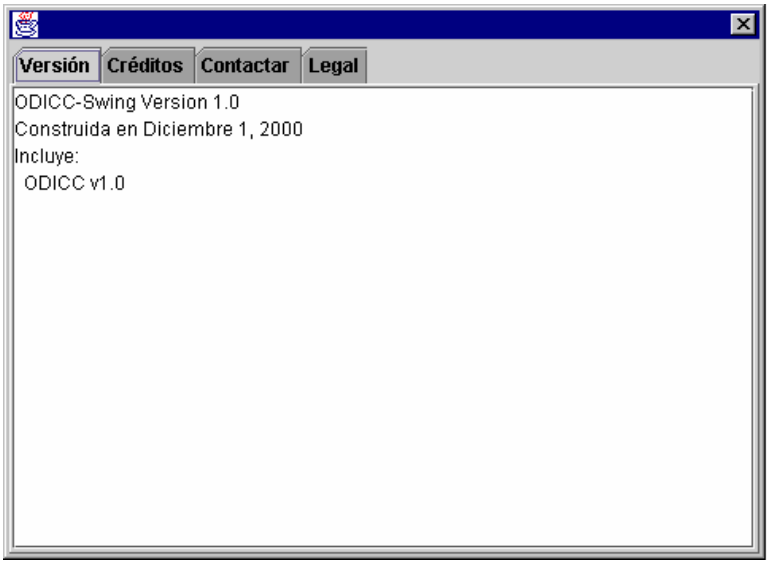


Figura 4.18 Figura de información de la herramienta.

La figura 4.19 muestra los créditos a las personas involucradas en la realización del proyecto, además del lugar de realización del mismo.

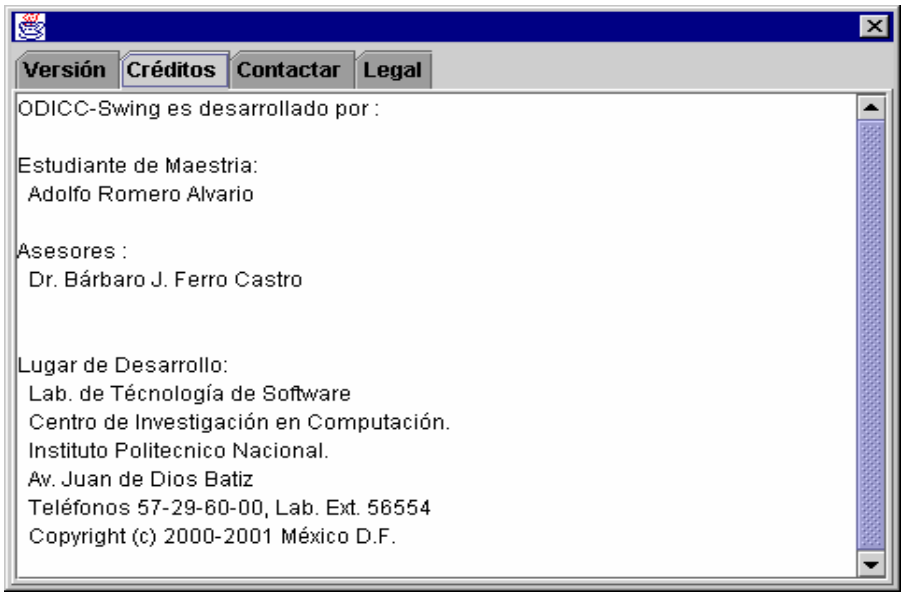


Figura 4.19 Muestra los créditos de las personas que participaron en la realización de este proyecto.

La figura 4.20, muestra la información necesaria para realizar un contacto con el autor del proyecto, así como el sitio WEB, en donde se puede encontrar información referente al mismo.

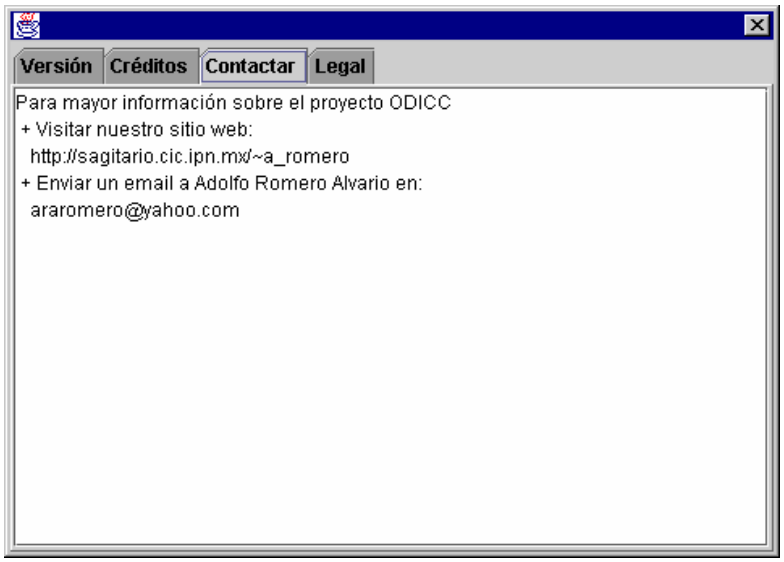


Figura 4.20 Información referente al proyecto.

La figura 4.21 muestra información legal acerca de los derechos de autor, sobre el código y los productos que de él pueden derivarse.

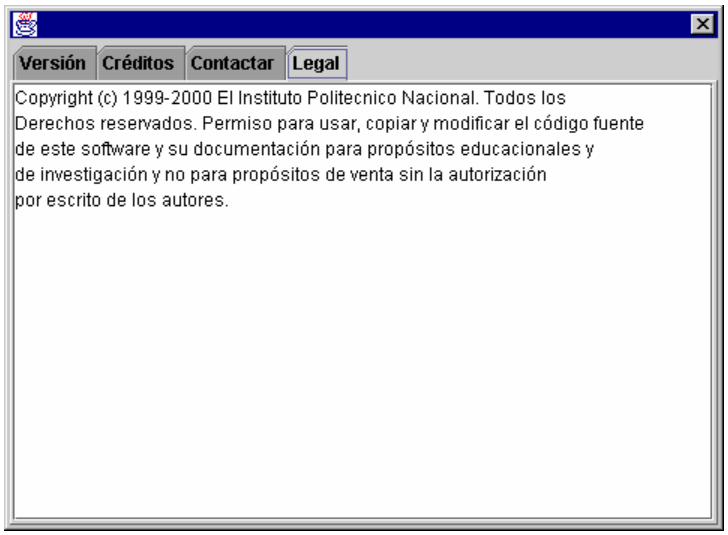


Figura 4.21 Información legal del proyecto.

Capítulo

5

Evaluación Experimental

En este capítulo se describe la forma en que se llevó a cabo la evaluación de la herramienta generada en este proyecto de tesis. Se enuncian las preguntas de investigación, las variables y términos de la investigación y sus definiciones, la hipótesis de investigación, la definición de los casos de estudio, los instrumentos de medición aplicados, el resumen del procedimiento en el desarrollo del experimento, los resultados de las pruebas, y el resumen de resultados.

5.1 VARIABLES Y TÉRMINOS DE INVESTIGACIÓN Y SUS DEFINICIONES

La herramienta generada en el presente proyecto de tesis es de cierta forma una herramienta de documentación y análisis de sistemas, ya que a través de ella se pueden obtener diferentes diagramas de interacción de los objetos participantes en una aplicación. También se puede hacer un análisis de la forma en como reacciona una aplicación al disparo de eventos dentro de la interfaz gráfica de una aplicación. Linda H. Rosenberg y Lawrence E. Hyatt, en [LIN01], proponen que la reingeniería es examinar, analizar y alterar los sistemas de software reconstruyéndolos a una nueva forma, así como una subsiguiente implantación. La meta es comprender el sistema de software existente (especificación, diseño e implantación) para entonces rehacerlo incrementado así su funcionalidad, desempeño o implantación.

Aunque los objetivos de las tareas de ingeniería en reversa son determinadas por las metas de los propietarios y usuarios de un sistema, hay dos objetivos generales:

1. *Incrementar la calidad*: Típicamente, los sistemas de software son de baja calidad, porque necesitan muchas modificaciones. Los usuarios y la documentación usualmente no están actualizados con las versiones del software que utilizan. La reingeniería esta intentando mejorar la calidad del software así como producir documentación actual. Incrementar la calidad es necesario para incrementar la rentabilidad, reducir los costos de mantenimiento, y preparar al software para un desarrollo funcional.

2. *Migración*: El software antiguo puede trabajar de acuerdo a las necesidades del usuario, pero esta basado sobre plataformas de hardware, sistemas operativos o lenguajes que han llegado a ser obsoletos y estos sistemas pueden ser rediseñados para migrarlos a nuevas plataformas de hardware o lenguaje. La migración puede involucrar un rediseño extenso si las nuevas plataformas son completamente diferentes como por ejemplo migrar un sistema de mainframe a un sistema basado en un ambiente de red.

En nuestro caso, nos concentraremos en evaluar si nuestra herramienta de software cumple los siguientes dos objetivos:

1. Que ayude en la comprensión de un código existente, concretamente en la parte de la interfaz gráfica de usuario, permitiendo analizar como una aplicación reacciona al disparo de eventos.

2. Construir documentación adicional de diseño, ya que permite construir diferentes diagramas de interacción de los objetos participantes en una aplicación.

5.2 DEFINICIÓN DE CASO DE ESTUDIO

La herramienta generada por el presente proyecto de tesis, se probó con una aplicación típica de Swing, la cual fue obtenida de los programas ejemplos que se encuentran en el sitio de Sun Microsystem [SUN1]. La característica de éste programa es que engloba las relaciones típicas que se pueden representar en un diagrama de interacción.

5.3 Instrumentos de medición aplicados

En el objetivo de esta tesis el único parámetro medible es que la herramienta sea capaz de construir el diagrama de interacción de clases a partir del análisis de código fuente, aplicando el modelo para la obtención de dicho diagrama planteado en el presente trabajo.

Para tales efectos no es relevante medir otras cosas, como: tiempo de respuesta, interfaz de usuario, etc., aunque sí se trató de maximizar la eficiencia de éstos. El instrumento de evaluación aplicado fue propiamente el sistema desarrollado en ésta tesis, el cual nos sirvió para demostrar que el modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente sirve para analizar como una aplicación se comporta desde diferentes métodos miembros de las clases participantes en una aplicación.

5.4 RESUMEN DEL PROCEDIMIENTO EN EL DESARROLLO DEL EXPERIMENTO

- Antes de construir la herramienta se formuló el modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente, el cual dicta las reglas del funcionamiento de la herramienta. Para más información consultar el capitulo 4, y el tema 4.2 y los subtemas 4.2.1, 4.2.2 y 4.2.3.
- La herramienta toma como entrada la lista de archivos con la extensión `.Java` que se encuentra en el subdirectorio que selecciono el usuario (para más información consultar el tema 4.3.1 del capitulo 4) entonces genera la taxonomía de objetos.
- Una vez generada la taxonomía, el módulo para la transformación del código Java-Swing a la forma canónica genera dicha forma, ver tema 4.3.2.
- Con la forma canónica como entrada, el módulo de visualización gráfica del diagrama de interacción, aplica la función de transformación descrita en el apartado 4.2.3 y realiza la construcción del diagrama de interacción.

5.5 CASO DE PRUEBA

El programa con el cual fue probada la herramienta es un programa demostrativo acerca de la utilización de Swing para la construcción de las interfaces gráficas, dicho programa consta de tres clases, las cuales fueron analizadas por ODICC-Swing. A continuación se describen los pasos realizados de acuerdo con el modelo para la obtención

del diagrama de interacción de objetos a partir del análisis de código fuente así como las interfaces generadas por la herramienta en el análisis del programa llamado AppMultiListenerPanel.

El programa se inicia con la opción *Archivo*, *apertura de proyecto*, entonces el sistema presenta una pantalla como la que se muestra en la figura 5.1.

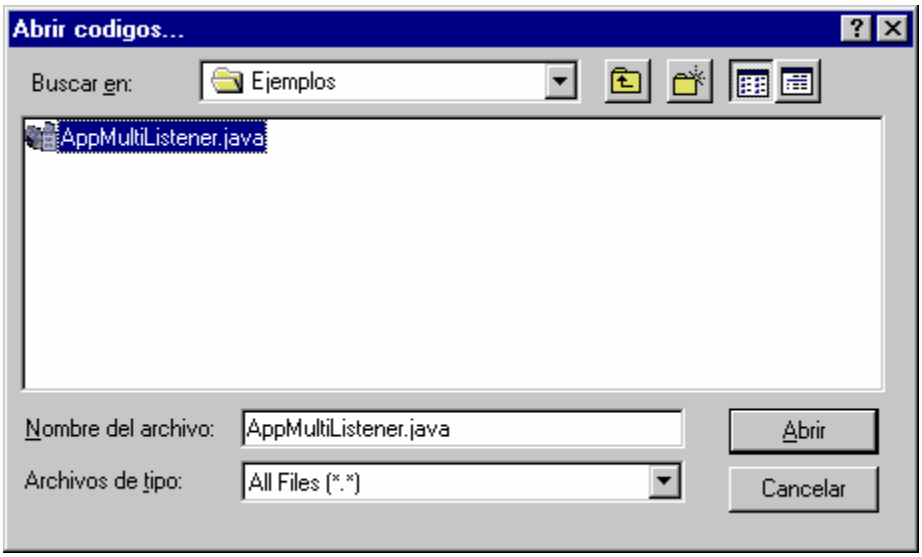


Figura 5.1 Pantalla de carga de las clases de la aplicación AppMultiListener

Una vez realizada la carga del archivo, el sistema analiza el código fuente y genera una clasificación de las clases participantes en la construcción de la aplicación, correspondiente a la figura 5.3.

5.5.1 Algoritmo de clasificación de objetos

En el ejemplo encontramos, tres clases, las cuales debido a la definición que se presentó en el modelo en su parte de clasificación se tiene que:

`AppMultiListener`, debido a que no hereda ni implementa de ninguna clase o interface de los paquetes definidos por Swing para tales propósitos, se considera como un objeto del dominio de aplicación.

`Eavesdropper` implementa una interface llamada `ActionListener` que corresponde al paquete `Java.awt.event`, se considera como de manejo de eventos.

La clase `MultiListenerPanel` extiende de una clase definida dentro del paquete `Javax.Swing`, y por lo tanto se considera como un componente Swing.

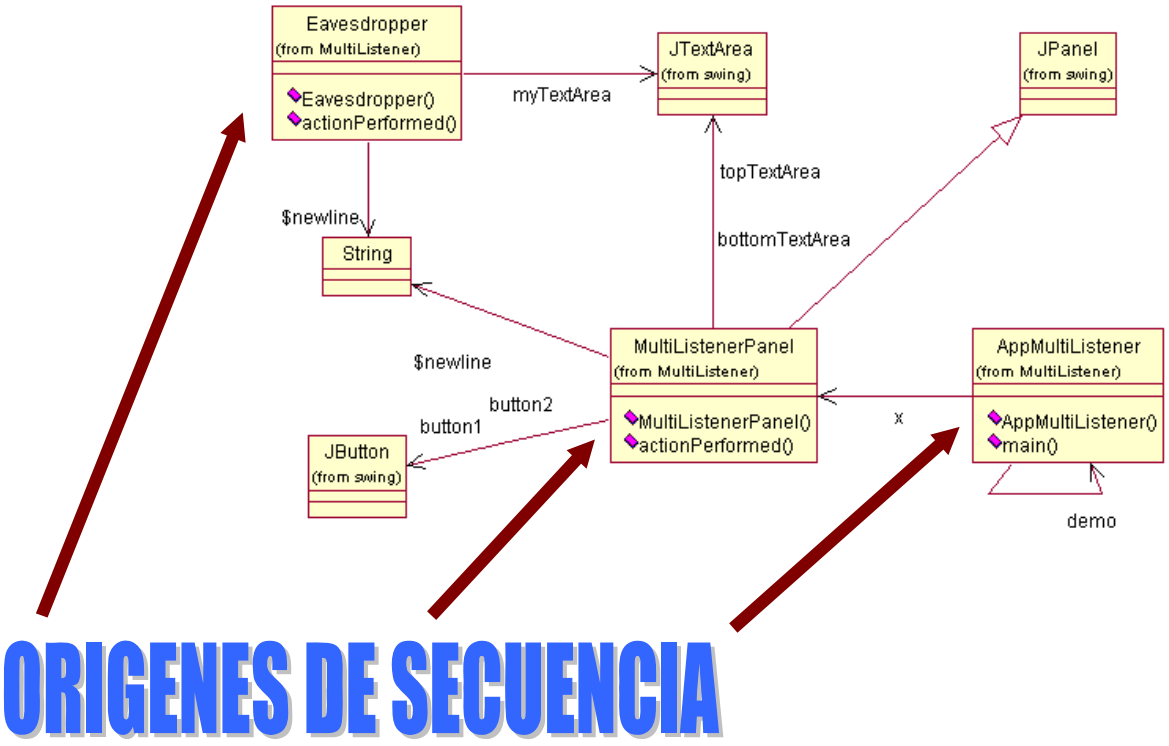


Figura 5.2 Muestra de los orígenes de secuencia

En la figura 5.2 se muestra el diagrama de clases de la aplicación analizada por ODICC-Swing, así como los orígenes de secuencia detectados durante su análisis.

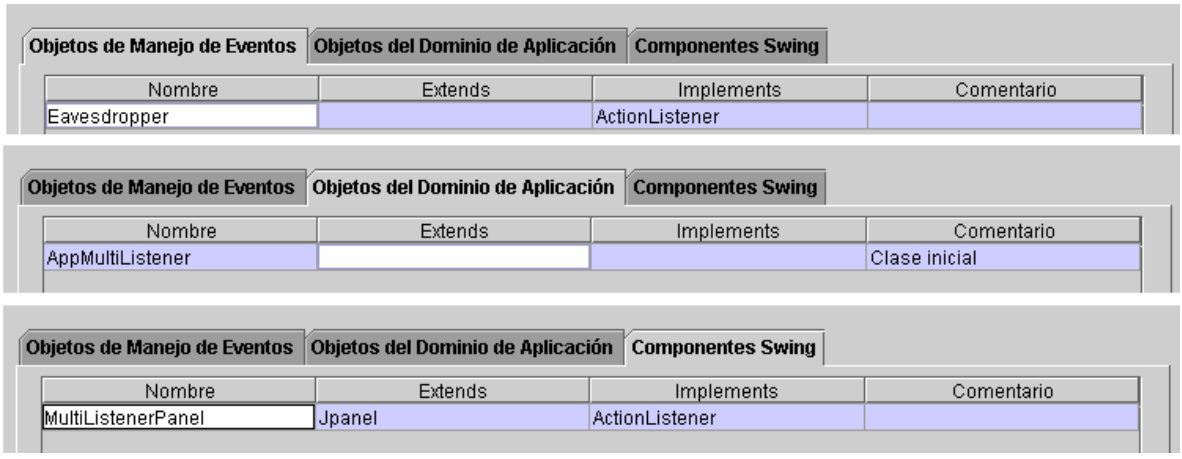


Figura 5.3 Clasificación obtenida por la aplicación.

De ésta manera se pueden obtener los orígenes de las secuencias por analizar quedando como se ilustra en la siguiente tabla:

Número de Origen	Nombre de la Clase	Firma Método
1	AppMultiListener	public AppMultiListener()
2	AppMultiListener	public static void main(String args[])
3	MultiListenerPanel	public MultiListenerPanel()
4	MultiListenerPanel	public void actionPerformed(ActionEvent e)
5	Eavesdropper	public Eavesdropper(JTextArea ta)
6	Eavesdropper	public void actionPerformed(ActionEvent e)

Identificación de Origenes de la clase AppMultiListener

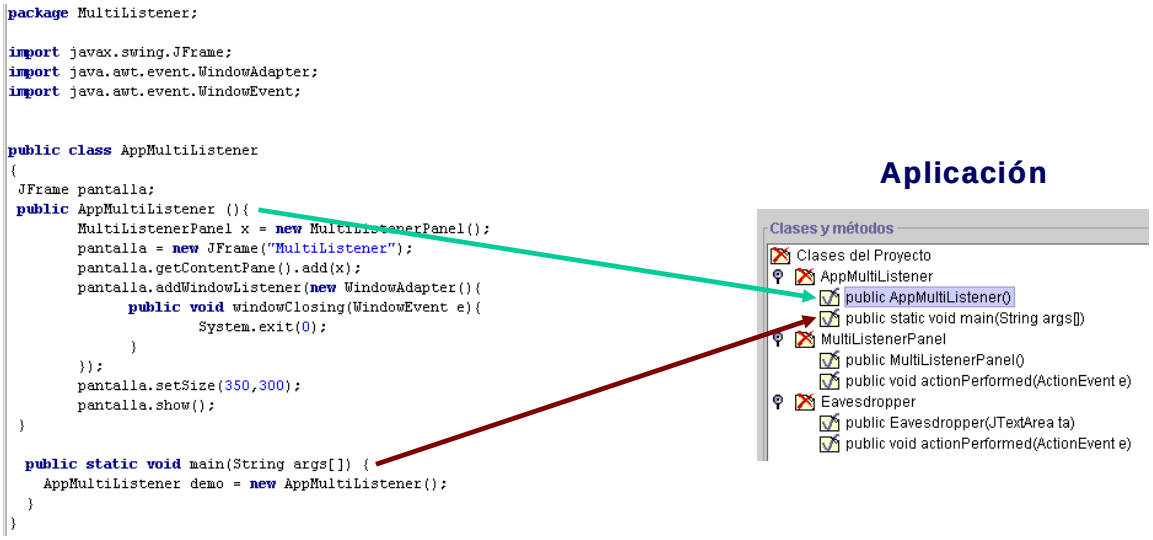


Figura 5.4 Muestra del mapeo del código fuente al despliegue que se realiza dentro de la aplicación.

En la figura 5.4 se muestra el mapeo del código fuente de los orígenes de secuencia identificados durante el análisis de código fuente por la herramienta ODICC-Swing.

5.5.2 Forma canónica para la representación de la interacción entre objetos

A continuación en la figura 5.5 se presenta la forma canónica que construye la aplicación al momento de seleccionar el origen número 1, que corresponde a la clase AppMultiListener, y el método corresponde al constructor.

Origen: 1	public AppMultiListener()
Forma canónica.	

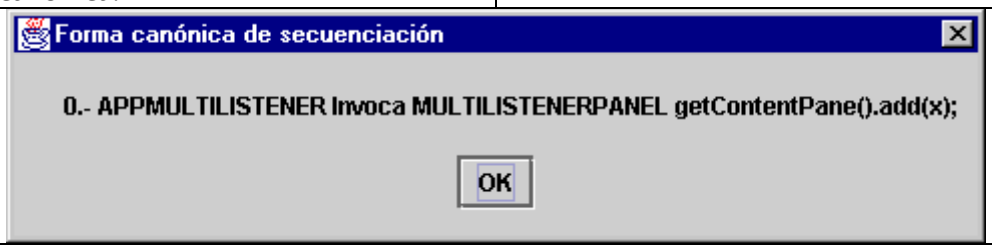
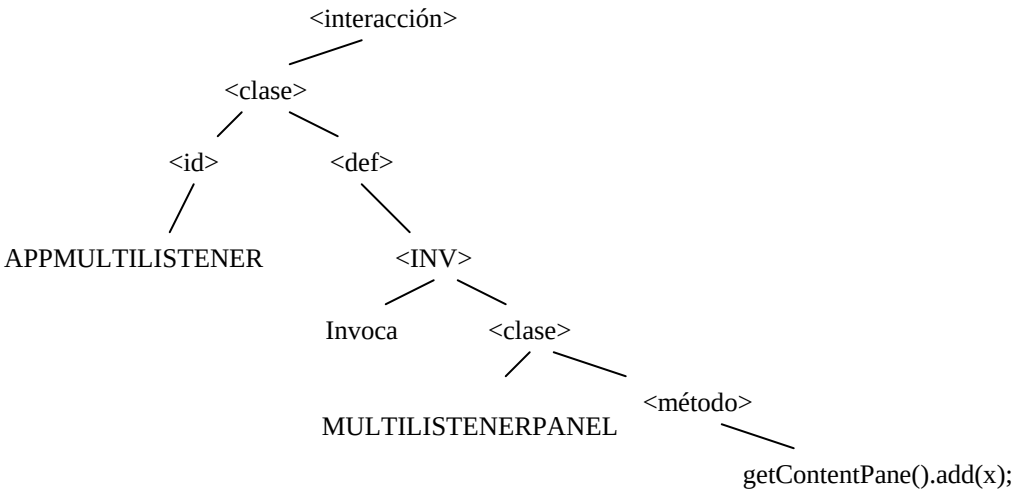
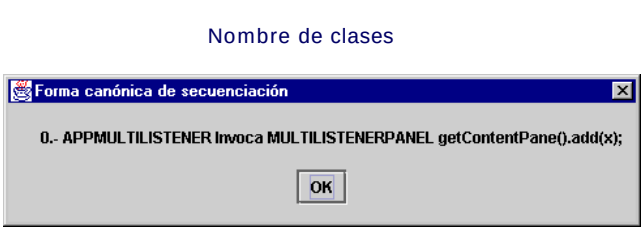


Figura 5.5 Forma canónica para el origen número 1.

En la figura 5.6 se presenta el árbol de derivación mediante el cual se construyó la forma canónica.





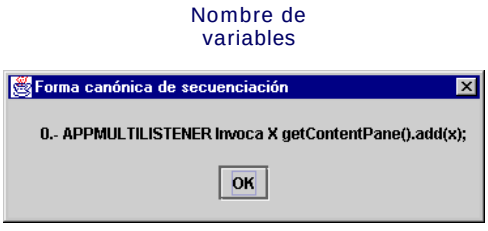


Figura 5.6 Árbol de derivación

Al final de la figura 5.6 se muestra la interfaz que presenta la aplicación cuando se selecciona la opción para que el modelo canónico sea mostrado con el nombre de clases o bien sea desplegado con el nombre de variables utilizadas en el código fuente, esto para darle mayor claridad al ingeniero durante el análisis de las clases de la aplicación.

5.5.3 Función de transformación de la forma canónica a la representación visual de interacción entre objetos

A continuación se presenta el resultado de aplicar la función de transformación a la forma canónica, del ejemplo presentado. Seleccionado el origen

`public void actionPerformed(ActionEvent e) de la clase MultiListenerPane`

Se toman como ejemplos ilustrativos las formas número 0 y número 20, analizando el origen con los nombres de las clases.

0 MULTILISTENERPANE Invoca JPANEL setLayout(gridbag);
donde: MULTILISTENERPANE equivale a A
JPANEL equivale a B

Para quedar el diagrama como se ilustra en la figura 5.7:

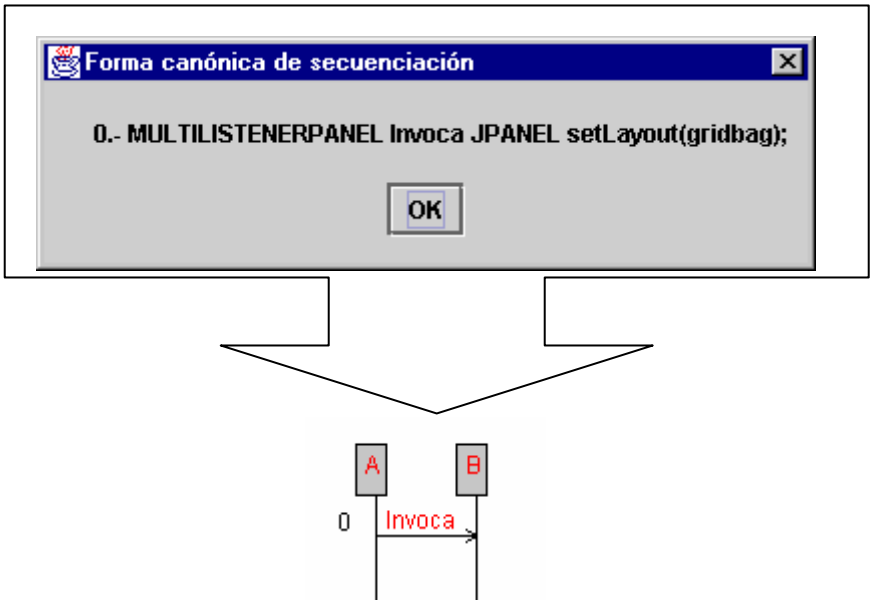


Figura 5.7 Muestra de la construcción de un diagrama de interacción a partir de su forma canónica.

A continuación se ilustra el caso para la forma canónica número 20, que es

20 MULTILISTENERPANEL Ejecuta add(contentPane);

donde: MULTILISTENERPANE equivale a A

quedando el diagrama como se ilustra en la figura 5.8.

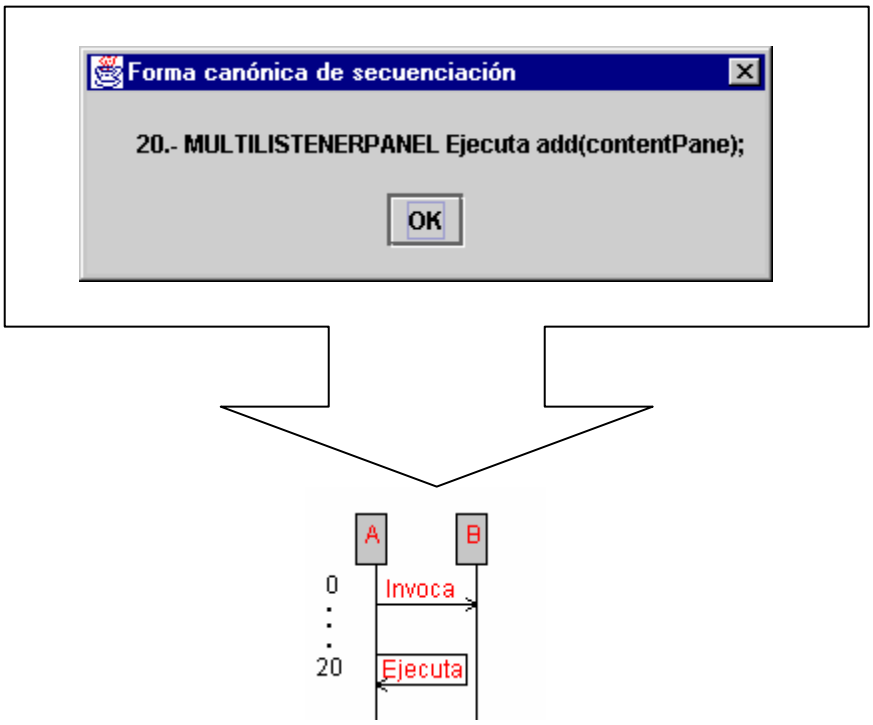


Figura 5.8 Muestra de la construcción de un diagrama de interacción a partir de su forma canónica. Ilustra la relación this.

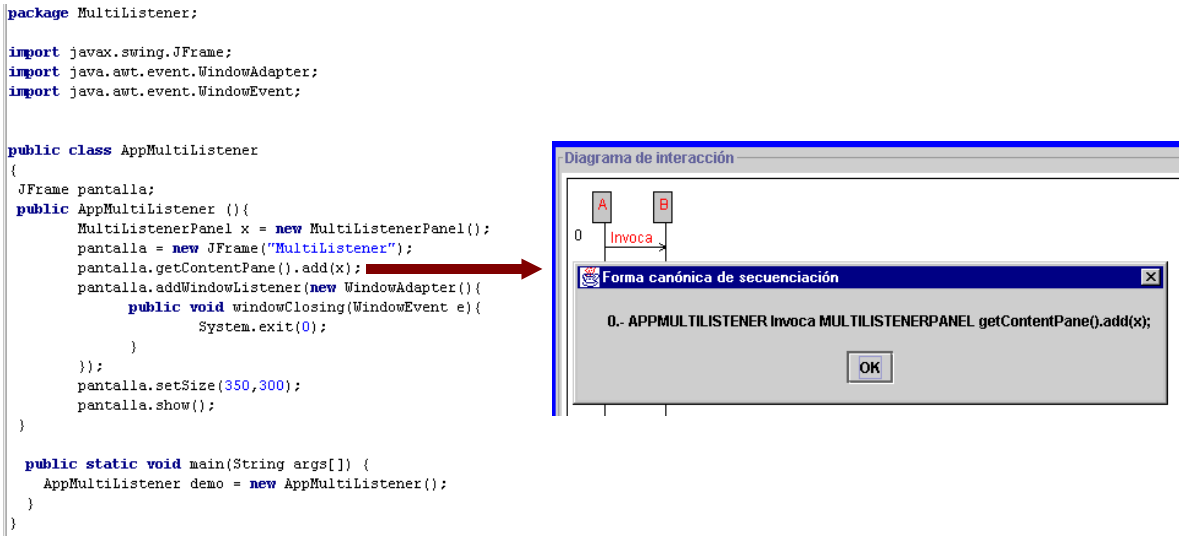


Figura 5.9 Muestra de la invocación desde un origen de un método miembro de otra clase.

La figura 5.9 muestra el mapeo del análisis que realiza ODICC-Swing de una invocación desde un método origen de un método miembro de otra clase participante en la construcción de la aplicación. En la parte de la derecha se ilustra como se muestra lo anterior en el diagrama de interacción, así como la forma canónica generada para tal interacción.

Capítulo

6

Conclusiones

Este capítulo describe los alcances logrados en el presente trabajo de tesis, así como también se mencionan algunas mejoras y trabajos futuros relacionados con esta herramienta.

6.1 CONCLUSIONES GENERALES

El proceso de separación de los objetos que intervienen en la construcción de la interfaz gráfica de usuario de la lógica del negocio es realmente complejo, lo cierto es que la mayoría de las aplicaciones (que usan una interfaz gráfica) se diseñan y construyen de manera monolítica, mezclando las diferentes partes del sistema con acoplamientos demasiado fuertes, que hace complicada la tarea de separación. Sin embargo al identificar la firma de los objetos que pertenecen a la interfaz gráfica (proporcionada por el paquete de Swing) hace más claro el procedimiento a seguir en la búsqueda de los objetos gráficos, y una vez identificados, se puede cambiar su comportamiento para orientarlos hacia una visualización remota.

Ningún método de comprensión de código puede ser enteramente automático, así como también el trabajo de mantenimiento de sistemas. Ya que la comprensión del código fuente es una actividad inteligente, involucra la comprensión de lo que el programador ha hecho, como es que fue alcanzado el diseño original, y porque una forma de implementarlo fue elegida.

Es posible la extracción del diagrama de secuencia de clases a partir del análisis de código fuente, con algunas limitaciones tales como:

1. Definir un origen de análisis, para así tener un punto de partida, en la construcción de la forma canónica de secuenciación de mensajes entre objetos.

2. Al seleccionar el origen, únicamente se obtiene el diagrama de interacción del origen hacia otras clases y no de las otras clases hacia el origen, ya que ello implicaría elegir otro origen.

A pesar de las limitaciones mencionadas anteriormente, se puede concluir que la construcción del diagrama de secuencia de clases a partir del análisis de código fuente es posible.

La decisión de definir el punto de origen para realizar la construcción del diagrama de interacción, fue motivado de la necesidad de contar con un método en particular que fungiera como punto de partida para construir las interacciones de ese método hacia otros objetos del código analizado. Con lo anterior, se puede analizar el comportamiento de la aplicación desde un método en particular, permitiendo ver el comportamiento de los objetos desde una perspectiva de disparo de eventos.

La conversión de un código normal escrito de manera monolítica a un código de visualización remota se vislumbra más cercano ya que el *"modelo para la obtención del diagrama de interacción de objetos a partir del análisis de código fuente"* resulta ser una buena opción para el entendimiento del código a trasformar.

6.2 TRABAJOS FUTUROS

La presente tesis representa un primer acercamiento a la Ingeniería en Reversa, que es área de interés en la actualidad, por lo tanto, es necesario seguir trabajando en la misma línea de investigación para lograr un avance sustancioso en el área. Esta tesis puede ser el origen del desarrollo de trabajos futuros. A continuación se describe brevemente cada uno de ellos:

- Actualmente en el repositorio (ver apéndice C), se almacena información de una sola aplicación analizada a la vez, en el futuro se piensa en mejorar la estrategia de almacenamiento, para que en el mismo repositorio se pueda tener más de una aplicación analizada.

- Al lograr lo mencionado en el párrafo anterior, se está en posibilidad de construir un módulo adicional para realizar comparaciones de interacciones similares, para así ayudar en la detección de patrones de interacción entre objetos.

- Actualmente ODICC-Swing solamente analiza código en lenguaje Java, por lo tanto, otro de los trabajos futuros propuestos, es que analice también código en otros lenguajes como C++, Visual Basic, Delphi, etc.

- La herramienta realiza la construcción del diagrama de secuencia, en un futuro se planea la construcción del diagrama de colaboración, el cual puede ser también construido a partir de la misma información fuente.

- La construcción de una aplicación que a partir del repositorio de información manipule el código analizado convirtiéndolo a un código orientado a la visualización remota de aplicaciones.

Apéndice

A

Diagrama de Secuencia

En este apéndice se presenta la definición, semántica, notación y opciones de presentación del diagrama de secuencia, que forma parte del diagrama de interacción definido en el Unified Modeling Languaje (UML, Lenguaje Unificado de Modelado) [OMG].

Semántica.

Un diagrama de secuencia presenta una interacción, la cual es un conjunto de mensajes entre los roles dentro de una colaboración para afectar el resultado de una operación o resultado.

Opciones de presentación.

El orden horizontal de la línea de vida es arbitrario. Usualmente las flechas son enlazadas en una sola dirección sobre la página, sin embargo esto no es siempre posible y el orden no conviene a la información.

Los ejes pueden ser intercambiados, así que el tiempo puede ser horizontalmente a la derecha y diferentes objetos son mostrados como líneas horizontales.

Mensajes.

Un mensaje es una especificación de estímulo, especifica los roles que el objeto emisor y el receptor deben conformar para la acción que se ejecutara. Un estímulo es una comunicación entre dos objetos que transporta información con la expectativa de que la acción sobrevenga. Un estímulo causa a que una operación sea invocada, levante una señal, o cause que un objeto sea creado o destruido.

Notación.

Un diagrama de secuencia tiene dos dimensiones: la primera, dimensión vertical representa el tiempo y la segunda, dimensión vertical representa diferentes objetos. Normalmente el tiempo se extiende hacia abajo de la página. Las dimensiones pueden ser intercambiadas si se desea. Usualmente solo las secuencias de tiempo son importantes, pero en las aplicaciones de tiempo real el tiempo puede no ser métrico. No hay un significado para el orden horizontal de los objetos.

En un diagrama de secuencia un estímulo es mostrado como una flecha horizontal desde la línea de vida de uno de los objetos a otra línea de vida de otro objeto. En caso de un estímulo de un objeto a si mismo, la flecha inicia y termina en la misma línea de vida del objeto. La flecha es etiquetada con el nombre del estímulo (operación o señal) y sus argumentos o valores.

La flecha puede ser etiquetada con un número secuencial para demostrar la secuencia de los estímulos en toda la interacción. Sin embargo, los números secuenciales son usualmente omitidos en el diagrama de secuencia, debido a que la ubicación de las flechas demuestra la secuencia relativa que hay necesariamente en los diagramas de colaboración. Los números de secuencia son útiles para identificar los hilos de control concurrentes.

Para fines de nuestro trabajo, se utiliza únicamente la siguiente figura para indicar los mensajes entre objetos:

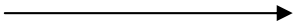


Figura A.1 Notación de mensaje entre objetos.

Llamada a un procedimiento o a otro flujo de control anidado. La secuencia anidada entera es completada antes que el nivel de secuencia exterior finalice. Puede ser usado para llamadas ordinarias de procedimientos[OMG].

Ejemplo de un diagrama de secuencia simple.

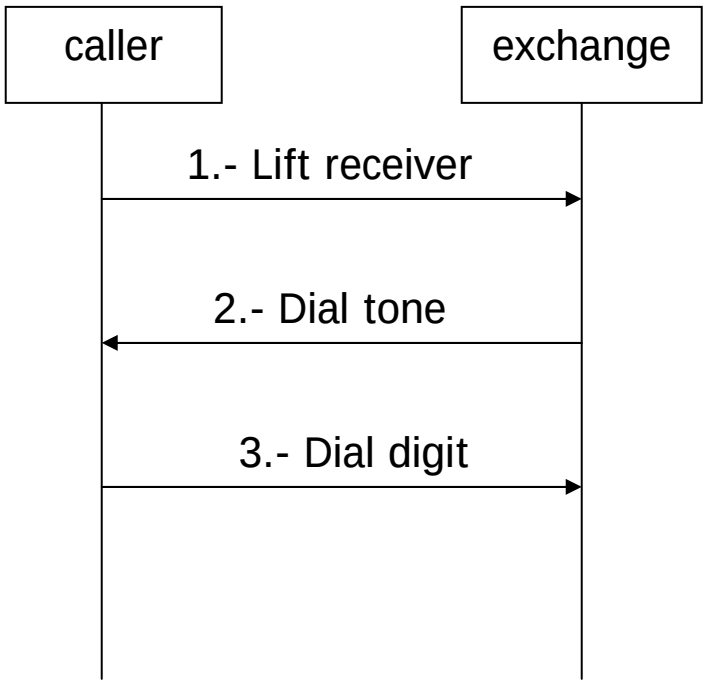


Figura A.2 Ejemplo de diagrama de secuencia.

Apéndice

B

Paquetes

En este apéndice se presenta la descripción de los paquetes que forma el API (Aplication Program Interface) de Swing, sobre el cual se basa la presente tesis, para lograr la identificación y clasificación de las clases que intervienen en una aplicación.

Swing simplifica el desarrollo de aplicaciones independientes de plataforma. La colección de clases Swing consiste de 70 paquetes, cada uno de los cuales tiene su propio propósito, a continuación se describen algunos de ellos.

<table>
<tr><th>Paquete</th><th>Descripción</th></tr>
<tr><td>Javax.Swing</td><td>El paquete de más alto nivel consiste de componentes, adaptadores, modelos de componentes por default y de interfaces.</td></tr>
<tr><td>Javax.Swing.border</td><td>El paquete de borders declara la interface border, las cuales definen el border específico para los estilos de visualización.</td></tr>
<tr><td>Javax.colorchooser</td><td>El paquete colorchooser contiene soporte para colorear los componentes.</td></tr>
<tr><td>Javax.Swing.event</td><td>El paquete de eventos contiene los tipos específicos de eventos Swing y listeners adicionales a los tipos declarados en Java.awt.event, los componentes Swing pueden generar sus propios eventos.</td></tr>
<tr><td>Javax.Swing.filechooser</td><td>El paquete filechooser contiene soporte para las clases del componente filechooser.</td></tr>
<tr><td>Javax.Swing.plaf.*</td><td>El pluggable look-and-feel (PLAF) contiene las clases de interfaz de usuario (UI) las cuales implementan los diferentes aspectos del look-and-feel para los componentes Swing. Hay más paquetes PLAF bajo la jerarquia Javax.Swing.plaf.</td></tr>
<tr><td>Javax.Swing.table</td><td>Contiene las interfaces de soporte y clases para el componente table.</td></tr>
<tr><td>Javax.Swing.text</td><td>Contiene las clases de soporte para el marco de trabajo de documentos Swing.</td></tr>
<tr><td>Javax.Swing.text.html.*</td><td>Contiene clases de soporte para una visualización y análisis de texto HTML versión 3.2.</td></tr>
<tr><td>Javax.Swing.text.rtf</td><td>Contiene clases para una visualización de texto en formato Rich Text Format (RTF).</td></tr>
<tr><td>Javax.Swing.tree</td><td>Contiene clases e interfaces para dar soporte al componente Jtree.</td></tr>
<tr><td>Javax.Swing.undo</td><td>El paquete undo proporciona las clases de soporte para implementar las capacidades de hacer/deshacer en una interfaz gráfica de usuario (GUI).</td></tr>
<tr><td>Java.awt.event</td><td>Provee interfaces y clases para realizar el manejo de eventos que son generados por los componentes gráficos. Dentro de ese paquete se encuentran las siguientes interfaces.
<table>
<tr><td>ActionListener</td><td>Esta interface es usada para recibir acciones de evento.</td></tr>
<tr><td>AdjustmentListener</td><td>Sirve para recibir eventos ajustados.</td></tr>
<tr><td>ComponentListener</td><td>Sirve para recibir eventos de componentes. Los eventos de los componentes son provistos únicamente para propósitos de</td></tr>
</table>
</td></tr>
</table>

Paquete	Descripción	
		notificación únicamente. El AWT manejará automáticamente el movimiento de los componentes y los redimensionará internamente.
	ContainerListener	La interfaz para recibir los eventos de los contenedores. Los eventos de los contenedores son provistos para propósitos de notificación únicamente. El AWT automáticamente maneja la adición y remoción de operaciones internamente.
	FocusListener	Interfaz para recibir eventos del teclado sobre un componente
	ItemListener	Interfaz para recibir los eventos de un item.
	KeyListener	Interfaz para recibir eventos del teclado.
	MouseListener	Interfaz para recibir eventos del ratón sobre un componente.
	MouseMotionListener	Interfaz para recibir eventos de movimiento del ratón sobre un componente.
	TextListener	Interfaz para recibir eventos de ajuste de texto.
	WindowListener	Interfaz para recibir eventos de ventanas.

Apéndice

C

Repositorio

En este apéndice se presenta la descripción del repositorio de información, que contiene la definición de los paquetes y clases que forman parte de Swing, así como también las tablas en las que se almacena información necesaria para la construcción del diagrama de secuencia entre objetos.

En el repositorio se almacenan dos tipos de información, la primera enfocada a la definición de paquetes y clases de Swing, con la finalidad de poder detectar en la aplicación analizada dichos objetos, que servirán además para definir la taxonomía de objetos, el segundo tipo de información que se almacena es producto del análisis del código fuente, y sirve para posteriormente construir el modelo canónico de interacción entre objetos, mediante consultas realizadas a la base de datos o repositorio.

En la siguiente tabla se muestran las tablas que sirven para almacenar las definiciones de los componentes Swing.

Nombre de Tabla	Descripción
Clase_Swing	Contiene todas las clases pertenecientes a Swing 1.1
Interface	Contiene las interfaces que forman parte de Swing 1.1
Paquetes	Contiene los paquetes que forma parte de Swing 1.1

Para almacenar la información de las interacciones entre clases, producto del análisis de la aplicación se utilizan las siguientes tablas.

Nombre de Tabla	Descripción
Clase	Contiene todas las clases analizadas en el código fuente.
Clase_Metodos	Almacena los orígenes de secuencias, representadas con las firmas de los métodos de las clases contenidas en la aplicación analizada.
Rel_this	Almacena las relaciones this para cada una de las clases, no importa el origen.
Solicitudes	Almacena la relación del origen de la clase con su clase refenciada y la firma del método.
Tipos_objetos	Almacena los diferentes tipos de objetos que se encuentran en el análisis. Es una tipificación, donde: 1.- Corresponde a Objetos Swing. 2.- Corresponde a Objetos de manejo de eventos. 3.- Corresponde a Objetos del dominio de aplicación.
Clases_referenciadas	Contiene las clases referenciadas, para cada una de las clases, no importa el origen.

A continuación se presenta el diagrama entidad relación del repositorio de información que sirve como base a la herramienta ODICC-Swing, en la identificación y clasificación de los objetos participantes en la aplicación analizada.

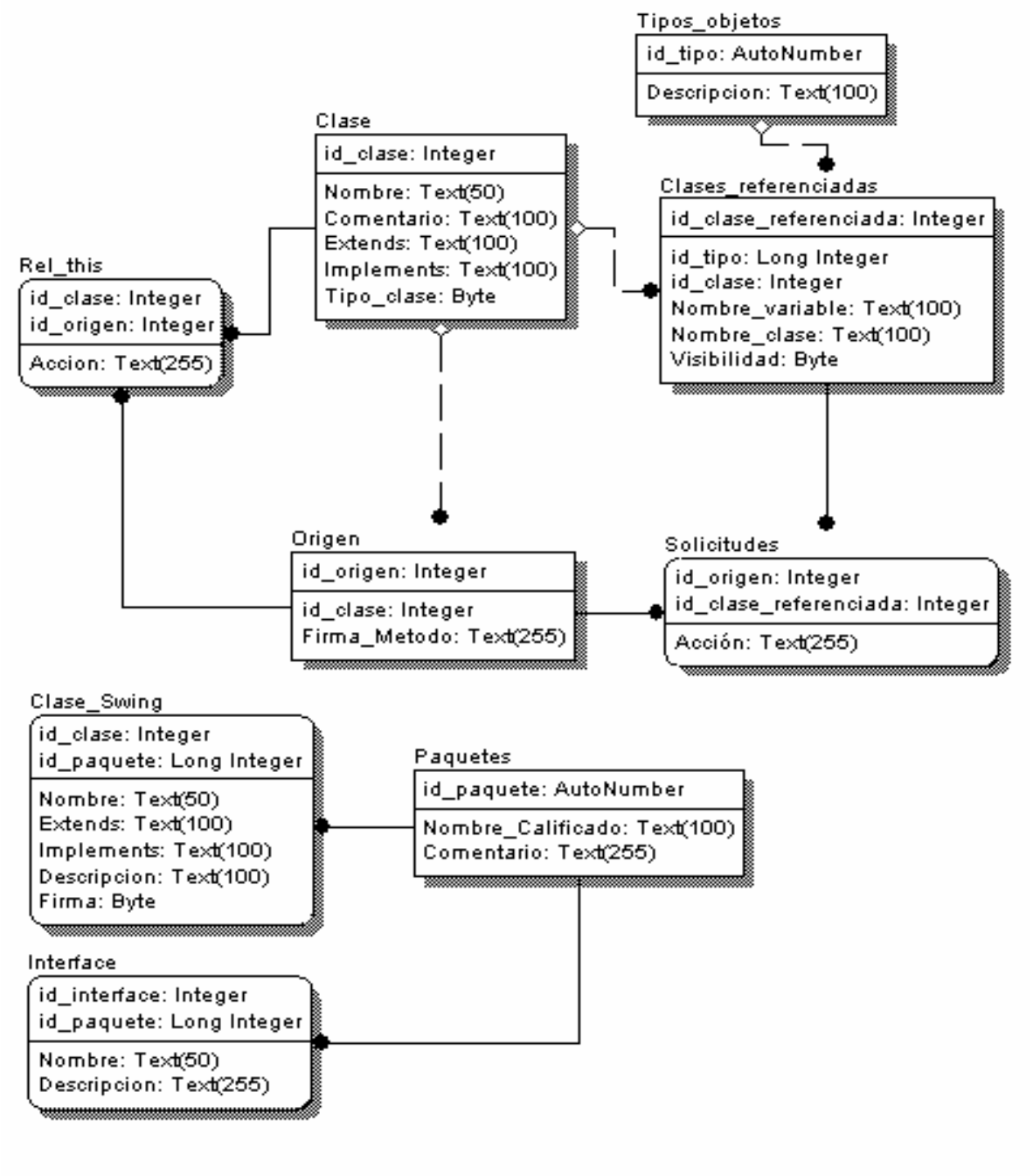


Figura C.1 Diagrama Entidad-Relación del repositorio.

El repositorio se encuentra creado sobre una base de datos en Microsoft Acess 2000, y es utilizado por la herramienta a través de una conexión JDBC (***Java Database Connectivity***) [SUN3], utilizando el driver nativo de Java.

A continuación se presenta el diccionario de datos completo del repositorio.

Nombre de la Tabla	Nombre del campo	Tipo de campo	Descripción
Clase	id_clase	Integer	Identificador de la clase.
	Extends	Text(100)	Nombre de clase de la cual extiende ésta clase.
	Implements	Text(100)	Nombre de las interfaces que implementa ésta clase.
	Comentario	Text(100)	Comentario si es que existe en el código fuente, de acuerdo con Javadoc.
	Nombre	Text(50)	Nombre de la clase.
	Tipo_clase	Byte	Tipo de clase, puede ser: 1.- Componente Swing 2.- Manejo de Eventos. 3.- Objetos de la aplicación. 4.- Objetos del lenguaje.
Clase_Swing	id_clase	Integer	Identificador de la clase
	id_paquete	Long Integer	Clave del paquete al que pertenece ésta clase.
	Implements	Text(100)	Nombre de las interfaces que implementa.
	Nombre	Text(50)	Nombre de la clase Swing.
	Extends	Text(100)	Nombre de la clase, de la cual extiende ésta clase.
	Descripcion	Text(100)	Descripción de la clase.
	Firma	Text(100)	Contiene la cadena de declaración de la clase.
Clases_referenciada	id_tipo	Long Integer	Clave del tipo de objeto, contiene . el tipo de clase
	id_clase_referenciada	Integer	Identificador de la clase referenciada.
	Nombre_clase	Text(100)	Nombre de la clase.
	Visibilidad	Byte	Si es public, private o protected.
	id_clase	Integer	Identificador de la clase.
	Nombre_variable	Text(100)	Nombre de la variable con la cual se hace referencia a la clase.
Interface	id_paquete	Long Integer	Clave del paquete.
	Descripcion	Text(255)	Descripción de la interface.
	id_interface	Integer	Clave de la interface.
	Nombre	Text(50)	Nombre de la interface.
Origen	Firma_Metodo	Text(255)	Firma del método que identifica un origen de secuencia.

Nombre de la Tabla	Nombre del campo	Tipo de campo	Descripción
	id_clase	Integer	Identificador de la clase.
	id_origen	Integer	Identificador del origen de los datos.
Paquetes	id_paquete	AutoNumber	Clave del paquete.
	Nombre_Calificado	Text(100)	Nombre calificado del paquete.
	Comentario	Text(255)	Descripción del paquete.
Rel_this	id_clase	Integer	Identificador de la clase.
	id_origen	Integer	Identificador del origen de los datos.
	Accion	Text(255)	Firma calificada del método invocado.
Solicitudes	id_clase_referenciada	Integer	Identificador de la clase referenciada.
	Accion	Text(255)	Contiene la invocación que aparece en el diagrama de secuencia.
	id_origen	Integer	Identificador del origen de los datos.
Tipos_objetos	Descripcion	Text(100)	Descripción del tipo de objeto.
	id_tipo	AutoNumber	Clave del tipo de objeto.

Apéndice

D

Diagrama de Clases

En este apéndice se presenta el diagrama de clases de la aplicación en notación OMT, así como una breve descripción acerca de su funcionamiento. Para describir las clases participantes en la creación de la aplicación el diagrama de clases se divide en cuatro, los cuales son explicados a continuación.

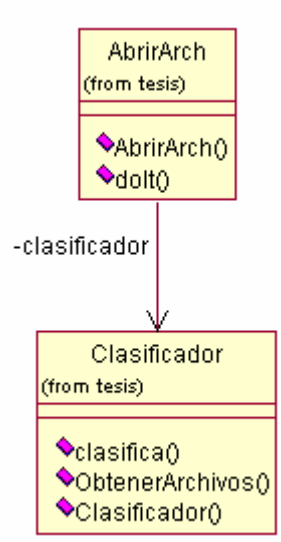


Figura D.1 Diagrama de clases para leer la aplicación por analizar.

La clase *`AbrirArch`*, es la encargada de realizar el despliegue de la interfaz gráfica que sirve para capturar la ubicación del código fuente por analizar. Cabe destacar que la clase *`Clasificador`* es invocada desde la clase *`AbrirArch`* dentro de su método *`doit()`* que es donde se realiza el proceso de análisis de código fuente. En la figura D.1 se muestra las dos clases mencionadas anteriormente.

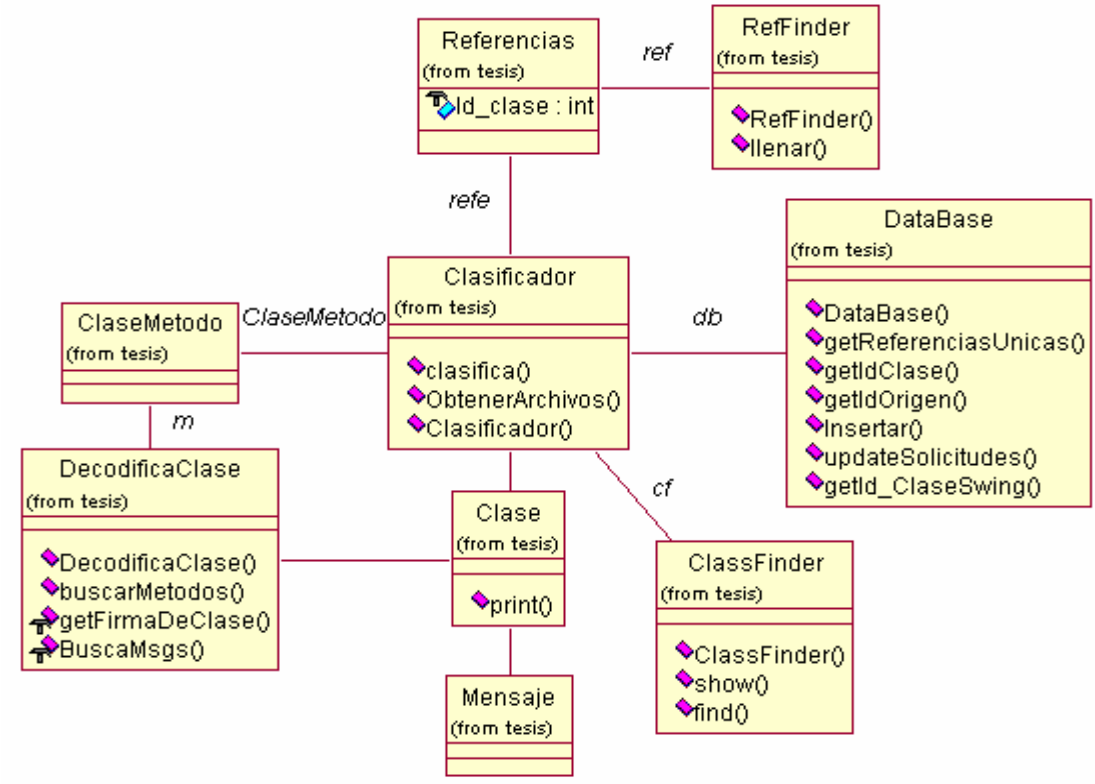


Figura D.2 Diagrama que ilustra las clases participantes en la clasificación e identificación de interacciones entre las clases de un código analizado.

La clase ***clasificador***, es la responsable de realizar el análisis del código fuente, identificar las clases participantes en la aplicación, definir los orígenes de análisis e identificar las interacciones que existen entre las clases pertenecientes a un código analizado. Esta clase hace uso de las siguientes clases: ***ClaseMetodo, ClassFinder, Clase, RefFinder, Referencias*** y ***DataBase***. La clase ***ClassFinder*** es la que encuentra las porciones de código que corresponden a una clase, desde la definición de la clase hasta el último de sus métodos. La clase ***ClaseMetodo*** se encarga de identificar los métodos pertenecientes a las clases, además realiza la identificación y definición de los orígenes de secuencia, esta clase hace uso de la clase ***DecodificaClase*** quien es la encargada de identificar las variables de objeto, que posteriormente se utilizan para que la clase ***Referencias*** identifique las interacciones de la clase analizada con las demás dentro de la aplicación, cabe señalar que la clase ***Referencias*** utiliza a la clase llamada ***Clase*** y ésta a su vez a la clase ***Mensaje*** para realizar la identificación de la invocación de los métodos de las demás clases participantes en la construcción de la aplicación analizada. La clase ***Referencias*** en el procedimiento que tiene que seguir para encontrar referencias entre las clases hace uso de la clase llamada ***RefFinder***, que es la encargada de realizar el barrido de todas las clases analizadas hasta el momento, cabe señalar que al terminar de analizar clase por clase, la clase ***RefFinder*** vuelve a realizar todo el barrido de las clases, esta vez para encontrar las relaciones que no haya detectado por no tener identificadas a todo el universo de clases participantes.

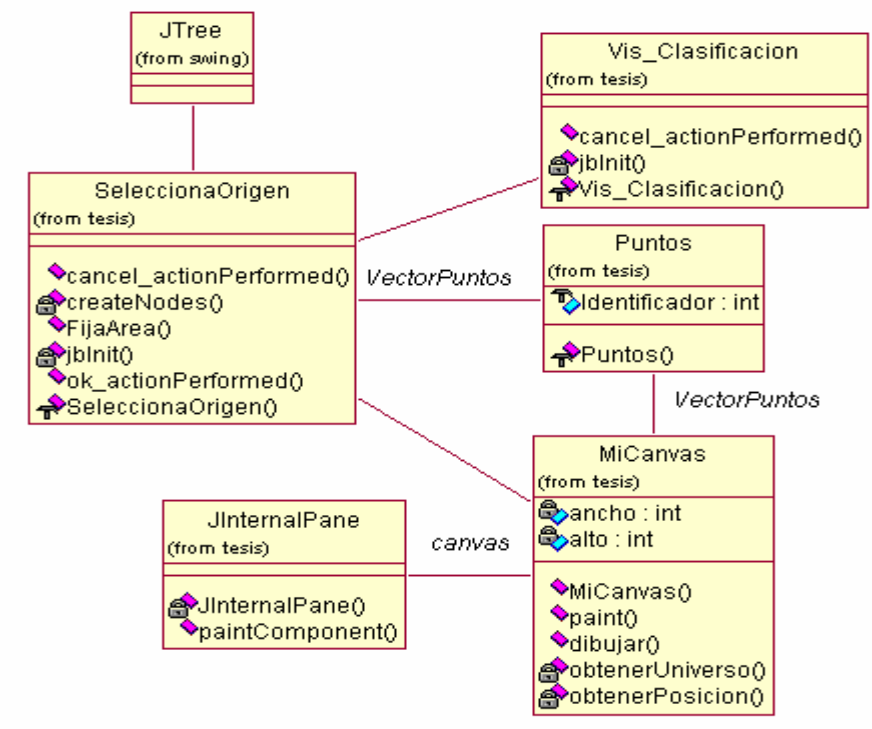


Figura D.3 Diagrama de clases para realizar la construcción de la forma canónica y la posterior visualización de las interacciones entre objetos.

La clase `SeleccionaOrigen,` es la encargada de visualizar al usuario un árbol (clase *`JTree`*) conteniendo las clases encontradas en el análisis así como los métodos de dichas clases a partir de los cuales se puede generar el diagrama de interacción de clases (orígenes de secuencia), esta clase hace uso de la clase *`Vis_Clasificacion`* que es la encargada de controlar el proceso de construcción de la forma canónica de las interacciones presentes en el origen seleccionado y la posterior visualización del diagrama de interacción, para realizar lo anterior se hace uso de la clase *`MiCanvas`*, que es la responsable de realizar la visualización del diagrama de interacción, y como una funcionalidad adicional en la interfaz de usuario se hace uso de la clase *`JInternalPane,`* para realizar el *`Scroll`* sobre la clase *`MiCanvas.`* La clase *`Puntos`* define la forma canónica para cada interacción presentada en el diagrama. La figura D.3 muestra las clases explicadas anteriormente.

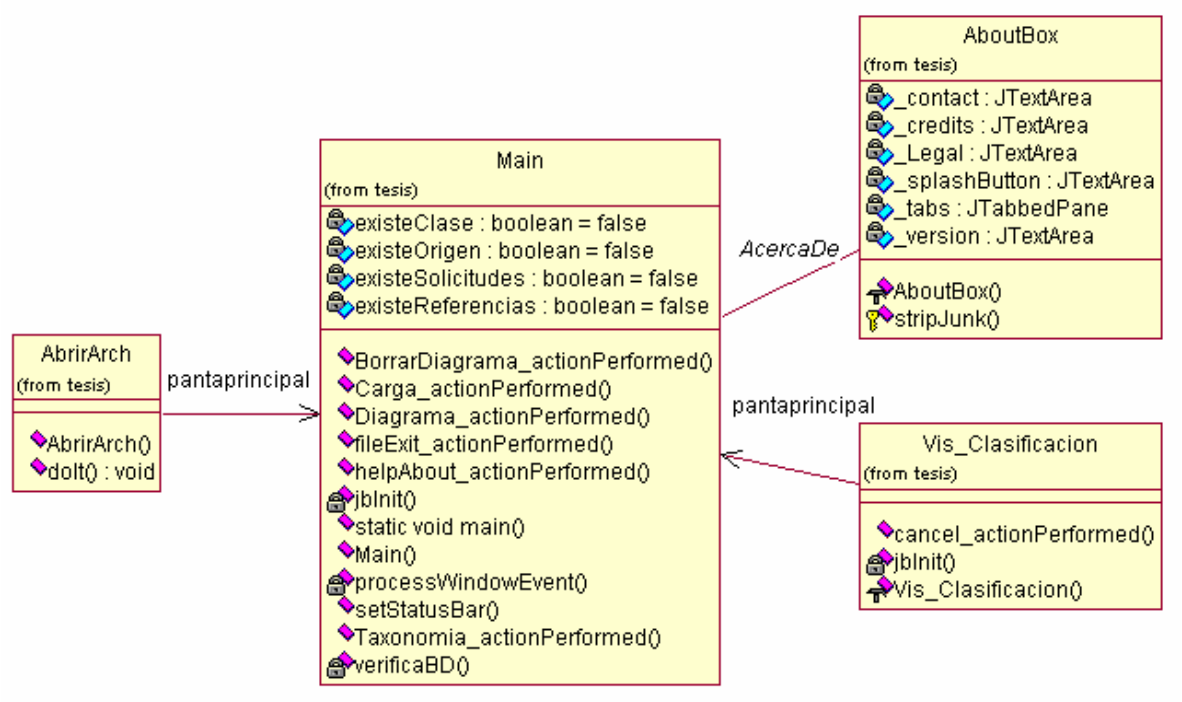


Figura D.4 Diagrama de clases de la interfaz gráfica de la aplicación.

La clase *`Main`*, es la clase principal y de arranque de la aplicación, contiene la creación de los objetos necesarios para la construcción de la interfaz gráfica, así como las validaciones acerca del estado de la base de datos. Hace uso de la clase *`AbrirArch`*, que es la encargada de realizar la lectura de los archivos y fijar el directorio raíz de análisis de la aplicación, además realiza la invocación de la clase *`Clasificador`* que es la encargada de iniciar el proceso de análisis de código fuente. La clase *`AboutBox`*, visualiza la información de ayuda acerca del proyecto. La clase *`Vis_Clasificacion`* es la encargada de realizar la visualización del diagrama de interacción de objetos.

Bibliografía

[AHO90] Alfred V. Aho, Ravi Seth, Jeffrey D. Ullman. Compiladores, principios, técnicas y herramientas. Addison Wesley Iberoamericana, S.A. 1990.

[AHR95] Judith D. Ahrens, Drexel University and Computer Command and Control Company, Noah S. Prywes, University of Pennsylvania and Computer Command and Control Company, "Transition to a Legacy and Reuse Based Software Life Cycle", IEEE Computer, October, 1995, pp. 27-36.

[BUR95] Margaret Burnett et al (eds). *"Visual object-oriented programming"* concepts and environments, Manning Publications, 1995.

[BRO--] Kyle Brown. "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk". clasic thesis. http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm.

[BRE00] P.T. Brewer, K. Lano. "Creating Specifications from Code: Reverse Engineering Techniques". Oxford University Computer Laboratory, Group Research Group, 11 Keble Rd., Oxford, U.K.

[CAS99] Félix Agustín Castro Espinoza. SISTEMA PARA IDENTIFICACIÓN DE PATRONES DE DISEÑO EN CÓDIGO C++. Centro Nacional de Investigación y desarrollo Tecnológico. CENIDET.

[COO98] James W. Cooper. "User Interfaces That Vary with Your Data". Fawcette Technical Publications, junio/julio 1998

[ECK99] Bruce Eckel. Thinking in Java. http://www.BruceEckel.com, 1999.

[GAM95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison Wesley Professional Computing Series.1995

[GRA97] W. K. Grassmann/J.P. Tremblay. MATEMÁTICA DISCRETA Y LÓGICA. Prentice-Hall,1997.

[HOP93] John E. Hopcroft, Jeffrey D. Ullman. Introducción a la teoría de autómatas, lenguajes y computación. Compañía Editorial Continental S.A. de C.V. 1993.

[JAY98] Jay Alonis. A reverse-Engineering Environment Framework**.** Carnegie Mellon University.

[JAC00] Jacques Surveyer. Java and UML**.** http://www.uml-zone.com/

[JHO93] J. Howard Johson. Identifing Redundancy in Source Code using Fingerprints**.** Software Engineering Laboratory, Institute for Information Technology.

[KAZ97] Kazman, Rick and Carriére, S. Jeromy. "Playing Detective: Reconstructing Software Architecture from Available Evidence". Software Engineering Institute, Carnegie Mellon University, 1997.

[KRA96] Christian Krämer, Lutz Prechelt. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software". Working Conference on Reverse Engineering, IEEE CS Press, Monterey CA, November 8-10, 1996.

[LIN01] Linda H. Rosenberg, Lawrence E. Hayatt, Hybrid Re-engineering. http://satc.gsfc.nasa.gov/support/STC_APR97/hybrid/HYBRIDRE1.doc

[LAM01] Margaret Lamb. A Quick Introduction to Javadoc http://www.cs.queensu.ca/home/cisc121/2000f/assignments/Javadoc.html

[OMG] OMG Unified Modeling Language Specification. http://www.omg.org

[PTB00] P.T. Brewer K. Lano. Creating Specifications from Code: Reverse-Engineering Techniques**,** Oxford University Computer Laboratory.

[RAT00] Sitio Web de Rational Corp. www.rational.com

[RAJ00] Vaclav Rajlich. "Comprensión and Evolution of Legacy Software". Department of Computer Science, Wayne State University, Detroit, USA.

[STO97] Storey, Margaret-Anne D; Wong, Kenny; and Miller, Hausi A. "How Do Program Understanding Tools Affect How Programmers Understand Programs?". IEEE Computer Society Press, 1997.

[SUN1] Sun Microsystems."Java Compiler Compiler". http://www.suntest.com/JavaCC/

[SUN2] Sun Microsystems."Event Handling of Swing". http://java.sun.com/docs/books/tutorial/uiswing/overview/event.html

[SUN3] Sun Microsystems: JDBC^{tm} http://java.sun.com/products/jdbc/index.html http://webopedia.internet.com/TERM/J/JDBC.html

[TILL96] Tilley, Scott R.; Santanu Paul; and Smith, Denis B. "Toward a Framework for Program Understanding". 19-28. *Proceedings of the 4th Workshop on*

Program Comprehension. Berlin, Germany, March 29-31, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[TILL98] Scott Tilley. A reverse-Engineering Environment Framework. Technical Report. April 1998

[TOP98] Kim Topley, *CORE Java Foundation Classes*, Prentice Hall, Inc. 1998.

[VON92] Von Mayrhauser, Anniliese and Vans, Marie. *An industrial experience with an Integrated Code Comprehension Model (Technical Report CS-92-205) Ft. Collins, CO: Colorado state University, 1992.*

[VON95] Von Mayrhauser, Anniliese and Vans, Marie. "Program Comprehension During Software Maintenance and Evolution", Computer 28,8 (August 1995): 44-45.

[WE1] Peter Wegner, Draft of ECOOP'99 Banquet Speech, http://www.cs-brown.edu/people/pw

[WEG00] Peter Wegner, Dina Goldin, David Keil. An interactive Viewpoint on the Role of UML. University of Massachusetts, Boston. August 2000.

Glosario

Abstracción:	Facilidad mental que permite a los humanos ver los problemas del mundo real con grados variables de detalle, dependiendo del contexto vigente del problema.
Análisis orientado a objetos:	Método de análisis en el que los requisitos se examinan desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.
BNF:	Backus Naur Form, formalismo mediante el cual se definió el lenguaje ALGOL.
Clase:	Una clase define una interfaz de objetos y su implementación. Se especifica la representación interna de los objetos y se definen las operaciones que los objetos pueden desempeñar. Conjunto de objetos que comparten una estructura común y un comportamiento común.
CASE:	Computer Assisted Software Engineering, programas para el desarrollo de software.
Diagrama de clase:	Es un diagrama que describe las clases, sus estructuras internas y las operaciones, así como las relaciones estáticas entre ellas.
Gramática:	Es un sistema matemático para definir un lenguaje, así como un mecanismo para proporcionar las sentencias en el lenguaje en una estructura útil.
IDL:	Lenguaje IDL (Interface Definition Languaje). Mediante el IDL se especifican interfaces, consistentes en conjuntos de operaciones que los objetos que actúan como servidores proporcionan a los clientes.

Objeto:	Concepto, abstracción o cosa con frontera y significado débil, perteneciente al problema que se trata; instancia de una clase.
Parsing o análisis sintáctico:	Es el proceso de encontrar la estructura sintáctica asociada con una sentencia de entrada.
Programación orientada a objetos (POO):	Método de programación, en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de algún tipo, y cuyos tipos miembros de una jerarquía de tipos unidos mediante relaciones que no son de herencia.
Referencia a objeto:	Es un valor que identifica otro objeto.