



Universidad Nacional de San Luis

Facultad de Ciencias Físico

Matemáticas y Naturales

**Trabajo Final para optar al título de Licenciado en Ciencias de la
Computación**

**Análisis de Identificadores para
Abstraer conceptos del Dominio del
Problema**

Autor

A.P.U. Javier Azcurra Marilungo

Directores

Dr. Mario Marcelo Berón

Dr. Germán Antonio Montejano

San Luis - Argentina

2015

Resumen

Las demandas actuales en el desarrollo de software implican una evolución y mantenimiento constante con el menor costo de tiempo y de recursos. La Ingeniería de Software (IS) se encarga de llevar adelante esta tarea. Dentro de la IS existe el área de Comprensión de Programas (CP). Esta área se dedica a agilizar la comprensión de los sistemas de software, en base al desarrollo de Métodos, Técnicas, Estrategias y Herramientas.

La extracción de la información es un área de investigación muy usada por la CP. Esta área tiene como uno de sus principales objetivos la recolección de diferentes elementos del programa, los cuales pueden ser utilizados para diferentes propósitos en el contexto de CP. Los identificadores (ids) son uno de tales elementos. Estudios indican [22, 46, 38, 29] que los ids esconden detrás de sus abreviaturas indicios de las funcionalidades de los sistemas. Por ende, construir herramientas automatizadas que extraigan y analicen ids es relevante para facilitar la comprensión de programas. La extracción de ids es lo más sencillo, y una forma de lograrlo es a través de un Analizador Sintáctico construido para que pueda extraer ids. Sin embargo, el análisis del significado de los ids en el código fuente es más complicado, una razón de esto es que el nombre de cada identificador (id) depende de cada programador. Una estrategia para analizar ids es expandir las abreviaturas que lo componen.

La expansión de las abreviaturas del id a palabras completas es compleja, pero si se hace correctamente puede ayudar a comprender el sistema. Para lograr esta expansión, se emplean recursos propios del programa de estudio: comentarios, literales, documentación o bien, con fuentes externas al sistema basándose en diccionarios de palabras.

En este trabajo final de licenciatura, se describe *Identifier Analyzer* (IDA), una herramienta útil para el análisis de ids de programas escritos en JAVA. IDA implementa técnicas de análisis de ids que involucran la expansión de abreviaturas presentes en los ids, con el propósito de facilitar la comprensión de los sistemas de software.

Agradecimientos

“Hay en el mundo un lenguaje que todos comprenden: es el lenguaje del entusiasmo, de las cosas hechas con amor y con voluntad, en busca de aquello que se desea o en lo que se cree.”

Paulo Coelho

Quiero agradecer en primer lugar a Dios, Amo y Señor de todas las cosas por brindarme salud y las fuerzas que fueron necesarias para finalizar mi carrera universitaria.

Agradezco a mi familia, que me brindó el sustento afectivo y económico, que fueron indispensables para conseguir los objetivos académicos. A mi mamá Gladys por el apoyo incondicional, a mi papá Carlos por la confianza permanente, a mi hermana Soledad por haberme acompañado en mi trayecto universitario.

Quiero dar un reconocimiento notable a mis directores Prof. Mario Berón y Prof. Germán Montejano, sin su ayuda y dedicación nunca hubiera podido terminar este trabajo final de licenciatura. También quiero dar gracias a los demás profesores que estuvieron involucrados en mi proceso de formación profesional.

Envío un especial agradecimiento a mi amigo Dario que siempre estuvo en todo momento cuando lo necesité y también a todas las demás amistades que brindaron aportes a mi crecimiento personal y profesional. No puedo dejar de destacar a mis ex compañeros de facultad que actualmente son colegas, por compartir horas de estudio y ayudarme a asimilar los conocimientos adquiridos durante mi carrera.

Índice general

Índice de figuras	4
Índice de tablas	6
1. Introducción: Problema y Solución	7
1.1. Introducción	7
1.2. Mantenimiento del Software	7
1.3. Evolución del Software	8
1.4. Migración del Software	9
1.5. Comprensión de Programas	10
1.6. Extracción de Información	11
1.7. Análisis de Identificadores	12
1.8. Problema y la Solución	13
1.9. Contribución	15
1.10. Organización del Trabajo Final	15
2. Comprensión de Programas	16
2.1. Introducción	16
2.2. Modelos Cognitivos	17
2.3. Extracción de Información	19
2.4. Administración de la Información	20
2.5. Visualización de Software	21
2.6. Estrategias de Interconexión de Dominios	22
2.7. Notas y Comentarios	23

3. Análisis de Identificadores: Estado del Arte	24
3.1. Introducción	24
3.2. Conceptos claves	26
3.3. Nombres de Identificadores	27
3.3.1. Clasificación	27
3.3.2. Importancia en los Nombres	28
3.3.3. Herramienta: Identifier Dictionary	32
3.4. Traducción de Identificadores	36
3.4.1. Conceptos y Desafíos observados	36
3.4.2. Algoritmo de División Greedy	38
3.4.3. Algoritmo de División Samurai	42
3.4.4. Algoritmo de Expansión Básico	49
3.4.5. Algoritmo de Expansión AMAP	52
3.4.6. Herramienta: Identifier Restructuring	63
3.5. Notas y Comentarios	68
4. Identifier Analyzer (IDA)	69
4.1. Introducción	69
4.2. Arquitectura	70
4.3. Analizador Sintáctico	72
4.4. Base de Datos Embebida	73
4.5. Proceso de Análisis de Identificadores	74
4.5.1. Barra de Menú	74
4.5.2. Lectura de Archivos JAVA	76
4.5.3. Panel de Elementos Capturados	77
4.5.4. Ventana de Análisis	80
4.5.5. Palabras Capturadas y Diccionarios	82
4.5.6. Nuevamente al Panel de Elementos Capturados	86
5. Casos de Estudio	88
5.1. Introducción	88
5.2. Juego Buscaminas	88
5.3. Editor de Texto	96

5.4. Juego de Aviones	100
5.5. Notas y Comentarios sobre los casos de estudio	103
6. Conclusiones y Trabajos Futuros	106
6.1. Conclusiones	106
6.1.1. La Investigación sobre el Análisis de Identificadores	106
6.1.2. La Construcción de la Herramienta IDA	108
6.1.3. Los Casos de Estudio probados en IDA	109
6.2. Trabajos Futuros	110
6.2.1. Ampliar la Captura del Analizador Sintáctico	110
6.2.2. Implementar otro Algoritmo de Expansión	112
6.2.3. Expandir Identificadores en el Código	113
6.2.4. Acoplar a Entornos de Desarrollo	114
Apéndice A.	115
A.1. Extensión de la Herramienta IDA	115

Índice de figuras

2.1. Modelo de Comprensión de Programas	17
3.1. Trozo de Código de un Sistema Comercial	29
3.2. Visualización de Identifier Dictionary	32
3.3. Visualización de Identifier Dictionary	33
3.4. Visualización de Identifier Dictionary	34
3.5. Ejemplos de distintas Búsquedas dentro del Código	59
3.6. Arquitectura de Identifier Restructuring	63
3.7. Etapas de Identifier Restructurer	64
3.8. Gramática que determina la Función de los Ids	65
3.9. Visualización de Identifier Restructuring	66
4.1. Arquitectura de IDA	70
4.2. Barra de Menú de IDA	75
4.3. Ventana para seleccionar Archivos JAVA	76
4.4. Aviso sobre Archivo JAVA ya abierto en IDA	77
4.5. Panel de Elementos Capturados	78
4.6. Pestaña de Literales Capturados	78
4.7. Pestaña de Comentarios Capturados	79
4.8. Ventana de Análisis	81
4.9. Ventana de Palabras Capturadas	82
4.10. Ventana TagCloud (Nube de Etiquetas)	83
4.11. Ventana de Diccionarios	84
4.12. Panel de Elementos Capturados	85
4.13. Captura del Navegador Web con el Análisis de Ids realizado	86

5.1. Captura del Juego Buscaminas programado en JAVA	89
5.2. Captura del Editor de Texto programado en JAVA	96
5.3. Captura del juego Top Gun programado en JAVA	100
5.4. Porcentajes de Ids divididos correctamente e incorrectamente .	103
5.5. Porcentajes de Ids expandidos correctamente e incorrectamente	104
6.1. Comparación de un código, antes y después de expandir los Ids	113
A.1. Extensión de IDA	115
A.2. Ejemplo de Archivo XML de entrada.	118
A.3. Ejemplo de Archivo XML de salida.	119

Índice de tablas

3.1. Análisis Léxico de códigos JAVA	25
3.2. Algunas Frecuencias Relativas de Ids en JAVA 5	61
5.1. Identificadores extraídos por el AS ANTLR	91
5.2. Comentarios extraídos por el AS ANTLR	92
5.3. Literales extraídos por el AS ANTLR	92
5.4. Análisis Realizado a los Ids extraídos de Minesweeper.java . .	95
5.5. Parte del Análisis Realizado a los Ids de Editor.java	99
5.6. Parte del Análisis Realizado a los Ids de TopGun.java	102

Capítulo 1

Introducción: Problema y Solución

1.1. Introducción

Cuando se desarrollan aplicaciones de software se busca que el producto final satisfaga las necesidades de los usuarios, que el buen funcionamiento perdure en el mayor tiempo posible y en caso de hacer una modificación para mejorarlo no implique grandes costos. Estos puntos conllevan a un software de alta calidad y la disciplina encargada de conseguirlo es la *Ingeniería de Software* (IS). Para que la IS logre las metas antes mencionadas se utilizan, entre otras tantas, tres temáticas importantes que están orientadas al desarrollo de buenos sistemas: el *mantenimiento del software*, la *evolución del software* y la *migración del software*.

1.2. Mantenimiento del Software

La etapa de mantenimiento es importante en el desarrollo del software, dado que los sistemas están sujetos a cambios y a una permanente evolución [63].

El mantenimiento del software según el estándar 610 de IEEE [40], *es la modificación del software que se realiza posterior a la entrega del producto al usuario con el fin de arreglar fallas, mejorar el rendimiento, o adaptar el*

sistema al ambiente que ha cambiado.

De la definición anterior se desprenden 4 tipos de cambios que se pueden realizar en la etapa de mantenimiento. El *mantenimiento correctivo* cambia el software para reparar las fallas detectadas por el usuario. El *mantenimiento adaptativo* modifica el sistema para adaptarlo a cambios externos, como por ejemplo actualización del sistema operativo o el motor de base de datos. El *mantenimiento perfectivo* se encarga de agregar nuevas funcionalidades al software que son descubiertas por el usuario. Por último, el *mantenimiento preventivo* realiza cambios al programa para facilitar futuras correcciones o adaptaciones que puedan surgir en el futuro [55].

Por lo antedicho, entre otras diversas razones [12, 58, 7, 20] indican que el mantenimiento del software consume mucho esfuerzo y dinero. En algunos casos, los costos de mantenimiento pueden duplicar a los que se emplearon en el desarrollo del producto. Las causas que pueden aumentar estos costos son, el mal diseño de la arquitectura del programa, la mala codificación, la ausencia de documentación.

No es descabellado pensar estrategias de automatización que puedan ser aplicadas en fases del mantenimiento del software que ayuden a reducir estos costos, lo cual requiere una comprensión del objeto que se va a modificar antes de realizar algún cambio que sea de utilidad.

Algunos autores [12, 61, 55, 63] concluyen que el mantenimiento del software y la comprensión de los programas son conceptos que están fuertemente relacionados. Una frase conocida en la jerga de la IS es: “*Mientras más fácil sea comprender un sistema, más fácil será de mantenerlo*”.

1.3. Evolución del Software

Los sistemas complejos evolucionan con el tiempo, los nuevos usuarios y requisitos durante el desarrollo del mismo causan que el producto final posiblemente no sea el que se planteó en un comienzo.

La evolución del software básicamente se atribuye al crecimiento de los sistemas a través del tiempo, es decir, tomar una versión operativa y generar una nueva versión ampliada o mejorada en su eficiencia [24, 12, 68, 67].

Generalmente, los ingenieros del software recurren a los modelos evolutivos. Estos modelos indican que cuando se desarrolla un producto de software es conveniente que se divida en distintas iteraciones. A medida que avanza el desarrollo, cada iteración retorna una versión entregable cada vez más compleja [55]. Estos modelos son muy recomendados sobre todo si se tienen fechas ajustadas donde se necesita una versión funcional lo más rápido posible. De esta manera, no se requiere esperar una versión completa al final del proceso de desarrollo.

Es inevitable y fundamental comprender correctamente la iteración actual del producto antes de comenzar con una nueva. Nuevos requerimientos del usuario pueden aparecer, como así también nuevas dificultades en el desarrollo del sistema. Es por esto que la comprensión del sistema durante su evolución es crucial.

1.4. Migración del Software

Las tareas de mantenimiento de software no solo se realizan para mejorar cuestiones internas del sistema, como es el caso de arreglos de errores de programación o requisitos nuevos del usuario. También se necesitan para adaptar el software al contexto cambiante. Es aquí donde la *migración del software* entra en escena.

De acuerdo con el estándar 1219 de IEEE [41], *la migración del software es convertir o adaptar un viejo sistema (sistema heredado) a un nuevo contexto tecnológico sin cambiar la funcionalidad del mismo*.

Las migraciones de sistemas más comunes que se llevan a cabo son causadas por cambios en el hardware obsoleto, nuevos sistemas operativos, cambios en la arquitectura y nuevas base de datos. Sin duda, la que más se ha acentuado en los últimos años es la aparición de nuevas tecnologías web y las mobile (basadas en dispositivos móviles) [65].

La migración de grandes sistemas es fundamental, sin embargo está demostrado que es costosa y compleja [65, 69, 6, 70]. Para llevarla a cabo reduciendo estos costos se recomienda hacer previamente una buena comprensión del sistema antiguo.

Una técnica de comprensión de sistemas ideada recientemente para la migración de software, consiste en un modelado de datos [35]. Este modelado tiene como finalidad representar el sistema en distintos niveles de abstracción. De esta forma, el grupo encargado de la migración solo debe preocuparse por convertir el antiguo código a la nueva tecnología.

Nuevamente la comprensión de programas se hace presente en temáticas relacionadas al desarrollo de proyectos de software.

1.5. Comprensión de Programas

Como se explicó en las secciones anteriores, es crucial lograr comprender el sistema para llevar adelante las tareas de mantenimiento, evolución y migración del software. Por esta razón, existe un área de la IS que se dedica al desarrollo de técnicas de inspección y comprensión de software. Esta área se conoce con el nombre de *Comprensión de Programas* (CP).

La CP se define como: *Una disciplina de la IS encargada de ayudar al desarrollador a lograr un entendimiento acabado del software. De forma tal que se pueda analizar disminuyendo en lo posible el tiempo y los costos* [13].

La CP asiste al equipo de desarrollo de software proveyendo métodos, técnicas y herramientas que faciliten el entendimiento del sistema de estudio.

Las investigaciones realizadas en el área de la CP determinaron que el foco de estudio se centra en relacionar el *Dominio del Problema* con el *Dominio del Programa* [16, 13, 3, 28]. El primero hace referencia a los elementos que forman parte de la salida del sistema, mientras que el segundo indican las componentes del programa empleados para generar dicha salida. Esta relación es compleja de realizar y representa el principal desafío de la CP.

Una aproximación para relacionar ambos dominios consiste en realizar los siguientes pasos:

- I) Construir una representación del *Dominio del Problema*.
- II) Elaborar una representación para el *Dominio del Programa*.
- III) Unir las representaciones de I) y II) con una *Estrategia de Vinculación*.

Para llevar a cabo los tres pasos mencionados, se necesitan diferentes temáticas las cuales se mencionan a continuación:

- Los *Modelos Cognitivos* son relevantes para la CP porque destacan como el programador utiliza procesos mentales para comprender el software.
- La *Visualización del Software* analiza las distintas partes del sistema y genera representaciones visuales que agilizan la CP.
- La *Interconexión de Dominios* trata de como interrelacionar los elementos de un dominio con otro. Es útil para la CP en la reconstrucción de la relación entre Dominio del Problema y el Dominio del Programa.
- La *Extracción de Información* en los sistemas de software, es necesaria para las estrategias de cognición, visualización, interconexión de dominios, etc.
- Al extraer gran cantidad de información se necesita la *Administración de Información*. La misma brinda técnicas de almacenamiento y acceso eficiente a la información extraída.

Todos estos temas se explican con mayor precisión en el próximo capítulo. En la siguiente sección se amplía la *extracción de la información* ya que es la antesala al *análisis de identificadores*, eje central de este trabajo final.

1.6. Extracción de Información

En la CP, la *Extracción de la Información* se define como *el uso/desarrollo de técnicas que permitan extraer información desde el sistema de estudio*. Esta información puede ser: Estática o Dinámica, dependiendo de las necesidades del ingeniero de software o del equipo de trabajo.

La extracción de información estática recupera los distintos elementos en el código fuente de un programa [1].

Por otro lado, la extracción dinámica de información implica obtener los elementos que se utilizaron en la ejecución del sistema [10]. Generalmente no

todas las partes del código son ejecutadas, por lo tanto no todos los elementos se pueden recuperar en una sola ejecución.

La principal diferencia que radica entre ambas técnicas, es que las estáticas se basan en la información presente en el código, mientras que las dinámicas es obligatorio ejecutar el sistema.

En este trabajo final se hace énfasis en la extracción de información estática. Es importante aclarar que la información dinámica es tan importante como la información estática, sin embargo su extracción requiere del estudio de otro tipo de aproximaciones y escapan al alcance de este trabajo.

Como se mencionó previamente, en los códigos de software abundan componentes que conforman la información estática. Alguno de ellos son, nombres de variables, tipos de las variables, los métodos de un programa, las variables locales a un método, constantes. Todos estos componentes se representan mediante los identificadores (ids). Los ids abarcan gran parte de los elementos de un código por lo tanto su análisis no debe pasarse por alto.

1.7. Análisis de Identificadores

Estudios realizados [25, 17, 31, 30] indican que gran parte de los códigos están conformados por identificadores (ids), por ende abundante información estática está representada por ellos (Ver Capítulo 3).

Generalmente, los ids están compuestos por más de una palabra en forma de abreviaturas. Varios autores coinciden [22, 46, 38, 29] que detrás de estas abreviaturas, se encuentra oculta información que es propia del dominio del problema.

Una vía posible para exponer la información oculta consiste en traducir las abreviaturas antes mencionadas, en sus correspondientes palabras expresadas en lenguaje natural. Para lograrlo: I) primero se extraen los identificadores del código; II) luego se les aplican técnicas de división, donde se descompone a cada identificador en las distintas palabras abreviadas que lo componen. Por ejemplo: `inp_fl_sys` \rightarrow `inp fl sys`; y III) se emplean estrategias de expansión a las abreviaturas para transformar las mismas en palabras completas. Siguiendo con el ejemplo anterior: `inp fl sys` \rightarrow `input file system`.

Normalmente, los nombres de los ids son elegidos en base a criterios del programador [46, 29]. Lamentablemente, estos criterios son desconocidos para las técnicas que expanden abreviaturas en los ids. Una forma de afrontar esta dificultad, es recurrir a fuentes de información informal que se encuentran disponible en el código fuente. Por información informal se entiende aquella contenida en los comentarios de los módulos, comentarios de las funciones, literales strings, documentación del sistema y todos lo demás recursos descriptivos del programa que estén escritos en lenguaje natural.

Sin duda, los comentarios tienen como principal finalidad ayudar a comprender un segmento de código [34, 5, 8, 33]. Por esta razón, se puede ver a los comentarios como una herramienta natural para entender el significado de los ids en el código, como así también el funcionamiento del sistema.

Por otro lado para poder entender la semántica de los ids, se toman literales o constantes strings. Estos representan un valor constante formado por secuencias de caracteres. Ellos son generalmente utilizados en la muestra de carteles por pantalla, y comúnmente se almacenan en variables de tipo string.

Los literales string como los comentarios pueden brindar indicios del significado de las abreviaturas que se desean expandir.

En caso de que estas fuentes de información informal sean escasas dentro del mismo sistema, se puede acudir a alternativas externas como es el caso de los diccionarios predefinidos de palabras en lenguaje natural.

Las estrategias de análisis de ids, explicadas en esta sección, se han ido mejorando a lo largo del tiempo. Al principio contaban con diccionarios extensos y ocupaban mucho espacio, luego estos diccionarios se fueron reemplazando por listados de palabras que son acordes a la ciencias de la computación. Estos listados son más eficientes que los diccionarios, ya que contienen palabras más precisas y ocupan menos espacio de almacenamiento (Ver Capítulo 3).

1.8. Problema y la Solución

En síntesis, un camino para tratar de comprender los sistemas, es fijar la atención en los ids. En la actualidad entender el significado de los ids en los programas se realiza generalmente de manera manual leyendo el código, esto

implica grandes esfuerzos para el lector sobre todo si el código del sistema es grande. Las herramientas automáticas que analizan ids simplifican en gran medida estas arduas tareas. Normalmente, estas herramientas para realizar sus tareas, efectúan la traducción de ids, e involucran los pasos que fueron descriptos en el apartado anterior: I) capturar los ids del código, II) dividir las palabras abreviadas que componen un id, III) expandir las abreviaturas en base a lo observado en comentarios/literales y/o diccionarios de palabras.

Para abordar las iniciativas planteadas y hacer algunos aportes a la solución del problema descripto en el párrafo anterior, se pretende:

- Construir un analizador sintáctico que permita extraer los ids, comentarios y literales encontrados en el código fuente del sistema de estudio. La herramienta seleccionada para realizar esta tarea es ANTLR¹.
- Armar una estrategia para construir un diccionario de palabras en lenguaje natural que sirva como alternativa a las fuentes de información informal.
- Investigar técnicas de división de ids y técnicas expansión de abreviaturas. Algunas de estas utilizan como principal recurso la información brindada en los comentarios y los literales extraídos del código. Como segunda posibilidad se recurre al diccionario.
- Implementar algunas de las técnicas mencionadas en el ítem anterior en una herramienta denominada *Identifier Analyzer* (IDA). Esta herramienta también permite visualizar atributos del id (que representa el id, de que tipo es, número de línea, etc.) mostrando el sector del código donde se encuentra ubicado.
- Probar la herramienta IDA con casos de estudios, para demostrar la utilidad de la misma.
- Comparar el desempeño de cada técnica implementada en IDA y sacar las conclusiones pertinentes.

¹ANother Tool for Language Recognition - <http://wwwantlr.org>

1.9. Contribución

Como se mencionó previamente, las herramientas que ayudan a comprender los ids son importantes para la CP, dado que facilitan encontrar indicios sobre las funcionalidades del sistema. Teniendo en cuenta la escasez de este tipo de herramientas en el ámbito de la CP, el correspondiente trabajo final consiste en el diseño y la implementación de una nueva herramienta que analiza ids en programas escritos en JAVA. Esta herramienta llamada *Identifier Analyzer* (IDA), le permite al usuario ingresar un programa escrito en JAVA, luego IDA captura los ids de forma automática y finalmente mediante la ejecución de técnicas específicas, ayuda al usuario a encontrar el significado de los ids en el programa ingresado. La herramienta IDA, además de analizar ids, puede ser utilizada como base para construir futuras herramientas de comprensión.

1.10. Organización del Trabajo Final

El trabajo está organizado de la siguiente manera. El Capítulo 2 define conceptos teóricos relacionados a la comprensión de programas. El Capítulo 3 describe el estado del arte asociado a las técnicas de análisis de identificadores conocidas. El Capítulo 4 trata sobre la herramienta *Identifier Analyzer* (IDA), en donde se explican los distintos módulos que la herramienta posee y que técnicas de análisis de identificadores tiene implementadas. En el Capítulo 5, se describen algunos casos de estudio que demuestran la utilidad de la herramienta IDA. Finalmente, en el Capítulo 6 se explican las conclusiones elaboradas y se proponen trabajos futuros.

Capítulo 2

Comprensión de Programas

2.1. Introducción

La CP es un área de la Ingeniería de Software que tiene como meta primordial desarrollar métodos, técnicas y herramientas que ayuden al programador a comprender en profundidad los sistemas de software.

Diversos estudios e investigaciones demuestran que el principal desafío en la CP es vincular el *Dominio del Problema* y el *Dominio del Programa* [16, 13, 14, 28]. El primero se refiere al resultado de la ejecución del sistema, mientras que el segundo indica todos los componentes de software que causaron dicho resultado.

La comunidad de CP sostiene, basado en estudios de los modelos cognitivos, que un programador entiende un programa cuando puede relacionar el comportamiento del mismo con su operación [56, 15, 53]. En otras palabras cuando puede vincular el *Dominio del Problema* y el *Dominio del Programa*. Una posibilidad para lograr la reconstrucción del vínculo antes mencionado es a través de los siguientes pasos: I) Elaborar una representación para el Dominio del Problema; II) Construir una representación del Dominio del Programa y III) Elaborar un procedimiento de vinculación. La Figura 2.1 muestra un modelo de comprensión que refleja lo mencionado previamente.

Para lograr con éxito los pasos antedichos se deben tener en cuenta conceptos muy importantes que son los cimientos sobre los cuales está sustentada



Figura 2.1: Modelo de Comprensión de Programas

la CP: los *modelos cognitivos*, la *extracción de información*, la *administración de la información*, la *visualización de software* y la *interconexión de dominios*. Estos conceptos fueron descriptos brevemente en el capítulo anterior, a continuación se desarrolla una explicación más amplia de cada uno.

2.2. Modelos Cognitivos

Los *Modelos Cognitivos* se refieren a las estrategias de estudio y las estructuras de información usadas por los programadores para comprender los programas [64, 66, 19, 60]. Estos modelos son importantes porque indican de que forma el programador comprende los sistemas y como incorpora nuevos conocimientos (en términos generales aprende nuevos conceptos). Los Modelos Cognitivos están compuestos por: *Conocimientos*, *Modelos Mentales* y *Procesos de Asimilación* [14]. Estos conceptos se detallan a continuación.

Conocimiento: Existen dos tipos de conocimientos, por un lado está el conocimiento interno el cual se refiere al conocimiento que el programador ya posee antes de analizar el código del sistema de estudio. Por otro lado existe el conocimiento externo que representa un soporte de información que le brinda al programador nuevos conceptos. Este conocimiento externo es el que proporciona el sistema que se quiere

entender, con todos los artefactos asociados al mismo. Los ejemplos más comunes son la documentación del sistema, otros desarrolladores que conocen el dominio del problema, códigos de otros sistemas con similares características, entre otras tantas fuentes de información.

Modelo Mental: Este concepto hace referencia a representaciones mentales que el programador elabora al momento de estudiar el código del sistema. Algunos ejemplos de modelos mentales son: el grafo de llamadas a funciones, el grafo de dependencias de módulos, diagramas de flujo, etc. Es importante mencionar que algunos modelos mentales permiten una representación visual que posibilita que dicho modelo sea exteriorizado. Esta además decir que esta característica ayuda al programador a entender un programa.

Proceso de Asimilación: Está compuesto por estrategias de aprendizaje que el programador usa para llevar adelante la comprensión de un programa. Los procesos de asimilación se pueden clasificar en tres grupos: Bottom-up, Top-Down e Híbrido [52, 61]. A continuación se explica cada uno.

Bottom-up: El proceso de comprensión *Bottom-up*, indica que el desarrollador primero lee el código del sistema. Luego, mentalmente las líneas de código leídas se agrupan en distintas abstracciones de más alto nivel. Estas abstracciones intermedias se van construyendo hasta lograr una abstracción comprensiva total del sistema.

Top-down: Brooks explica el proceso teórico *top-down*, donde el programador primero elabora hipótesis generales del sistema en base a su conocimiento del mismo. En ese proceso se elaboran nuevas hipótesis las cuales se intentan validar y en ese proceso se generan nuevas hipótesis. Este proceso se repite hasta que se encuentre un trozo de código que implementa la funcionalidad deseada.

Para resumir, el proceso de aprendizaje bottom-up procede de lo más específico (código fuente) a lo más general (abstracciones). Mientras que el top-down es a la inversa.

Híbrido: Este proceso combina los dos conceptos mencionados anteriormente *top-down* y *bottom-up*. Durante este proceso de aprendizaje del sistema, el programador combina libremente ambas políticas (*top-down* y *bottom-up*) hasta lograr comprender el sistema.

La teoría descrita en esta sección sobre Modelos Cognitivos, marca la importancia de esta temática en el ámbito de la CP y de lo difícil que es desarrollar métodos, técnicas y herramientas en este ámbito.

Para llevar a cabo estrategias involucradas en Modelos Cognitivos, es importante extraer información desde los códigos de estudio, en el próximo apartado se hace hincapié en este tema.

2.3. Extracción de Información

En la Ingeniería del Software existe un área que se encarga de la *Extracción de la Información* desde los sistemas de software [36, 50, 39]. Existen dos tipos de informaciones: la Estática y la Dinámica. A continuación se explica cada una y se dan ejemplos conocidos de técnicas que las extraen.

Información Estática: Está presente en los componentes del código fuente del sistema. Alguno de ellos son identificadores, procedimientos, comentarios, documentación. Una forma para extraer la información estática consiste en utilizar técnicas tradicionales de compilación [1]. Estas técnicas, utilizan un analizador sintáctico similar al empleado por un compilador. Este analizador sintáctico, por medio de acciones semánticas específicas procede a capturar elementos presentes en el código del sistema de estudio. En la actualidad, existen herramientas automáticas que ayudan a construir este tipo de analizador sintáctico. Entre las más conocidas se pueden mencionar yacc y lex, bison, lisa, javacc, antlr, entre otras tantas herramientas.

Información Dinámica: Se basa en elementos del programa presentes durante una ejecución específica del sistema [10]. Un ejemplo conocido de una técnica que extrae información dinámica es la instrumentación de

código. Esta técnica inserta sentencias nuevas en el código fuente. La finalidad de las nuevas sentencias es registrar las actividades realizadas durante la ejecución del programa. Estas sentencias nuevas no deben modificar la funcionalidad original del sistema, por ende la inserción debe realizarse con sumo cuidado y de forma estratégica para no alterar el flujo normal de ejecución. Otras técnicas (de extracción dinámica de información) en el ámbito del desarrollo de sistemas son el debugging, que sirve para identificar errores (bugs) en el código; el profiling, que ayuda a optimizar los sistemas, observando el tiempo que demora en ejecutarse las distintas partes del código; entre otras.

Tanto las estrategias de extracción de información estáticas, como las dinámicas son importantes ya que permiten elaborar técnicas para comprender programas. A veces, se recomienda complementar el uso de ambas estrategias para obtener mejores resultados, sobre todo si el sistema que se está analizando es grande y complejo [26].

Cabe destacar que extraer información de los sistemas implica tomar ciertos recaudos. Si la magnitud de información es demasiado grande se puede dificultar el acceso y el almacenamiento de la misma. Es por esto que se recurre a las técnicas de administración de información. En la próxima sección se explica con más detalles esta afirmación.

2.4. Administración de la Información

Teniendo cuenta que los sistemas son cada vez más amplios y complejos. El volumen de la información extraída de los sistemas crece notoriamente, por lo tanto se necesita administrar la información.

Las técnicas de *Administración de Información* se encargan de brindar estrategias de almacenamiento y acceso eficiente a la información recolectada de los sistemas.

Dependiendo del tamaño de la información se utiliza una determinada estrategia. Estas estrategias indican el tipo de estructura de datos a utilizar y las operaciones de acceso sobre ellas [2, 59]. La eficiencia en espacio de almacenamiento y tiempo de acceso son claves a la hora de elegir una estrategia.

Cuando la cantidad de datos son de gran envergadura, se recomienda emplear una base de datos con índices adecuados para realizar las consultas [27].

Después que se administra la información, se aconseja que la misma sea representada por alguna técnica de visualización. Esta representación, permite esclarecer la información del sistema de una mejor manera para que sea interpretada por el programador. El área encargada de llevar adelante esta tarea es la Visualización de Software.

2.5. Visualización de Software

La *Visualización de Software* es una disciplina de la Ingeniería del Software. Esta disciplina, se encarga de visualizar la información presente en los programas, con el propósito de facilitar el análisis y la comprensión de los mismos [11, 62, 51, 54]. Esta cualidad es interesante ya que en la actualidad los sistemas son cada vez más amplios y complicados de entender. Esta disciplina además brinda apoyo en lo que respecta a la comprensión de las distintas etapas involucradas en el desarrollo de los sistemas, como es el caso del análisis, diseño, implementación y mantenimiento. Por ende, la Visualización de Software colabora en la CP.

Para lograr lo descripto en el párrafo anterior, la Visualización de Software provee distintos sistemas de visualización. Estos sistemas son herramientas útiles que se encargan de analizar los distintos módulos de un programa y generar vistas. Las vistas son una representación visual de la información contenida en el software. Generalmente, una herramienta de comprensión de programas posee varias vistas que ayudan a comprender un programa, dependiendo de la información que se requiera visualizar existe una vista adecuada a cada caso. Las vistas en el contexto de la CP, representan puentes cognitivos que disminuyen la brecha entre los conocimientos del programador y los conceptos usados en el software de estudio.

Para concluir, el objetivo primordial de la visualización de software orientada a la CP es generar vistas (representaciones visuales), que ayuden a reconstruir el vínculo entre el Dominio del Problema y el Dominio del Programa. De manera más sencilla, el objetivo es representar visualmente la

salida del sistema, los componentes utilizados para dicha salida y la relación que existe entre ambos.

2.6. Estrategias de Interconexión de Dominios

Los conceptos explicados en los puntos anteriores como la *visualización de software* y la *extracción de la información* forman la base para elaborar estrategias de interrelación de dominios.

La *Interconexión de Dominios* en la Ingeniería del Software, tiene como objetivo principal la transformación y vinculación de un dominio específico en otro dominio. Este último dominio puede estar en un nivel más alto o más bajo de abstracción. Lo primordial aquí, es que cada componente de un dominio se vea proyectado en uno o más componentes de otro dominio y viceversa.

Un ejemplo sencillo de *Interconexión de Dominios*, es cuando el dominio del código fuente de un programa, se puede transformar en un Grafo de Llamadas a Funciones (Dominio de Grafos). Cada nodo del grafo representa una función particular y cada arco las funciones que puede invocar. En este ejemplo la relación entre ambos dominios (código y grafo) es clara y directa.

Es importante aclarar que existe una amplia gama de transformaciones entre dominios. La más escasa y difícil de conseguir es aquella que relaciona el Dominio del Problema con el Dominio del Programa (Ver Figura 2.1).

Sin embargo, actualmente existen técnicas recientemente elaboradas, que conectan visualmente el Dominio del Problema y el Dominio del Programa mediante la información estática y dinámica que se extrajo del sistema de estudio, algunos ejemplos son *Simultaneous Visualization Strategy* (SVS), *Behavioral-Operational Relation Strategy* (BORS) y *Simultaneous Visualization Strategy Improved* (SVSI) [16, 13, 14]. Estas técnicas son muy útiles para la CP, ya que vinculan la salida del sistema con los componentes del programa que se utilizaron para producir dicha salida.

2.7. Notas y Comentarios

Para resumir las ideas tratadas en este capítulo, el área de la CP le da mucha importancia a la relación entre el *Dominio del Problema* y el *Dominio del Programa*. Esta relación es clave, por que ayuda al programador a entender con facilidad los programas, ya que encuentra las partes del sistema que produjeron una determinada salida. Dado que los especialistas determinan que es complicado este vínculo [3, 13, 14, 28], un camino para aproximarse a la difícil solución de unir ambos dominios, es construir una representación de cada uno y luego unirlos con una estrategia de relación. Para llevar a cabo estos pasos, se necesitan estudiar temáticas pertinentes tales como: los *modelos cognitivos*, que indican como el programador emplea procesos mentales para comprender los sistemas; la *visualización de software*, que se encarga de crear representaciones visuales de los sistemas y con esto facilita su comprensión; la *extracción de información* en los sistemas de software, es importante para elaborar técnicas de CP; si la información extraída es mucha se recomienda *administrar la información*; por último la *interconexión de dominios* es una pieza clave para elaborar estrategias de CP, dado que brinda teorías útiles que aproximan a reconstruir el vínculo entre el Dominio del Problema y el Dominio del Programa.

Los conceptos antedichos son claves para la CP y sirven como punto de partida para la construcción de Herramientas de Comprensión de Sistemas. Estas herramientas presentan diferentes perspectivas del sistema, facilitando su análisis y su inspección. De esta manera, evitan que el programador invierta tiempo y esfuerzo en entender los módulos de los sistemas. Por ende, se agilizan las tareas de evolución y mantenimiento del software.

En el próximo capítulo se presentan distintas estrategias que se encargan de extraer información de los sistemas, y luego en analizar la información extraída. Este análisis, puntualmente es sobre los identificadores presentes en el código del sistema de estudio. Algunas de estas estrategias están implementadas en forma de Herramientas de CP.

Capítulo 3

Análisis de Identificadores: Estado del Arte

3.1. Introducción

En el capítulo anterior se introdujo el ámbito de comprensión de programas con las definiciones de los conceptos más importantes. Este capítulo se centra en el estado del arte de algunas técnicas y herramientas orientadas a la CP. Las mismas basan su análisis en los identificadores (ids) situados en los códigos de programas. También se explica de la importancia que tienen los comentarios y los literales al momento de examinar ids. Al final del capítulo se brindan algunos comentarios sobre los temas tratados. A continuación se desarrolla una introducción sobre la temática asociada a identificadores.

Los equipos de desarrollo de software frecuentemente enfocan todo su esfuerzo en el análisis, diseño, implementación y mantenimiento de los sistemas, restándole importancia a la documentación. Por lo tanto, es común encontrar paquetes de software carentes de documentación, lo cual indica que la lectura de los códigos de los sistemas es la única manera de interpretarlos. Es necesaria la interpretación del sistema sobre todo en grandes equipos de desarrollo, por el simple hecho de que un integrante del equipo puede tomar código ajeno para continuar con su desarrollo o realizar algún tipo de mantenimiento.

Teniendo en cuenta que los códigos crecen con los nuevos requerimientos y el frecuente mantenimiento, los sistemas son cada vez más complejos y difíciles de entender. Un camino para lograr un entendimiento ágil y facilitar las arduas tareas de interpretación de códigos, es a través del uso de las herramientas de comprensión. Estas herramientas presentan diferentes perspectivas del sistema, y con esto se agiliza su análisis e inspección. Para diseñar y construir este tipo de herramientas, se recurre al área de CP. Como se mencionó en el capítulo anterior, la CP es un área de la Ingeniería de Software que brinda métodos, técnicas y herramientas que facilitan al programador entender los programas. Un aspecto importante de la CP es la extracción de información estática (Ver Capítulo 2). El correspondiente trabajo final hará más hincapié en esta temática. Cabe destacar que el resto de las temáticas asociadas a la CP no dejan de ser importantes, sin embargo su análisis escapan a los objetivos de este trabajo.

Una forma de extraer información estática es aplicar técnicas de compilación conocidas, con estas técnicas se extrae información que hay detrás de los componentes visibles en los códigos. Entre los distintos componentes visibles, los lectores de los códigos fijan su atención principalmente en los ids y los comentarios, ambos son una fuente de información importante para la CP. Sin embargo, cuando en el código no abundan los comentarios, el foco de atención se centra en los ids.

En la tabla 3.1 se muestra un análisis léxico que se realizó sobre 2.7 millones de líneas de códigos escritos en lenguaje JAVA [25, 17].

Elementos de un código	Cant. por elemento	% / total de elementos	Caracteres utilizados	% / total de caracteres
Palabras claves	1321005	11.2 %	6367677	12.7 %
Delimitadores	5477822	46.6 %	5477822	11.0 %
Operadores	701607	6.0 %	889370	1.8 %
Literales	378057	3.2 %	1520366	3.0 %
Identificadores	3886684	33.0 %	35723272	71.5 %
Total	11765175	100.0 %	49978507	100.0 %

Tabla 3.1: Análisis Léxico de códigos JAVA

En la tabla 3.1 se ve claramente que más de las dos terceras partes (71.5 %) de los caracteres en el código fuente forman parte de un id. Por ende, en el ámbito de CP los ids son una fuente importante de información que el lector del código o encargado de mantenimiento debe tener en cuenta. Utilizar una herramienta que analice los ids dando a conocer su significado ayuda a revelar esta información, mejora la comprensión, aumenta la productividad y agiliza el mantenimiento de los sistemas.

Por lo antedicho, construir herramientas de CP que analicen ids en los códigos fuentes de los programas constituye un aporte importante al ámbito de CP. Antes de comenzar con la incursión de herramientas existentes que analizan ids, se detallan algunos conceptos claves relacionados con la temática.

3.2. Conceptos claves

*“Un **Identificador** (**id**) básicamente se define como una secuencia de letras, dígitos o caracteres especiales de cualquier longitud que sirve para identificar las entidades del programa”*

Cada lenguaje tiene sus propias reglas que definen como pueden estar contruidos los nombres de sus ids. Por ejemplo, en JAVA no está permitido declarar ids que coincidan con palabras reservadas o que contengan operadores relacionales o matemáticos (+ − & ! %), a excepción del guión bajo (_) o signo peso (\$). Ejemplo: `var_char`, `var$char`.

Generalmente, la buena práctica de programación recomienda que al momento de construir un id dentro del código, el mismo debe estar asociado a un concepto [25, 17, 45].

Identificador \Leftrightarrow Concepto

Por ejemplo, el id `userAccount` está asociado al concepto ‘cuenta de usuario’, en este caso si el sistema que se analiza es bancario, el concepto antedicho pertenece al Dominio del Problema. Por lo tanto, uno de los requisitos importantes que debe reunir un programa para facilitar su comprensión es que sus ids sean claros, de esta manera se podrán ver más fácilmente los conceptos

que cada id representa. Sin embargo, dicho requerimiento no es tenido muy en cuenta por los programadores [25, 44, 48, 45].

En la siguiente sección se menciona, como los nombres asignados a los ids impacta enormemente en la interpretación del concepto que el id tiene asociado, por lo tanto afecta la lectura comprensiva del código. Todas estas características, son tenidas en cuenta en el ámbito de la CP.

3.3. Nombres de Identificadores

Durante el desarrollo del sistema, las reglas de construcción de ids se enfocan más en el formato del código y el formato de la documentación, en lugar de enfocarse en el concepto que el id representa. Más adelante de la etapa de desarrollo, viene la etapa de mantenimiento del sistema, aquí es probable que el encargado de hacerlo, no sea el mismo que desarrolló el sistema y generalmente no tiene en cuenta los nombres de los ids, ni que conceptos representan a la hora de interpretar el código.

Antes de proseguir sobre la importancia que tienen los nombres de los ids en la CP, a continuación se clasifican las distintas formas que se puede nombrar un id multi-palabra¹.

3.3.1. Clasificación

Estudios realizados con 100 programadores [48, 49] sobre comprensión de ids indican que existen tres formas principales de construir ids multi-palabras (tomando como ejemplo el concepto `File System Input`):

- Palabras completas (`fileSystemInput`).
- Abreviaturas (`flSyslpt`).
- Una sola letra² (`fsi`).

También pueden existir la combinación entre dos tipos de clasificaciones: por ejemplo `flSystemlpt`.

¹Que contiene más de una palabra.

²Esta clasificación se la conoce como acrónimo.

Los estudios antes mencionados le posibilitaron a los investigadores concluir que las palabras completas son las más comprendidas, sin embargo las estadísticas marcan en algunos casos que las abreviaturas que se ubican en segundo lugar, no demuestran una diferencia notoria con respecto a las palabras completas [48, 57].

Los investigadores Feild, Binkley, Lawrie [30, 31, 18, 17], clasifican los nombres de los ids con dos términos conocidos en la jerga del análisis de ids: *hardwords* y *softwords*.

Los *hardwords* destacan la separación de cada palabra que compone al id multi-palabra a través de una marca específica; algunos ejemplos son: `fileSystem` en donde se marca bien la separación de cada palabra con el uso de mayúscula entre las minúsculas¹, o todas mayúsculas seguidas de todas minúsculas `FILEsystem`. También se utilizan para dividir las palabras, un símbolo especial, como es el caso del guión bajo: `file_system`.

En cambio los *softwords* no poseen ningún tipo de separador o marca que de indicios de las palabras que lo componen; por ejemplo: `textInput` o `TEXTINPUT` que está compuesto por `text` y por `input` sin tener una marca que destaque la separación. La nomenclatura de *hardwords* y *softwords* se utilizará en el resto de este trabajo final.

Habiendo explicado algunas clasificaciones en lo que respecta al tipo de nombres asignados en los ids, en la próxima sección se retoma la importancia que tienen los nombres utilizados en los ids.

3.3.2. Importancia en los Nombres

En la actualidad existen innumerables convenciones en cuanto a la construcción sintáctica de los ids, alguno de ellos son:

- En el caso de JAVA, los nombres de los paquetes deben ser con minúscula (`main.packed`). Las clases con mayúscula en la primer letra de cada palabra que compone el nombre (`MainClass`).

¹Este caso es conocido como camel-case ya que las letras en mayúsculas se asemejan a las jorobas de un camello.

```
function mr_mr_1(mr, mr_1)
  if Null(mr) or Null(mr_1) then
    exit function
  end if
  mr_mr_1 = (mr - mr_1)
end function
```

Figura 3.1: Trozo de Código de un Sistema Comercial

- En el caso de C#, las clases se nombran igual que JAVA. Pero para el caso de los paquetes deben comenzar con mayúscula y el resto minúscula (Main.Packed).

Lo mencionado en los ítems precedentes, indica que se concentra más en los aspectos sintácticos del id y no tanto en los aspectos semánticos, a la hora de asignar nombres a los ids.

Una evidencia fehaciente de la importancia en la semántica de nombres, son las técnicas que se aplican para protección de código. Algunas de ellas se encargan de reemplazar los nombres originales de los ids por secuencias de caracteres aleatorios y de esta manera se reduce la comprensión. Estas técnicas se conocen con el nombre de ofuscación de código. La ofuscación es común en los sistemas de índole comercial, en la Figura 3.1 se puede observar un ejemplo tomado de un caso real, en donde la función `mr_mr_1` no parece complicada pero se desconoce la finalidad de su ejecución [25].

A su vez, los programadores cuando desarrollan sus aplicaciones, restan importancia a la semántica de nombres asignados a los ids. Existen tres razones destacadas que conllevan a esto:

1. Los ids son escogidos por los programadores, sin tener en cuenta los conceptos que tienen asociados.
2. Los desarrolladores tienen poco conocimiento de los nombres usados en los ids ubicados en otros sectores del código fuente.
3. Durante la evolución del sistema, los nombres de los ids se mantienen y no se adaptan a nuevas funcionalidades (o conceptos) que puedan tener asociado.

En este sentido, la construcción de ids poco claros, se combate con la programación “menos egoísta”. Esta consiste en hacer programas más comprensibles para el futuro lector que no está familiarizado con el código. Para lograrlo se deben respetar dos reglas sobre los nombres que se le asignan a los ids [25, 44]:

Nombre Conciso: El nombre de un id es conciso, si la semántica del nombre coincide exactamente con la semántica del concepto que el id representa.

Nombre Consistente: Cada id, debe tener asociado si y solo si, un único concepto (relación biyectiva entre los ids y los conceptos).

Un ejemplo de *conciso* es `output_file_name` que representa el concepto de ‘nombre de archivo de salida’, distinto sería un id nombrado como `file_name`, el cual está incompleto y no representa de forma semánticamente concisa el concepto mencionado.

Los propiedades que violan nombrar a un id de manera *consistente* son conocidas en el lenguaje natural como sinónimos y homónimos. Los homónimos son palabras que pueden tener más de un significado. Por ende, si el nombre de un id está asociado a más de un concepto, no estará claro que concepto representa. Por ejemplo, un id con el nombre `file` generalmente se asocia al concepto de ‘archivo’, pero puede que se refiera a una estructura del tipo cola.

Por otro lado, los sinónimos indican que para un mismo concepto pueden tener asociados diferentes nombres. Por ejemplo, un id con el nombre `accountBankNumber` y otro `accountBankNum` son sinónimos porque hacen referencia al mismo concepto ‘número de cuenta bancaria’.

Esta demostrado [25, 44, 4, 17] que la ausencia de nombres consistentes tales como se mencionó anteriormente, hacen que se dificulte identificar con claridad los conceptos en el dominio del problema, lo que hace aumentar los esfuerzos de comprensión del programa.

Por lo tanto, si los ids están contruidos de forma *concisa* (identificando bien al concepto) y la *consistencia* está presente, se pueden descubrir los

conceptos que representan en el dominio del problema más fácilmente. De esta manera, se agiliza la comprensión, aumenta la productividad, mejora la calidad durante la etapa de mantenimiento [25, 44].

Intuitivamente, se necesita que los ids representen bien al concepto, ya que mayor será el impacto que tendrá en la interpretación del sistema [25, 44, 21]. Sin embargo, durante las etapas de desarrollo y mantenimiento de software, es muy difícil mantener una consistencia global de nombres en los ids, sobre todo si el sistema es grande. En este sentido, cada vez que un concepto se modifica el nombre del id asociado debe cambiar y adaptarse a la modificación.

Los autores Deissenboeck y Pizka [25] elaboraron una herramienta que ayuda a la construcción y mantenimiento de ids con nombres concisos y consistentes. Dada la dificultad que conlleva construir una herramienta totalmente automática que se encargue de nombrar y mantener correctamente los ids, ellos elaboraron una herramienta que necesita la intervención del programador. Esta herramienta, a medida que el sistema se va desarrollando, construye y mantiene un diccionario de datos compuesto con información de ids. En el ámbito de la Ingeniería de Software el concepto de diccionarios de datos es importante.

Diccionarios de Datos: Este concepto conocido también como ‘glosario de proyecto’ se recomienda en los textos orientados a la administración de proyectos de software. Con los diccionarios se describe en forma clara todos los términos utilizados en los grandes sistemas de software. También brindan una referencia completa a todos los participantes de un proyecto durante todo el ciclo de vida del producto [55].

Este concepto sirvió de inspiración a los autores para construir la herramienta, que a continuación se describe.

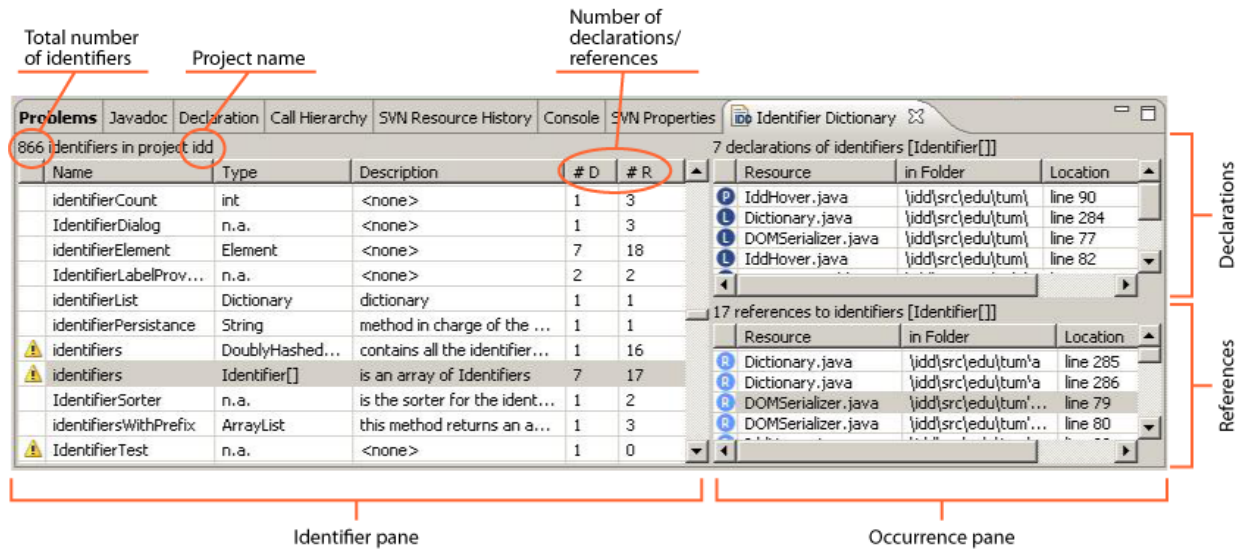


Figura 3.2: Visualización de Identifier Dictionary

3.3.3. Herramienta: Identifier Dictionary

La herramienta conocida con el nombre de *Identifier Dictionary* (IDD)¹ construida por Deissenboeck y Pizka [25] actúa como un diccionario de datos, que ayuda al desarrollador a mantener la consistencia de nombres en los ids de un proyecto JAVA. Es una base de datos que almacena información de los ids tales como el nombre, el tipo del objeto que identifica y una descripción comprensiva.

La herramienta IDD ayuda a reducir la creación de nombres sinónimos (Ver apartado anterior) y asiste a escoger un nombre adecuado para los ids siguiendo el patrón de nombres existentes. Además, aumenta la velocidad de comprensión del código en base a las descripciones de cada id. El equipo encargado de tareas de mantenimiento localiza un concepto del dominio del problema y luego su correspondiente id de manera ágil. Otro aporte que hace la herramienta es asegurar la calidad de los nombres (nombres concisos) de los ids con un esfuerzo moderado, usando como ayuda la descripción comprensiva ubicada en la base de datos [25, 46].

Se implementó como extensión de la IDE Eclipse 3.1². Se visualiza en el

¹<http://www4.informatik.tu-muenchen.de/~ccsm/idd/index.html>

²<http://www.eclipse.org/jdt>

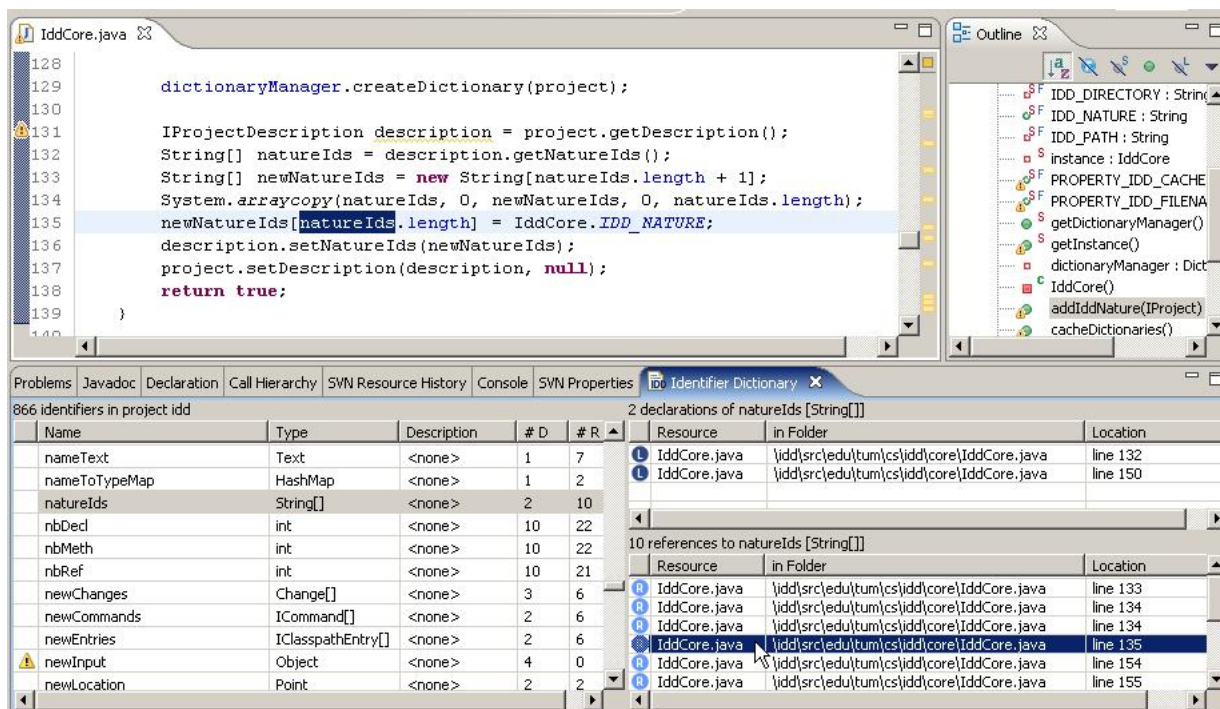


Figura 3.3: Visualización de Identifier Dictionary

panel de las vistas de la IDE y consiste de tres secciones (Ver Figura 3.2):

- Una tabla con información de los ids en el proyecto: nombre, tipo, descripción, cantidad de declaraciones y cantidad de referencias (Identifier pane).
- Una lista de ids declarados en el proyecto (Occurrence pane).
- Una lista de referencias de los ids en el proyecto (Occurrence pane).

Mientras se realiza el desarrollo del código la herramienta asiste al programador a llevar buenas prácticas de asignación de nombres en los ids, a través de las siguientes características:

Navegación en el código fuente: Si se selecciona un id, en la tabla de ids (inferior izquierda), mostrará la ubicación exacta en donde se encuentra cada declaración y referencia (Ver Figura 3.3).

Advertencias (warnings): Mientras se realiza la recolección de ids, los íconos de advertencia indican potenciales problemas en el nombre asignado.

```
String message;
if (selectedProject == null) {
    message = "No project selected.";
} else {
    String projectName = selectedProject.getName();
    message = "Project " + projectName
        + " has no Ident";
}
setContentDescription(message);
splitter.setVisible(false);
```

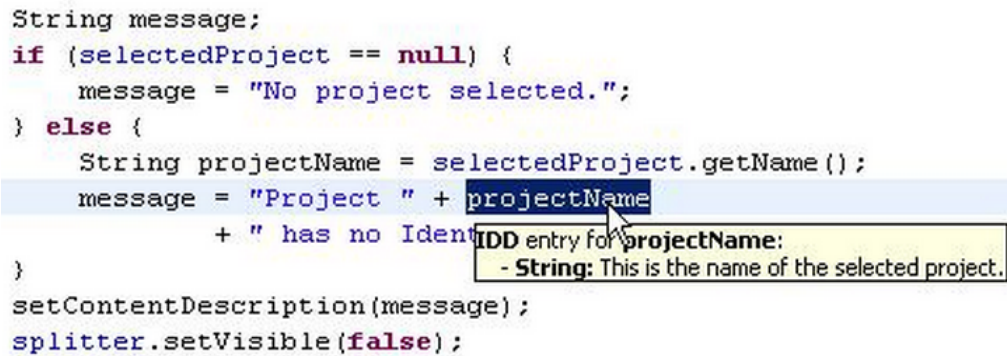


Figura 3.4: Visualización de Identifier Dictionary

Los dos tipos de mensajes que se muestran son: dos ids con el mismo nombre pero distinto tipos y el id es declarado pero no referenciado¹.

Mensajes pop-up: Se puede visualizar información tales como: la descripción del id mientras se está programando, esto se logra posicionando el cursor sobre el id en el código fuente (Ver Figura 3.4).

Auto-completar nombres: Las IDE² actuales proveen la función de auto-completar. Sin embargo, esta funcionalidad falla cuando los nombres de los ids no están declarados dentro del alcance actual de edición. Con el plugin IDD a la hora de auto-completar mira todos los ids del proyecto sin importar el ambiente en el que se encuentre.

Renombre global de ids: Esta función permite renombrar cualquier id generando una vista previa y validando el nombre de los ids a medida que el sistema va evolucionando. De esta forma se preserva la consistencia global de nombres.

La herramienta IDD trabaja internamente con un colector de ids que está acoplado al proceso de compilación del proyecto (Build Project) de Eclipse. Los ids se van recolectando a medida que el programa se va compilando. Los nombres, el tipo, la descripción se van guardando en un archivo XML. También se puede exportar en un archivo en formato HTML el cual permite una lectura más clara de los ids con toda información asociada [25].

¹Similar a los warnings de Eclipse

²Entornos de desarrollos integrados, por su siglas en inglés. Netbeans, Eclipse etc.

La herramienta IDD colabora en mejorar los nombre de los ids con un esfuerzo moderado como se describió antes. Sin embargo, los investigadores Feild, Binkley, Lawrie [46, 44] determinaron que en IDD, los esfuerzos son moderados solo para sistemas que se empiezan a programar desde el comienzo y no con sistemas ya existentes.

Para concluir con esta sección, la buena calidad en los nombres de los ids mejora el entendimiento del código. Además, muchos expertos sostienen que las técnicas de Ingeniería Inversa se emplean con mayor precisión si el código está bien escrito. Algunas de estas técnicas tienen como objetivo mejorar la CP, y una forma conocida de lograrlo es traduciendo los nombres abreviados de los ids a palabras más completas en lenguaje natural. En la siguiente sección se describen este tipo de técnicas de regresión.

3.4. Traducción de Identificadores

Los lectores de códigos de programas tienen inconvenientes para entender el propósito de los ids y deben invertir tiempo en analizar el significado de su presencia. Por esta razón, las estrategias automáticas dedicadas a facilitar este análisis son bienvenidas en el contexto de la CP.

Para aliviar el inconveniente mencionado anteriormente, un camino viable, consiste en descubrir la información que ocultan los ids detrás de sus abreviaturas. Esta información es relevante ya que pertenece al dominio del problema [29, 46].

3.4.1. Conceptos y Desafíos observados

Una manera de descubrir la información oculta detrás de los ids, es intentar convertir estas abreviaturas en palabras completas del lenguaje natural. Por ende, el foco del análisis de los ids se basa en la traducción de palabras abreviadas a palabras completas.

El proceso que se encarga de realizar la traducción de ids consta de dos pasos [46]:

1. **División:** Separar el id en las palabras que lo componen usando algún separador especial¹. (Ejemplo: `flSys` \Rightarrow `fl-sys`).
2. **Expansión:** Expandir las abreviaturas que resultaron como producto del paso anterior. (Ejemplo: `fl-sys` \Rightarrow `file system`).

Cabe mencionar que el ejemplo mostrado en ambos pasos corresponde a un caso de *hardword* en donde la separación de las palabras es destacada. Sin embargo, la dificultad se presenta en los *softwords* (Ver sección anterior) ya que la división no está marcada (Ejemplo: `hashtable` \Rightarrow `hash-table`). Existen también casos híbridos (Ejemplo: `hashtable_entry`). En este caso el id tiene una marca de separación (guión bajo) con dos *hardwords* `hashtable` y `entry`. A su vez el *hardword* `hashtable` posee dos *softwords* `hash` y `table`, mientras que `entry` es un *hardword* compuesto por un único *softword*.

¹Siempre y cuando el id sea multi-palabra (contenga más de una palabra).

El principal desafío (y el más complejo) en la traducción de ids, es detectar los casos de *softword*. Luego proceder a separar las palabras abreviadas que la componen para posteriormente realizar la expansión [30, 46].

Para afrontar este objetivo los especialistas deciden recurrir a fuentes de palabras en lenguaje natural. Existen 2 tipos de fuentes, dentro del mismo código extrayendo palabras presentes en comentarios, literales y documentación. La otra fuente se encuentra fuera del programa consultando diccionarios o listas de palabras predefinidas.

Habiendo explicado el proceso encargado de expandir las abreviaturas de un id a palabras completas, el siguiente paso es describir las herramientas conocidas que lo implementan.

La autora Emily Hill [38] con alto reconocimiento por su investigación en lo que respecta a expandir ids en códigos JAVA, explica algunas amenazas y desafíos a tener en cuenta a la hora de desarrollar herramientas que analizan ids. A continuación se explican algunas de ellas.

Dificultad para armar diccionarios apropiados: La mayoría de los diccionarios, se usan para corregir la ortografía. Las palabras que incluyen son sustantivos propios, abreviaturas, contracciones¹ y demás palabras que puedan aparecer en un software. Sin embargo, la inclusión de muchas palabras genera que una simple abreviatura (*char*, *tab*, *id*) se trate como una palabra expandida y no se expanda. Por el contrario, si el diccionario contiene pocas palabras, la expansión se realiza más frecuente de lo normal.

Dificultad para traducir abreviaturas cortas: Si un id está abreviado como ‘def’, es complicado determinar con precisión cual es la mejor traducción entre tantos candidatos *definition*, *default*, *defect*. Otra observación hecha, es que mientras más corta es la abreviatura más candidatos posee, el ejemplo más común es *i* que generalmente es *integer* pero

¹Palabras en inglés que llevan apostrofes, ejemplo: let’s.

podrían ser otros tantos `interface`, `interrupt`, etc. Se requieren procesos inteligentes para solucionarlo.

Distintas políticas de expansión: Si la abreviatura se mira como prefijo tiene menos candidatos a traducirse, un ejemplo es `'str'` el cual coincide con `STRing`, `STReam`. En cambio si las letras de `str` forman parte de la palabra tiene más posibilidades de expansión `SubsTRing`, `SToRe`, `SepTembeR`, `SaTuRn`.

Las palabras abreviadas, usadas en los ids dependen mucho de la idiosincrasia del programador. Por lo tanto, construir herramientas automáticas que analicen ids representa un verdadero desafío en el área de CP.

En las próximas dos secciones se explican algoritmos encargados de la división de ids, y en la secciones subsiguientes se describen algoritmos de expansión. En todos los casos se presentan pseudo-códigos de los algoritmos que ejecuta cada técnica, de esta manera ayudarán al lector a entender el funcionamiento de cada una.

3.4.2. Algoritmo de División Greedy

El Algoritmo de división Greedy elaborado por Lawrie, Feild, Binkley [47, 30, 31, 46, 29], emplea tres listas para realizar su trabajo:

Palabras de diccionarios: Contiene palabras de diccionarios públicos y del diccionario que utiliza el comando de Linux `ispell`¹.

Abreviaturas conocidas: La lista se arma con abreviaturas extraídas de distintos programas y de autores expertos. Se incluyen abreviaturas comunes (ejemplo: `alt` \Rightarrow `altitude`) y abreviaturas de programación (ejemplo: `txt` \Rightarrow `text`).

Palabras excluyentes (stop list): Posee palabras que son irrelevantes para realizar la división de los ids. Incluye palabras claves (ejemplo:

¹Comando de Linux generalmente utilizado para corregir errores ortográficos, en archivos de texto. <http://wordlist.aspell.net>

`while`), ids predefinidos (ejemplo: `NULL`), nombres y funciones de librerías (ejemplo: `strcpy`, `errno`), y todos los ids que puedan tener un solo caracter.

El algoritmo de Greedy utiliza las 3 listas nombradas al comienzo de la sección en forma de variable global. Esto ocurre porque las 3 listas son usadas por subrutinas más tarde. El algoritmo procede de la siguiente manera (Ver Algoritmo 1), el id que recibe como entrada primero se divide (con espacios en blanco) en las hardwords que lo componen (ejemplo: `fileinput.txt` \Rightarrow `fileinput` y `txt` en la línea 2 - algoritmo 1, si es camel-case `fileinputTxt` \Rightarrow `fileinput` y `txt` en la línea 3 - algoritmo 1). Luego, cada palabra resultante en caso que esté en alguna de las 3 listas, se distingue como un único softword (ejemplo: `txt` pertenece a la lista de abreviaturas conocidas, línea 5). Si alguna palabra no está en alguna lista se considera como múltiples softwords que necesitan subdividirse (ejemplo: `fileinput` \Rightarrow `file` y `input`, línea 5). Para subdividir estas

Algoritmo 1: División Greedy

```

Var Global: ispellList // Palabras de ispell + Diccionario
Var Global: abrevList // Abreviaciones conocidas
Var Global: stopList // Palabras Excluyentes
Entrada   : idHarword // identificador a dividir
Salida    : softwordDiv // id separado con espacios

1 softwordDiv  $\leftarrow$  ""
2 softwordDiv  $\leftarrow$  dividirCaracteresEspecialesDigitos(idHarword)
3 softwordDiv  $\leftarrow$  dividirCamelCase(softwordDiv)
4 para todo (s | s es un substring separado por ' ' en softwordDiv)
  hacer
5     si (s no pertenece a (stopList  $\cup$  abrevList  $\cup$  ispellList ))
6         entonces
7             resultadoPrefijo  $\leftarrow$  buscarPrefijo(s, "")
8             resultadoSufijo  $\leftarrow$  buscarSufijo(s, "")
           // Se elige la división que mayor particiones hizo.
           s  $\leftarrow$  maxDivisión(resultadoPrefijo, resultadoSufijo)

9 devolver softwordDiv // Retorna el id dividido por ' '
```

palabras se buscan los prefijos y los sufijos más largos posibles dentro de ellas, para hacer esta búsqueda se utilizan las mismas 3 listas antes mencionadas (líneas 6 y 7).

Por un lado se buscan prefijos con un proceso recursivo (Ver Función **buscarPrefijo**). Este proceso comienza analizando toda la palabra por completo. Se van extrayendo caracteres del final hasta encontrar el prefijo más largo o no haya más caracteres (líneas 5 - 7, **buscarPrefijo**).

Tomando como ejemplo $s = \text{flinptxt}$, en las línea 5 y 6 se realiza lo siguiente, el caracter que se toma de la última posición de flinptxt se coloca en $sDiv$ en el mismo orden (ejemplo: $s = \text{flinptx}$ y $sDiv = t$), estos nuevos parámetros son pasados en la misma función **buscarPrefijo** (línea 7). Este proceso recursivo se repetirá ($s = \text{flinpt}$ y $sDiv = xt$; $s = \text{flinp}$ y $sDiv = txt$; etc), hasta que sucedan dos posibilidades: a) s no tenga más caracteres (línea 1), en ese caso se retorna $sDiv$ con el resultado de la ejecución; o b) s tenga un prefijo válido (pertenezca a algún diccionario - línea 3), en caso de que así sea, se procede a separarlo colocando un separador (‘ ’) y luego se invoca

Función buscarPrefijo

Entrada: s // *cadena a dividir*

Salida : $sDiv$ // *Cadena dividida con espacios*

```

1 si ( $length(s) = 0$ ) entonces
    // Punto de parada de la recursión, retorna el
    // resultado de la división.
2     devolver  $sDiv$ 

3 si ( $s$  pertenece a ( $stopList \cup abbrevList \cup ispellList$ )) entonces
    // Se separa el prefijo encontrado y el resto de la
    // cadena se sigue procesando.
4     devolver ( $s + ' ' + \text{buscarPrefijo}(sDiv, "")$ )

    // Se extrae y se guarda el último caracter de  $s$ .
5  $sDiv \leftarrow s[length(s) - 1] + sDiv$ 

    // Llamar nuevamente a la función sin el último caracter.
6  $s \leftarrow s[0, length(s) - 1]$ 
7 devolver  $\text{buscarPrefijo}(s, sDiv)$ 

```

nuevamente a **buscarPrefijo** para buscar más subdivisiones (línea 4). Para aclarar más el punto b) anterior, tomando el ejemplo antedicho, si *fl* es el prefijo más largo encontrado en *flinptxt*, entonces *fl* (variable *s*) se separa del resto con ‘ ’ y la palabra *inptxt* (variable *sDiv*) se procesa con **buscarPrefijo** (línea 4). Si el resultado de este último fuera, por ejemplo, que *inp* se divida de *txt*, el resultado final será *fl inp txt*.

De manera simétrica, otro proceso recursivo se hace cargo de los sufijos (Ver Función **buscarSufijo**). También extrae caracteres, pero en este caso desde la primer posición hasta encontrar el sufijo más largo presente en alguna lista o no haya más caracteres (líneas 5 - 7, **buscarSufijo**). De la misma forma que la función de prefijos, cuando encuentra una palabra, se inserta un separador (‘ ’) y el resto se procesa por la función **buscarSufijo** para buscar más subdivisiones (línea 4).

Una vez que ambos procesos terminaron, los resultados son retornados al algoritmo principal (*resultadoPrefijo*, *resultadoSufijo* líneas 6 y 7, algoritmo 1). Mediante una función de comparación se elije el que obtuvo mayores par-

Función **buscarSufijo**

Entrada: *s* // *cadena a dividir*

Salida : *sDiv* // *Cadena separada con espacios*

```

1 si (length(s) = 0) entonces
    // Punto de parada de la recursión, retorna el
    // resultado de la división.
2     devolver sDiv

3 si (s pertenece a (stopList  $\cup$  abrevList  $\cup$  ispellList)) entonces
    // Se separa el sufijo encontrado y el resto de la
    // cadena se sigue procesando.
4     devolver (buscarSufijo(sDiv, “”) + ‘ ’ + s)

    // Se extrae y se guarda el primer caracter de s.
5 sDiv  $\leftarrow$  sDiv + s[0]

    // Llamar nuevamente a la función sin el primer caracter.
6 s  $\leftarrow$  s[1 , length(s)]
7 devolver buscarSufijo(s,sDiv)

```

ticiones (línea 8). Finalmente, el algoritmo Greedy retorna el id destacando las palabras que lo componen mediante el separador espacio (`file input txt`).

La ventaja de hacer dos búsquedas (prefijo y sufijo) radica en aumentar las chances de dividir al id. A modo de ejemplo, suponga que la palabra abreviada `fl` no se encuentra en ninguno de los 3 listados y las palabras `input` y `txt` si están. Dada esta situación, si el id `flinputtxt` se procesa por ambas rutinas, el resultado será que **buscarPrefijo** no divida al id. Esto sucede porque al retirar caracteres del último lugar, nunca se encontrará un prefijo conocido. Más precisamente al no dividirse entre `fl` e `input` el resto de la cadena no se procesará y tampoco se dividirá entre `input` y `txt`.

Sin embargo, este inconveniente no lo tendrá **buscarSufijo** porque al retirar los caracteres del principio de la palabra, `input txt` será separado. Como `input` es una palabra conocida se agregará un espacio entre `fl input`. De esta manera el id queda correctamente separado `fl input txt` por **buscarSufijo**.

Este algoritmo de división generalmente se emplea como punto de comparación con técnicas más eficientes de separación de ids multi-palabras. En la próxima sección, se describe un algoritmo de división más avanzado.

3.4.3. Algoritmo de División Samurai

Esta técnica pensada por Eric Enslen, Emily Hill, Lori Pollock, Vijay-Shanker [29] divide a los ids en secuencias de palabras al igual que Greedy, con la diferencia que la separación es más efectiva. La estrategia utiliza información que está presente en los códigos, para llevar a cabo el objetivo, asume que el id está compuesto con palabras que son utilizadas en otras partes del sistema. Esto permite que no sea necesario utilizar diccionarios predefinidos, y por lo tanto, Samurai no está limitado por el contenido de estos diccionarios. De esta manera, la técnica va evolucionando con el tiempo a medida que aparezcan nuevas tecnologías, y nuevas palabras se incorporen al vocabulario de los programadores. La estrategia de separación Samurai está inspirada en un técnica de expansión de abreviaturas AMAP [38] que se describe en próximas secciones.

El algoritmo Samurai, selecciona la partición más adecuada en los ids

multi-palabra¹ en base a una función de puntuación (scoring). Esta función, utiliza información que está presente en dos tablas. Una es la *tabla de frecuencias locales*, que posee dos entradas, una entrada tiene el listado de palabras extraídos del programa actual bajo análisis (cada palabra es única en la tabla), y la otra es el número de ocurrencias (frecuencia de aparición) de cada palabra. Por otro lado, se encuentra la *tabla de frecuencias globales*. Esta tabla contiene las mismas columnas que la tabla anterior, las palabras y sus frecuencias. La diferencia es que las palabras de la tabla global, se recolectan de distintos programas de gran envergadura, y no del programa actual. Esto indica que la *tabla de frecuencias global* está predefinida de antemano.

Las palabras de ambas tablas, se extraen de los comentarios, literales strings, documentación y a su vez en los propios ids. Para el caso de los ids que son multi-palabras, si son hardwords se dividen y se toma a cada palabra por separado (ejemplo: `fileSystem` \Rightarrow `file system` o `file_system` \Rightarrow `file system`).

Los autores de la estrategia Samurai sostienen, que los ids multi-palabras también se encuentran dentro de los comentarios y literales strings del código. Es por esta razón, que el parámetro de entrada del algoritmo se nombra de una manera más genérica: *token*, y no simplemente “identificador”. Por tal motivo, se menciona la expresión “dividir al token”, en lugar del “dividir al identificador”.

La ejecución del algoritmo Samurai, invoca dos rutinas primero se ejecuta *divisiónHardWord* y después *divisiónSoftWord*. La primera básicamente se encarga de dividir los hardwords (palabras que poseen guión bajo o son del tipo camel-case), luego cada una de las palabras obtenidas (en forma de softword), son pasadas a la segunda rutina para continuar con el análisis. A continuación se explican ambas rutinas.

División Hardword

En la rutina *divisiónHardWord* (Ver Algoritmo 2) primero se ejecutan dos funciones (líneas 1 y 2). La primera *dividirCaracteresEspecialesDigitos*, toma al token y reemplaza todos los posibles caracteres especiales y números por

¹Que posee más de una palabra.

espacio en blanco. La segunda *dividirMinusSeguidoMayus*, agrega un blanco entre dos caracteres que sea una minúscula seguido por una mayúscula. En este punto solo quedan tokens de la forma softword o que contengan una mayúscula seguido de minúscula (Ejemplos: List, ASTVisitor, GPSstate, state, finalstate, MAX).

Los casos de softword que se obtuvieron (state, finalstate, MAX) van directo a la rutina *divisiónSoftWord*. El resto del tipo, mayúscula seguido de minúscula (List, ASTVisitor, GPSstate) continúa con el proceso de división. Con respecto a estos tipos, el autor, hace la siguiente clasificación:

Camel-case sencillo: La mayúscula indica el comienzo de la nueva palabra.

Ejemplos: List, AST Visitor.

Variante Camel-case: El autor a través de estudios de datos, se encontró con variantes en donde la mayúscula indica el fin de una palabra. Ejemplos: SQL list, GPS state.

El algoritmo para determinar cual es el tipo correcto y no tomar malas decisiones (Ejemplos: SQ Llist o GP Sstate), divide a la palabra por ambas consideraciones (ejemplo: GP Sstate y GPS state), luego calcula el puntaje (score) de la parte derecha de ambas divisiones (líneas 7 y 8). Aquella con puntaje más alto entre las dos será por la cual se decida (línea 9). La raíz cuadrada se explica más adelante. Tomando como ejemplo el id GPSstate, para el caso camel-case calculará $score(Sstate)$ y para la otra variante $score(state)$. Intuitivamente, la función score (a través de las tablas de frecuencias de palabras) elegirá *state* sobre *sstate* ya que esta última debería tener un puntaje inferior (frecuencia menor de aparición). Por ende Samurai detecta que GPSstate es del tipo Variante de Camel-Case. La división elegida se lleva a cabo en las líneas 11 y 13 (según el caso). Finalmente, todas las partes que fueron divididas se envían a *divisiónSoftWord* (línea 17, 18).

División Softword

La rutina recursiva *divisiónSoftWord* (Ver Algoritmo 3) recibe como entrada un substring *s*, el cual puede tener tres tipos de variantes: a) todos los

Algoritmo 2: divisiónHardWord

Entrada: *token* // *token a dividir*
Salida : *tokenSep* // *token separado con espacios*

```

1 token ← dividirCaracteresEspecialesDigitos(token)
2 token ← dividirMinusSeguidoMayus(token)
3 tokenSep ← ""
4 para todo (s | s es un substring separado por ' ' en token) hacer
5     si (  $\exists \{i | esMayus(s[i]) \wedge esMinus(s[i + 1])\}$  ) entonces
6          $n \leftarrow \text{length}(s) - 1$ 
7         // se determina con la función score si es del tipo
8         // camelcase u otra alternativa
9          $scoreCamel \leftarrow \text{score}(s[i,n])$ 
10         $scoreAlter \leftarrow \text{score}(s[i+1,n])$ 
11        si ( $scoreCamel > \sqrt{scoreAlter}$ ) entonces
12            si ( $i > 0$ ) entonces
13                 $s \leftarrow s[0,i - 1] + ' ' + s[i,n]$  // GP Sstate
14            en otro caso
15                 $s \leftarrow s[0,i] + ' ' + s[i + 1,n]$  // GPS state
16        tokenSep ← tokenSep + ' ' + s
17 token ← tokenSep
18 tokenSep ← ' '
19 para todo (s | s es un substring separado por ' ' en token) hacer
20     tokenSep ← tokenSep + ' ' + divisiónSoftWord(s,score(s))
21 devolver tokenSep

```

caracteres en minúsculas (*visitor*), b) todos con mayúsculas (*VISITOR*), c) el primer caracter con mayúscula seguido por todas minúsculas (*Visitor*). El otro parámetro de entrada es el puntaje original ($score_{sd}$) correspondiente a *s*.

La rutina primero examina cada punto posible de división en *s* dividiendo en *split_{izq}* y *split_{der}* respectivamente (líneas 4 y 5). La decisión de cual es la mejor división (línea 9), se basa en: a) substrings que no tengan prefijos o

sufijos conocidos, los mismos están disponibles en la página web del autor¹ (línea 6), b) que el puntaje de la división elegida sobresalga del resto de los puntajes (líneas 7-8).

Para aclarar los puntos a) y b) mencionados en el párrafo precedente, para cada partición (izquierda o derecha) obtenida se calcula el score (líneas 4 y 5). Luego este se compara con el puntaje máximo ($\max(..)$) entre el puntaje de la palabra original (score_{sd} , score original) y el puntaje de la palabra actual ($\text{score}(s)$). En un principio ambos puntajes serán iguales, ya que se trata de la misma palabra, pero a medida que avance la recursión $\text{score}(s)$ puede variar con respecto a score_{sd} (líneas 7 y 8). La raíz cuadrada se explica más adelante.

El si de la línea 9, controla que no tenga prefijos y sufijos ordinarios, y que la parte derecha e izquierda sean serios candidatos a ser divididos, a su vez, se determina con esto, que la parte derecha es una palabra que no necesita dividirse más. En las líneas 10 y 11 se comprueba que la división se realice siempre y cuando, supere al máximo puntaje obtenido hasta el momento. En la línea 12 procede a separar la palabra.

El si de la línea 13, controla que no tenga prefijos y sufijos ordinarios, y que únicamente la parte izquierda es un candidato serio a ser separado, mientras que la cadena de la parte derecha se invoca recursivamente con la misma rutina *divisiónSoftWord*, porque podría seguir dividiéndose en más partes (línea 14).

Si la parte derecha finalmente se divide (línea 15 - el si es verdadero), luego entre la parte izquierda y la derecha también. Por ejemplo, el id *countrownumber* primero se analiza *rownumber* (parte derecha - línea 14) como este finalmente se separará en *row number*, la palabra *count* (parte izquierda) se divide del resto (línea 16), dando como resultado *count row number*. Sin embargo, cuando la parte derecha no se divide tampoco se debería separar entre ambas partes (línea 15 - el si es falso). Los análisis de datos hechos por el autor [29] obligan a hacer este control, ya que se encontraron abundantes casos erróneos de división, uno de ellos es *string ified*. Con este nuevo control al no dividirse *ified* tampoco lo hará del resto dejando la palabra

¹Listas de prefijos y sufijos <http://www.eecis.udel.edu/~enslen/Site/Samurai>.

Algoritmo 3: divisiónSoftWord

Entrada: s // *softword string*
Entrada: $score_{sd}$ // *puntaje de s sin dividir*
Salida : $tokenSep$ // *token separado con espacios*

```

1  $tokenSep \leftarrow s$ ,  $n \leftarrow \text{length}(s) - 1$ 
2  $i \leftarrow 0$ ,  $maxScore \leftarrow -1$ 
3 mientras ( $i < n$ ) hacer
4      $score_{izq} \leftarrow \text{score}(s[0,i])$ 
5      $score_{der} \leftarrow \text{score}(s[i+1,n])$ 
6      $preSuf \leftarrow \text{esPrefijo}(s[0,i]) \vee \text{esSufijo}(s[i+1,n])$ 
7      $split_{izq} \leftarrow \sqrt{score_{izq}} > \max(\text{score}(s), score_{sd})$ 
8      $split_{der} \leftarrow \sqrt{score_{der}} > \max(\text{score}(s), score_{sd})$ 
9     si ( $!presuf \wedge split_{izq} \wedge split_{der}$ ) entonces
10         si ( $(split_{izq} + split_{der}) > maxScore$ ) entonces
11              $maxScore \leftarrow (split_{izq} + split_{der})$ 
12              $tokenSep \leftarrow s[0,i] + ' ' + s[i+1,n]$ 
13         sinó, si ( $!presuf \wedge split_{izq}$ ) entonces
14              $temp \leftarrow \text{divisiónSoftWord}(s[i+1,n], score_{sd})$ 
15             si ( $temp$  se dividió?) entonces
16                  $tokenSep \leftarrow s[0,i] + ' ' + temp$ 
17          $i \leftarrow i+1$ 
18 devolver  $tokenSep$ 

```

correctamente unida stringified.

Otro problema detectado son las palabras de pocos caracteres (menor a 3). Estas palabras, tienen mucha aparición en los códigos y por lo general el puntaje es más alto que el resto. Por esta razón, el autor [29] en base a un análisis sustancial decide colocar la raíz cuadrada en algunos resultados de score antes de comparar (línea 7 y 8), sino la división frecuentemente sería errónea. Un ejemplo es la palabra **per formed**, esto ocurre por que **per** tiene un alto puntaje y fuerza la separación.

En el algoritmo anterior *divisiónHardWord*, la presencia de la raíz cua-

drada (línea 9), cuando se compara el caso camel-case y el caso alternativo también es para solucionar este mismo problema.

Función de Scoring

Para que la técnica Samurai pueda llevar a cabo la tarea de separación de ids, se necesita la función de scoring. Como bien se explicó anteriormente, esta función participa en 2 decisiones claves durante el proceso de división:

- En la rutina *divisiónHardWord*, para determinar si el la división del id es un caso de camel-case o no (líneas 7 y 8).
- En la rutina *divisiónSoftWord*, para puntuar las diferentes particiones de substrings y elegir la mejor separación (líneas 4, 5, 7 y 8).

Dado un string s , la función $score(s)$ retorna un valor que indica: i) la frecuencia de aparición de s en el programa bajo análisis, y ii) la frecuencia de aparición de s en un conjunto grande de programas predefinidos. Las tablas de frecuencias de aparición de palabras local y global le brindan información a la función score. La inclusión de la tabla de frecuencias global, está dada porque los programas pequeños en general tienen frecuencias bajas en la tabla local, y a veces, la función score no trabaja correctamente a causa de esto.

La fórmula es la siguiente:

$$Frec(s, p) + (globalFrec(s) / \log_{10}(totalFrec(p)))$$

Donde p es el programa de estudio, $Frec(s, p)$ es la frecuencia de ocurrencia de s en p . La función $totalFrec(p)$, es la frecuencia total de todos los strings en el programa p . La función $globalFrec(s)$, es la frecuencia de aparición de s en una gran conjunto de programas tomados como muestras¹. Esta fórmula de score fue desarrollada por los autores, a través del análisis de datos obtenidos [29].

¹Estos programas son alrededor de 9000 y están escritos en JAVA

3.4.4. Algoritmo de Expansión Básico

El algoritmo de expansión de abreviaturas ideado por Lawrie, Feild, Binkley (mismos autores que la técnica de separación Greedy) [46] trabaja con cuatro listas para realizar su tarea:

- Una lista de palabras (en lenguaje natural) que se construye a partir del código fuente.
- Una lista de frases (en lenguaje natural) presentes también en el código.
- Una lista de palabras irrelevantes (stop-list).
- Diccionario de palabras.

La primer lista se confecciona de la siguiente manera, para cada método m dentro del código se crea una sub-lista de palabras que se extraen de los comentarios que están antes (comentarios JAVA Doc) o dentro del método m . También se incorporan palabras *hardwords* (camel-case o guión bajo) encontradas en los id ubicados en el código, por ejemplo: del id **easyCase**, se agregan las palabras **easy** y **case** al listado. Las palabras que se agreguen en este listado servirán como candidatas a expandir abreviaturas, por eso antes de agregar una palabra en este listado, se filtra utilizando la lista de palabras irrelevantes (esta lista se explica más adelante), para excluir palabras que no colaboran mucho con la expansión y no brindan mucha información.

La lista de frases también contiene una sub-lista por cada método y se construye con una técnica que extrae frases en lenguaje natural [32], el principal recurso son los comentarios y los ids multi-palabras. Este listado de frases se utiliza para expandir acrónimos¹, si un id en forma de acrónimo coincide con alguna frase, la misma es considerada como potencial expansión [42] (Ejemplo: la frase **file status** es una expansión posible para el id **fs**).

El listado de palabras irrelevantes y el diccionario, están predefinidos con antelación. El primero está compuesto con palabras relacionadas a artículos/-conectores (the, an, or), palabras reservadas del lenguaje de programación que se utilicen (**while**, **for**, **if**, etc.). Además posee palabras que no aportan

¹Abreviatura formada por las primeras letras de cada palabra en una frase. Ejemplo gif: Graphics Interchange Format.

información importante en la comprensión del código, y que son fácilmente reconocidas por los programadores (esta lista se usa con la misma política que el algoritmo Greedy). Por otro lado, las palabras del diccionario, contiene palabras en lenguaje natural, de diccionarios públicos (similar a utilizado por Greedy).

Una vez que las listas de palabras y frases potenciales se confeccionan, comienza la ejecución del algoritmo. La política que utiliza, es dar prioridad primero a la información extraída del código y más aún, a la cercanía de donde este ubicada la abreviatura a expandir, es decir, se busca primero en la sub-lista correspondiente al método donde se encuentra la abreviatura, luego en el resto de las listas con información extraída del código y por último en el diccionario público.

Este algoritmo (Ver Algoritmo 4) recibe como entrada la abreviatura a expandir y las 4 listas antes descriptas. El primer paso, es ver si la abreviatura forma parte de la lista de palabras irrelevantes (stop-list línea 1 y 2, algoritmo 4), en caso afirmativo finaliza la ejecución, ya que no es importante expandir, en caso negativo continúa la ejecución. En caso de continuar, se chequean si alguna de las frases extraídas del código se correspondan con la abreviatura en forma de acrónimo¹ (línea 5, algoritmo 4), dando prioridad aquellas que se encuentren cercanas al método (a través de la sub-lista correspondiente). Si no hubo resultados, la búsqueda prosigue en la lista de palabras recolectadas del código, en este caso las letras de la abreviatura deben coincidir en el mismo orden que las letras de una palabra en la lista (línea 8, algoritmo 4), si esto se cumple, la palabra elegida es una candidata, ejemplos: **horiz** \Rightarrow **HORIZ**ontal, **trg** \Rightarrow **TR**ianGle. Aquí se emplea la misma política de expansión que la búsqueda anterior, dando la preferencia a las palabras que están cerca del método (a través de la sub-lista correspondiente).

Finalmente, en caso no tener éxito, con las palabras extraídas del código, la búsqueda continúa en el diccionario predefinido como último recurso (línea 10, algoritmo 4). Al igual que la búsqueda anterior, una palabra candidata dentro del diccionario es aquella que contenga todas las letras en el mismo

¹Las letras de la abreviatura coincidan con las primeras letras de cada palabra en la frase. Ejemplo: fs \Rightarrow file system.

Algoritmo 4: Expansión Básica

Entrada: *abrev* // Abreviatura a expandir
 Entrada: *listaPalabras* // Palabras extraídas del código
 Entrada: *listaFrases* // Frases extraídas del código
 Entrada: *stopList* // Palabras Excluyentes
 Entrada: *dicc* // Diccionario en Inglés
 Salida : *únicaExpansión* // Abreviatura expandida, o null

```

1 si (abrev pertenece stopList) entonces
2   └─ devolver null
3 listaExpansión ← [ ]

   // Buscar coincidencia de acrónimo.
4 para todo (Frase | Frase es una frase en listaFrases) hacer
5   └─ si (abrev es un acrónimo de Frase) entonces
6     └─ // Se prioriza aquella Frase que está en el mismo
        └─ método que abrev
        └─ devolver Frase

   // Buscar abreviatura común.
7 para todo (Pal | Pal es una palabra en listaPalabras) hacer
8   └─ si (abrev es una abreviatura de Pal) entonces
9     └─ // Se prioriza aquella Pal que está en el mismo
        └─ método que abrev
        └─ devolver Pal

   // Si no hay éxito, buscar en el diccionario.
10 listaCandidatos ← buscarDiccionario(abrev, dicc)
    listaExpansión.add(listaCandidatos)
11 únicaExpansión ← null

   // Debe haber un solo resultado, sino no retorna nada.
12 si (length(listaExpansión) = 1) entonces
13   └─ únicaExpansión ← listaExpansión[0]
14 devolver únicaExpansión

```

orden que la abreviatura a expandir, ejemplo: `rctgl` \Rightarrow ReCTanGLE. Dado que el diccionario posee muchas palabras, puede retornar más de un resultado posible.

La técnica de expansión descrita en esta sección, devuelve una única expansión potencial para una abreviatura determinada y en caso contrario no retorna un resultado (líneas 11 - 14, algoritmo 4). El motivo de esto, es porque no tiene programado como decidir una única opción ante múltiples alternativas de expansión. A esta característica de mejora, los autores lo presentan como trabajo futuro [46, 38].

3.4.5. Algoritmo de Expansión AMAP

El algoritmo de expansión de abreviaturas que construyó Emily Hill, Zachary Fry, Haley Boyd [38] conocido como *Automatically Mining Abbreviation Expansions in Programs* (AMAP), además de buscar expansiones potenciales al igual que el algoritmo anterior, también se encarga de seleccionar la expansión que mejor se ajusta en caso de que haya más de un resultado posible. Otra mejora destacable, con respecto al algoritmo previo es que no se necesita un diccionario con palabras en lenguaje natural. Este tipo de diccionarios incluyen demasiadas palabras e implica disponer de un gran almacenamiento.

Las fuentes de palabras que se utilizan son:

- Una lista de abreviaturas comunes: Estas abreviaturas se obtienen automáticamente desde distintos programas. También se puede incorporar palabras en forma personalizada.
- Una lista palabras irrelevantes (stop-list).
- Una lista de contracciones¹ más comunes.

Para agilizar la lectura se asigna el nombre de “palabras largas” a las palabras normales que no están abreviadas y son potenciales expansiones de las abreviadas.

¹Palabras en inglés que llevan apostrofes, ejemplo: let’s.

La técnica automatizada AMAP, busca palabras largas candidatas para una palabra abreviada dentro del código, con la misma filosofía que se usa en la construcción de una tabla de símbolos en un compilador.

Se comienza con el alcance estático más cercano donde se examinan sentencias vecinas a la palabra abreviada. Luego gradualmente el alcance estático crece para incluir métodos, comentarios de métodos, y los comentarios de la clase. Si la técnica no encuentra una palabra larga adecuada para una determinada palabra abreviada, la búsqueda continúa mirando todo el programa y finalmente examina las librerías de JAVA SE 1.5.

Los autores asumen que una palabra abreviada está asociada a una sola palabra larga dentro de un método. No es frecuente que dentro de un método una palabra abreviada posea más de una expansión posible. En caso de que esto se cumpla, se puede cambiar la asunción. Se puede estipular que una palabra abreviada solo tiene una sola expansión posible dentro de los bloques o, en un contexto más reducido, solo dentro de las sentencias de código.

El algoritmo AMAP ejecuta los siguientes pasos:

1. Buscar palabras largas candidatas dentro de un método.
2. Elegir la mejor alternativa de expansión.
3. Buscar nuevas palabras si en el alcance local no es suficiente, utilizando el método EMF (Expansión más Frecuente).

A continuación, se explican cada uno de estos pasos.

Comenzando por el paso 1, la búsqueda de las palabras largas contiene dos algoritmos, uno que recibe como entrada palabras abreviadas compuestas por una sola palabra (singulares) y el otro algoritmo se encarga de procesar multi-palabras.

Búsqueda por Palabras Singulares

El primer paso para buscar palabras largas consiste en construir una expresión regular con un patrón de búsqueda. Este patrón se encarga de

seleccionar las palabras largas que coincidan con las letras de la palabra abreviada.

Los patrones se construyen a partir de la palabra abreviada, a continuación se detalla como se arman estos patrones:

Patrón prefijo: Se construye colocando la palabra abreviada (***pa***) seguida de la expresión regular $[a-z]^+{}^1$. Las palabras que coinciden si o si deberán comenzar con ***pa***. La expresión regular queda: ***pa*** $[a-z]^+$.

Ejemplo: Dada ***pa*** = ***arg***, el patrón ***arg*** $[a-z]^+$ coincide con la palabra ***argument*** (entre otras).

Patrón compuesto por letras: La expresión regular se construye insertando $[a-z]^+{}^1$ después de cada letra de la palabra abreviada (***pa***). Si ***pa*** = $c_1, c_2, c_3, \dots, c_n$, donde n es la longitud de la palabra abreviada. El patrón queda: $c_1[a-z]^+c_2[a-z]^+c_3[a-z]^+\dots c_n$.

Ejemplo: Dada ***pa*** = ***pgm***, el patrón ***p*** $[a-z]^+g[a-z]^+m[a-z]^+$ coincide (entre otras) con ***program*** (entre otras).

La búsqueda de palabras singulares se presenta en el algoritmo 5. Los parámetros de entrada son: la palabra abreviada a expandir y la expresión regular formada por el patrón elegido.

En la línea 1 (del algoritmo 5), se impide básicamente dos cosas:

a) Que no se procesen palabras abreviadas con muchas vocales consecutivas (segundo argumento del \wedge en el **si**). Este control lo determinaron los autores de AMAP [38], ya que comprobaron que la mayoría de las palabras abreviadas con vocales consecutivas se expanden como multi-palabras (ejemplos: es el caso de los acrónimos ***gui*** \Rightarrow ***graphical user interface***, ***ioe*** \Rightarrow ***invalid object exception***). El algoritmo de la próxima sección es el encargado de expandirlos (multi-palabras).

b) En caso de que el patrón sea el *compuesto por letras* (no sea el prefijo), se hacen dos controles más (primer argumento del \wedge en el **si**). El primero verifica que la abreviatura no posea muchas vocales consecutivas (“ $[^aeiou]^+$ ”

¹La expresión $[a-z]^+$ significa, una o más letras entre la a y la z.

¹La expresión $[a-z]^*$ significa, cero o más letras entre la a y la z.

logra eso) y el segundo controla que la longitud sea mayor que 3. Los autores a través del análisis de datos determinaron esta restricción [38], ya que el *patrón compuesto por letras* tiene el inconveniente que es muy flexible y tiende a capturar muchas palabras largas incorrectas. Por ejemplo: *lang* \Rightarrow (*loading*, *language*), o *br* \Rightarrow (*bar*, *barrier*, *brown*), entre otros.

En las líneas 2-10 se describe el proceso de búsqueda. Si alguna de estas sentencias de búsqueda encuentra una única palabra larga candidata, el algoritmo finaliza y retorna el resultado.

En la línea 2 la búsqueda se realiza en los comentarios Java Doc, donde la expresión regular es “@param *pa patrón*”. Por ejemplo, si en Java Doc se tiene el comentario “@param ind index” donde *pa* = *ind*, *patrón* = “ind[a-z]+”. La expresión regular “@param ind ind[a-z]+” coincidirá y devolverá el resultado “index” como expansión de *ind*.

Si el algoritmo no encuentra resultados, sigue la búsqueda en la línea 3 con los nombres de los tipos ubicados en las variables declaradas, donde la expresión regular es “*patrón pa*”. Por ejemplo, si se tiene una declaración “**component comp**” donde *pa* = **comp**, *patrón* = “comp[a-z]+” la expresión regular “comp[a-z]+ comp” coincidirá y devolverá el resultado “component” como expansión de **comp**.

Si el algoritmo continúa sin resultados, sigue en la línea 4 donde se busca coincidir con “*patrón*” en el nombre del método. En caso de seguir sin resultados, prosigue en la línea 5 con distintas variantes “*patrón pa*” o “*pa patrón*” en las sentencias comunes del método.

Si la ejecución continúa sin encontrar resultados, la línea 6 se restringe una búsqueda por palabras que tengan al menos 3 caracteres ya que generalmente aquellas con 2 tienden a ser multi-palabras (Ejemplo: *fl* \Rightarrow *file system* / Ver próxima sección). Luego en la línea 7 se busca con *patrón* solamente en palabras del método (ejemplo: para una abreviatura *setHor* coincide con una llamada a función con el nombre de *setHorizontal()*). Después en la línea 8 se busca en palabras de comentarios dentro del método con *patrón*.

Para finalizar, en la línea 10 si la palabra abreviada tiene más de un caracter y el patrón es de tipo prefijo, se busca usando (*patrón*) en los comentarios de la clase. En la línea 9 se restringe esta búsqueda, porque los

Algoritmo 5: Búsqueda por Palabras Singulares

Entrada: *pa* // Palabra Abreviada
Entrada: *patrón* // Expresión regular
Salida : *palCand* // Palabras candidatas, o null si no hay
 // Las expresiones regulares están entre comillas

- 1 **si** (*patrón* prefijo \vee *pa* coincide “[a-z][^aeiou]+” \vee length(*pa*) > 3)
 \wedge (*pa* no coincide con “[a-z][aeiou][aeiou]+”) **entonces**
 - // Si alguna de las siguientes búsquedas encuentra un
 único resultado, el algoritmo lo retorna
 finalizando la ejecución
- 2 Buscar en Comentarios JavaDoc con “@param *pa patrón*”, si el
 resultado es único **devolver** en *palCand*
- 3 Buscar en Nombres de Tipos y la correspondiente Variable
 declarada con “*patrón pa*”, si el resultado es único **devolver** en
palCand
- 4 Buscar en el Nombre del Método con “*patrón*”, si el resultado es
 único **devolver** en *palCand*
- 5 Buscar en las Sentencias con “*patrón pa*” y “*pa patrón*”, si el
 resultado es único **devolver** en *palCand*
- 6 **si** (length(*pa*) \neq 2) **entonces**
 - 7 Buscar en palabras del Método con “*patrón*”, si el resultado
 es único **devolver** en *palCand*
 - 8 Buscar en palabras que están en los Comentarios del Método
 con “*patrón*”, si el resultado es único **devolver** en *palCand*
- 9 **si** (length(*pa*) > 1) \wedge (*patrón* prefijo) **entonces**
 - // Solo se busca con patrones prefijos
- 10 Buscar en palabras que están en los Comentarios de la Clase
 con “*patrón*”, si el resultado es único **devolver** en *palCand*

autores sostienen [38], que buscar con un solo caracter en comentarios implica tener muchos resultados y más aun si el patrón es el compuesto por letras.

Búsqueda por Multi-Palabras

El algoritmo de búsqueda por multi-palabras a diferencia del explicado anteriormente, expande abreviaturas que contienen dos o más palabras. Algunos ejemplos son: `gui` \Rightarrow `graphical user interface`, `fs` \Rightarrow `file system`. Como bien se definió en secciones anteriores estas abreviaturas se las conoce con el nombre de acrónimos, que generalmente están conformadas por 2 ó 3 caracteres. El algoritmo anterior intenta detectar este tipo de abreviaturas y no analizarlas para que sea procesado por el multi-palabras.

Al igual que el algoritmo de palabras singulares, el algoritmo de multi-palabras utiliza expresiones regulares conformada por patrones de búsqueda. Los patrones utilizados en las búsquedas multi-palabras se construyen de la siguiente manera:

Patrón acrónimo: Se elabora colocando la expresión regular $[a-z][]^+{}^1$, después de cada letra de la palabra abreviada (***pa***). Si ***pa*** = $c_1, c_2, c_3, \dots, c_n$, donde n es la longitud de la palabra abreviada. El patrón queda: $c_1[a-z][]^+c_2[a-z][]^+c_3[a-z][]^+ \dots [a-z][]^+c_n$. Permite encontrar acrónimos tales como `pdf` \Rightarrow ***p***ortable ***d***ocument ***f***ormat.

Patrón de Combinación de Palabras: En este caso el patrón se construye de manera similar al anterior pero se usa la expresión regular $[a-z]^*[]^*{}^2$ después de cada caracter de la palabra abreviada (***pa***). Si ***pa*** = $c_1, c_2, c_3, \dots, c_n$, donde n es la longitud de la palabra abreviada. El patrón queda: $c_1[a-z]^*[]^*c_2[a-z]^*[]^*c_3[a-z]^*[]^* \dots [a-z]^*[]^*c_n$. De esta manera se pueden capturar palabras del tipo `arg` \Rightarrow ***a***ccess ***r***ights, permitiendo más capturas que el patrón anterior.

En el algoritmo 6, se presenta la búsqueda por multi-palabras [38]. Las variables de entrada son: la abreviatura multi-palabra a expandir y la expresión regular formada por el patrón elegido.

¹La expresión $[a-z][]^+$ significa, una o más letras entre la a y la z, seguido de espacio.

²La expresión $[a-z]^*[]^*$ significa, cero o más letras entre la a y la z, seguido de espacio o no.

Algoritmo 6: Búsqueda por Multi Palabras

```

Entrada: pa // Palabra Abreviada
Entrada: patrón // Expresión regular
Salida : palCand // Palabras candidatas, o null si no hay
           // Las expresiones regulares están entre comillas
1 si (patrón acrónimo  $\vee$   $\text{length}(pa) > 3$ ) entonces
    // Si alguna de las siguientes búsquedas encuentra un
    // único resultado, el algoritmo lo retorna
    // finalizando la ejecución
2   Buscar en Comentarios JavaDoc con “@param pa patrón”, si el
    resultado es único devolver en palCand
3   Buscar en Nombres de Tipos y la correspondiente Variable
    declarada con “patrón pa”, si el resultado es único devolver en
    palCand
4   Buscar en el Nombre del Método con “patrón”, si el resultado es
    único devolver en palCand
5   Buscar en todos los ids (y sus tipos) dentro del Método con
    “patrón”, si el resultado es único devolver en palCand
6   Buscar en Literales String con “patrón”, si el resultado es único
    devolver en palCand
    // En este punto se buscó en todos los lugares
    // posibles dentro del método
7   Buscar en palabras que están en los Comentarios del Método con
    “patrón”, si el resultado es único devolver en palCand
8   si (patrón acrónimo) entonces
    // Solo se busca con patrones Acrónimos
9   Buscar en palabras que están en los Comentarios de la Clase
    con “patrón”, si el resultado es único devolver en palCand

```

Los patrones de *combinación de palabras* son menos restrictivos que los patrones de *acrónimos* y frecuentemente conllevan a malas expansiones. En caso que no sea acrónimo, la búsqueda se restringe a palabras abreviadas ingresadas con longitud 4 ó mayor (línea 1, algoritmo 6). Esto genera la sensación de que se pierden casos de 2 ó 3 caracteres pero estudios indican

<pre> /** * Copies characters from this string into the destination character * array. * * @param srcBegin index of the first character in the string * to copy. * @param srcEnd index after the last character in the string * to copy. * @param dst the destination array. * @param dstBegin the start offset in the destination array. * @exception NullPointerException if <code>dst</code> is <code>>null</code> */ public abstract void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin); </pre>	Comentarios JAVA Doc
<pre> private void circulationPump(ControlFlowGraph cfg, InstructionContext start, final Random random = new Random(); InstructionContextQueue icq = new InstructionContextQueue(); Object source = event.getSource(); if (source instanceof Component) { Component comp = (Component)source; comp.dispatchEvent(event); } else if (source instanceof MenuComponent) { </pre>	Nombres de los Tipos
<pre> public void setBarcodeImg(int type, String text){ StringBuffer bcCall = new StringBuffer("it.businesslogic //boolean isFormula = text.trim().startsWith("\$"); bcCall.append(type); </pre>	Nombre del Método
<pre> final int nConstructors = constructors.size(); final int nArgs = _arguments.size(); final Vector argsType = typeCheckArgs(stable); </pre>	Sentencias

Figura 3.5: Ejemplos de distintas Búsquedas dentro del Código

que son la minoría [38].

Al igual que el algoritmo anterior en las líneas 2-4 se realiza la búsqueda primero en comentarios JAVA Doc, luego en nombres de tipos, después en el nombre del método. La Figura 3.5 muestra algunos ejemplos antedichos.

Dado que las expresiones regulares son más complejas en este algoritmo, los tiempos de respuestas son más elevados. Por esta razón, la búsqueda en sentencias no se realiza, a diferencia del algoritmo de palabras singulares.

En las siguientes líneas 5-7 se examinan los ids (incluyendo declaraciones), palabras de literales strings y palabras de comentarios del método. En los tres casos solo se utiliza “*patrón*”.

Luego en la línea 9 se busca en comentarios de la clase con el *patrón acrónimo*. Cabe aclarar que *patrón de combinación de palabras* en este caso

no se usa (línea 8) ya que puede tomar palabras largas incorrectas.

Finalmente después de observar cientos de casos de palabras largas, los autores [38] concluyen que el mejor orden de ejecución de las técnicas de búsqueda es ejecutar los patrones: acrónimo (multi-palabra), prefijo (una sola palabra), compuesto por letras (una sola palabra), combinación de palabras (multi-palabra).

Si ninguna de las estrategias de expansión funciona en el ámbito local dentro de un método, se procede a buscar la palabra abreviada en un listado de contracciones (inglés).

En caso de seguir sin éxito, se recurre a la técnica conocida como expansión más frecuente (EMF). Antes de explicar EMF, esta pendiente describir la forma en que AMAP decide ante varias alternativas de expansión.

Decidir entre Múltiples Alternativas

Existe la posibilidad de que una abreviatura posea múltiples alternativas potenciales de expansión, dentro del mismo alcance estático. Por ejemplo, el patrón prefijo para **val** puede coincidir **value** o **valid**. La técnica de elección entre múltiples candidatos procede de la siguiente manera:

1. Se elije la palabra larga dentro del alcance estático con mayor frecuencia de aparición. Tomando el ejemplo anterior para **val** si **value** aparece 3 veces y **valid** una sola vez, se elije la primera.
2. En caso de haber paridad en el ítem 1, se agrupan las palabras largas con similares características. Por ejemplo, si **def** coincide con **defaults**, **default** y **define** donde todas aparecen 2 veces, en este caso se agrupa las dos primeras en solo **default**, sumando la cantidad total a 4 predominando sobre **define** con 2.
3. En caso de que la igualdad persista, se acumulan las frecuencias de aparición entre las distintas búsquedas para determinar un solo candidato. Por ejemplo, si el id **fs** coincide con **file system** y **file socket** ambas con una sola aparición en los comentarios de JAVA Doc. Para llegar a una decisión, primero se almacenan ambas opciones. Después, prosigue con

el resto de las búsquedas (nombres de tipos de ids, literales, comentarios) en cuanto aparezca una de las dos, por ejemplo `file socket` este termina prevaleciendo sobre `file system`.

4. Finalmente si todas las anteriores fallan se recurre al método de expansión más frecuente (EMF).

Expansión Más Frecuente (EMF)

La estrategia EMF [38] es una técnica que se utiliza en 2 casos. Por un lado, encuentra una expansión cuando todas las búsquedas fracasan y por el otro, ayuda a decidir entre varias alternativas de expansión.

La idea consiste en ejecutar la misma estrategia local de expansión de abreviaturas explicada anteriormente, pero sobre el programa entero. Para cada palabra abreviada, se cuenta el número de veces que esa palabra se le asigna una palabra larga candidata. Luego, se calcula la frecuencia relativa de una palabra abreviada con respecto a cada palabra larga encontrada. La palabra larga con mayor frecuencia relativa se considera la expansión más frecuente. Al final del proceso se agrupan las palabras largas potenciales en un listado de EMF. Sin embargo, suele pasar que la expansión más probable es la incorrecta. Para evitar que esto suceda, una palabra larga debe, a su vez, superar la frecuencia relativa más de la mitad (0.5). Inclusive, la palabra abreviada debe tener al menos 3 asignaciones de palabras largas candidatas en todo el programa.

La técnica EMF tiene dos niveles de análisis, el primero es a nivel de programa y el otro más general a nivel JAVA. El nivel de programa es ideal ya que expande las abreviaturas con palabras propias del dominio del problema.

Abreviatura	Palabra Expandida	Frecuencia Relativa
int	integer	0.821
impl	implement	0.840
obj	object	1.000
pos	position	0.828

Tabla 3.2: Algunas Frecuencias Relativas de Ids en JAVA 5

El nivel más general se arma con la API¹ de JAVA. En la tabla 3.2 se muestra algunos casos de frecuencias relativas más altas de JAVA 5. En caso de que una palabra abreviada no obtenga un candidato de expansión, EMF también puede entrenarse sobre muchos programas JAVA para mejorar la precisión. A su vez, existe la posibilidad de armar una lista a mano para casos puntuales de expansión que no son de frecuente aparición. Otras soluciones propuestas son entrenar sobre documentación online relacionada a JAVA o documentación vinculada a la ciencias de computación. El algoritmo de expansión de abreviaturas AMAP es totalmente automático y se implementó como una extensión de Eclipse.

Hasta ahora se han descripto algoritmos y técnicas que recientemente se pensaron y elaboraron. En la próxima sección se presenta una herramienta que fue construida en los comienzos de los estudios basados en ids. Esta herramienta es tomada como objeto de estudio por varios autores de las técnicas antes mencionadas [38, 48, 44, 46].

¹Interfaz de programación de aplicaciones, por su siglas en Inglés.

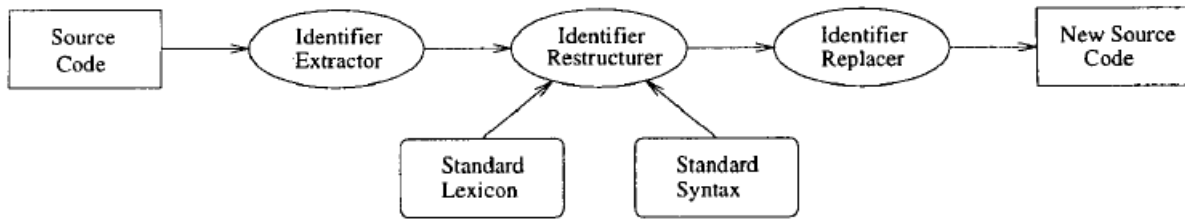


Figura 3.6: Arquitectura de Identifier Restructuring

3.4.6. Herramienta: Identifier Restructuring

La herramienta Identifier Restructuring [23] se encarga de recibir como entrada un código fuente escrito en lenguaje C. Luego a través de un proceso de transformación cada id del código, se expande a palabras completas. La salida es el mismo código pero con los ids expandidos. Cabe destacar que esta herramienta es semi-automática, en algunas situaciones necesita intervenir el usuario.

Los ids se cambian por nombres más explicativos, los cuales incluyen un verbo que indica la función del id en el código. Más precisamente después de renombrar los ids, se visualiza claramente el rol que cumple el id en el programa.

El código fuente se convierte de esta manera en un código más claro y mejora la comprensión. El proceso consta de tres etapas (Ver Figura 3.6 ¹):

1. **Identifier Extractor:** Recupera una lista con los nombres de los ids presentes en el código. Este módulo se programó con un parser de lenguaje C, que fue modificado para que reconozca los ids y los extraiga.
2. **Identifier Restructurer:** Genera una asociación entre el nombre actual del id y un nuevo nombre estándar expandido. El primer paso consiste en segmentar el id en las palabras que lo constituyen. Después, cada palabra se expande usando un diccionario de palabras estándar (estándar léxico). Finalmente, la secuencia de palabras expandidas deben coincidir con reglas predefinidas por una gramática para determinar

¹Todas las figuras de esta sección se obtuvieron del artículo que describe la herramienta Identifier Restructuring [23].

que acción cumple el id en el código (estándar sintáctico).

3. **Identifier Replacer:** Transforma el código original en el nuevo código usando las asociaciones que se construyeron en la etapa anterior. Se emplea un scanner léxico para evitar reemplazar posibles nombres de ids contenidos en literales strings y en comentarios.

Los pasos 1 y 3 están totalmente automatizados. Sin embargo, para lograr que la expansión de nombres sea efectiva, se necesita que en algunos casos del paso 2 intervenga el usuario.

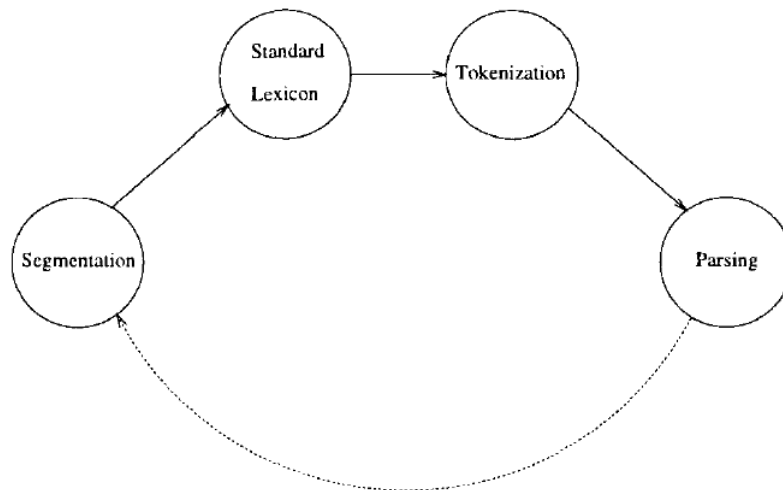


Figura 3.7: Etapas de Identifier Restructurer

A continuación, se detalla el paso 2 que es el más importante de esta herramienta, en la Figura 3.7 se desglosa las diferentes etapas.

Segmentation: El id se separa en las palabras que lo componen. De manera automática se utilizan estrategias simples de separación (basada en guión bajo o camel-case: hardword - ejemplo: `get_txt` \Rightarrow `get txt`). En caso que existan softwords, la división se debe hacer en forma manual. Por ejemplo: `get_txtinput` \Rightarrow `get txt input` la separación entre `txt` e `input` la realiza el usuario. Sin embargo, los autores proponen de manera conceptual (no implementado), utilizar un algoritmo programado (por los mismos autores) en LISP, para detectar y separar casos de softwords [22].

FunctionId	::=	[Context] (Action PropertyCheck Transformation)	
Context	::=	Qualifier <noun>	
Qualifier	::=	(<adjective> <noun>)*	
Action	::=	SimpleAction ComplexAction	
SimpleAction	::=	DirectAction IndirectAction	
ComplexAction	::=	ActionOnObject DoubleAction	
IndirectAction	::=	Qualifier <noun> ActionSpecifier	{Head word = <noun>}
DirectAction	::=	<verb> ActionSpecifier	{Head word = <verb>}
ActionOnObject	::=	<verb> Qualifier <noun> ActionSpecifier	{Head words = <verb>, <noun>}
DoubleAction	::=	(DirectAction ActionOnObject) ² {Head words from DirectAction and/or ActionOnObject}	
ActionSpecifier	::=	(<adjective> <adverb> <preposition> Qualifier <noun>)*	
PropertyCheck	::=	"is" Qualifier (<adjective> <noun>) ActionSpecifier	{Head word = <adjective> <noun>}
Transformation	::=	Source TransformOp Target	{Head words from Source and Target}
Source	::=	Qualifier (<adjective> <noun>)	{Head word = <adjective> <noun>}
Target	::=	Qualifier (<adjective> <noun>)	{Head word = <adjective> <noun>}
TransformOp	::=	"to" "2"	

Figura 3.8: Gramática que determina la Función de los Ids

Standard Lexicon: Lograda la separación de las palabras estas son mapeadas a una forma estándar (expandidas) con la ayuda de un diccionario léxico [22] (Ejemplo: `upd` \Rightarrow `Update`). Una idea que mejora esta propuesta, es incorporar al diccionario términos extraídos del código fuente. También aquí, el usuario puede intervenir para realizar la expansión manualmente. Los autores de la herramienta construyeron los diccionarios de manera genérica tomando como muestra 10 programas [23]. Sin embargo, se aconseja que con el tiempo los diccionarios deben crecer con la inclusión de nuevos términos.

Tokenization: Una vez obtenidas las palabras a una forma estándar (expandidas) en el paso anterior, se procede a asignar cada palabra a un *tipo léxico* (verbo, sustantivo, adjetivo). Por ejemplo, la palabra `Update` se transforma en `<Update,verb>`, `Standard` a `<Standard,adjective>`. Esta tuplas se denominan tokens y se utiliza un ‘diccionario de tipos’ para generarlos de manera automática, este diccionario al igual que los otros se arma previamente a gusto del programador [22]. Sin embargo, existen casos que se necesita la intervención humana para determinar el tipo correcto. Por ejemplo, `free` en inglés es un verbo, un adjetivo y a la vez un adverbio.

Parsing: Finalmente, la secuencia de tokens obtenidos en la etapa anterior se analiza usando una gramática predefinida. Este análisis permite determinar cuál es el rol/acción del id en el código fuente y de esta manera, se determina la “acción semántica” del id. En la Figura 3.8 se muestra un ejemplo de gramática construida por los autores. Cabe aclarar que cada usuario puede elaborar su propia gramática. Es una gramática regular donde los símbolos terminales están indicados con $\langle \rangle$. Las producciones con negrita, determinan en función del tipo léxico asignado a cada palabra la acción semántica del id. Por ejemplo, el verbo expresa la acción y el sustantivo representa al objeto de la acción, con **ActionOnObject** $\Rightarrow \langle \text{verb} \rangle, \langle \text{noun} \rangle \equiv \langle \text{go}, \text{home} \rangle$. Otro ejemplo es, **IndirectAction** $\Rightarrow \langle \text{adjective} \rangle, \langle \text{noun} \rangle \equiv \langle \text{order}, \text{textfield} \rangle$ donde el adjetivo representa una cualidad del sustantivo.

En caso de que el análisis falle el proceso se reinicia desde el comienzo partiendo nuevamente de la etapa de segmentación [23] (Ver Figura 3.7).

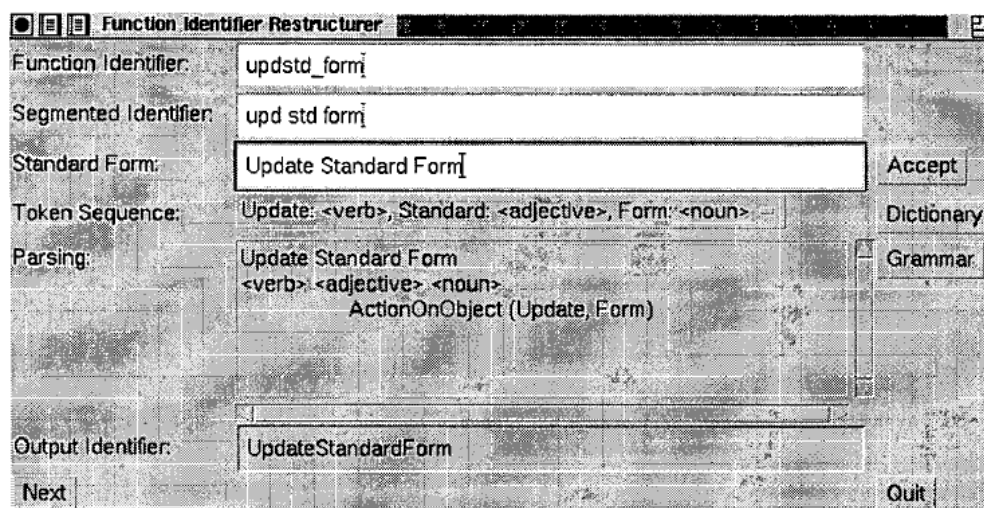


Figura 3.9: Visualización de Identifier Restructuring

Interfaz para el Usuario

La interfaz para el usuario de **Identifier Restructurer** se visualiza en la Figura 3.9, el id de entrada se muestra en el primer cuadro de texto `updstd_form`. Se usan heurísticas sencillas (guión bajo, camel-case) para separar las palabras del id, en este caso `updstd` y `form`. Como esta segmentación está incompleta, el usuario puede separar manualmente en el segundo cuadro de texto la palabra `upd` y `std` (Ver Figura 3.9). En el tercer cuadro de texto se propone la forma estándar de cada palabra. Cuando una palabra no se puede expandir la herramienta muestra un signo de pregunta en su lugar (?). En este caso `upd` \Rightarrow `Update`, `std` \Rightarrow (?), `form` \Rightarrow `Form`, como `std` no está presente en el diccionario, se necesita la intervención del desarrollador para que se complete correctamente a `Standard`. Luego las palabras expandidas son asociadas a la función gramatical. En esta etapa puede existir para una secuencia de palabras más de una función gramatical (la gramática es ambigua y puede generar más de una secuencia de tokens). En caso de suceder esto el usuario puede elegir cual es la secuencia más adecuada. En el ejemplo de la Figura 3.9 solo existe una única función gramatical y es reflejada en el cuarto cuadro de texto.

Luego, en el cuadro de Parsing (Ver Figura 3.9) se puede apreciar la acción que aplica el id, en este caso **ActionOnObject(Update,Form)** ‘actualizar formulario’. Finalmente el resultado se detalla en el último cuadro de texto de más abajo.

Cuando se arma la asociación de los nombres ids con los nuevos nombres generados la misma debería cumplir con la propiedad de inyectividad, de esta forma se evita que haya conflictos de nombres entre los distintos ids del programa. La herramienta ayuda al programador a conseguir este objetivo resaltando los posibles conflictos en los nombres.

Para concluir con las etapas de esta herramienta, la última fase llamada **Identifier Replacer** toma todas las ocurrencias del id `updstd_form` y se reemplaza por `UpdateStandarForm`, como se mencionó con anterioridad.

3.5. Notas y Comentarios

Las observaciones que se destacan en el estado del arte de las técnicas de análisis de ids apuntan por un lado, que asignar buenos nombres a los ids ayudan a comprender el sistema. Al comienzo de este capítulo se explicó la herramienta Identifier Dictionary (IDD) que ayuda a lograr este cometido. Sin embargo, no trascendió ya que es costosa de utilizar sobre grandes proyectos de software y solo es efectiva cuando se emplea desde el arranque del desarrollo de un sistema.

La investigación realizada por los autores de IDD, dejaron en claro que los nombres de los ids es crucial para la comprensión de los sistemas, un código con ids más descriptivos y claros se entiende mucho mejor. Además, en este contexto, las herramientas/técnicas de análisis de ids mejoran sus resultados.

Con respecto a otras herramientas/técnicas de análisis de ids, las mismas han ido evolucionando con el pasar del tiempo. Al principio algunas etapas necesitaban la intervención del usuario para realizar las tareas, se puede decir que usaban procesos semi-automatizados (ejemplo: **Identifier Restructurer**). A medida que se construyeron nuevas técnicas, se buscó más la automatización haciendo que el usuario se involucre menos (ejemplo: Samurai, AMAP).

Habiendo explicado en este capítulo algunas de las estrategias asociadas al análisis de ids en códigos, en el próximo capítulo, se describe una herramienta de análisis de ids que se construyó tomando como referencia algunas de las características propias de las técnicas vistas en este capítulo.

Capítulo 4

Identifier Analyzer (IDA)

4.1. Introducción

En el capítulo anterior, se explicó la importancia de analizar identificadores (ids) ubicados en el código fuente de un sistema de software. Los ids de un programa normalmente están compuestos por más de una palabra en forma de abreviatura, por ejemplo: `inp_fl_sys`. Detrás de estas abreviaturas, los ids ocultan información es propia del Dominio del Problema [22, 46, 38, 29]. Desafortunadamente, las personas ajenas al código, no comprenden a simple vista la información que los ids poseen en sus abreviaturas e invierten tiempo en entenderlas. Es por esto, que las herramientas automáticas de análisis de ids son bienvenidas en el ámbito de la Comprensión de Programas (CP). Con estas herramientas se logra disminuir los tiempos de comprensión de ids y revelar la información que estos contienen en sus abreviaturas.

Dada la importancia que tienen las herramientas de análisis de ids, se tomó la iniciativa de desarrollar una llamada Identifier Analyzer (IDA) [9]. Esta herramienta le permite al usuario ingresar un archivo JAVA, luego IDA analiza los ids que están en el archivo. IDA lleva a cabo el análisis de ids en tres pasos, los cuales se mencionan a continuación: I) Extraer los ids del código de estudio. II) Aplicar una técnica de división, en donde se descomponen a los ids en las distintas abreviaturas que lo componen. Por ejemplo: `inp_fl_sys` \Rightarrow `inp fl sys`. III) Emplear una técnica de expansión de abreviaturas

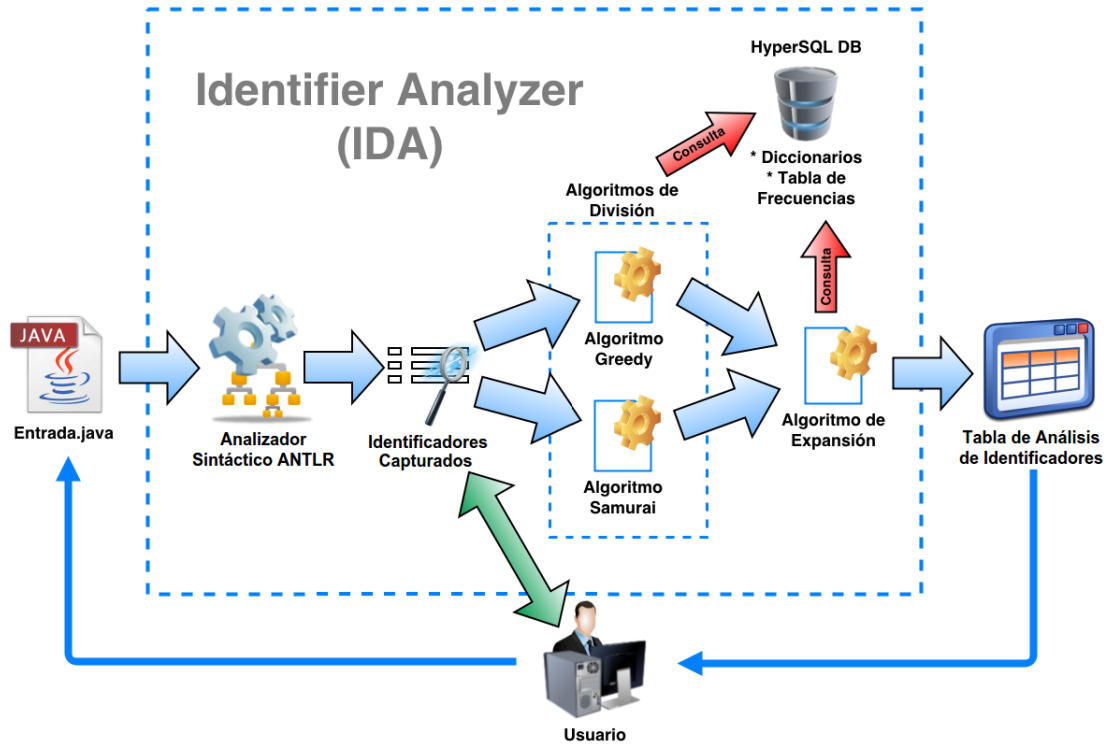


Figura 4.1: Arquitectura de IDA

que las expande a palabras completas. Por ejemplo: `inp fl sys` \Rightarrow `input file system`. Una vez realizado los tres pasos, los resultados de las expansiones de ids se exhiben en una tabla. El objetivo de IDA es lograr que el usuario comprenda más rápidamente el propósito de los ids en los archivos JAVA, y de esta manera mejorar la comprensión del código analizado.

El correspondiente capítulo, está destinado a explicar los distintos módulos que IDA posee, y que proceso de ejecución debe realizar el usuario para analizar los ids. Para comenzar con la descripción de IDA, en la siguiente sección se explica la arquitectura y cuales son sus componentes principales.

4.2. Arquitectura

En la Figura A.1 se puede apreciar la arquitectura de IDA. Esta arquitectura describe tres partes principales, la primera consiste en la *extracción de datos*, la segunda trata sobre la *división de ids* y la tercera sobre *expansión*

de *ids*. A continuación se detallan cada una de ellas.

Módulo de Extracción de Datos: Este módulo recibe como entrada un archivo JAVA que ingresa el usuario (Ver Figura A.1 Entrada.java), luego este archivo se procesa por un Analizador Sintáctico (AS) (Ver Figura A.1 Analizado Sintáctico ANTLR). El AS, extrae y almacena, en estructuras internas, la información estática perteneciente al código del archivo ingresado. Esta información, está relacionada con *ids*, literales y comentarios (Ver próxima sección para más detalles). El usuario a través de la interfaz de IDA, puede visualizar esta información capturada del código por medio de tablas claramente definidas (Ver Figura A.1 - Flecha Verde).

Módulo de División de Ids: Una vez completada la extracción de información, el proceso continua en el módulo de división de *ids*. Aquí se encuentran implementados dos algoritmos de división; uno es el Algoritmo Greedy y el otro es el Algoritmo Samurai ambos explicados en la sección 3.4.2 y 3.4.3 del capítulo anterior. Estos algoritmos reciben como entrada la información capturada en el módulo de extracción de datos (*ids*, comentarios, literales), y luego estos algoritmos dividen los *ids* del archivo JAVA (Ver Figura A.1 - Algoritmos de División). Los resultados de las divisiones se almacenan en estructuras internas que serán utilizadas por el módulo de expansión. Cabe recordar que estos algoritmos de división necesitan datos externos para funcionar, uno es el diccionario de palabras (en caso de Greedy) y el otro es lista de frecuencias globales de aparición de palabras (en caso de Samurai). Estos datos externos se encuentran almacenados en una base de datos embebida (Ver Figura A.1 - HyperSQL DB).

Módulo de Expansión de Ids: La tercera y última parte, tiene implementado el Algoritmo de Expansión Básico de abreviaturas que fue explicado en la sección 3.4.4 del capítulo anterior. Este algoritmo, toma como entrada los *ids* separados en el módulo de división de *ids* (tanto de Greedy como de Samurai), luego el Algoritmo de Expansión expande las abreviaturas resultantes producto de la división de *ids* (Ver Figura A.1 - Algoritmo de Expansión). Los resultados de las expansiones se retornan en dos grupos: las expansiones provistas por el Algoritmo de división Greedy y las provistas

por el Algoritmo de división Samurai.

El Algoritmo de Expansión también necesita de un diccionario de palabras, por eso se realizan consultas a la base de datos embebida (Ver Figura A.1 - HyperSQL DB). Finalmente, los resultados de las divisiones y las expansiones de los ids, se muestran en una tabla (Ver Figura A.1 - Tabla de análisis de identificadores).

4.3. Analizador Sintáctico

Como se explicó en la sección previa, cuando el usuario ingresa un archivo JAVA, IDA examina y extrae información estática presente en el archivo ingresado. Esta información está compuesta por identificadores, comentarios y literales. La manera en que IDA extrae esta información es a través de un Analizador Sintáctico (AS). La construcción de este AS se llevó a cabo, primero investigando herramientas encargadas de construir AS. Se dio preferencia a aquellas que emplean la teoría asociada a las gramáticas de atributos [1]. De la investigación previamente descrita, se determinó que la herramienta *ANTLR*¹ era la que mejor se ajustaba a las necesidades antes planteadas. Esta herramienta permite agregar acciones semánticas (escritas en JAVA) para el cálculo de los atributos, en una gramática de lenguaje JAVA². Estas acciones semánticas deben estar correctamente insertadas en la gramática para, por ejemplo, implementar estructuras de datos y algoritmos que capturan los ids utilizados en un programa [2]. Una vez insertadas estas acciones, ANTLR lee la gramática y genera el AS adicionando acciones que fueron programadas. De esta manera, se obtiene un AS que recolecta ids mientras examina el código. A su vez a estas acciones semánticas, se le agregan otras acciones que extraen comentarios y literales strings. Estos elementos son necesarios ya que sirven para los algoritmos de análisis de ids que serán explicados en próximas secciones.

¹ANother Tool for Language Recognition. <http://wwwantlr.org>

²<http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>

4.4. Base de Datos Embebida

Como se describió en secciones previas, IDA posee una base de datos embebida. Esta base de datos utiliza una tecnología llamada HSQLDB¹. Dado que HSQLDB esta desarrollada en JAVA, al momento de incorporarla en IDA (que está programada en JAVA) no resultó una tarea difícil. Otra ventaja por la cual se eligió esta tecnología, es que responde rápidamente las consultas. Esto es importante ya que todos los algoritmos de IDA consultan a HSQLDB. Dentro de esta base de datos embebida, se encuentran almacenadas los diccionarios/listas de palabras que IDA necesita para llevar adelante sus tareas. Estos diccionarios/listas se describen a continuación, nombrando también que algoritmo de IDA consulta cada diccionarios/listas.

Diccionario en Inglés (ispell): Contiene palabras en Inglés que pertenecen a la lista de Palabras Comando de Linux *Ispell*². Se utiliza en el Algoritmo de Greedy y en el Algoritmo de Expansión (Ver Capítulo 3).

Lista de Palabras Excluyentes (stop-list): Esta compuesta con palabras que son poco importantes o irrelevantes en el análisis de ids³. Se utiliza en el Algoritmo de Greedy y en el Algoritmo de Expansión (Ver Capítulo 3).

Lista de Abreviaturas y Acrónimos Conocidas: Contiene abreviaturas comunes del idioma Inglés y Acrónimos conocidos de programación⁴ (gif, jpg, txt). Se emplea en el Algoritmo Greedy (Ver Capítulo 3).

¹Hyper SQL Data Base. <http://www.hsqldb.org>

²<http://wordlist.aspell.net>

³<http://www.lextek.com/manuals/onix/stopwords1.html>

⁴<http://langs.eserver.org/acronym-dictionary.txt>

Lista de Prefijos y Sufijos Conocidos: Posee Sufijos y Prefijos conocidos en Inglés¹, esta lista fue confeccionada por el autor del Algoritmo Samurai (Ver Capítulo 3). Se consulta solo en dicho algoritmo.

Frecuencias Globales de Palabras: Lista de palabras, junto con su frecuencia de aparición. Esta lista fue construida por el autor del Algoritmo Samurai². Se emplea solo en dicho algoritmo, más precisamente en la función de *Scoring* (Ver Capítulo 3).

Cabe destacar que las listas y diccionarios que fueron descriptos poseen palabras que pertenecen al idioma Inglés, dado que los autores así lo determinaron. Por lo tanto, para que la herramienta IDA analice correctamente los ids, se deben ingresar en IDA archivos JAVA con comentarios, literales e ids acordes a la lengua Inglesa.

Habiendo descripto los principales módulos de la herramienta, en la próxima sección se explicará el proceso que debe seguir el usuario para analizar ids a través de IDA.

4.5. Proceso de Análisis de Identificadores

En esta sección, se describe el proceso que debe seguir el usuario con la herramienta IDA para llevar a cabo el análisis de los ids, en los archivos JAVA. Se explicará que función cumple cada elemento de IDA (ventanas, botones, paneles, etc.), y de como estos elementos ayudan al usuario a analizar los ids.

4.5.1. Barra de Menú

Al ejecutar la herramienta IDA, el primer componente de interacción es una simple barra de menú ubicada en el tope de la pantalla, los botones de esta barra son *Archivo*, *Diccionarios* y *Ayuda* (Ver Figura 4.2).

¹<http://www.eecis.udel.edu/~enslen/Site/Samurai>

²Esta lista no está disponible en la web, por ende se construyó una aproximación.

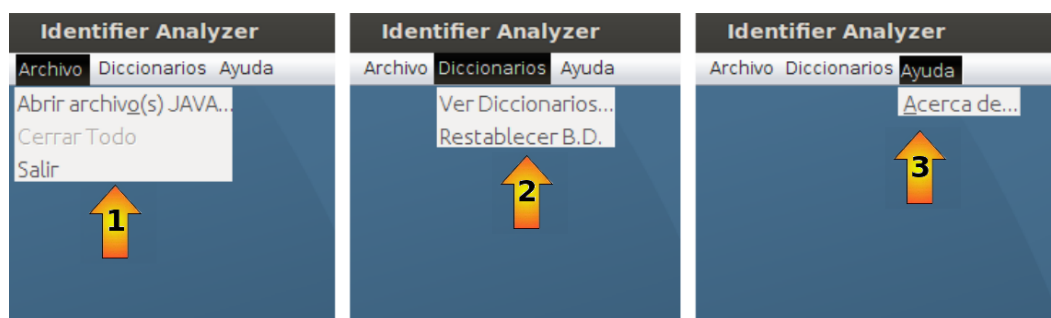


Figura 4.2: Barra de Menú de IDA

Al pulsar¹ en *Archivo* de la barra antedicha, se despliega un menú con las opciones que se describen a continuación (Ver Figura 4.2 - Flecha 1):

Abrir archivo(s) JAVA: Abre una ventana que permite elegir uno o varios archivos con extensión JAVA (Ver Figura 4.3). Los archivos seleccionados serán analizados por IDA (en la próxima sección, serán dados más detalles).

Cerrar Todo: Cierra todos los archivos JAVA abiertos actualmente en la aplicación.

Salir: Cierra la Aplicación.

Cuando se pulsa en *Diccionarios* de la barra de menú, se despliega otro menú con las siguientes opciones (Ver Figura 4.2 - Flecha 2):

Ver Diccionarios: Abre una ventana, que muestra un listado de palabras en Inglés correspondiente al diccionario *ispell* (explicado en la sección anterior). La ventana antedicha, también muestra un listado de palabras irrelevantes o stoplist (explicado en la sección anterior). Esta ventana, se explica con más detalles en la sección 4.5.5.

Restablecer B.D.(Base de Datos): Genera nuevamente la base de datos HSQldb. En caso de haber problemas con la base de datos, es útil restablecerla.

¹El término ‘pulsar’ o ‘presionar’ se utilizará a lo largo del capítulo, significa hacer click con el puntero del ratón.

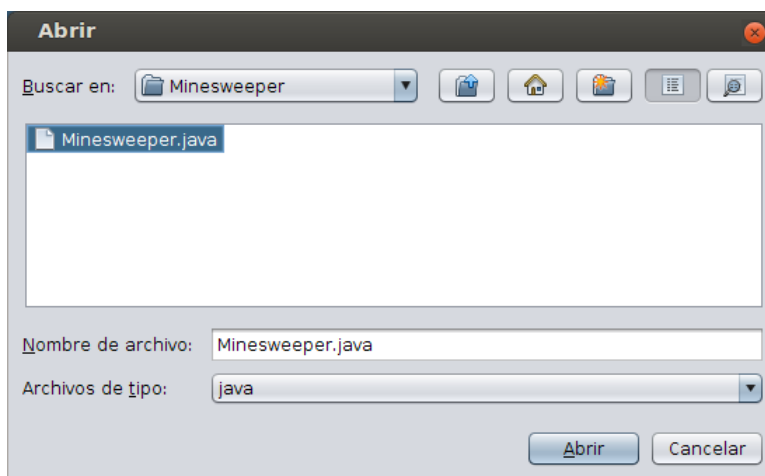


Figura 4.3: Ventana para seleccionar Archivos JAVA

Finalmente al presionar *Ayuda* de la barra de menú, se despliega un solo botón, el cual se describe a continuación (Ver Figura 4.2 - Flecha 3):

Acerca de: Brinda información sobre el autor y directores involucrados en la construcción de la herramienta IDA.

4.5.2. Lectura de Archivos JAVA

Cuando se pulsa en el botón *Abrir archivo(s) JAVA*, (Ver Figura 4.2 - Flecha 1), se despliega una ventana para que el usuario elija uno o varios archivos JAVA (Ver Figura 4.3).

Una vez que el usuario elige el/los archivo/s, IDA utiliza un programa externo llamado JACOB¹. Este programa JACOB, recibe como entrada un archivo JAVA y embellece el código fuente que esta contenido en el. Este embellecimiento se realiza para facilitar la lectura del código al usuario. En la próxima sección se describe el panel que IDA tiene para visualizar el código leído del archivo.

La herramienta IDA, además realiza un control de los archivos abiertos, impidiendo que se abra el mismo archivo más de una vez. En caso de que esto suceda la situación antes mencionada, se muestra un cartel informando al

¹<http://www.tiobe.com/jacobe>

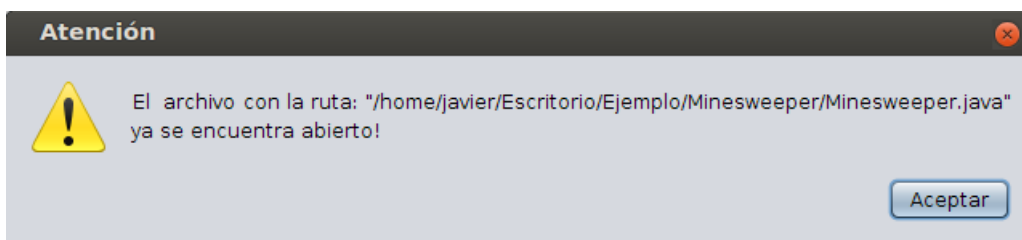


Figura 4.4: Aviso sobre Archivo JAVA ya abierto en IDA

usuario (Ver Figura 4.4). Este control se realiza por cuestiones de coherencia al momento de analizar los archivos.

4.5.3. Panel de Elementos Capturados

Después que el programa JACOBÉ embellece el código contenido en el archivo JAVA, el mismo se procesa por el AS explicado en secciones previas. Luego que el AS termina sus tareas de extracción de elementos (ids, comentarios y literales), el *Panel de Elementos Capturados* aparece (Ver Figura 4.5). Este panel en la parte superior posee pestañas, cada pestaña posee un rótulo con el nombre del archivo que está siendo analizado (Ver Figura 4.5 - Flecha 1). Es posible elegir de a varios archivos para analizar, mediante la ventana de selección de archivos (Ver Figura 4.3), o también se puede ir eligiendo de a un archivo por apertura de esta ventana. En caso de querer finalizar el análisis de un archivo particular y cerrar la pestaña, se puede pulsar en la cruz ubicada al lado del rótulo de cada pestaña (Ver Figura 4.5 - Flecha 1).

Cada pestaña en su interior posee el mismo subpanel que se divide en dos partes principales. La parte superior contiene el código leído y embellecido (por JACOBÉ) del archivo JAVA (Ver Figura 4.5 - Flecha 2). La parte inferior, muestra toda la información extraída por el AS referente a ids, literales y comentarios. Estos últimos tres poseen una pestaña cada uno (Ver Figura 4.5 - Flecha 3). Al pulsar sobre cada pestaña, se muestra la información clasificada correspondiente (a ids, literales y comentarios), a continuación se describe como se exhibe esta información.

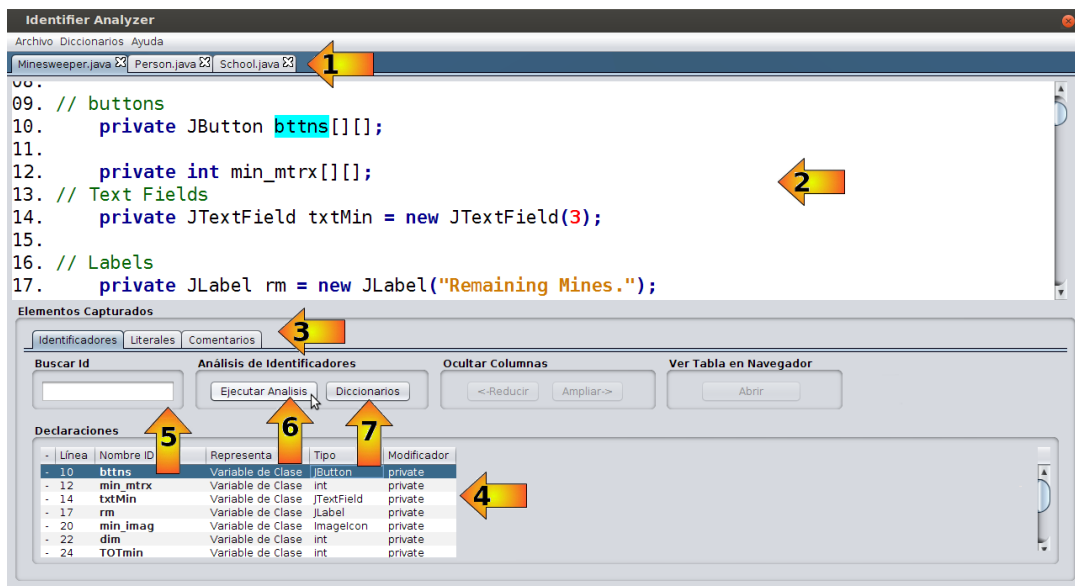


Figura 4.5: Panel de Elementos Capturados

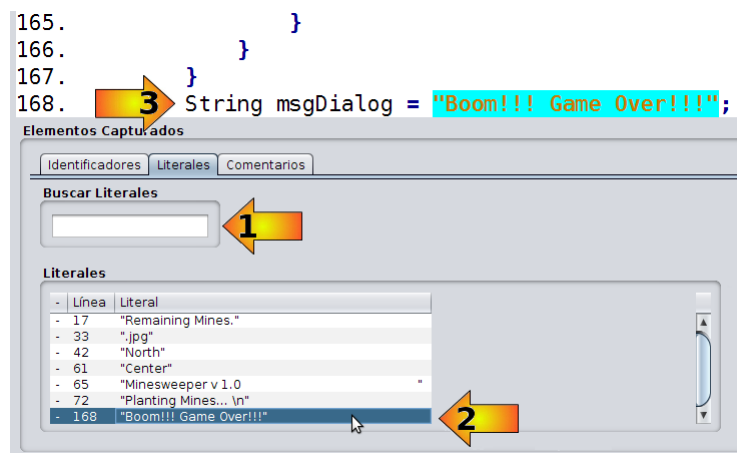


Figura 4.6: Pestaña de Literales Capturados

Pestaña de Identificadores Capturados

Al pulsar la *Pestaña de Identificadores* (Ver Figura 4.5 - Flecha 3), se muestra la *Tabla de Declaraciones* (Ver Figura 4.5 - Flecha 4). Esta tabla enumera los ids capturados por el AS y cada columna se corresponde a: el número de línea donde esta declarado el id, el nombre del id, el tipo

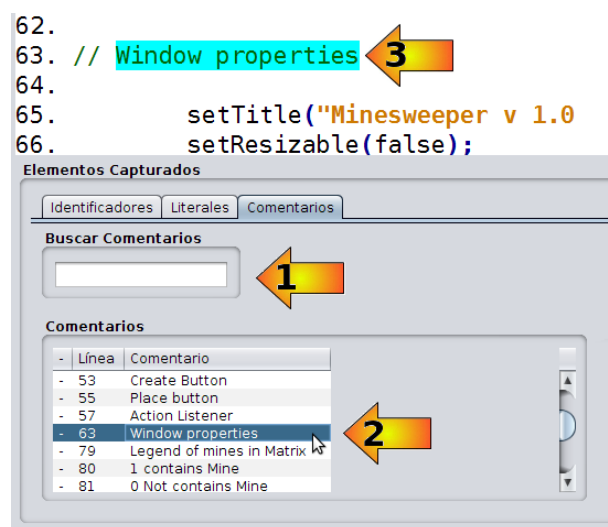


Figura 4.7: Pestaña de Comentarios Capturados

(int, char, etc.), el modificador (público, privado, protegido), lo que el id representa (variable de clase, constructor, método de clase, etc.). Esta *Tabla de Declaraciones* si se presiona sobre una fila, inmediatamente se resalta con color en el código ubicado en la parte superior, la declaración del id correspondiente (Ver Figura 4.5 - Flecha 2). Cabe destacar que si el usuario lo desea, puede realizar búsquedas en la *Tabla de Declaraciones* por nombre de id; para realizar las estas búsquedas se debe escribir en el cuadro de texto ubicado dentro del recuadro *Buscar Id* (Ver Figura 4.5 - Flecha 5), a medida que se escriba en este cuadro de texto, se irán filtrando los resultados en la *Tabla de Declaraciones*.

Pestaña de Literales Capturados

Al pulsar la *Pestaña de Literales* (Ver Figura 4.5 - Flecha 3), se puede apreciar que aparece una *Tabla de Literales* (Ver Figura 4.6 - Flecha 2) y un buscador (Ver Figura 4.6 - Flecha 1). Esta tabla contiene dos columnas, número de línea del literal y el literal propiamente dicho. También al pulsar sobre alguna fila, automáticamente el literal correspondiente se resalta en el código que está ubicado en la parte superior (Ver Figura 4.6 - Flecha 3). Por

otro lado, el buscador en forma de cuadro de texto, a medida que se escriba en este cuadro, se irán filtrando los resultados en la *Tabla de Literales*.

Pestaña de Comentarios Capturados

Al presionar la *Pestaña de Comentarios* (Ver Figura 4.5 - Flecha 3), se visualiza la *Tabla de Comentarios* y un buscador de comentarios en esta tabla (Ver Figura 4.7 - Flecha 1). La *Tabla de Comentarios* posee dos columnas que corresponden, por un lado al comentario y por el otro al número de línea donde se encuentra el comentario dentro del código (Ver Figura 4.7 - Flecha 2). Al igual que se describió en el párrafo anterior, al presionar en una de las filas de la *Tabla de Comentarios* inmediatamente se resalta en el código de la parte superior, la ubicación del comentario seleccionado (Ver Figura 4.7 - Flecha 3).

Hasta aquí, solo se ha descripto como IDA exhibe la información útil que fue capturada del código por el AS. A continuación, se explicará como se emplea IDA para analizar los ids. Para ello, se debe pulsar en la *Pestaña de Identificadores* (Ver Figura 4.5 - Flecha 3), luego pulsar en el botón *Ejecutar Análisis* ubicado dentro del cuadro *Análisis de Identificadores* (Ver Figura 4.5 - Flecha 6), al hacerlo se abrirá la *Ventana de Análisis* que será explicada en la próxima sección.

4.5.4. Ventana de Análisis

La *Ventana de Análisis* (Ver Figura 4.8) contiene 3 partes principales, (de izquierda a derecha):

Parte Izquierda: Posee un listado con los ids capturados, en el cuadro inferior izquierdo (Ver Figura 4.8 - Flecha 1). Arriba de estos se encuentran dos botones *Palabras Capturadas* y *Diccionarios* (Ver Figura 4.8 - Flecha 2), al pulsarlos le brindan información al usuario sobre los datos que se utilizan para ejecutar los algoritmos de análisis, y ambos botones serán explicados con más detalles en la próxima sección.



Figura 4.8: Ventana de Análisis

Parte Central: Ubicado en la parte central, en el cuadro superior (Ver Figura 4.8 - Flecha 3) se pueden seleccionar los dos algoritmos de división de ids (Greedy y Samurai), el botón *Dividir* del mismo cuadro ejecuta el algoritmo seleccionado. Los resultados obtenidos se muestran en el cuadro central inferior, en una tabla con los resultados. En la Figura 4.8 - Flecha 4 se muestran ambas técnicas (Greedy y Samurai) ya ejecutadas y enumerando los resultados.

Parte Derecha: Al presionar el botón *Expandir*, situado en el cuadro superior derecho (Ver Figura 4.8 - Flecha 5), ejecuta el algoritmo de expansión básico tomando como entrada los ids divididos desde Greedy o desde Samurai, según haya seleccionado el usuario en este mismo cuadro (Ver Figura 4.8 - Flecha 5). Los resultados obtenidos de las expansiones (desde Greedy y desde Samurai), se muestran en la tabla situada en el cuadro inferior derecho. En la Figura 4.8 - Flecha 6 se pueden apreciar las expansiones realizadas.



Figura 4.9: Ventana de Palabras Capturadas

4.5.5. Palabras Capturadas y Dicionarios

En la sección anterior, se describieron dos botones *Palabras Capturadas* y *Diccionarios*, ubicados en la *Ventana de Análisis* (Ver Figura 4.8 - Flecha 2). A continuación, se explicarán la función de cada uno de ellos.

Ventana de Palabras Capturadas

Al pulsar el botón *Palabras Capturadas*, se abre una ventana que posee dos cuadros (Ver Figura 4.9), el cuadro de la izquierda contiene una tabla que muestra las frecuencias correspondiente al Algoritmo Samurai (Ver Figura 4.9 - Flecha 1). Esta tabla tiene tres columnas, en la primera posee palabras¹, las mismas fueron capturadas y procesadas por el AS ANTLR. La segunda columna, contiene la frecuencia local de cada palabra, cabe recordar que la frecuencia local se construye en función de la frecuencia absoluta de aparición de las palabras en el código del archivo actual. La tercera y última columna de la tabla denota la frecuencia global de cada palabra, la misma está predefinida en la base de datos HSQLDB (Ver sección 4.4).

El cuadro de la derecha, lista en una tabla las frases capturadas (Ver

¹En este contexto, significa las palabras que están contenidas en literales, comentarios e ids, este último necesita un proceso especial, para más detalles ver Cap. 3 - sección 3.4.3

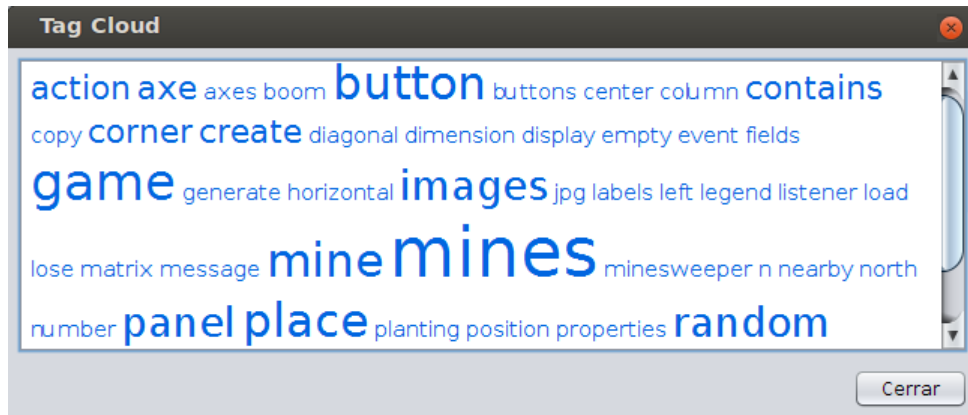


Figura 4.10: Ventana TagCloud (Nube de Etiquetas)

Figura 4.9 - Flecha 2), estas frases se obtienen de los comentarios, los literales strings, y algunos ids multi-palabras; las otras dos columnas de la tabla muestran la clase y el método donde se encuentra ubicado cada frase (Ver Figura 4.9 - Flecha 2). El Algoritmo de Expansión es el encargado de utilizar toda esta información (Ver Capítulo 3 - sección 3.4.4). Este algoritmo usa las frases, ya que son posibles candidatas para expandir las abreviaturas en forma de acrónimo que puede contener un id (Ejemplo: `fs` \Rightarrow `file system`). También cada una de las palabras por separado, puede ser utilizada para expandir abreviaturas comunes (Ejemplos: `sys` \Rightarrow `system`, `fl` \Rightarrow `file`). La ubicación de cada frase (en que método y/o clase se encuentra), el Algoritmo de Expansión la utiliza para dar prioridad a aquellas frases que están en el mismo método y/o clase que la palabra a expandir, sino encuentra frase candidatas observa el resto de las frases en la tabla.

A modo de agilizar las búsquedas (de palabras) en las tablas descriptas anteriormente (la de frecuencias de Samurai y la de frases), se pueden llevar a cabo utilizando los cuadros de texto con rótulo *Buscar Palabra* situados arriba de cada tabla correspondiente (Ver Figura 4.9).

Ventana TagCloud (Nube de Etiquetas)

Como se puede observar en la Figura 4.9 - Flecha 3, existe un botón con el

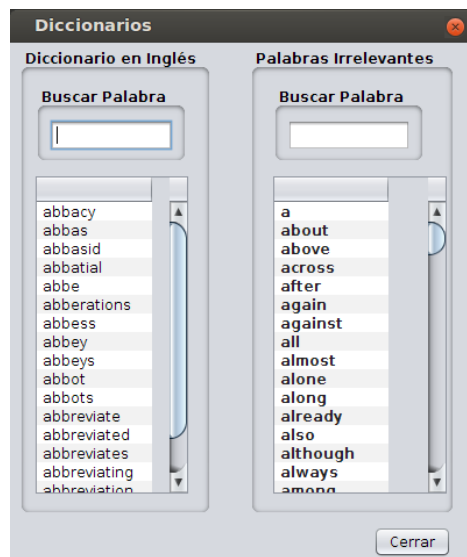


Figura 4.11: Ventana de Diccionarios

nombre de *TagCloud de Palabras*, al pulsar sobre este, abre una ventana que contiene una *Nube de Etiquetas* (Ver Figura 4.10). Esta nube posee palabras y resalta en tamaño más grande aquellas palabras que más frecuencia de aparición tienen (En la Figura 4.10 las palabras *mine*, *mines*, *game* etc. son las que mayor aparición tienen). Para generar esta nube se emplea una librería de JAVA llamada *OpenCloud*¹. Esta *Nube de Etiquetas* ayuda a ver con más claridad que palabras son más frecuentes, dado un conjunto de palabras pasado por entrada.

La *Nube de Etiquetas* correspondiente a la tabla de frecuencias de Samurai (Ver Figura 4.9 - Flecha 3), el tamaño de cada palabra depende de la Frecuencia Local de cada palabra (Ver Figura 4.9 - Flecha 1), mientras mayor sea la frecuencia que tenga una palabra determinada, mayor será el tamaño de la misma en la nube. Con esta visualización, se pretende ayudar al usuario a determinar que conceptos son más abundantes en el sistema analizado.

¹<http://opencloud.mcavallo.org>

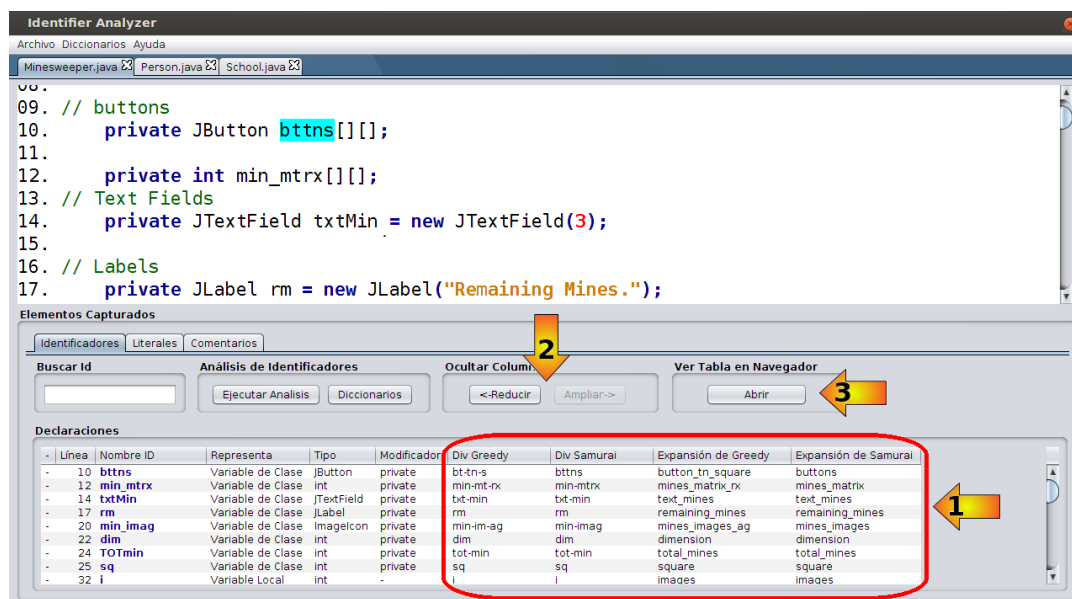
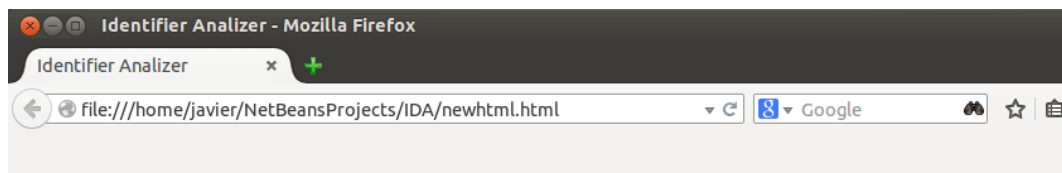


Figura 4.12: Panel de Elementos Capturados

Ventana de Diccionarios

Volviendo a la *Ventana de Análisis* si se pulsa el botón *Diccionarios* (Ver Figura 4.8 - Flecha 2), se abre la *Ventana de Diccionarios* (Ver Figura 4.11). Esta ventana posee dos tablas, la tabla de la izquierda lista todas las palabras en inglés que tiene el diccionario del comando de Linux *ispell*, esta lista de palabras es utilizada por el Algoritmo Greedy y el Algoritmo Expansión Básica (Ver sección 4.4 de este capítulo). La segunda tabla de la derecha enumera las palabras que pertenecen a la stoplist o lista de palabras irrelevantes, también utilizada por los dos algoritmos antedichos. Convenientemente, ambas tablas poseen un buscador por palabras dado que el contenido de cada una es amplio (Ver Figura 4.11). Esta *Ventana de Diccionarios* puede ser invocada desde otros lugares de la herramienta IDA. Uno de ellos es desde la barra de menú (Ver Figura 4.2 - Flecha 2). Otro sitio donde puede abrirse, es desde el *Panel de Elementos Capturados*, pulsando el botón que esta al lado de *Ejecutar Análisis* (Ver Figura 4.5 - Flecha 7).

Todas las ventanas que fueron descriptas previamente (palabras capturadas, tagcloud, diccionarios), en caso de haber sido abiertas por el usuario, el



Identifier Analyzer

Análisis de Identificadores: Minesweeper.java

Identificador	División Greedy	División Samurai	Expansión desde Greedy	Expansión desde Samurai
bttns	bt-tn-s	bttns	button_tn_square	buttons
min_mtrx	min-mt-rx	min-mtrx	mines_matrix_rx	mines_matrix
txtMin	txt-min	txt-min	text_mines	text_mines
rm	rm	rm	remaining_mines	remaining_mines
min_imag	min-im-ag	min-imag	mines_images_ag	mines_images
dim	dim	dim	dimension	dimension
TOTmin	tot-min	tot-min	total_mines	total_mines
sq	sq	sq	square	square
i	i	i	images	images
topPanel	top-panel	top-panel	top_panel	top_panel
middlePanel	middle-panel	middle-panel	middle_panel	middle_panel
j	j	j	jpg	jpg
plantmines	plant-mines	plant-mines	planting_minesweeper	planting_minesweeper

Figura 4.13: Captura del Navegador Web con el Análisis de Ids realizado

mismo debe cerrarlas si desea continuar con el proceso de análisis de ids.

4.5.6. Nuevamente al Panel de Elementos Capturados

Una vez que los ids fueron analizados (Divididos y Expandidos) mediante la *Ventana de Análisis*, la misma debe ser cerrada presionando el botón *Cerrar* (Ver Figura 4.8 - Flecha 7). Esta acción, retorna al *Panel de Elementos Capturados* nuevamente (Ver Figura 4.12). Como se puede observar, en la tabla *Declaraciones* que detalla los ids extraídos e información asociada a estos, se le suman nuevas columnas (Ver Figura 4.12 - Flecha 1). Estas nuevas columnas contienen los resultados obtenidos de los algoritmos de división (Greedy, Samurai) y el algoritmo de expansión ejecutados en la *Ventana de Análisis*; las columnas nuevas son: División Greedy, División Samurai,

Expansión desde Greedy y Expansión desde Samurai.

Al agregar las columnas nuevas se habilita el cuadro *Ocultar Columnas*. En el se encuentran dos botones *Reducir* y *Ampliar* (Ver Figura 4.12 - Flecha 2). El primero de ellos a modo de facilitar la visualización, oculta las columnas que hay entre los ids y las columnas que contienen el análisis de ids (las columnas que se ocultan son: tipo, modificador, Representa), de esta manera el usuario puede comparar más claramente las distintas divisiones y expansiones de ids. Mientras que el botón *Ampliar* restablece las columnas originales.

Luego si el usuario lo decide, puede presionar el botón *Abrir* en el cuadro *Ver Tabla en Navegador* (Ver Figura 4.12 - Flecha 3). Esta acción abre automáticamente el navegador web por defecto del sistema operativo, y mediante una página web en formato html, se muestra una tabla con los resultados obtenidos producto del análisis de ids (Ver Figura 4.13). De esta manera, el usuario puede visualizar más claramente el análisis realizado, permitiendo también imprimir los resultados en papel (mediante el navegador web), si el usuario lo desea.

Dando por finalizado el proceso necesario que debe realizar el usuario para analizar los ids con la herramienta IDA, resta destacar que la misma a su vez posee una extensión. Esta extensión permite que IDA pueda interactuar con otras aplicaciones y de esta manera forme parte de un proceso de análisis más extenso, la extensión antedicha se describe en el Apéndice A.

Para continuar con el desarrollo de este trabajo final, en el próximo capítulo se explican algunos casos de estudio que demuestran la importancia de haber construido IDA.

Capítulo 5

Casos de Estudio

5.1. Introducción

En este capítulo, se presentan tres casos de estudio que fueron seleccionados para que sean analizados por Identifier Analyzer (IDA). En cada uno de estos casos se examinan los identificadores (ids) de una aplicación JAVA. A través de tablas se irán mostrando los resultados obtenidos, producto de la ejecución de las técnicas que IDA posee. Con estos casos, se pretende manifestar la utilidad de la herramienta IDA en lo que respecta al análisis de ids y demostrar que es un aporte al área de la CP. A continuación, se listan las aplicaciones JAVA que serán analizadas por IDA:

- Juego Buscaminas: Minesweeper
- Editor de Texto
- Juego de Aviones: Top Gun

En las próximas secciones, se describe el análisis realizado de cada una.

5.2. Juego Buscaminas

El programa JAVA denominado Minesweeper.java, al ejecutarlo posee el clásico y conocido juego llamado Buscaminas (Minesweeper en Inglés - Ver

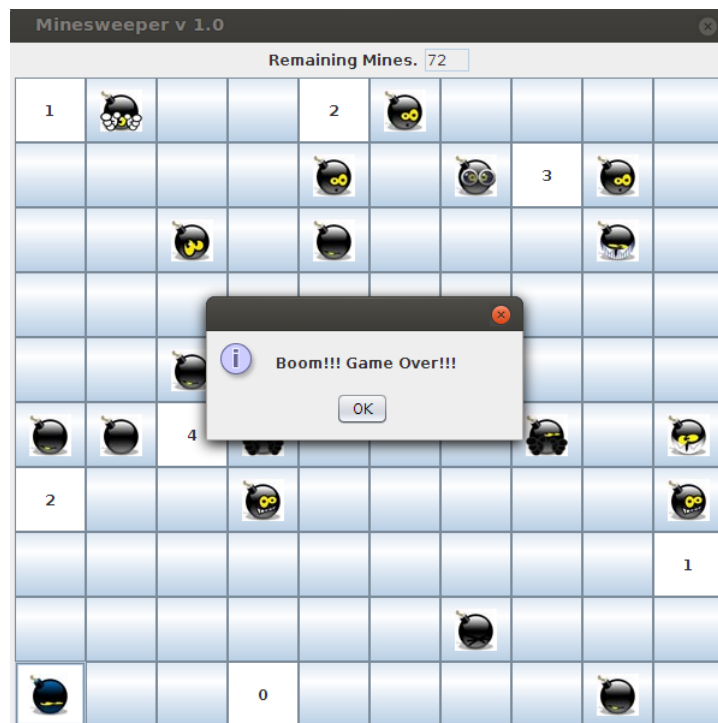


Figura 5.1: Captura del Juego Buscaminas programado en JAVA

Figura 5.1). El juego consiste en despejar los casilleros de un tablero que posee minas, estas minas están ocultas y el jugador pierde en caso de detonar alguna. Este programa contiene un módulo de 250 líneas aproximadamente que fueron analizadas por IDA.

Captura de Información

Al ingresar el programa Minesweeper.java a la herramienta IDA, se da comienzo a la fase de extracción de datos y el analizador sintáctico (AS) captura información referente a los ids. Esta información la exhibe IDA al usuario en el *Panel de Elementos Capturados*, en la tabla *Declaraciones* (Ver Capítulo anterior). En la tabla 5.1 se puede apreciar esta información (por columnas): la línea donde está declarado el id, el nombre, que representa en el código analizado, el tipo del id y que modificador posee.

Para hacer una descripción más detallada sobre los ids capturados, en la

tabla 5.1 se puede observar que el archivo Minesweeper.java tiene ids del tipo *hardwords* y *softwords* (Ver Capítulo 3 - sección 3.3.1). Algunos *hardwords* que se pueden observar son `min_mtrx`, `TOTmin`, `topPanel` (entre otros), ya que estos poseen una marca de separación que destacan las palabras que lo componen. Por otro lado, algunos de los *softwords* que se capturaron son `ae`, `plantmines`, `bttms`, `xar` (entre otros). A su vez, se capturaron los comentarios, los mismos se muestran en tabla 5.2¹. De la misma forma, los literales Strings que se extrajeron se exhiben en la tabla 5.3. En ambas tablas se muestran las líneas en donde se ubica cada comentario y literal en el código. Tanto los literales y los comentarios se visualizan en IDA a través del *Panel de Elementos Capturados*, eligiendo la pestaña correspondiente (Ver Capítulo anterior).

Es importante recordar, que los comentarios y literales de las tablas 5.2 y 5.3 son usados para construir la lista de frases que se muestra en la ventana de *Palabras Capturadas* (Ver Capítulo anterior). Esta información es útil para el Algoritmo de Expansión. De la misma manera, con los comentarios y literales se construye una parte importante del listado de frecuencias locales de aparición de palabras², que son destinadas a ser usadas por el Algoritmo de división Samurai. Esta lista se visualiza en IDA también dentro de la ventana de *Palabras Capturadas* (Ver Capítulo anterior).

Habiendo explicado con este caso de estudio la información capturada por el AS, asociada a ids, literales y comentarios, en la próxima sección se procede a analizar (para este mismo caso de estudio), los resultados obtenidos producto del análisis de los ids.

¹Los comentarios de más de una línea (comprendidos entre `/* */`), se descomponen en líneas diferentes.

²En este contexto, se refiere a palabras que están contenidas en literales, comentarios e ids, este último necesita un proceso especial, para más detalles ver Cap. 3 - sección 3.4.3

Línea	Nombre ID	Representa	Tipo	Modificador
7	Minesweeper	Clase	–	public
10	bttns	Variable de Clase	JButton	private
12	min_mtrx	Variable de Clase	int	private
14	txtMin	Variable de Clase	JTextField	private
17	rm	Variable de Clase	JLabel	private
20	min_imag	Variable de Clase	ImageIcon	private
22	dim	Variable de Clase	int	private
24	TOTmin	Variable de Clase	int	private
25	sq	Variable de Clase	int	private
37	topPanel	Variable Local	JPanel	–
48	middlePanel	Variable Local	JPanel	–
71	plantmines	Método de Clase	void	private
71	mins	Parámetro	int	–
102	main	Método de Clase	void	public
102	args	Parámetro	String[]	–
107	actionPerformed	Método de Clase	void	public
107	ae	Parámetro	ActionEvent	–
124	uncoverEmpty	Método de Clase	void	private
124	j	Parámetro	int	–
124	i	Parámetro	int	–
150	win	Método de Clase	void	private
159	boom	Método de Clase	void	private
172	msgDialog	Variable Local	String	–
179	nearbyMines	Método de Clase	int	private
188	MINSnum	Variable Local	int	–
179	xar	Parámetro	int	–
179	yac	Parámetro	int	–

Tabla 5.1: Identificadores extraídos por el AS ANTLR

Línea	Comentario	Línea	Comentario
9	buttons	85	Generate random position
13	Text Fields	91	Place mine
16	Labels	93	Display mines panel
19	Mines images	107	Action Event
21	Dimension	126	Uncover an empty square
23	total mines	129	Nearby Mines
27	Time tp	136	restart game
31	load Images	152	Win the game
36	Top Panel	161	lose the game
47	Button panel	165	Mines Random Images
50	Create and place button	183	x axe row
53	Create Button	184	y axe column
55	Place button	186	return the number of mines
57	Action Listener	192	horizontal
63	Window properties	199	vertical
80	Legend of mines in Matrix	207	diagonal
81	1 contains Mine	208	Top left corner
82	0 Not contains Mine	209	copy of axes
83	Place random mine	224	top right corner

Tabla 5.2: Comentarios extraídos por el AS ANTLR

Línea	Literal
17	“Remaining Mines.”
33	“.jpg”
42	“North”
61	“Center”
65	“Minesweeper v 1.0 ”
73	“Planting Mines...”
153	“You Win!!! Game Over!!!”
155	“Message”
173	“Boom!!! Game Over!!!”
175	“Message”

Tabla 5.3: Literales extraídos por el AS ANTLR

Análisis de Resultados

Cuando el usuario ejecuta las técnicas de análisis de ids, lo lleva a cabo mediante la *Ventana de Análisis* (Ver Capítulo anterior), aquí se aplican las técnicas de división (Greedy y Samurai) y después la técnica de expansión de abreviaturas. Esta misma ventana, muestra debajo los resultados obtenidos, luego de ejecutar las técnicas. Para el caso del programa Minesweeper.java, los resultados que se obtuvieron producto del análisis de ids, se pueden apreciar en la Tabla 5.4.

Con respecto a la información mostrada en la tabla mencionada previamente, las columnas de *Greedy* y *Samurai* muestran los resultados de división de dichos Algoritmos. En las columnas *Expansión desde Greedy* y *Expansión desde Samurai* se enumeran las expansiones realizadas de las distintas partes del id, que fueron efectuadas desde los Algoritmos Greedy y Samurai respectivamente.

En lo que respecta a los ids analizados de Minesweeper.java (Ver Tabla 5.4), se pueden observar que algunos son del tipo *hardwords*. Dentro de esta clasificación, se pueden encontrar ids con guión bajo, por ejemplo: `min_imag`, `min_mtrx`; también algunos casos del tipo camel-case, por ejemplo: `uncoverEmpty`, `msgDialog`; a su vez, también existen casos de la variante camel-case, por ejemplo: `TOTmin`, `MINsnum`.

El algoritmo Greedy manifiesta irregularidades a la hora de dividir ya que siempre considera que la mayor cantidad de divisiones es la mejor opción (Ver Capítulo 3 - sección 3.4.2), esto puede observarse en casos como `min-im-ag`, `bt-tn-s`, `min-sn-um` (Ver Tabla 5.4 - columna Greedy). Para los ids que son del tipo especial: `MINsnum`, `TOTmin` (variante camel-case), es interesante observar como Samurai realiza la división correctamente: `mins-num`, `tot-min` (Ver Capítulo 3 - sección 3.4.3), y no la considera un caso común de camel-case que la separa antes de la mayúscula seguido de minúscula. Esto sucede justamente con Greedy ya supone que es del tipo camel-case, y realiza la separación incorrecta: `min-sn-um` (entre min y sn), `to-tm-in` (entre to y tm).

En lo que respecta a *softwords* se aprecia la presencia de acrónimos como `rm` y `ae` (entre otros); ambos algoritmos de división no los dividen. La

razón de esto es porque poseen solo dos caracteres y el algoritmo lo considera acrónimos. A la hora de expandir **rm**, **ae**, el Algoritmo de Expansión consulta la lista de frases conformada en su mayoría por los comentarios y literales capturados (Ver Tablas 5.2 y 5.1), al encontrar coincidencia con el literal “**remaining mines**” y el comentario **action event**, selecciona ambos como la expansión correspondiente de **rm** y **ae** (Ver Tabla 5.4 - Columnas de Expansión). El resto de los softwords se puede considerar a **mins**, **bttns**, **dim**, aquí Greedy también acusa inconvenientes y procede a separar estos ids siendo que no se deben separar, caso contrario ocurre con **Samurai** (Ver Tabla 5.4 - Columnas de División).

Continuado en la tabla 5.4, los ids *i*, *j* son comunes en la mayoría de los códigos, y generalmente se utilizan como contadores en estructuras de control iterativas (**for**, **while**, etc). En estos casos, el Algoritmo de Expansión busca palabras que estén dentro del *Dominio del Problema*. Para lograrlo, recurre a las tablas de Literales y Comentarios (Ver Tablas 5.2 y 5.3), luego trata de encontrar palabras que comiencen con las letras respectivas, para este caso **image**, **jpg**. De esta manera el algoritmo, trata de darle una traducción válida en este contexto. En caso de no haber coincidencia con ninguna, el algoritmo de expansión buscará en el diccionario de palabras en Inglés.

Para concluir, en las columnas de expansión de la tabla 5.4 se visualizan palabras como: explosión, vacío, plantar, minas, recuadro, eje x, eje y, dimensión (entre otras). Estas palabras intuitivamente se consideran más cercanas al *Dominio del Problema* correspondiente al programa Minesweeper.java.

Id	Greedy	Samurai	Exp. desde Greedy	Exp. desde Samurai
Minesweeper	minesweeper	minesweeper	minesweeper	minesweeper
bttns	bt-tn-s	bttns	button tn square	buttons
min mtrx	min-mt-rx	min-mtrx	mines matrix rx	mines matrix
txtMin	txt-min	txt-min	text mines	text mines
rm	rm	rm	remaining mines	remaining mines
min_imag	min-im-ag	min-imag	mines images ag	mines images
dim	dim	dim	dimension	dimension
TOTmin	to-tm-in	tot-min	to total mines in	total mines
sq	sq	sq	square	square
topPanel	top-panel	top-panel	top panel	top panel
middlePanel	middle-panel	middle-panel	middle panel	middle panel
plantmines	plant-mines	plant-mines	planting minesweeper	planting minesweeper
mins	min-s	mins	mines square	mines
actionPerformed	action-performed	action-performed	action performed	action performed
ae	ae	ae	action event	action event
uncoverEmpty	uncover-empty	uncover-empty	uncover empty	uncover empty
j	j	j	jpg	jpg
i	i	i	images	images
win	win	win	window	window
boom	boom	boom	boom	boom
msgDialog	msg-dialog	msg-dialog	message dialog	message dialog
nearbyMines	nearby-mines	nearby-mines	nearby minesweeper	nearby minesweeper
MINSnum	min-sn-um	mins-num	mines sn um	mines number
xar	xa-r	xar	xa row	x axe row
yac	y-ac	yac	yellow action	y axe column

Tabla 5.4: Análisis Realizado a los Ids extraídos de Minesweeper.java

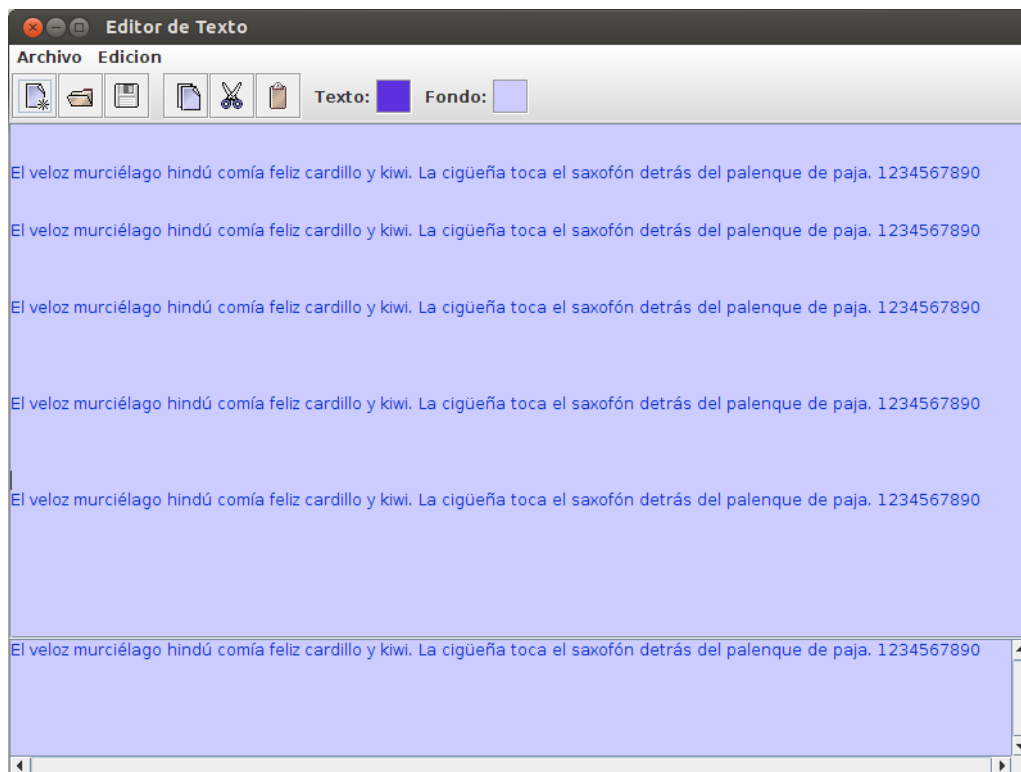


Figura 5.2: Captura del Editor de Texto programado en JAVA

5.3. Editor de Texto

En el próximo caso de estudio, se seleccionó un programa escrito en JAVA que corresponde a un editor de texto (Ver Figura 5.2). Este editor posee las herramientas básicas para crear o modificar archivos de textos sin formato, permite cambiar la visualización de colores en las letras y en el fondo, también imprime los archivos que se abran en el. Este editor esta programado con alrededor de 500 líneas de código, las cuales están escritas en un único archivo llamado Editor.java.

Cabe destacar que, a modo de facilitar la lectura, de aquí en adelante, por cada caso de estudio, se mostrará únicamente la tabla con resultados obtenidos producto del análisis de los ids y se excluirán el resto de las tablas que se mostraron en el caso de estudio anterior. El proceso es el mismo y por lo tanto no tiene sentido volverlo a escribir.

A continuación, se brindan los resultados obtenidos de analizar con la herramienta IDA el editor de textos, el análisis efectuado en los ids se exhibe en la tabla 5.5. Dado que este programa posee muchos ids, los resultados que se presentan en la tabla 5.5 son los más destacados y no son la totalidad.

Análisis de Resultados

A simple vista, se puede observar en la tabla 5.5, que nuevamente las divisiones de Samurai son las que mejor se realizan. Como se explicó anteriormente, esto ocurre por que Greedy siempre selecciona la mayor cantidad de divisiones en la palabra como la mejor opción (Ver Capítulo 3 - sección 3.4.2). Algunos ejemplos de divisiones mal hechas por Greedy son `bckCol` → `b-c-k-col`, `btAccept` → `bt-ace-pt`, `jChoColor` → `j-c-ho-color`, entre otros.

Sin embargo, en este caso de estudio a diferencia del anterior existen algunas divisiones en las que Samurai falla, los ids que se pueden observar son: `flreader`, `ptjob` los cuales directamente no se dividieron como lo hizo Greedy en `fl-reader`, `pt-job`. La primer hipótesis que se maneja sobre este resultado, es que Greedy encuentra en el diccionario en Inglés las palabras `reader` y `job` y con esto basta para llevar a cabo la división. Mientras que Samurai en su función de *Score*, las palabras `fl` con `reader` y `pt` con `job` no representan puntajes (score) altos para ser divididas entre ellas (Ver Capítulo 3 - sección 3.4.3). En este caso de estudio, al igual que el anterior, también existen casos de variante de camel-case, algunos son: `TEXTarea`, `PRGname` y el algoritmo Samurai los divide correctamente, mientras que Greedy no (Ver Tabla 5.5).

En lo que respecta a la expansión de ids, la mayoría de las abreviaturas resultantes de la división de ids fueron expandidas; algunos ejemplos son `sel` → `select`, `tl` → `tool`, `cl` → `close`, `bck` → `background`, entre otras. Estas palabras son expandidas por el algoritmo de expansión, gracias a los comentarios y literales capturados por el AS. Por otro lado, existen algunas abreviaturas que no se expanden como es el caso de `bt` a `button`, `it` a `item`; que forman parte de los ids `bt-accept` y `men-it-new`. La hipótesis de este comportamiento en el Algoritmo de Expansión, se debe a que estas abreviaturas son tratadas

como acrónimos (abreviaturas con más de una palabra) dado que tienen solo dos caracteres y no existe una frase (de comentarios o literales) capturada por el AS, que coincida. Como se consideran abreviaturas con más de una palabra, tampoco se utiliza el diccionario en Inglés para expandir.

También existen casos de abreviaturas mal expandidas, uno de ellos es **bar** → **background** en el id **menBarEdit**. Si bien **bar** hace referencia a una barra de menú, aquí simplemente el algoritmo de expansión entiende que **bar** es una abreviatura y la expande con un candidato fuerte que se encuentra en el listado de frases capturadas **background**.

La mayoría de los ids analizados (Ver Tabla 5.5 columnas Expansión desde Greedy y Expansión desde Samurai), las palabras presentes se corresponden a distintos elementos que son propios de un editor de textos convencional. De estos elementos se pueden enumerar algunos: texto, copiar, pegar, imprimir, imagen, color, archivo, herramienta, nuevo, guardar, buscar. Estos elementos sin duda están asociados al Dominio del Problema en el programa Editor.java.

Id	Greedy	Samurai	Exp. desde Greedy	Exp. desde Samurai
menBarFile	men-bar-file	men-bar-file	menu background file	menu background file
menItNew	men-it-new	men-it-new	menu it new	menu it new
menBarEdit	men-bar-edit	men-bar-edit	menu background editor	menu background editor
menItCopy	men-it-copy	men-it-copy	menu it copy	menu it copy
jTlBar	j-tl-bar	j-tl-bar	java tool background	java tool background
jBtSave	j-bt-save	j-bt-save	java bt save	java bt save
popUpMenu	pop-up-menu	pop-up-menu	pop up menu	pop up menu
imlcPrint	im-ic-print	im-ic-print	images icon printing	images icon printing
PRGname	pr-g-name	prg-name	program gif name	program name
jLabelColTex	j-label-col-t-ex	j-label-col-tex	java label color text exit	java label color text
bckCol	b-c-k-col	bck-col	bar copy kilo color	background color
TEXTarea	t-ex-tare-a	text-area	text exit tare areas	text areas
textAREAEerrors	t-ex-tare-ae-rors	text-area-errors	text exit tare areas rrors	text areas errors
jScrPANtxtAr	j-scr-pa-nt-xt-ar	j-scr-pan-txt-ar	java scrollbar program north xt areas	java scrollbar pan text areas
selCl	sel-c-l	sel-cl	select copy literalizes	select close
fc	f-c	fc	finish copy	fc
freader	fl-reader	freader	file reader	freader
ioe	io-e	ioe	icon editor	invoiced
printText	print-text	print-text	printing text	printing text
ptjob	pt-job	ptjob	printing job	ptjob
pg	pg	pg	print graphics	print graphics
linNum	l-in-nu-m	lin-num	lab in nu menu	linearised numerically
i	i	i	images	images
jLbFind	j-lb-find	j-lb-find	java lb find	java lb find
jTxFind	j-tx-find	j-tx-find	java text find	java text find
jPanBut	j-pan-but	j-pan-but	java pan but	java pan but
jChoColor	j-c-ho-color	j-cho-color	java copy ho color	java choose color
btAccept	bt-ace-pt	bt-accept	bt ace printing	bt accept

Tabla 5.5: Parte del Análisis Realizado a los Ids de Editor.java

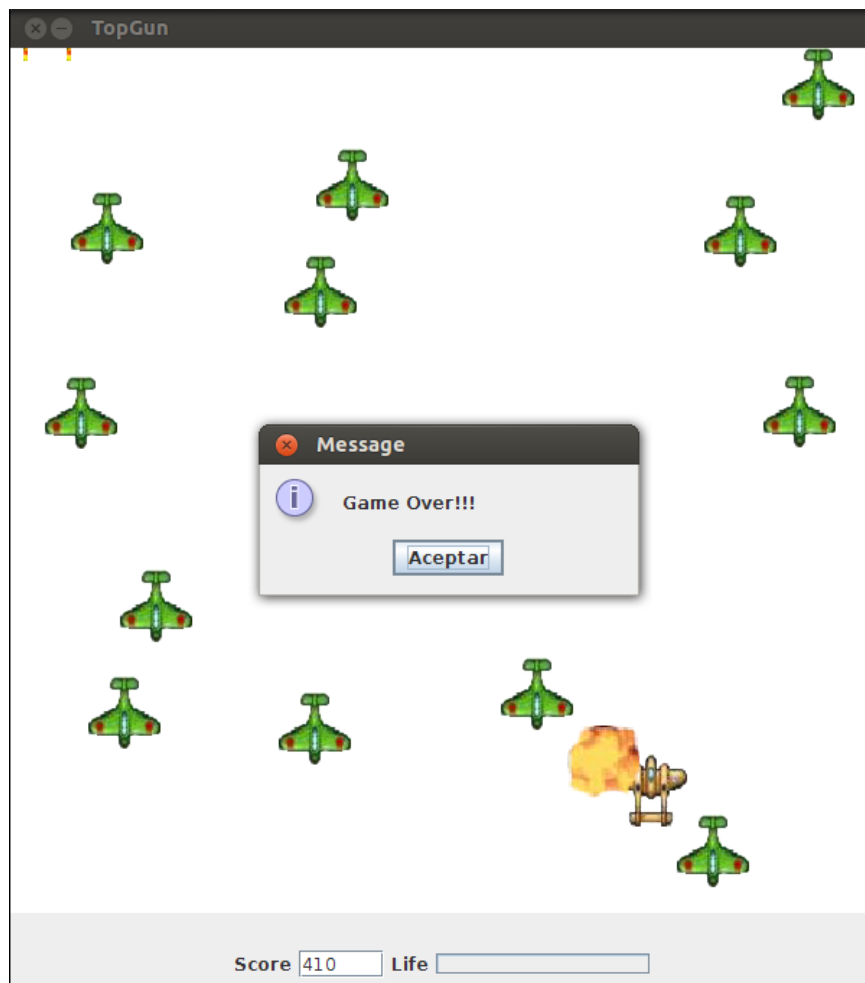


Figura 5.3: Captura del juego Top Gun programado en JAVA

5.4. Juego de Aviones

Continuando con los casos de estudio, el próximo es un programa JAVA llamado TopGun.java. Cuando este programa se ejecuta aparece un juego de aviones (Ver Figura 5.3), en este juego el usuario conduce un avión que dispara proyectiles. El objetivo consiste en derribar la mayor cantidad de aviones enemigos y un contador suma puntos por cada avión derribado. El juego finaliza cuando al avión se le agota la barra de energía por completo, esta barra disminuye debido a los disparos enemigos que recibe. Este programa, posee un módulo de aproximadamente de 600 líneas que fueron

analizadas por la herramienta IDA. Al igual que el caso de estudio anterior (Editor de texto), debido a que el programa TopGun.java contiene muchos ids, en la tabla 5.6 se lista el análisis de los ids más relevantes.

Análisis de Resultados

A diferencia de los casos de estudios antes vistos, en la tabla 5.6 se puede observar la presencia de ids capturados conformados por tres palabras; algunos ejemplos de esto son: `LIFEprogressBar`, `hitShotEnemy`, `hitPlaneEnemy`. En la división de ids persiste la tendencia en Greedy con respecto a Samurai, de separar mal algunos ids (al igual que los casos de estudio anteriores). Algunos ejemplos son: `lnEne` → `ln-en-e`, `enelimage` → `en-e-image`, `strt_but` → `str-t-but` (entre otros).

Las expansiones de ids en general están bastante precisas, salvo casos como `hit` → `height`, en donde el algoritmo interpreta que `hit` es la abreviatura de **height** y no debería expandirse. El problema aquí se da porque `height` forma parte de un comentario, entonces el algoritmo le da prioridad (lo mismo sucedía con `bar` en el caso de estudio anterior). Otro caso similar ocurre con `key` → **key**listener.

Por otro lado, la abreviatura `but` que representa **buttons** no se expande, se estima que esto ocurre por que **buttons** no está en ninguna frase del código (comentario y literal), y no se expande con palabras del diccionario en Inglés porque `but` es una palabra válida del diccionario.

De los ids analizados (Ver Tabla 5.6 en las columnas Expansión desde Greedy y Expansión desde Samurai), las palabras que más frecuentemente aparecen están asociados al juego. De estas palabras se pueden nombrar, disparos, aviones, enemigos, imágenes, actualizar pantalla, movimiento. Estas palabras en gran parte están relacionadas al Dominio del Problema perteneciente al programa TopGun.java.

Id	Greedy	Samurai	Exp. desde Greedy	Exp. desde Samurai
strt_but	str-t-but	strt-but	start topgun but	start but
scoLabel	sco-label	sco-label	score label	score label
scoText	sco-text	sco-text	score text	score text
lifeLabel	life-label	life-label	life label	life label
LIFEProgressBar	l-if-e-progress-bar	life-progress-bar	load if enemies progress background	life progress background
init	init	init	initiate	initiate
sn	sn	sn	shoot number	shoot number
pm	pm	pm	plane movement	plane movement
shot	shot	shot	shoot	shoot
lnEne	ln-en-e	ln-ene	launch enemies enemies	launch enemies
rfsScreen	rfs-h-screen	rfs-h-screen	refresh hit screen	refresh screen
shoImage	sho-image	sho-image	shot images	shot images
eneImage	en-e-image	ene-image	enemies enemies images	enemies images
bangImage	bang-image	bang-image	bang images	bang images
tg	tg	tg	topgun	topgun
hitPlaneEnemy	hit-plane-enemy	hit-plane-enemy	height planes enemys	height planes enemys
intExp	int-exp	int-exp	int exploit	int exploit
enNum	en-num	en-num	enemies number movement	enemies number
getYac	get-y-ac	get-yac	get yin active	get y axe column
getXar	get-xa-r	get-xar	get xa row	get x axe row
ie	ie	ie	initiate enemies	initiate enemies
updatePlane	update-plane	update-plane	update planes	update planes
updateShot	update-shot	update-shot	update shoot	update shoot
hitShotEnemy	hit-shot-enemy	hit-shot-enemy	height shoot enemys	height shoot enemys
j	j	j	jFrame	jFrame
updatePlane	update-plane	update-plane	update planes	update planes
updateShot	update-shot	update-shot	update shoot	update shoot
keyReleased	key-released	key-released	keylistener released	keylistener released
plalma	pla-im-a	pla-ima	plane_images_attributes	plane_images

Tabla 5.6: Parte del Análisis Realizado a los Ids de TopGun.java

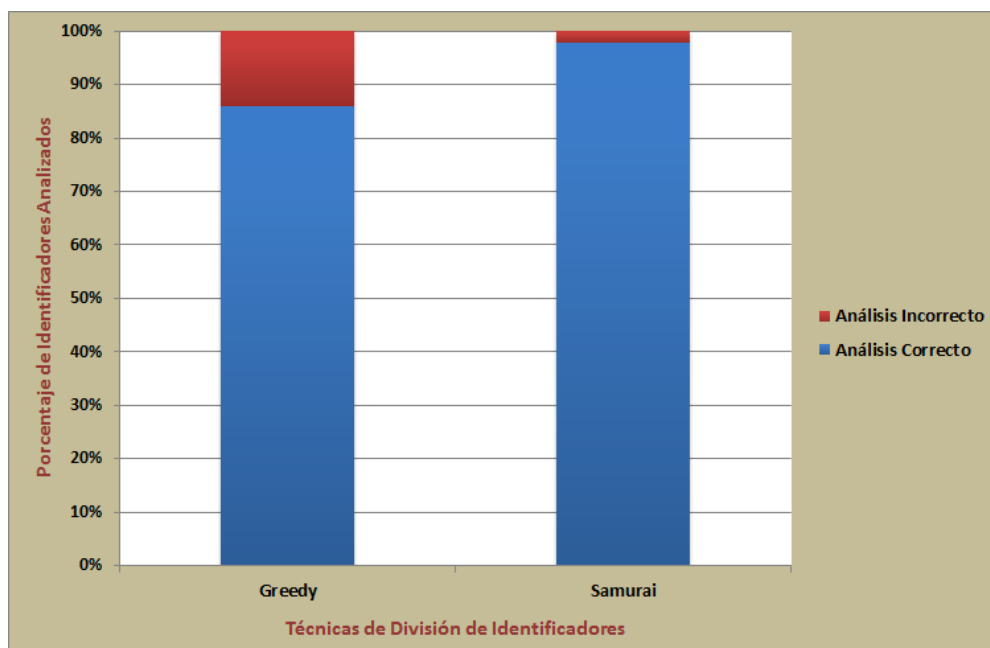


Figura 5.4: Porcentajes de Ids divididos correctamente e incorrectamente

5.5. Notas y Comentarios sobre los casos de estudio

Una vez que IDA analizó alrededor de 400 ids en los tres casos de estudio presentados previamente, los resultados obtenidos de las divisiones se resumen en el gráfico de barras de la Figura 5.4. Como se puede observar en este gráfico, el eje vertical representa el porcentaje de ids analizados sobre la cantidad total. El color azul de cada barra indica que porcentaje sobre total de ids se separó correctamente, mientras que el rojo señala lo contrario. Por otro lado, el eje horizontal corresponde a cada una de las técnicas empleadas para dividir ids (Greedy, Samurai). Claramente, el gráfico de la Figura 5.4 muestra que entre las dos técnicas de división, Samurai contiene una tasa de fallo mucho menor que Greedy. Esto era previsible por que el algoritmo Greedy es bastante simple y divide sin tener en cuenta el contexto en el que esta situado el id, no ocurriendo lo mismo con Samurai, ya que emplea un análisis más complejo a la hora de realizar su tarea.

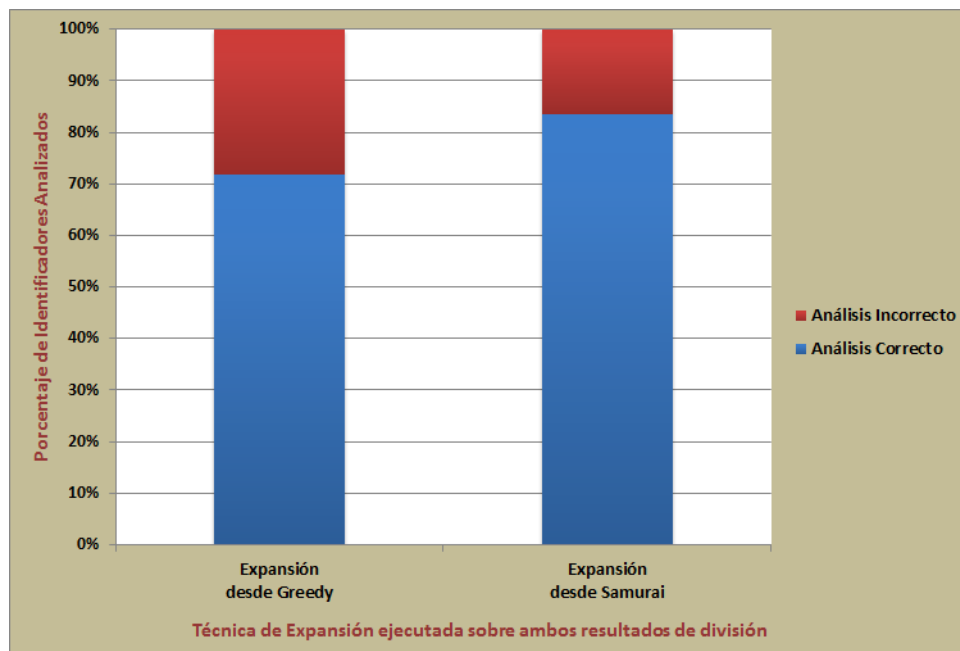


Figura 5.5: Porcentajes de Ids expandidos correctamente e incorrectamente

Por otro lado, en la Figura 5.5, se aprecia una comparativa similar a la anterior, pero esta vez es entre las expansiones realizadas con la técnica de expansión básica, que recibe como entrada los dos grupos de resultados de las técnicas de división (Expansión desde Greedy y Samurai). Como se puede observar en la Figura 5.5, la expansión de los ids que dividió Greedy tienen más casos incorrectos que las expansiones de los ids que dividió Samurai. Esto tiene sentido dado que mientras más acertada esté la división del id, mayor acierto tendrá la expansión. Si se compara las expansiones realizadas con las divisiones (Entre la Figura 5.4 y 5.5), en el caso de las expansiones existe un aumento notable de casos incorrectos. Este comportamiento ocurre por que el algoritmo de expansión es básico y posee ciertos inconvenientes, por ejemplo, no expande una abreviatura ante múltiples posibilidades de expansión, en algunos casos utiliza un diccionario de palabras en lenguaje natural, el cual es extenso, ocupa mucho espacio y deriva en expansiones imprecisas (Ver Capítulo 3 - sección 3.4.4). Por último, las expansiones efectuadas luego de ser procesadas por el algoritmo Greedy son las que tienen mayor fallo, tomando en cuenta las dos figuras, esto ocurre debido a que se ejecutaron las dos

técnicas más básicas y que más inconvenientes tienen (Greedy y Expansión Básica).

Habiendo explicado los casos de estudio que muestran la utilidad de la herramienta IDA, en el próximo y último capítulo de este trabajo final, se describen las conclusiones obtenidas.

Capítulo 6

Conclusiones y Trabajos Futuros

6.1. Conclusiones

En este último Capítulo, se describen las conclusiones obtenidas luego de la realización de este trabajo final, más adelante se proponen trabajos futuros a realizar.

Las conclusiones que se llevaron a cabo, están relacionadas a:

- La Investigación sobre el Análisis de Identificadores.
- La Construcción de la Herramienta Identifier Analyzer (IDA).
- Los Casos de Estudio probados en IDA.

A continuación, se desarrollan cada uno de estos ítems.

6.1.1. La Investigación sobre el Análisis de Identificadores

Luego de haber realizado un extenso estudio sobre la temática de análisis de identificadores (el cual fue descrito en el Capítulo 3), se arribaron a las conclusiones que se describen a continuación.

Si bien existen diferentes técnicas de división de identificadores (Gent-Test, Dynamic Time Warping, Identifier Name Tokeniser Tool, entre otros) los algoritmos más populares son Greedy y Samurai. El primero porque es sencillo de implementar, y además es muy utilizado como medida de comparación con técnicas más avanzadas [30, 31, 43, 37]. El segundo porque es un algoritmo bastante automatizado y según el autor [29] tiene un mejor desempeño que la mayoría de las técnicas conocidas que separan ids.

En lo que respecta a estrategias de expansión de ids, la técnica básica de expansión es muy sencilla y trae como consecuencias expansiones incorrectas. Esto ocurre debido a que las abreviaturas con pocas letras, generalmente se expanden por medio de diccionarios con palabras en lenguaje natural. Estos diccionarios, normalmente contienen amplias variedades de palabras que no están muy relacionadas al dominio del problema y tampoco a las ciencias de la computación. Siguiendo con las técnicas investigadas, el algoritmo de expansión Automatically Mining Abbreviation Expansions in Programs (AMAP) [38] contiene mejoras notables con respecto al algoritmo de Expansión Básico. Con la estrategia AMAP, se expanden abreviaturas de manera más precisa y sin utilizar amplios diccionarios, lo que conlleva a tener resultados más efectivos.

Algunas técnicas de análisis de identificadores emplean diccionarios/listas para realizar su trabajo, en el estado de arte descrito en el capítulo 3 se encuentran algunos ejemplos de este tipo de técnicas, la herramienta Identifier Restructurer, el Algoritmo Greedy y el Algoritmo de Expansión Básico. Los mismos utilizan diccionarios/listas extensos lo que conlleva a un gasto alto en espacio y a una precisión baja de aciertos en sus resultados. A medida que se avanzó en la investigación, se pudo comprobar la existencia de técnicas más nuevas como Samurai y AMAP que disminuyen el espacio utilizado y aumentan la efectividad de sus tareas. La forma en que consiguen esto, es explotando más los recursos propios del sistema, y no apoyándose tanto en recursos externos. Esto a su vez permite, que las técnicas Samurai y AMAP sean más escalables a nuevas tecnologías y a nuevos vocablos en el ámbito de la ciencias de la computación.

6.1.2. La Construcción de la Herramienta IDA

Luego de haber construido la herramienta Identifier Analyzer (IDA) (que fue descrita en el Capítulo 4), se puede concluir que, implementar técnicas de análisis de identificadores requiere de la aplicación de diferentes conocimientos allegados a áreas dentro de las ciencias de la computación, tales como:

Lenguajes de Programación: Se utilizaron conceptos de lenguajes de programación, relacionados a la sintaxis y la semántica. Se hizo hincapié en el lenguaje JAVA, y de como los nombres de los ids impactan en la comprensión de los sistemas.

Base de Datos: Se adquirieron conocimientos para seleccionar un motor de base de datos adecuado. El mismo, debe mantener y gestionar los diccionarios/listados de palabras para que las técnicas que utiliza IDA, funcionen lo más eficientemente posible.

Sistemas Operativos: Dado que IDA emplea un programa externo para embellecer el código de entrada, se necesitó investigar como realizar las llamadas a programas por línea de comandos, según el Sistema Operativo que se utilice.

Ingeniería del Software: Se implementaron técnicas de ingeniería inversa, IDA recibe un código de entrada, y retorna una tabla descriptiva con los ids analizados, para lograrlo se adquirieron conocimientos sobre técnicas para construir un Analizador Sintáctico, con el objeto de capturar ids, comentarios y literales.

La selección del lenguaje Java para la implementación de la herramienta IDA resultó apropiada porque dicho lenguaje posee un conjunto amplio de librerías útiles. Algunas de las librerías escritas en JAVA, que se incorporaron en IDA son: ANTLR¹, OpenCloud², HSQLDB³.

¹ANother Tool for Language Recognition. <http://wwwantlr.org>

²<http://opencloud.mcavallo.org>

³Hyper SQL Data Base. <http://www.hsqldb.org>

La interfaz gráfica de IDA fue construida de manera tal, de que sea simple de usar. Para lograr tal objetivo fue necesario la implementación de técnicas de visualización tanto textuales, como gráficas. Las estrategias antes mencionadas posibilitaron que la herramienta IDA tenga un cuadro con el código leído desde el archivo, este código se resalta con color para destacar los distintos elementos que lo componen, en sintonía con la interacción con el usuario. Además se requirió llevar a cabo, una correcta distribución de las tablas, ventanas, botones, menús y todos los componentes visuales que la herramienta IDA tiene. Con esto se intentó hacer una interfaz lo más sencilla posible.

6.1.3. Los Casos de Estudio probados en IDA

Los casos de estudios presentados en este trabajo final, son programas que han sido implementados por otros programadores y por lo tanto son susceptibles a tareas de comprensión. En el caso que se requieran hacer futuras modificaciones, se sabe como funcionan.

Los resultados que arrojaron la ejecución de estos casos de estudio en IDA, indican que la técnica Samurai fue más efectiva que Greedy. Una causa de esto, se debe a que Greedy es una técnica que siempre tiende a dividir a los ids en mayor proporción que lo deseado, y esto se reflejó en los resultados de los casos de estudio. La mayoría de los resultados incorrectos de Greedy fueron causados por el exceso de separaciones. Otro aspecto que influyó en los resultados, es que Greedy basa sus criterios de división, solo en la información provista por diccionarios/listados de palabras predefinidos, sin tener en cuenta la información provista en el código (comentarios y literales), que es propia del dominio del problema.

Las expansiones de las abreviaturas de los ids, si se observan los casos fallidos de expansión, los mismos fueron causados por dos motivos principales: cuando la división previa no se efectuó correctamente, y cuando no se encontraron palabras candidatas completas dentro de los comentarios y literales ubicados en el código del programa. En muy pocos casos hubo acierto en la expansión, cuando la fuente de búsqueda era el diccionario de palabras en lenguaje natural. Esto apoya la teoría de que estos diccionarios son

imprecisos cuando se trata de analizar ids.

A través de los casos de estudio, se pretendió mostrar que IDA es útil a la hora de comprender un sistema por medio del análisis de los ids. Como se describió en capítulos anteriores, el principal objetivo de la CP es relacionar el Dominio del Programa y el Dominio del Problema. En los tres casos de estudio presentados en el capítulo anterior, la información obtenida producto de las expansiones acertadas de los ids revela, en general, que es propia del Dominio del Problema. Por ende, IDA sirve como aporte a la CP y como punto de partida para desarrollar nuevas herramientas, que faciliten el entendimiento de los ids en los códigos de los sistemas de software.

Habiendo descripto las conclusiones pertinentes, en la próxima sección se proponen algunos trabajos futuros a realizar.

6.2. Trabajos Futuros

En esta sección se describen propuestas vinculadas a trabajos futuros de la herramienta IDA. Se tomará como punto de partida el actual estado de desarrollo de IDA y en función de este estado, se proponen mejoras y/o expansiones. Los trabajos futuros propuestos son:

- Ampliar la Captura del Analizador Sintáctico.
- Implementar otro Algoritmo de Expansión.
- Expandir Identificadores en el Código.
- Acoplar a Entornos de Desarrollo.

A continuación, se describen cada uno de ellos.

6.2.1. Ampliar la Captura del Analizador Sintáctico

El Analizador Sintáctico (AS) actual que posee la herramienta IDA no se capturan todos los ids dentro del código, solo se extraen los ids en su punto de declaración. Esto quiere decir, que todos los ids que se declaran son

capturados (ya sean locales dentro de una función, dentro de una estructura de control (if, while, etc.) o una variable de clase), pero no se extraen las ocurrencias, es decir, el uso del id. Desarrollar un AS que también capture las ocurrencias de los ids, implica un gran esfuerzo debido a la complejidad del mismo. Esta conclusión, se determinó mediante la experiencia obtenida producto de la construcción del actual AS de IDA. Sin embargo, para el caso de los comentarios y literales, el AS los extrae en forma completa.

En función de lo antedicho, una propuesta a futuro es ampliar el AS para que capture todos los ids presentes en los archivos JAVA. Con esto, se brindará al usuario más y mejor información estática asociada a los ids. Por otro lado, se perfeccionarán las técnicas de análisis de ids. Sin duda, la técnica más beneficiada en este sentido será Samurai, más precisamente en las tablas de frecuencias de aparición de palabras, a continuación se explican más detalles al respecto.

Las dos tablas de frecuencias de aparición de palabras que son utilizadas por el Algoritmo Samurai en la función score, son la tabla de frecuencias local y la tabla de frecuencias global (Ver Capítulo 3 - sección 3.4.3). En la tabla local se considera la frecuencia de aparición de palabras en el código de estudio actual, mientras que la global, se asocia a un conjunto grande de programas externos. A continuación, se explican las mejoras que recibiría cada tabla, si el AS se amplía:

Tabla de Frecuencia Local: Intuitivamente, cada palabra en esta tabla que esté asociada a un id¹, tendrá la frecuencia de aparición local mucho más precisa dado que se capturan todas las ocurrencias del id.

Tabla de Frecuencia Global: Esta tabla originalmente fue construida por los autores, a partir del análisis de 9000 programas JAVA (Ver Capítulo 3 - sección 3.4.3), y la misma no está disponible. Por lo tanto, se tomó la iniciativa de construir una aproximación. Esta aproximación se efectuó ejecutando el mismo AS que se utiliza en IDA, pero para 20 programas JAVA tomados como muestra. Una vez capturadas las

¹Producto de la separación Hardword (easyCase se obtienen las palabras easy case, por ejemplo).

palabras provistas por el AS de ids, comentarios y literales, se calculó la frecuencia de aparición de cada una. Teniendo en cuenta que este AS se va ampliar para que capture todos los ids del código. Si se corre nuevamente el AS para el mismo lote de 20 programas antedicho, se mejorará aun más la precisión de la tabla de frecuencias globales construida, ya que la cantidad de palabras vinculadas a ids serán más precisas. A su vez, como mejora extra, se puede correr el AS para una cantidad de programas superior a 20 y de esta manera tener mayor variedad de palabras.

6.2.2. Implementar otro Algoritmo de Expansión

Esta propuesta consiste en implementar una nueva técnica de análisis de ids en IDA, más precisamente un nuevo algoritmo de expansión. Un algoritmo interesante es AMAP (Automatically Mining Abbreviation Expansions in Programs) descrito en el capítulo 3 - sección 3.4.5. El mismo, observa gradualmente en el código las palabras presentes partiendo desde el lugar del id que se desea expandir. La búsqueda de palabras candidatas para expandir las abreviaturas comienza en un alcance cercano (palabras ubicadas en sentencias cercanas a la abreviatura), luego en caso de no tener éxito, el alcance se va expandiendo a los métodos (sentencias, comentarios de métodos, etc.), clases (comentarios de clase, variables de clase, etc.). En caso de ser necesario, también puede explorar palabras contenidas en el sistema completo, librerías estándar de JAVA y/o otros programas JAVA. Con estas características, la técnica AMAP no necesita grandes diccionarios con palabras en lenguaje natural como el caso del algoritmo Básico de Expansión. Además, AMAP, a diferencia del algoritmo de expansión convencional, posee criterios de selección inteligentes que permiten elegir la mejor expansión para una abreviatura, ante muchas posibilidades de expansión. Esta propuesta para la herramienta IDA brindaría al usuario mejores expansiones a los ids y una nueva opción en lo que respecta al análisis de ids.

```
public Person(int doc, String name, int age, String gen) {
    setDocPrs(doc);
    setNm(name);
    setAg(age);
    setGndr(gen);
}

public void setDocPrs(int value) {
    this.docPrs = value;
}

public Person(int document, String name, int age, String gender) {
    set_document_person(document);
    set_name(name);
    set_age(age);
    set_gender(gender);
}

public void set_document_person(int value) {
    this.document_person = value;
}
```

Figura 6.1: Comparación de un código, antes y después de expandir los Ids

6.2.3. Expandir Identificadores en el Código

Para ubicarse en el contexto de esta mejora, IDA tiene un panel (Panel de Elementos Capturados - Ver Capítulo 4) en donde se visualiza el código del archivo ingresado para que el usuario lo visualice.

Una propuesta de mejora en la herramienta IDA, consiste en traducir los ids que se muestran en esta visualización del código. Esta traducción implica reemplazar cada id ubicado en el código por la expansión que fue llevada a cabo, dado que hay dos tipos de expansiones por cada id (expansión desde Greedy/Samurai), lo que se permitirá es que el usuario pueda elegir entre ambas alternativas la que mejor le parezca. De esta forma, se obtendrá un código más legible y ayudará a comprenderlo más fácilmente. En la Figura 6.1 se explyea esta idea, aquí se compara un código, antes y después de expandir los ids.

Luego el nuevo código con los ids expandidos se podrá guardar en un nuevo archivo de salida JAVA, este nuevo archivo será funcionalmente equivalente al archivo ingresado, pero tendrá los ids expandidos. Esta idea de traducción y creación de un nuevo archivo, fue tomada de una de las características que posee la herramienta Identifier Restructuring cuyos autores son

Tonella y Caprile (Ver capítulo 3 - sección 3.4.6), ya que esta herramienta realiza una traducción similar de ids generando un código más comprensivo.

6.2.4. Acoplar a Entornos de Desarrollo

Una interesante extensión futura para la herramienta IDA, consiste en adaptarla como extensión (plugin) para un entorno de desarrollo integrado, como es el caso de NetBeans o Eclipse. Esto permitiría que el usuario abra un proyecto JAVA desde estos entornos, e inmediatamente con IDA expanda los ids para mejorar la comprensión. Esta propuesta, en parte es similar a una de las características de la herramienta Identifier Dictionary (IDD), que fue desarrollada por Deissenboeck y Pizka (Ver Capítulo 3 - sección 3.3.3). La herramienta IDD es un plugin de eclipse que al compilar un proyecto en JAVA, automáticamente captura y enumera dentro de una tabla los ids presentes en el proyecto, luego el usuario puede renombrar cada id desde esta tabla a una forma más comprensiva.

Para la herramienta IDA se propone construir un plugin similar al de IDD, en donde se enumeran los ids en una tabla, pero el renombre de ids en IDA a diferencia de IDD es más automático, ya que IDA expande los ids por medio de las estrategias que tiene implementadas. El usuario solo deberá intervenir para determinar que expansión es la más adecuada, entre los distintos resultados que se obtengan, producto de las diferentes estrategias de análisis de ids ejecutadas.

Apéndice A

A.1. Extensión de la Herramienta IDA

La herramienta Identifier Analyzer (IDA) posee una característica adicional. Dicha característica permite recibir como parámetro un archivo XML (Extensible Markup Language). Este archivo debe contener información asociada a identificadores (ids), literales y comentarios (propia de una aplicación JAVA). La herramienta IDA lee la información provista por el archivo xml y ejecuta directamente los algoritmos de análisis de ids que tiene implementados. Por último, los resultados de la ejecución se almacenan en otro archivo XML que será creado en la misma ruta que el archivo leído como entrada (Ver Figura A.1).

IDA soporta integración por medio de archivos XML. Esta característica es una ventaja que posibilita, compartir información con otras aplicaciones (herramientas) sin tener problemas de compatibilidad, dado que los archivos XML permiten un intercambio de datos estándar entre aplicaciones. De esta forma, IDA puede formar parte de un proceso de análisis más extenso que involucre otras herramientas asociadas a la comprensión de sistemas.

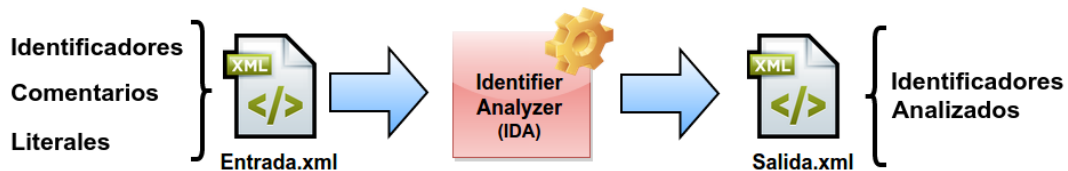


Figura A.1: Extensión de IDA

La invocación de IDA con un archivo XML se realiza a través del archivo JAR¹ (correspondiente a la herramienta), ejecutando la siguiente orden en la línea de comandos del sistema operativo²:

```
java -jar IDA.java <argumento>
```

En <argumento> se coloca la ruta donde se encuentra ubicado el archivo XML a procesar (un ejemplo en linux es `/home/entrada.xml`³). Este argumento no es obligatorio, y en caso de no pasarlo, se ejecuta la interfaz normal de IDA que fue descripta en el capítulo 4. La herramienta IDA procesa el archivo XML ingresado ejecutando los algoritmos de análisis de ids que tiene implementados (Greedy, Samurai y Expansión Básica) y produce los resultados correspondientes a cada uno de ellos. A continuación, se describe como debe estar estructurado el archivo XML de entrada.

Archivo XML de Entrada

El archivo XML de entrada debe comenzar con <entrada> y finalizar con </entrada> (Ver Figura A.2), entre las etiquetas antes mencionadas se pueden especificar los siguientes elementos:

Lista de Identificadores: Contiene los ids que van a ser analizados, se delimita con <lista_ids> y </lista_ids>; cada elemento de esta lista se especifica con <id> y </id>; dentro de cada uno de estos elementos se coloca: el nombre del id con <nombre>**nmId**</nombre>, y el número de línea con <linea>**10**</linea> (Ver Figura A.2).

Lista de Frases: Tiene las frases (asociadas a comentarios y literales) el inicio y fin de esta lista se indica con <lista_frases> y </lista_frases>; cada elemento de esta lista se delimita con <frase> y </frase>; dentro de cada uno de estos elementos de la lista se especifica: la frase correspondiente con <texto>**File System**</texto>, y el número de línea

¹JAVA Archive.

²Se recomienda utilizar los sistemas operativos Windows o UNIX con *java runtime enviroment* instalado

³No es necesario que se llame entrada, pero si que tenga extensión xml.

con `<línea>19</línea>` (Ver Figura A.2).

Lista de Clases: Esta lista contiene las clases que posee el programa, se delimita con `<lista_clases>` y `</lista_clases>`; cada elemento de esta lista se indica con `<clase>` y `</clase>`; cada uno de estos elementos tiene: el nombre del método `<nombre>Person</nombre>`, el número de línea donde comienza la clase `<línea_inicio>7</línea_inicio>`, y la línea donde finaliza la clase `<línea_fin>30</línea_fin>` (Ver Figura A.2).

Lista de Métodos: Similar a la lista anterior pero para métodos, se delimita con `<lista_metodos>` y `</lista_metodos>`; cada elemento de este listado se indica con `<metodo>` y `</metodo>`; en cada uno de estos elementos se coloca: el nombre del método `<metodo>getPerson</metodo>`, la línea donde comienza el método `<línea_inicio>11</línea_inicio>`, y la línea donde finaliza la método `<línea_fin>19</línea_fin>` (Ver Figura A.2).

Cabe aclarar, que los nombres de los ids (Ver Figura A.2) son el único dato que necesita ser especificado de manera obligatoria en el archivo proporcionado a IDA (dado que son la principal fuente de análisis de IDA). Por otro lado, el resto de los datos: Métodos, Clases, Frases, números de líneas (de cualquier elemento), también son importantes, pero solo colaboran con el análisis de los ids y no son indispensables.

Luego de que IDA analiza los ids, el próximo paso es almacenar los resultados de cada ejecución en un nuevo archivo XML de salida que se describe en la siguiente sección.


```
<entrada>
  <lista_ids>
    <id>
      <nombre>nmId</nombre>
      <linea>10</linea>
    </id>
    <id>
      <nombre>fs</nombre>
      <linea>12</linea>
    </id>
  </lista_ids>
  <lista_frases>
    <frase>
      <texto>name identifier</texto>
      <linea>9</linea>
    </frase>
    <frase>
      <texto>file system</texto>
      <linea>19</linea>
    </frase>
  </lista_frases>
  <lista_clases>
    <clase>
      <nombre>Person</nombre>
      <linea_inicio>7</linea_inicio>
      <linea_fin>30</linea_fin>
    </clase>
  </lista_clases>
  <lista_metodos>
    <metodo>
      <nombre>getPerson</nombre>
      <linea_inicio>11</linea_inicio>
      <linea_fin>19</linea_fin>
    </metodo>
  </lista_metodos>
</entrada>
```

Figura A.2: Ejemplo de Archivo XML de entrada.

Archivo XML de Salida

El archivo de salida contiene los ids analizados por las distintas técnicas y se crea en la misma ubicación que el archivo XML pasado por entrada (siguiendo con el ejemplo de la sección anterior se creará en `/home/salida.xml`¹). El inicio de este archivo XML se indica con `<salida>` y el fin con `</salida>` (Ver Figura A.3), en su interior posee la siguiente lista:

Lista de Identificadores Analizados: Esta lista contiene los ids incluyendo el análisis realizado en cada uno, el inicio y fin de esta lista se especifica con `<lista_analisis_ids>` y `</lista_analisis_ids>`; cada elemento de la lista se indica con `<id>` y `</id>`; dentro de cada elemento de la lista se encuentra: el nombre del id analizado delimitado con `<nombre>nmId</nombre>`, la división greedy del id se ubica entre `<div_greedy>nm-id</div_greedy>`, la división samurai del id entre `<div_samurai>nm-id</div_samurai>`, la expansión desde greedy entre `<exp_greedy>name identifier</exp_greedy>`, y la expansión desde samurai entre `<exp_samurai>name identifier</exp_samurai>` (Ver Figura A.3).

```
<salida>
  <lista_analisis_ids>
    <id>
      <nombre>nmId</nombre>
      <div_greedy>nm-id</div_greedy>
      <div_samurai>nm-id</div_samurai>
      <exp_greedy>name identifier</exp_greedy>
      <exp_samurai>name identifier</exp_samurai>
    </id>
    <id>
      <nombre>fs</nombre>
      <div_greedy>fs</div_greedy>
      <div_samurai>fs</div_samurai>
      <exp_greedy>file system</exp_greedy>
      <exp_samurai>file system</exp_samurai>
    </id>
  </lista_analisis_ids>
</salida>
```

Figura A.3: Ejemplo de Archivo XML de salida.

¹Si ya existe un archivo con el nombre `salida.xml`, el mismo se sobrescribirá.

Bibliografía

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. ISBN: 978-0321486813.
- [2] Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. *Data structures and algorithms / Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman*. Addison-Wesley Reading, Mass, 1983. ISBN: 978-0201000238.
- [3] José Luís Albanes, Mario Berón, Pedro Henriques, and Maria João Pereira. Estrategias para relacionar el dominio del problema con el dominio del programa para la comprensión de programas. *XIII Workshop de Investigadores en Ciencias de la Computación*, pages 449–453, 2011.
- [4] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON '98: Proc. 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 4. IBM Press, 1998.
- [5] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.
- [6] Lerina Aversano, Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Migrating legacy systems to the web: An experience report. In *Fifth European Conference on Software Maintenance and Reengineering*, page 148–157, 2001.

-
- [7] Lerina Aversano, Gerardo Canfora, and Silvio Stefanucci. Understanding and improving the maintenance process: A method and two case studies. In *International Workshop on Program Comprehension*, page 199–208. IEEE Computer Society, 2001.
 - [8] Javier Azcurra, Mario Berón, Pedro Henriques, and Maria João Pereira. Análisis de información informal para facilitar la comprensión de programas. *XIV Workshop de Investigadores en Ciencias de la Computación*, pages 597–601, 2012.
 - [9] Javier Azcurra, Mario Berón, Germán Montejano, Augusto Farnese, Pedro Henriques, and Maria João Pereira. Aid: uma ferramenta para análise de identificadores de programas java. *Congreso Nacional de Ingeniería Informática/Sistemas de Información*, pages 880–892, 2014.
 - [10] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, page 216–234. Springer, 1999.
 - [11] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer, Bell Labs., 1000 East Warrenville Road, Naperville IL 60566*, 29(4):33–43, April 1996.
 - [12] Keith H. Bennett and Vaclav T. Rajlich. Software maintenance and evolution: a roadmap. In *Conference on The Future of Software Engineering*, page 73–87, Limerick, Ireland, June 2000. ACM Press.
 - [13] Mario Berón, Pedro Henriques, Maria João Pereira, and Roberto Uzal. Program inspection to interconnect behavioral and operational view for program comprehension. In *1st York Doctoral Symposium on Computing. York*. University of York, 2007.
 - [14] Mario Berón, Pedro Henriques, and Roberto Uzal. *Inspección de Programas para Interconectar las Vistas Comportamentaly Operacional para la Comprensión de Programas*. PhD thesis, Universidade do Minho, Braga, Portugal, 2010.

-
- [15] Mario Berón, Pedro Rangel Henriques, Maria João Varanda Pereira, and Roberto Uzal. Herramientas para la comprensión de programas. In *VIII Workshop de Investigadores en Ciencias de la Computación*, 2006.
 - [16] Mario Berón, Daniel Riesco, Germán Montejano, Pedro Rangel Henriques, and Maria J Pereira. Estrategias para facilitar la comprensión de programas. In *XII Workshop de Investigadores en Ciencias de la Computación*, 2010.
 - [17] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.
 - [18] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelCase or Under_score. In *International Conference on Program Comprehension*, page 158–167. IEEE, May 2009.
 - [19] Ruven Brook. A theoretical analysis of the role of documentation in the comprehension of computer programs. *Proceedings of the 1982 conference on Human factors in computing systems.*, pages 125–129, 1982.
 - [20] Evangeline Burch and Hsiang-Jui Kungs. Modeling software maintenance requests: a case study. In *International Conference on Software Maintenance.*, pages 40–47. IEEE, IEEE Computer Society, 1997.
 - [21] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: an empirical study. In *14th European Conf. on Software Maintenance and Reengineering*, page 159–168. IEEE Computer Society, 2010.
 - [22] Bruno Caprile and Paolo Tonella. Nomen est omen: analyzing the language of function identifiers. In *Proc. Sixth Working Conf. on Reverse Engineering*, page 112–122. IEEE, October 1999.

-
- [23] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *International Conference on Software Maintenance*, page 97–107. IEEE, 2000.
 - [24] Stephen Cook, He Ji, and Rachel Harrison. Dynamic and static views of software evolution. In *International Conference on Software Maintenance*, page 592–601, 2001.
 - [25] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.
 - [26] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *International Conference on Software Maintenance*, pages 602–611, 2001.
 - [27] Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999. ISBN: 978-0805317558.
 - [28] David W Embley. Toward semantic understanding: an approach based on information extraction ontologies. In *Proceedings of the 15th Australasian database conference-Volume 27*, pages 3–12. Australian Computer Society, Inc., 2004.
 - [29] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *6th IEEE International Working Conference on Mining Software Repositories.*, page 71–80. IEEE, may. 2009.
 - [30] Henry Feild, David Binkley, and Dawn Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications.*, 2006.
 - [31] Henry Feild, David Binkley, and Dawn Lawrie. Identifier splitting: A study of two techniques. In *Proceedings of the Mid-Atlantic Student*

- Workshop on Programming Languages and Systems.*, pages 154–160, 2006.
- [32] Fangfang Feng and W. Bruce Croft. Probabilistic techniques for phrase extraction. *Information processing & management*, 37(2):199–220, 2001.
- [33] Ruben Fonseca, Daniela Da Cruz, Pedro Rangel Henriques, and MJ Varranda Pereira. How to interconnect operational and behavioral views of web applications. In *The 16th IEEE International Conference on Program Comprehension*, pages 263–267. IEEE, 2008.
- [34] José Luís Freitas, Daniela da Cruz, and Pedro Rangel Henriques. The role of comments on program comprehension. In *Informatics Department, Universidade do Minho, Braga, Portugal*, 2008.
- [35] Wilhelm Hasselbring, Andreas Fuhr, and Volker Riediger. First international workshop on model-driven software migration (mdsm 2011). In *15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 299–300, Washington, DC, USA, März 2011. IEEE Computer Society.
- [36] Chengwan He, Zheng Li, and Keqing He. Identification and extraction of design pattern information in java program. In *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 828–834. IEEE, 2008.
- [37] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 2013.
- [38] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 5th Int’l Working Conf. on Mining Software Repositories.*, page 79–88. ACM, 2008.

-
- [39] Adrian Iasinschi and Mirel Cosulschi. Semi-automated wrappers using rule trees. In Viorel Negru, Tudor Jebelean, Dana Petcu, and Daniela Zaharie, editors, *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing.*, page 209–215. IEEE Computer Society, 2008.
 - [40] IEEE. Standard glossary of software engineering terminology, iee std 610.12-1990. *New York: Institute of Electrical and Electronics Engineers*, volume 1, 1990.
 - [41] IEEE. Ieee standard for software maintenance, iee std 1219-1998. *New York: Institute of Electrical and Electronics Engineers*, 1998.
 - [42] Leah S. Larkey, Paul Ogilvie, M. Andrew Price, and Brenden Tamilio. Acrophile: an automated acronym extractor and server. In *Proceedings of the fifth ACM conference on Digital libraries*, page 205–214, 2000.
 - [43] Dawn Lawrie and David Binkley. Expanding identifiers to normalize source code vocabulary. In *International Conference on Software Maintenance*, page 113–122. IEEE, 2011.
 - [44] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, page 139–148. IEEE Computer Society, 2006.
 - [45] Dawn Lawrie, Henry Feild, and David Binkley. An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229, 2007.
 - [46] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, page 213–222, Sept 2007.

-
- [47] Dawn Lawrie, Henry Feild, and David Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388, 2007.
 - [48] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What’s in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension*, page 3–12, June 2006.
 - [49] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.
 - [50] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In Irvin R. Katz, Robert L. Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen, editors, *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 480–486. ACM/Addison-Wesley, 1995.
 - [51] Welf Löwe, Morgan Ericsson, Jonas Lundberg, and Thomas Panas. Software comprehension-integrating program analysis and software visualization. *Software Engineering Research and Practice (SERPS)*, 2002.
 - [52] Michael P O’Brien. Software comprehension—a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report.*, 2003.
 - [53] Maria João Varanda Pereira, Mario Marcelo Beron, Daniela Carneiro da Cruz, Nuno Oliveira, and Pedro Rangel Henriques. Problem domain oriented approach for program comprehension. In Alberto Simões, Ricardo Queirós, and Daniela Carneiro da Cruz, editors, *Symposium on Languages, Applications. Universidade do Minho, Portugal*, volume 21 of *OASICS*, page 91–105. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
 - [54] Marian Petre, Ed de Quincey, et al. A gentle overview of software visualisation. *PPIG News Letter*, pages 1–10, 2006.

-
- [55] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 5 edition, 2001. ISBN: 978-8448132149.
 - [56] Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proc. 10th Int'l Workshop on Program Comprehension*, page 271–278. IEEE, 2002.
 - [57] Bonita Sharif and Jonathan I. Maletic. An eye tracking study on camel-case and under_score identifier styles. In *International Conference on Program Comprehension*, page 196–205. IEEE Computer Society, 2010.
 - [58] Maria Joao C. Sousa and H. M. Moreira. A survey on the software maintenance process. In *International Conference on Software Maintenance*, page 265–274, 1998.
 - [59] Thomas A. Standish. *Data structure techniques*. Computer Sciences. Addison-Wesley, 1980. ISBN: 978-0201072563.
 - [60] M-AD Storey, F David Fracchia, and Hausi A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *The Journal of Systems & Software.*, 44(3):171–185, 1999.
 - [61] Margaret-Anne Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension*, page 181–191, May 2005.
 - [62] Alfredo Raúl Teyseyre and Marcelo R. Campo. An overview of 3d software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, 2009.
 - [63] Pierre F. Tiako. Maintenance in joint software development. In *Proceedings. 26th Annual International Computer Software and Applications Conference*, page 1077–1080, 2002.
 - [64] Tim Tiemens. Cognitive models of program comprehension. *Software Engineering Research Center Technical Report*, 1989.

-
- [65] Marco Torchiano, Massimiliano Di Penta, Filippo Ricca, Andrea De Lucia, and Filippo Lanubile. Software migration projects in italian industry: Preliminary results from a state of the practice survey. In *23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops. ASE Workshops.*, page 35–42. IEEE, 2008.
 - [66] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995.
 - [67] YingHui Wang, XiuQing He, and QiongFang Wang. Lifecycle based study framework of software evolution. *International Conference on Computer Application and System Modeling (ICCASM)*, 4:7, 2010.
 - [68] Qian Zhao, Huiqiang Wang, Guangsheng Feng, and Xu Lu. Software evolution method considering software historical behavior. In *Fourth International Conference on Internet Computing for Science and Engineering (ICICSE)*, page 36–41, Dec 2009.
 - [69] Ying Zou. Incorporating quality requirements in software migration process. In *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, page 175–185. IEEE Computer Society, 2003.
 - [70] Ying Zou and Kostas Kontogiannis. Migration to object oriented platforms: A state transformation approach. In *International Conference on Software Maintenance*, pages 530–539. IEEE, 2002.