

Análisis de Identificadores para Abstraer conceptos del Dominio del Problema

Trabajo Final de Licenciatura en Ciencias de
la Computación

Autor

Javier Azcurra Marilungo

Director

Dr. Mario Marcelo Berón

Co-Director

Dr. Germán Antonio Montejano

Departamento de Informática

Facultad de Ciencias Físico Matemáticas y Naturales

Universidad Nacional de San Luis

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 del 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 son uno de tales elementos. Estudios indican [10, 25, 22, 16] 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 con un parser construido a medida. Sin embargo, el análisis del significado de los ids en el código fuente es más complicado por que el nombramiento de estos depende de cada programador. Una estrategia para analizar ids es expandir las abreviatura 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 identificadores de programas escritos en JAVA. IDA implementa técnicas de expansión de las abreviaturas de los identificadores con el propósito de facilitar la comprensión de los sistemas de software.

Índice general

1. Introducción: Problema y Solución	3
1.1. Mantenimiento del Software	3
1.2. Evolución del Software	4
1.3. Migración del Software	5
1.4. Comprensión de Programas	6
1.5. Extracción de Información	7
1.6. Análisis de Identificadores	8
1.7. Problema y la Solución	10
1.8. Contribución	11
1.9. Organización del Trabajo Final	11
2. Comprensión de Programas	12
2.1. Introducción	12
2.2. Modelos Cognitivos	13
2.3. Extracción de Información	15
2.4. Administración de la Información	16
2.5. Visualización de Software	17
2.6. Estrategias de Interconexión de Dominios	18
2.7. Comentarios	19
3. Análisis de Identificadores: Estado del Arte	21
3.1. Introducción	21
3.2. Conceptos claves	23
3.3. Nombramiento de Identificadores	24

3.3.1. Clasificación	24
3.3.2. Importancia del Nombramiento	25
3.3.3. Herramienta: Identifier Dictionary	29
3.4. Traducción de Identificadores	32
3.4.1. Conceptos y Desafíos observados	32
3.4.2. Algoritmo de División: Greedy	34
3.4.3. Algoritmo de División: Samurai	38
3.4.4. Algoritmo de Expansión Básico	44
3.4.5. Algoritmo de Expansión AMAP	46
3.4.6. Herramienta: Identifier Restructuring	57
3.5. Conclusiones	62
4. Identifier Analyzer (IDA)	63
4.1. Introducción	63
4.2. Arquitectura	64
4.3. Analizador Sintáctico (Parser)	66
4.4. Base de Datos Embebida	66
4.5. Proceso de Análisis de Identificadores	68
4.5.1. Barra de Menús	68
4.5.2. Lectura de Archivos JAVA	69
4.5.3. Panel de Elementos Capturados (Parte 1)	70
4.5.4. Panel de Análisis	73
4.5.5. Palabras Capturadas y Diccionarios	74
4.5.6. Panel de Elementos Capturados (Parte 2)	77
4.6. Casos de Estudio	79
4.6.1. Buscaminas (Minesweeper)	79
4.6.2. Ejemplo 2	85
5. Conclusiones	86

Capítulo 1

Introducción: Problema y Solución

Cuando se desarrollan los sistemas 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).

La IS abarca, 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*.

1.1. Mantenimiento del Software

La etapa de mantenimiento es importante en el desarrollo del software, por que está sujeto a cambios y a una permanente evolución [34].

El mantenimiento del software según la IEEE standard [24], *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 [30].

Por lo antedicho, entre otras diversas razones [4] 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 [4, 32, 30, 34] 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*”.

1.2. 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 [4].

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 [30].

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 imperioso 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.

1.3. 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 la IEEE standard [23], *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 son la aparición de nuevas tecnologías web y las mobile (basadas en dispositivos móviles) [36].

La migración de grandes sistemas es fundamental, sin embargo está demostrado que es costosa y compleja [36]. 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 mi-

gración de software, consiste en un modelado de datos [21]. Esta modelado se encarga de abstraer el código antiguo a niveles más altos, al hacerlo reduce los esfuerzos de comprensión. 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.

1.4. 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* [6].

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* [7, 6, 5, 15]. 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 elaborar la siguiente reconstrucción:

I) Armar una representación del *Dominio del Problema*. II) Construir una representación para el *Dominio del Programa*. III) Unir ambas representaciones con una *Estrategia de Vinculación*.

Para llevar a cabo los 3 pasos mencionados, se necesitan estudiar temáticas asociadas a la CP. Las más importantes son:

- 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 permite revelar información oculta en los programas que es valiosa para la CP. 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 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.

1.5. 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 [3]. 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. A veces, se recomienda complementar el uso de ambas estrategias para obtener mejores resultados [13].

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.

1.6. Análisis de Identificadores

Estudios realizados [12, 8, 18, 17] 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 [10, 25, 22, 16] 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 conseguir la traducción antedicha, I) primero se extraen los identificadores del código, II) luego se les aplica técnicas de división en donde se descompone al identificador en las distintas palabras abreviadas que lo componen. Por ejemplo: `inp.fl.sys` \rightarrow `inp fl sys`. Finalmente, III) se emplea 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 [25, 16]. Lamentablemente, estos criterios son desconocidos para las técnicas de expansión de abreviaturas de ids.

Para afrontar la dificultad que implica revelar los nombres abreviados de los ids, se recurre 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 [20]. 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.

En la actualidad se puede clasificar a las estrategias que analizan id en 2 grandes grupos. Las que usan diccionarios predefinidos y las que no. Las primeras son las más arcaicas, mientras que las segundas son las más recientes.

Desafortunadamente, las estrategias de análisis de ids en su gran mayoría, no han sido implementadas en herramientas que integren las distintas características que implica la traducción de ids: dividir las abreviaturas que componen un id, expandirlas en base a lo observado en comentarios, utilizar diccionarios, etc.

1.7. Problema y la Solución

Si bien existen técnicas/herramientas que extraen ids, dividen y expanden abreviaturas. No existen implementaciones que integren todas estas acciones en una única herramienta.

A su vez, tampoco hay herramientas que implementan distintas opciones de estrategias de división y de expansión de abreviaturas. De esta manera, se espera mejorar los resultados obtenidos por las aproximaciones actuales.

Para abordar las iniciativas planteadas y hacer algunos aportes a la solución de estos problemas planteados, se pretende:

- Construir un analizador sintáctico en lenguaje Java que permite extraer los ids, comentarios y literales encontrados en el código fuente del sistema de estudio. La herramienta a seleccionar para realizar esta tarea es ANTLR¹.
- Armar 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 empleando el lenguaje JAVA algunas de las técnicas del ítem anterior en una herramienta denominada *Identifier Analyzer* (IDA). Esta herramienta también permite visualizar atributos del id (ambiente, tipo de identificador, número de línea, etc) mostrando la parte del código donde se encuentra ubicados.
- Comparar el desempeño de cada técnica implementada en IDA y sacar las conclusiones pertinentes.

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

1.8. Contribución

El correspondiente trabajo final es un aporte al área de CP que basa su estudio sobre como capturar conceptos del dominio del problema a través del análisis de los ids en programas escritos en JAVA.

Este análisis consiste en técnicas de traducción de ids a palabras completas considerando distintas fuentes de información informal, como es el caso de los comentarios, los literales o diccionarios predefinidos de palabras. Luego algunas técnicas se implementan en una sola herramienta para comparar el desempeño con distintos casos de estudio.

1.9. 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 explica las técnicas de análisis de identificadores y que fuentes semánticas son utilizadas para su interpretación. El capítulo 4 trata sobre la herramienta *Identifier Analyzer* (IDA) que implementa algunas técnicas de análisis de identificadores sobre códigos escritos en JAVA. En el capítulo 5 se describen las conclusiones elaboradas y posibles trabajos futuros.

Capítulo 2

Comprensión de Programas

s

2.1. Introducción

En el capítulo 1 se explicó una aproximación para ayudar a comprender programas. Esta solución se basa en el análisis de los identificadores encontrados en el código del sistema de estudio.

En este capítulo se definen los conceptos más importantes sobre comprensión de programas (CP). Estos conceptos fueron encontrados en textos que narran sobre CP. Al final se redacta un breve comentario de los temas tratados en el capítulo.

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 está enmarcado en vincular el dominio del problema y el dominio del programa [7, 6, 5, 15]. El primero se refiere al resultado de la ejecución del sistema, mientras que el segundo indica todos los componentes de software involucrados que causaron dicho resultado. La figura 2.1 muestra un modelo de comprensión que refleja lo mencionado previamente.

El inconveniente es que la relación real mostrada en la figura 2.1, es



Figura 2.1: Gráfico de Comprensión de Programas

compleja de reconstruir, por ende se puede bosquejar una aproximación 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.

Para lograr con éxito los pasos antedichos se deben tener en cuenta conceptos muy importantes que son los cimientos sobre los cuales está sustentada la CP: los *modelos cognitivos*, la *extracción de información*, la *administración de la información*, la *visualización de software* y la *interconexión de dominios*.

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. Estos modelos [35, 37, 9, 33] son importantes porque indican de que forma el programador comprende los sistemas y como incorpora nuevos conocimientos (aprende nuevos conceptos).

Los modelos cognitivos están compuestos por: *conocimientos*, *modelos mentales* y *procesos de asimilación* [5].

Existen 2 tipos de *conocimientos* uno es 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 externo que le brinda al programador nuevos conceptos. 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.

El concepto *modelo mental* hace referencia a representaciones mentales que el programador elabora al momento de estudiar el código del sistema. Algunos modelos construidos por los arquitectos de software como por ejemplo el diagrama de clases o el diagrama de casos de uso, entre otros pueden verse como representaciones visuales de modelos mentales.

El *proceso de asimilación* son las 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 híbrida [29, 32].

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.

Por otro lado, 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.

Por último, el proceso *híbrido* combina los dos conceptos mencionados top-down y bottom-up. Durante el proceso de aprendizaje del sistema, el programador combina libremente ambas políticas hasta llegar a comprender

el sistema.

Para concluir sobre la temática de modelos cognitivos: el modelo mental es una representación mental que el programador tiene sobre el sector del sistema que esta analizando. Si esta representación se la vincula con los conocimientos que el propio programador posee, se logra un entendimiento de esa parte del sistema analizado, como así también incrementa los conocimientos del programador. Con esto se deja en claro la importancia de modelos cognitivos en el proceso de entendimiento de los sistemas.

2.3. Extracción de Información

En la ingeniería del software existe un área que se encarga de la *extracción de la información*. Esta extracción se realiza en los sistemas de software y las técnicas utilizadas para tal fin se clasifican según el tipo de información que extraen. Existen dos tipos de informaciones: la Estática y la Dinámica.

La información estática está presente en los componentes del código fuente del sistema. Alguno de ellos son identificadores, procedimientos, comentarios, documentación.

Una estrategia para extraer la información estática consiste en utilizar técnicas tradicionales de compilación [1]. Estas técnicas, utilizan un parser similar al empleado por un compilador. Este parser, por medio de acciones semánticas específicas procede a capturar elementos presentes en el código durante la fase de compilación. En la actualidad, existen muchas herramientas automáticas que ayudan a construir este tipo de parsers.

Por otro lado, la información dinámica se basa en elementos del programa presentes durante una ejecución específica del sistema [3]. Una técnica encargada de extraer 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 modifican 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.

Como se explico en secciones precedentes, para extraer la información

dinámica el sistema se debe ejecutar, mientras que no ocurre lo mismo a la hora de extraer la información estática.

La extracción de información cumple un rol fundamental en la CP. Como se explicó antes, en la CP se destaca la relación entre el dominio del problema y el dominio programa, como esta relación es compleja, se elabora una aproximación mediante el uso de representaciones (ver figura 2.1). Necesariamente, para poder bosquejar dichas representaciones se debe extraer información de ambos dominios.

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, 31]. 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 [14].

Después que se administra la información, se recomienda que la misma sea representada por alguna técnica de visualización. Esto permite esclarecer la información procesada del sistema al programador. El área encargada de

llevar adelante esta tarea es la *visualización de software*.

2.5. Visualización de Software

La *Visualización del Software* es una disciplina importante 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. Esta cualidad es interesante ya que en la actualidad los sistemas son cada vez más amplios y complicados de entender.

La *Visualización del Software* provee varias técnicas para implementar 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. Esto permite, interpretar los programas de manera más clara y ágil. Dependiendo de la información que se desea visualizar existe una vista específica [6].

En la actualidad, existen distintos tipos de sistemas de visualización, algunos autores son: Myers, Price, Roman, Kenneth, Storey [5]. Sin embargo, todos estos sistemas de visualización tienen algo en común, las vistas que son generadas por ellos representan conceptos situados solo en el dominio del programa, restando importancia al dominio del problema y a la relación entre ambos dominios.

Debido a esta falencia Berón [5] propone variantes en los sistemas de visualización orientados a la CP. Estas variantes contemplan la representación visual de los conceptos en el dominio del problema, inclusive la visualización de la relación entre los dos dominios mencionados.

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 (figura 2.1).

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 armar estrategias de interrelación de dominios.

La *Interconexión de Dominios* [7, 5] en la Ingeniería del Software básicamente se atribuye a la transformación y vinculación del dominio de la aplicación de software (dominio del problema) con el dominio del programa (ver figura 2.1). Esta afirmación, como se explico en secciones anteriores es clave en la CP.

El objetivo es que cada componente de un dominio se vea reflejado en uno o más componentes de otro dominio y viceversa.

Un ejemplo 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 usando la información estática y dinámica que se extrajo del sistema de estudio (ver figura 2.1). Como se dijo previamente, esta relación es de mucha importancia porque diagrama la salida del sistema y que elementos vinculados intervinieron.

Una de las técnicas es *Simultaneous Visualization Strategy* SVS, esta se encarga de mostrar los distintos componentes de un programa en plena ejecución, mediante distintas vistas, usando un inspector de sentencias, de esta manera se obtiene una visualización cuando el sistema se está ejecutando. Esta estrategia usa un esquema de instrumentación de código, donde las acciones semánticas le van indicando a un visualizador la traza de invocaciones

a funciones durante la ejecución del sistema [7, 6, 5].

La otra estrategia se denomina *Behavioral-Operational Relation Strategy* BORS, que a diferencia del SVS, espera a que termine la ejecución del sistema y luego la información recopilada por el instrumentador de código es procesada por BORS. Una vez procesada la información, BORS retorna una abstracción gráfica del código ejecutado. Un ejemplo de esta gráfica puede ser el Árbol de Ejecución de Funciones (ver sección anterior). BORS vincula los conceptos del código capturados en tiempo de ejecución con la información asociada al dominio del problema [7, 6, 5].

Existe también, a nivel conceptual una técnica que combina ambas estrategias mencionadas llamada *Simultaneous Visualization Strategy Improved* SVSI. Esta técnica disminuye los problemas que manifiestan tanto SVS como BORS y con ello se logra un mejor desempeño en cuanto a los resultados esperados [7, 6, 5].

2.7. 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 ayuda al programador a entender con facilidad los programas ya que encuentra las partes del sistema que produjeron una determinada salida. Como es demasiado complicado este vínculo, se necesitan estudiar las temáticas pertinentes: los *modelos cognitivos*, la *extracción de información*, la *administración de la información*, la *visualización de software* y la *interconexión de dominios*.

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

En el próximo capítulo se presenta distintas estrategias de análisis que se basan en la extracción de información de los sistemas. El análisis puntual

es sobre los identificadores presentes en el código. Algunas de ellas están implementadas en forma de Herramientas de CP.

Capítulo 3

Análisis de Identificadores: Estado del Arte

En el capítulo anterior se introdujo en 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 describe una conclusión sobre los temas tratados.

3.1. Introducción

Los equipos de desarrollos 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 más complejos y difíciles de entenderlos. He aquí la importancia del uso de las herramientas de comprensión, con ellas se puede lograr un entendimiento ágil y facilitar las arduas tareas de interpretación de códigos.

Como se mencionó en el capítulo anterior la CP brinda métodos, técnicas y herramientas que facilitan al programador entender los programas. Un aspecto importante de la CP es la extracción de información estática. Estas técnicas de extracción no necesitan ejecutar los programas para llevar a cabo la tarea. Una forma de implementarlas es aplicar técnicas de compilación conocidas para extraer información implícita detrás de los componentes visibles en los códigos. Entre los distintos componentes visibles se conocen los ids y los comentarios como principal fuente de información para la CP. Sin embargo, cuando en el código no abundan los comentarios la única fuente importante son los ids.

En la siguiente tabla se muestra un análisis léxico que se realizó sobre 2.7 millones de líneas de códigos escritos en lenguaje JAVA.

Tipo	Cantidad	%	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: Resultado del Análisis Léxico

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 [12, 8]. 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.

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 practica de programación recomienda que un id dentro del código este asociado a un concepto del programa.

Identificador \Leftrightarrow Concepto

Dicho de otra manera un id es un representante de un concepto ubicado en el dominio del problema [12, 8] (ver capítulo 2). Por ejemplo, el id `openWindow` está asociado al concepto ‘abrir una ventana’.

Uno los requisitos importantes que debe reunir un programa para facilitar su comprensión es que sus ids sean claros. Sin embargo, dicho requerimiento no es tenido en muy en cuenta por los programadores [12, 27, 26].

En la siguiente sección se menciona como la semántica de los ids impacta enormemente en la lectura comprensiva de los conceptos asociados y por ende también afecta a la CP.

3.3. Nombramiento de Identificadores

Durante los desarrollos de los sistemas, 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.

Un etapa importante en la vida de los sistemas es el mantenimiento (ver capítulo 2), generalmente el encargado de hacerlo no tiene en cuenta los nombres de los ids para interpretar el código.

Antes de proseguir sobre la importancia del nombramiento, a continuación se clasifican las distintas formas que se puede nombrar un id.

3.3.1. Clasificación

Estudios realizados con 100 programadores [26] sobre comprensión de ids indican que existen tres formas principales de construir (tomando como ejemplo el concepto **File System Input**):

- Palabras completas (`fileSystemInput`).
- Abreviaciones (`flSyslpt`).
- Una sola letra¹ (`fsi`).

De más está decir que los nombres de los ids pueden estar compuestos por más de una palabra como se describió en los ítems anteriores.

Los estudios antedichos arrojaron que las palabras completas son las más comprendidas, sin embargo las estadísticas marcan en algunos casos que las abreviaciones que se ubican en segundo lugar, no demuestran una diferencia notoria con respecto a las palabras completas [26].

Los investigadores Feild, Binkley, Lawrie [17, 18, 8], clasifican los nombres de los ids con 2 términos conocidos en la jerga del análisis de ids: *hardwords* y *softwords*.

Los *hardwords* destacan la separación de cada palabra que compone el identificador a través de una marca específica; algunos ejemplos son: `fileSystem`²

¹Este nombramiento lo llaman acrónimo algunos autores.

²Este nombramiento lo suelen llamar camel-case.

```
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

marca bien la separación de cada palabra con el uso de mayúscula entre las minúsculas o `fileSYSTEM` así también como utilizar 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` se compone 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. En la próxima sección se destacan afirmaciones sobre la importancia de los nombres utilizados en los ids.

3.3.2. Importancia del Nombramiento

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`).
- 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`).

Esto indica que se concentra más en los aspectos sintácticos del id y no tanto en los aspectos semánticos en lo que respecta al nombramiento.

Una evidencia fehaciente de la importancia en el nombramiento semántico 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 [12].

A su vez, los programadores cuando desarrollan sus aplicaciones, restan importancia al correcto nombramiento semántico de 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 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, el mal nombramiento de los ids se combate con la programación “menos egoísta”. Esta consiste en hacer programas más claros y entendibles para el futuro lector que no está familiarizado con el código. Para lograrlo se deben respetar dos reglas en cuanto al nombramiento [12, 27]:

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

Nombramiento Consistente: Para cada id, debe tener asociado si y solo si, un único concepto.

Un ejemplo de conciso es `output_file_name` que representa el concepto de ‘nombre de archivo de salida’, distinto es el nombre `file_name`, el cual no representa de forma semánticamente concisa el concepto mencionado.

Los propiedades que violan el nombramiento consistente en los ids son conocidos en el lenguaje natural con el nombre de sinónimos y homónimos.

Los homónimos son palabras que pueden tener más de un significado. Por ende, si el nombre de un id esta 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 o a una fila en una tabla.

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 [12, 27, 8] que la ausencia de nombramiento consistente tales como se menciono 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 nombrados 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 [12, 27].

Intuitivamente, se necesita que los ids representen bien al concepto, ya que mayor será impacto que tendrá en la interpretación del sistema [12, 27]. Sin embargo, durante las etapas de desarrollo y mantenimiento del software, es muy difícil mantener una consistencia global de nombres en los ids, sobre todo si el sistema es grande. Cada vez que un concepto se modifica el nombre del id asociado debe cambiar y adaptarse a la modificación.

Los autores Deissenboeck y Pizka [12] proponen utilizar una herramienta que solucione los problemas de mal nombramiento planteados anteriormente. Dada la dificultad que conlleva construir una herramienta totalmente automática que se encargue de nombrar correctamente los ids, ellos elaboraron una herramienta semi-automática 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 del 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.

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

Identifier pane

Name	Type	Description	# D	# R
identifierCount	int	<none>	1	3
IdentifierDialog	n.a.	<none>	1	3
identifierElement	Element	<none>	7	18
IdentifierLabelProv...	n.a.	<none>	2	2
identifierList	Dictionary	dictionary	1	1
identifierPersistance	String	method in charge of the ...	1	1
identifiers	DoublyHashed...	contains all the identifier...	1	16
identifiers	Identifier[]	is an array of Identifiers	7	17
IdentifierSorter	n.a.	is the sorter for the ident...	1	2
identifiersWithPrefix	ArrayList	this method returns an a...	1	3
IdentifierTest	n.a.	<none>	1	0

Occurrence pane

Declarations

Resource	in Folder	Location
IddHover.java	\idd\src\edu\tum\	line 90
Dictionary.java	\idd\src\edu\tum\	line 284
DOMSerializer.java	\idd\src\edu\tum\	line 77
IddHover.java	\idd\src\edu\tum\	line 82

References

Resource	in Folder	Location
Dictionary.java	\idd\src\edu\tum'a	line 285
Dictionary.java	\idd\src\edu\tum'a	line 286
DOMSerializer.java	\idd\src\edu\tum'...	line 79
DOMSerializer.java	\idd\src\edu\tum'...	line 80

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 [12] 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 y asiste a escoger un nombre adecuado para los ids siguiendo el patrón de nombres existentes. Aumenta la velocidad de comprensión del código en base a las descripciones de cada id. El equipo encargado de tareas de mantenimiento localiza un componente 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 [12, 25].

Se implementó como extensión de la IDE Eclipse 3.1². Se visualiza en el panel de las vistas de la IDE y consiste de tres secciones (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 un buen nombramiento 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 (Figura 3.3).

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

²<http://www.eclipse.org/jdt>

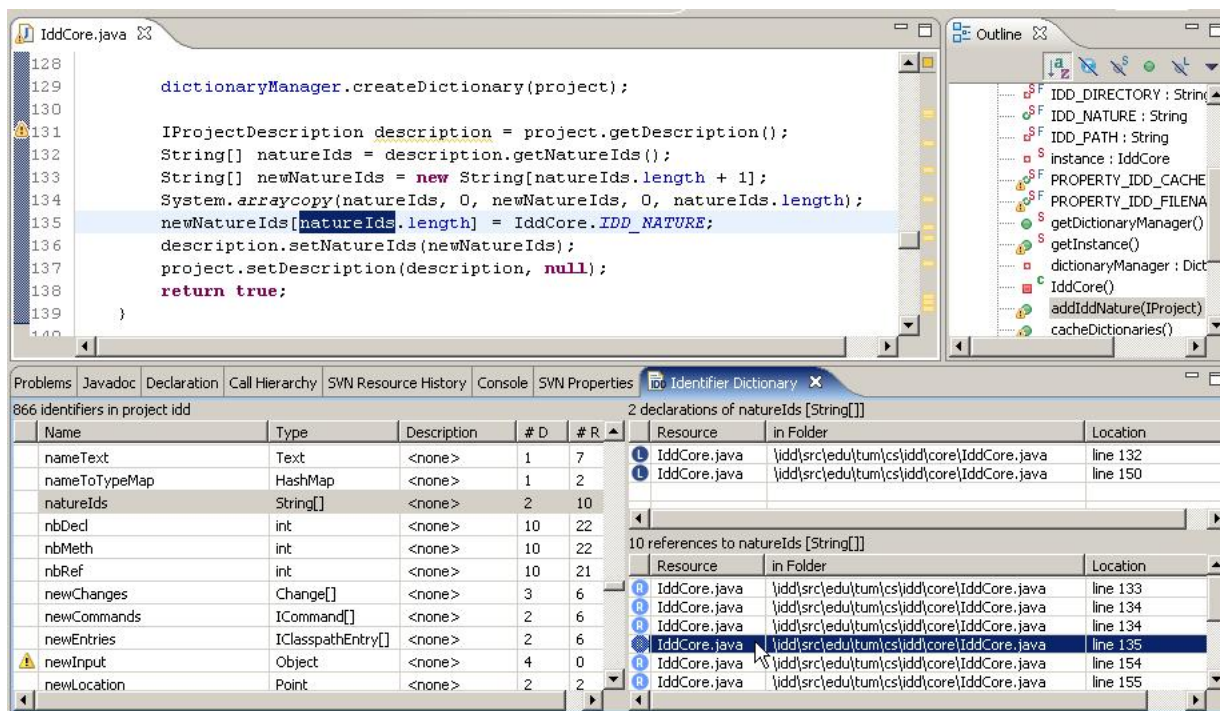


Figura 3.3: Visualización de Identifier Dictionary

Advertencias (warnings): Mientras se realiza la recolección de ids los íconos de advertencia indican potenciales problemas en el nombramiento. 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 posicionando el cursor sobre el id en el código fuente mientras se está programando (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.

¹Similar a los warnings de Eclipse

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


```
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);
```

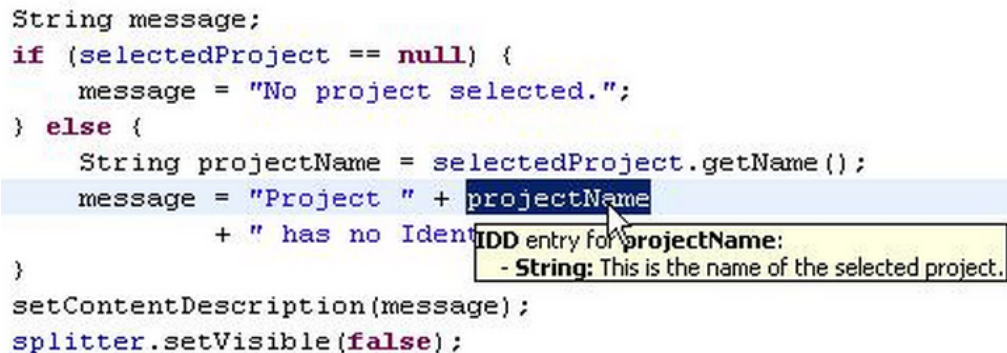


Figura 3.4: Visualización de Identifier Dictionary

Renombre global de ids: Esta función permite renombrar cualquier id generando una vista previa y validando el nombre de los ids a medida que 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 [12].

La herramienta IDD colabora en mejorar el nombramiento de los ids con un esfuerzo moderado como se describió antes.

Sin embargo, los investigadores Feild, Binkley, Lawrie [25, 27] determinaron que 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 el nombramiento de los ids descripta mejora el entendimiento del código. En este sentido, muchos expertos sostienen que las técnicas de Ingeniería Inversa se emplean con mayor precisión si el código esta bien escrito. Algunas de estas técnicas que tiene como objetivo mejorar la CP se encargan de traducir los nombres abreviados de los ids a palabras más completas en lenguaje natural. En la sección siguiente 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, se debe descubrir la información que ocultan los ids detrás de sus abreviaturas. Esta información es relevante ya que pertenece al dominio del problema [16, 25].

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 automático que se lleva a cabo para realizar la traducción de ids consta de dos pasos [25]:

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* (ver sección anterior) 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 contenga más de una palabra.

El objetivo primordial y más difícil 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 [17, 25].

Para afrontar este objetivo los especialistas deciden recurrir a fuentes de palabras en lenguaje natural (inglés en este caso). 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 [22] 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 en Inglés se usan para corregir la ortografía. Las palabras que incluyen son sustantivos propios, abreviaciones, contracciones¹ y demás palabras que puedan aparecer en un software. Sin embargo, la inclusión de muchas palabras genera que una simple abreviación (*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.

Las abreviaciones poseen muchos candidatos a expandir: Es complicado para un id abreviado con `def` determinar con precisión cual es la mejor traducción entre tantos candidatos `definition, default, defect`. Otra observación hecha, es que mientras más corta es la abreviación más candidatos posee, el ejemplo más común es `i` que generalmente es `integer` pero podrían ser otros `interface, interrupt`, etc. Se requieren procesos inteligentes para solucionarlo.

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

El tipo de la abreviación afecta el número de candidatos: Si la abreviatura se mira como prefijo tiene menos candidatos a traducirse, un ejemplo es `str` el cual tiene `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 2 secciones se explican algoritmos encargados de la división de ids, y en la secciones subsiguientes se describen algoritmos de expansión.

3.4.2. Algoritmo de División: Greedy

El algoritmo Greedy elaborado por Lawrie, Feild, Binkley [28, 17, 18, 25, 16] divide las palabras que forman parte de un id, es sencillo y emplea tres listas:

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

Abreviaciones conocidas: La lista se arma con abreviaciones extraídas de distintos programas y de autores expertos. Se incluyen abreviaciones comunes (ejemplo: `alt` → `altitude`) y abreviaciones de programación (ejemplo: `txt` → `text`).

Palabras excluyentes (stop list): Posee palabras que son irrelevantes para realizar la división de los ids. Incluye palabras claves (ejemplo: `while`), ids predefinidos (ejemplo: `NULL`), nombres y funciones de librerías (ejemplo: `strcpy`, `errno`), y todos los ids que puedan tener un solo caracter. Esta lista es muy grande.

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

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

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` → `fileinput` y `txt` en la línea 2, si es camelcase `fileinputTxt` → `fileinput` y `txt` en la línea 3). 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 abreviaciones conocidas - línea 5). Si alguna palabra no está en alguna lista se considera como múltiples softwords que necesitan subdividirse (ejemplo: `fileinput` → `file` y `input` - línea 5). Para subdividir estas palabras se buscan los prefijos y los sufijos más largos posibles dentro de ellas. Esta búsqueda también se realiza utilizando las 3 listas antes mencionada (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 de la función). Cuando una palabra se encuentra en alguna lista (línea 3 de la función) se coloca un separador (' '). El resto que fue descartado se procesa por **buscarPrefijo** para buscar más subdivisiones (línea 4).

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 de la función). 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** (línea 4).

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

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 ← ""
2 softwordDiv ← dividirCaracteresEspecialesDigitos(idHarword)
3 softwordDiv ← dividirCamelCase(softwordDiv)
4 para todo (s | s es un substring separado por ' ' en softwordDiv)
  hacer
5   si (s no pertenece a (stopList ∪ abrevList ∪ ispellList ))
6     entonces
7       resultadoPrefijo ← buscarPrefijo(s, "")
7       resultadoSufijo ← buscarSufijo(s, "")
8       // Se elige la división que mayor particiones hizo.
8       s ← maxDivisión(resultadoPrefijo, resultadoSufijo)
9 devolver softwordDiv // Retorna el id dividido por ' '
```

Función buscarPrefjjo

Entrada: s // Abreviaturas a dividir**Salida** : *abrevSeparada* // Abreviaturas separadas

// Punto de parada de la recursión.

1 **si** ($\text{length}(s) = 0$) entonces2 **devolver** *abrevSeparada*3 **si** (s pertenece a ($\text{stopList} \cup \text{abrevList} \cup \text{ispellList}$)) entonces4 **devolver** ($s + ' ' + \text{buscarPrefjjo}(\text{abrevSeparada}, "")$)// Se extrae y se guarda el último caracter de s .5 $\text{abrevSeparada} \leftarrow s[\text{length}(s) - 1] + \text{abrevSeparada}$

// Llamar nuevamente a la función sin el último caracter.

6 $s \leftarrow s[0, \text{length}(s) - 1]$ 7 **devolver** $\text{buscarPrefjjo}(s, \text{abrevSeparada})$

Función buscarSufijo

Entrada: s // Abreviaturas a dividir**Salida** : *abrevSeparada* // Abreviaturas separadas

// Punto de parada de la recursión.

1 **si** ($\text{length}(s) = 0$) entonces2 **devolver** *abrevSeparada*3 **si** (s pertenece a ($\text{stopList} \cup \text{abrevList} \cup \text{ispellList}$)) entonces4 **devolver** ($\text{buscarSufijo}(\text{abrevSeparada}, "") + ' ' + s$)// Se extrae y se guarda el primer caracter de s .5 $\text{abrevSeparada} \leftarrow \text{abrevSeparada} + s[0]$

// Llamar nuevamente a la función sin el primer caracter.

6 $s \leftarrow s[1, \text{length}(s)]$ 7 **devolver** $\text{buscarSufijo}(s, \text{abrevSeparada})$

3.4.3. Algoritmo de División: Samurai

Esta técnica pensada por Eric Enslen, Emily Hill, Lori Pollock, Vijay-Shanker [16] 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 presente en el código para llevar a cabo el objetivo. Esto permite que no sea necesario utilizar diccionarios predefinidos, además las palabras que se obtienen producto de la división no están limitadas 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.

El algoritmo selecciona la partición más adecuada en los ids multi-palabra¹ en base a una función de puntuación (scoring). Esta función, utiliza información que se recauda extrayendo las frecuencias de aparición de palabras dentro del código fuente. Estas palabras pueden estar contenidas en comentarios, literales strings y documentación.

La estrategia de separación Samurai está inspirada en una técnica de expansión de abreviaturas AMAP [22] que se describe en próximas secciones.

La técnica Samurai según los autores [16] no solo divide los ids, sino que también aquellos que aparezcan en los comentarios y en los literales strings. Por esta razón, el parámetro de entrada del algoritmo se denomina *token* en lugar de id.

El algoritmo primero se encarga de extraer información respecto a la frecuencia de tokens en el código fuente. Luego se construyen dos tablas de frecuencia de tokens. Para la construcción de una de las tablas primero se ejecuta el algoritmo que extrae del código fuente todos los tokens del tipo *hardword*. Estos tokens son agregados en la *tabla de frecuencias específicas*. Una entrada de esta tabla corresponde al listado de tokens extraídos del programa actual bajo análisis (cada token es único en la tabla). La otra entrada corresponde al número de ocurrencia de cada token.

Por otro lado, existe la *tabla de frecuencias globales*. Esta tabla contiene las mismas dos columnas que la tabla anterior, tokens y sus frecuencias. La

¹Que posee más de una palabra.

diferencia principal radica en que la información es recolectada de distintos programas de gran envergadura.

Durante el proceso de división del token, Samurai ejecuta la función de scoring que se basa en la información de ambas tablas antedichas.

El algoritmo ejecuta dos rutinas primero *divisiónHardWord* y después *divisiónSoftWord*. La primera básicamente se encarga de dividir los hard-words (palabras que poseen guión bajo o son del tipo camel-case), luego cada una de las palabras obtenidas son pasadas a la segunda rutina para continuar con el análisis.

En la rutina *divisiónHardWord* (ver algoritmo 2) primero se ejecutan dos funciones (líneas 1 y 2). La primera *dividirCaracteresEspecialesDigitos*, que reemplaza todos los posibles caracteres especiales y números que posea el token por espacio en blanco. La segunda *dividirMinusSeguidoMayus*, de la misma forma que la anterior 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 (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. Aquí se encontrarán casos del tipo camel-case donde la mayúscula indica el comienzo de la nueva palabra (ejemplo: List). Sin embargo, el autor a través de estudios de datos, se encontró con variantes en donde la mayúscula indica el fin de una palabra (ejemplo: SQLlist).

El algoritmo decide entre ambas opciones calculando el puntaje (score) de la parte derecha de las dos divisiones (líneas 7 y 8). Aquella con puntaje más alto entre las dos será por la cual se decida (línea 9). Tomando como ejemplo el id GPSstate, para el caso camel-case calculará $score(Sstate)$ y para la otra variante $score(state)$. Lógicamente, la función score elegirá *state* sobre *sstate* ya que esta última tiene un puntaje inferior, por ende GPSstate se corresponde a la variante de camel-case. La división elegida se lleva a cabo en las líneas 11 y 13 (según el caso). Finalmente, todas las partes divididas se envían a *divisiónSoftWord* (línea 18).

Algoritmo 2: divisiónHardWord

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

```

1 token  $\leftarrow$  dividirCaracteresEspecialesDigitos(token)
2 token  $\leftarrow$  dividirMinusSeguidoMayus(token)
3 tokenSep  $\leftarrow$  ""
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  $\leftarrow$  length(s) - 1
7         // se determina con la función score si es del tipo
8         // camelcase u otra alternativa
9         scoreCamel  $\leftarrow$  score(s[i,n])
10        scoreAlter  $\leftarrow$  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  $\leftarrow$  tokenSep + ' ' + s
17 token  $\leftarrow$  tokenSep
18 tokenSep  $\leftarrow$  ' '
19 para todo (s | s es un substring separado por ' ' en token) hacer
20     tokenSep  $\leftarrow$  tokenSep + ' ' + divisiónSoftWord(s,score(s))
21 devolver tokenSep

```

La rutina recursiva *divisiónSoftWord* (ver algoritmo 3) recibe como entrada un substring *s*, el cual puede tener tres tipos de variantes: a) todos los caracteres en minúsculas, b) todos con mayúsculas, c) el primer caracter

con mayúscula seguido por todas minúsculas (**Visitor**). El otro parámetro de entrada es el puntaje original score_{sd} que corresponde 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 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) el puntaje de la división elegida sobresalga del resto de los puntajes (líneas 7-9).

Para aclarar el punto anterior, para cada partición (izquierda o derecha) obtenida se calcula el score (líneas 4 y 5). Luego este es comparado con el puntaje de la palabra original (score_{sd} score original) y el puntaje de la palabra actual ($\text{score}(s)$). En un principio ambas son iguales pero a medida que avanza la recursión $\text{score}(s)$ varía con respecto a score_{sd} (líneas 7 y 8).

En caso de que no tenga prefijos y sufijos ordinarios, se considera que la parte izquierda es un candidato. Por otro lado, la cadena de la parte derecha se invoca recursivamente con la rutina porque podría seguir dividiéndose en más partes (línea 14).

Si la parte derecha finalmente se divide, 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 es dividida tampoco se debería separar entre ambas partes (el if de la línea 13 controla esto). Los análisis de datos hechos por el autor [16] obligan a hacer este control ya que se encontraron abundantes casos erróneos de división, uno de ellos es **string ified**.

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 [16] 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**. La presencia de la raíz cuadra-

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

da en el algoritmo *divisiónHardWord* (línea 9), cuando se compara el caso camel-case y el caso alternativo también es para solucionar este problema.

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 ( $\neg presuf \wedge split_{izq} \wedge split_{der}$ ) 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 ( $\neg presuf \wedge split_{izq}$ ) 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  $tokenSep$ 

```

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 $score(s)$ indica i) la frecuencia de aparición de s en el programa bajo análisis y ii) la frecuencia en un conjunto grande de programas predefinidos. 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 [16].

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) [25] trabaja con cuatro listas para realizar su tarea:

- Una lista de palabras (en lenguaje natural) que se extraen del código.
- Una lista de frases (en lenguaje natural) presentes también en el código.
- Una lista de palabras irrelevantes (stop list).
- Una lista de palabras de un diccionario en inglés.

La primer lista se confecciona de la siguiente manera, para cada método f dentro del código se crea una lista de palabras que se extraen de los comentarios que están antes (comentarios JAVA Doc) o dentro del método f . También se incorporan los ids del tipo `hardword` (si existen) dentro del alcance local de f .

La lista de frases se arma utilizando una técnica que extrae frases en lenguaje natural [19], el principal recurso son los comentarios y los ids multi-palabras. En este punto se construye un acrónimo¹ con las palabras de alguna frase, si ese acrónimo coincide con alguno de los ids extraídos, entonces esa frase se considera como potencial expansión (Ejemplo: la frase `file status` es una expansión posible para el id `fs_exists` \rightarrow `file status exists`).

Una vez que las listas de palabras y frases potenciales se confeccionan, la ejecución del algoritmo comienza. 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)². En caso de que así sea, no se retornan resultados. La razón de esto es porque estas palabras no aportan información importante en la comprensión del código y son fácilmente reconocidas por los ingenieros del software. Algunos casos son artículos/conectores (`the`, `an`, `or`) y palabras reservadas del lenguaje de programación que se utilicen (`while`, `for`, `if`, etc.).

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

²Esta lista se usa con la misma política que el algoritmo Greedy.

Algoritmo 4: Expansión Básica

Entrada: *abrev* // Abreviatura a expandir
 Entrada: *wordList* // Palabras extraídas del código
 Entrada: *phraseList* // 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 (phrase | phrase es una frase en phraseList) hacer
5   └─ si (abrev es un acrónimo de phrase) entonces
6     └─ devolver phrase

   // Buscar abreviatura común.
7 para todo (word | word es una palabra en wordList) hacer
8   └─ si (abrev es una abreviatura de word) entonces
9     └─ devolver word

   // 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

```

Siguiendo con la ejecución, se chequean si alguna de las frases extraídas del código se correspondan con la abreviatura en forma de acrónimo (línea

5).

Después, se busca si las letras de la abreviatura coinciden en el mismo orden que las letras de una palabra presente en la lista de palabras recolectadas del código (línea 8). Ejemplos: `horiz` → `horizontal`, `trgn` → `triangle`.

En caso de no tener éxito, la búsqueda continúa en el diccionario predefinido como último recurso (línea 10).

Esta técnica de expansión descripta, solo retorna una única expansión potencial para una abreviatura determinada y en caso contrario no retorna nada (líneas 13 y 14). El motivo de esto, es porque no tiene programado como decidir una sola opción ante múltiples alternativas de expansión. A esta característica, los autores lo presentan como trabajo futuro [25, 22].

3.4.5. Algoritmo de Expansión AMAP

El algoritmo de expansión de abreviaturas que construyó Emily Hill, Zachary Fry, Haley Boyd [22] 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 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. Los diccionarios (en inglés) incluyen demasiadas palabras e implica disponer de un gran almacenamiento.

Las fuentes de palabras que se utilizan son una lista de abreviaciones comunes. Estas abreviaciones se obtienen automáticamente desde distintos programas. También se puede incorporar palabras en forma personalizada. La lista palabras irrelevantes (stop-list) y la de contracciones más comunes se arman manualmente.

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.

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 achicando aun más 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 esos métodos.

Comenzando por el ítem 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 $[a-z]^+$. Las palabras que coinciden si o si deberán comenzar con ***pa***. La expresión regular queda: ***pa*** $[a-z]^+$.

Ejemplo: El patrón ***arg*** $[a-z]^+$ coincide (entre otras) con la palabra ***argument***.

Patrón compuesto por letras: La expresión regular se construye insertando $[a-z]^*$ 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: El patrón ***p*** $[a-z]^*$ ***g*** $[a-z]^*$ ***m*** $[a-z]^*$ coincide (entre otras) con ***program***.

La búsqueda de palabras singulares se presenta en el algoritmo 5. Los parámetros de entrada son la palabra abreviada a expandir, la expresión regular formada por el patrón elegido, los distintos comentarios que existan en el código (en la clase y en el método) y el cuerpo del método.

En la línea 1 se impide básicamente dos cosas:

a) Que no se procesen palabras abreviadas con muchas vocales consecutivas (segundo argumento del **and** en el **if**). La autora de AMAP comprobó [22] que la mayoría de las palabras abreviadas con vocales consecutivas se expanden como multi-palabras (ejemplos: es el caso de los acrónimos ***gui*** → ***graphical user interface***, ***ioe*** → ***invalid object exception***). El algoritmo de la próxima sección es el encargado de expandirlos.

b) En caso de que el patrón sea el *compuesto por letras* (no sea el prefijo), se hacen dos controles más (primer argumento del **and** en el **if**). Uno es, que la abreviatura no posea muchas vocales consecutivas (“ $[^aeiou]^+$ ” logra eso) y la otra es que longitud sea mayor a 3. La autora a través del análisis de datos determino esta restricción [22], 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*** → ***loading***, ***language*** o también ***br*** → ***bar***, ***barrier***, ***brown***.

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

Algoritmo 5: Búsqueda por Palabras Singulares

Entrada: *pa* // *Palabra Abreviada*
Entrada: *patrón* // *Expresión regular*
Entrada: Cuerpo y Comentarios del Método
Entrada: Comentarios de la Clase
Salida : Palabras largas 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”
3  Buscar en Nombres de Tipos y la correspondiente Variable
  declarada con “patrón pa”
4  Buscar en el Nombre del Método con “patrón”
5  Buscar en las Sentencias con “patrón pa” y “pa patrón”
6  si (length(pa)  $\neq$  2) entonces
7    Buscar en palabras del Método con “patrón”
8    Buscar en palabras que están en los Comentarios del Método
    con “patrón”
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”

```

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 de-

volverá el resultado “index” como expansión de **ind**.

Si no hay 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 no tiene éxito sigue en la línea 4 donde se busca coincidir con “**patrón**” en el nombre del método. En caso de seguir sin resultado alguno, prosigue en la línea 5 con distintas variantes “**patrón pa**” o “**pa patrón**” en las sentencias comunes del método.

Si la ejecución continúa, 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 → 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 función **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 la autora sostiene [22], 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 → graphical user interface, fl → 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 $[a-z][]+$, 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: $c_1[a-z][]+c_2[a-z][]+c_3[a-z][]+..[a-z][]+c_n$. Permite encontrar acrónimos tales como `xml` \rightarrow **e***x*tensible **m**arkup **l**anguage.

Patrón de Combinación de Palabras: En este caso el patrón se construye de manera similar al anterior pero se usa la expresión regular $[a-z]^*[]^*$ 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: $c_1[a-z]^*[]^*c_2[a-z]^*[]^*c_3[a-z]^*[]^*...[a-z]^*[]^*c_n$. 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 [22]. Las variables de entrada son: la abreviatura multi-palabra a expandir, la expresión regular formada por el patrón elegido, los distintos comentarios que existan en el código (en la clase y en el método) y el cuerpo del método.

Algoritmo 6: Búsqueda por Multi Palabras

Entrada: *pa* // *Palabra Abreviada*
Entrada: *patrón* // *Expresión regular*
Entrada: Cuerpo y Comentarios del Método
Entrada: Comentarios de la Clase
Salida : Palabras largas candidatas, o null si no hay
// Las expresiones regulares están entre comillas

```

1 si (patrón acrónimo  $\vee$  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”
3   Buscar en Nombres de Tipos y la correspondiente Variable
    declarada con “patrón pa”
4   Buscar en el Nombre del Método con “patrón”
5   Buscar en todos los ids (y sus tipos) dentro del Método con
    “patrón”
6   Buscar en Literales String con “patrón”
    // 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”
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”

```

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

<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 trozos de Código.

ingresadas con longitud 4 ó mayor (línea 1). Esto genera la sensación de que se pierden casos de 2 ó 3 caracteres pero estudios indican que son la minoría [22].

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 tienen cotas más altas. Es por esto que 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, la autora [22] concluye 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 abreviación 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 item 1, se agrupa 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 2 primeras en solo **default** con una cantidad de 4 predominando sobre **define**.
3. En caso de que la igualdad persista, se acumulan las frecuencias de aparición entre las distintas búsquedas para determinar un solo candi-

dato. 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 almacena ambas opciones. Después, se continúa 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 [22] 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 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 2 niveles, 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. El nivel más general se arma con la API¹ de JAVA. En la tabla 3.4.5 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

¹Application Programming Interface

Abreviatura	Palabra Exp.	Frec. 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

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 [22, 26, 27, 25].

3.4.6. Herramienta: Identifier Restructuring

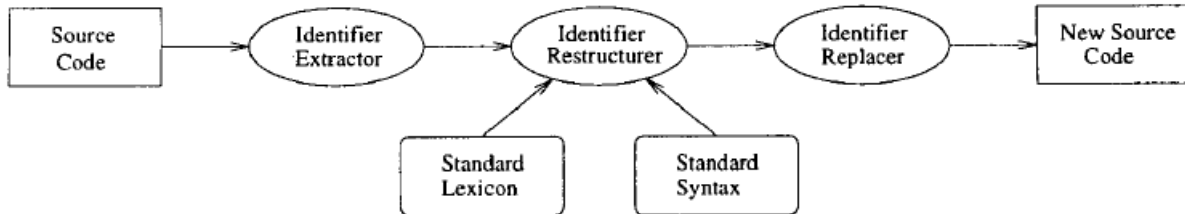


Figura 3.6: Etapas de Restructuring tool

La herramienta Restructuring Tool [11] 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 entendible y mejora la comprensión. El proceso consta de tres etapas (Figura 3.6):

1. **Identifier Extractor:** Recupera una lista con los nombres de los ids presentes en el código. Este módulo se programó con un parser modificado de C que reconoce los ids y los extrae.
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 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.

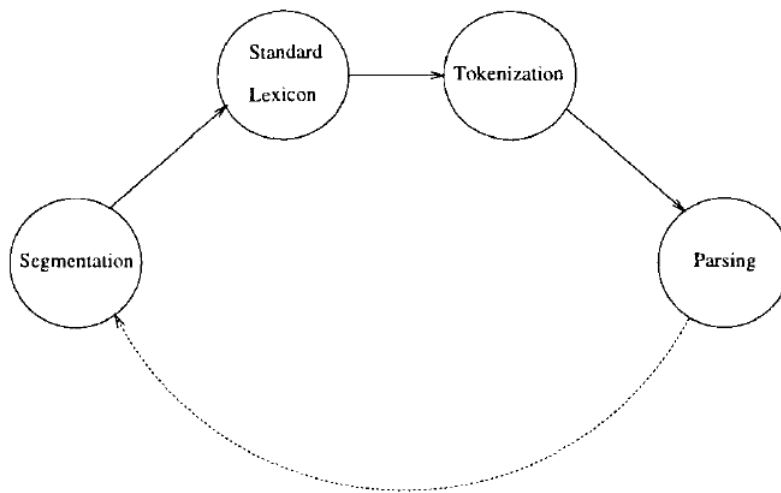


Figura 3.7: Etapas de Identifier Restructurer.

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` → `get txt`). En caso presencia de softwords, la división se debe hacer en forma manual. Por ejemplo: `get_txtinput` → `get txt input` la separación entre `txt` e `input` la realiza el usuario. De manera conceptual (no implementado), los autores proponen automatizar más esta fase. La propuesta consiste de un algoritmo hecho en LISP, este toma un string s como entrada. Se utiliza una estrategia greedy verificando a partir de la primer letra de

s un sub-string que pertenezca a un diccionario predefinido. Luego el sub-string se descarta y continúa el análisis con el resto hasta que no haya más sub-strings que separar [10].

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: Una vez lograda la separación de las palabras estas son mapeadas a una forma estándar (expandidas) con la ayuda de un diccionario léxico [10] (Ejemplo: `upd` → `Update`). Una idea de mejora propuesta es incorporar al diccionario términos extraídos del código fuente. También aquí, el usuario puede intervenir para realizar la expansión manualmente. Los autores [11] de la herramienta construyeron los diccionarios de manera genérica tomando como muestra 10 programas. 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 (expandida) 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 [10]. 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 parsea usando una gramática predefinida. Este parseo 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 parseo falle el proceso se reinicia desde el comienzo partiendo nuevamente de la etapa de segmentación [11] (figura 3.7).

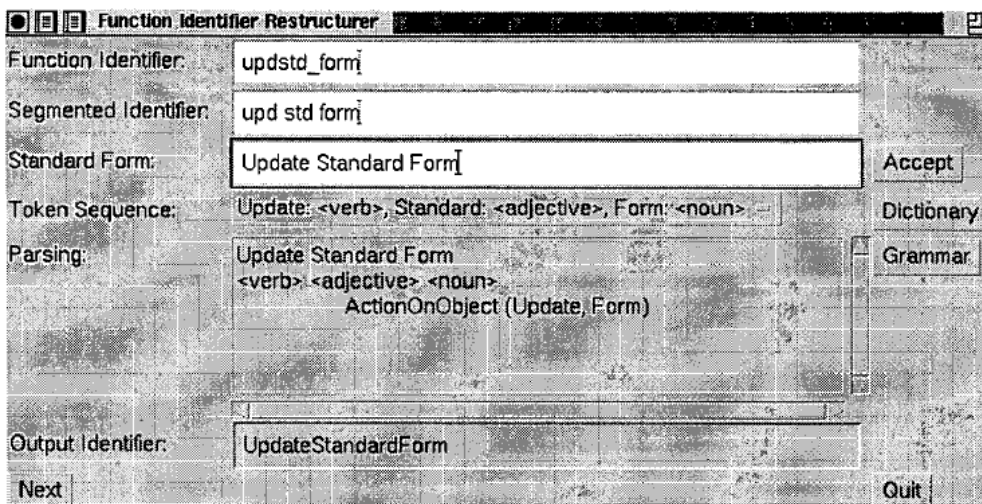


Figura 3.9: Visualización de Restructuring Tool.

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` → `Update`, `std` → (?), `form` → `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 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, la etapa **Identifier Replacer** toma todas las ocurrencias del id `updstd_form` y se reemplaza por `UpdateStandarForm`, como se mencionó con anterioridad.

3.5. Conclusiones

Las observaciones que se destacan en el estado del arte de las técnicas de análisis de ids apuntan por un lado al nombramiento correcto de los ids. Al comienzo de este capítulo se detalló una herramienta que ayuda a lograr esta meta. 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.

El correcto nombramiento en 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. De esta manera, es más sencillo extraer conceptos del dominio del problema desde los ids.

Las herramientas/técnicas de análisis de ids 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. A medida que se construyeron nuevas técnicas, se buscó más la automatización haciendo que el programador se involucre menos.

Como se mencionó en este capítulo, las primeras técnicas utilizaban netamente diccionarios de palabras en lenguaje natural, lo cual requiere mucho espacio de almacenamiento. Más tarde, se intentó disminuir el uso de estos diccionarios mirando más los recursos internos de información dentro los sistemas, como es el caso de los comentarios, literales strings y la documentación.

Sin embargo, suele ocurrir que estos recursos internos son escasos. Es por esto, que los autores de las recientes técnicas decidieron recurrir a procesos que examinan programas de gran envergadura. Estos procesos recolectan palabras útiles que son almacenadas en forma de diccionarios. Estos diccionarios no solo ayudan a traducir el significado de los ids, también tiene bajas exigencias de almacenamiento y están constituidos con palabras más adecuadas al ámbito de las ciencias de la computación.

Capítulo 4

Identifier Analyzer (IDA)

En este capítulo, se describe una herramienta útil que se construyó a partir de los conceptos explicados en el capítulo anterior. Esta herramienta llamada Identifier Analyzer (IDA), se encarga de analizar los ids que están presentes en archivos JAVA que son pasados por entrada. A lo largo de este capítulo, también se explicarán los distintos módulos que IDA posee y que flujo de ejecución debe realizar el usuario para analizar los ids. Al final del capítulo, se describen algunos casos de estudio que demuestran la importancia de haber construido IDA. Todas estas explicaciones, están acompañadas con capturas de pantalla y tablas que facilitarán al lector entender el funcionamiento de la herramienta IDA.

4.1. Introducción

La herramienta IDA le permite al usuario revelar la información estática oculta que hay detrás de los ids. Esto se logra mediante la ejecución de técnicas/algoritmos que fueron descritos en el capítulo anterior.

La iniciativa del desarrollo de la herramienta IDA surgió, porque en la actualidad no existe otra herramienta en el ámbito de la CP con similares características.

Para comenzar con la descripción de IDA, en la siguiente sección se explica la arquitectura y cuales son sus componentes principales.

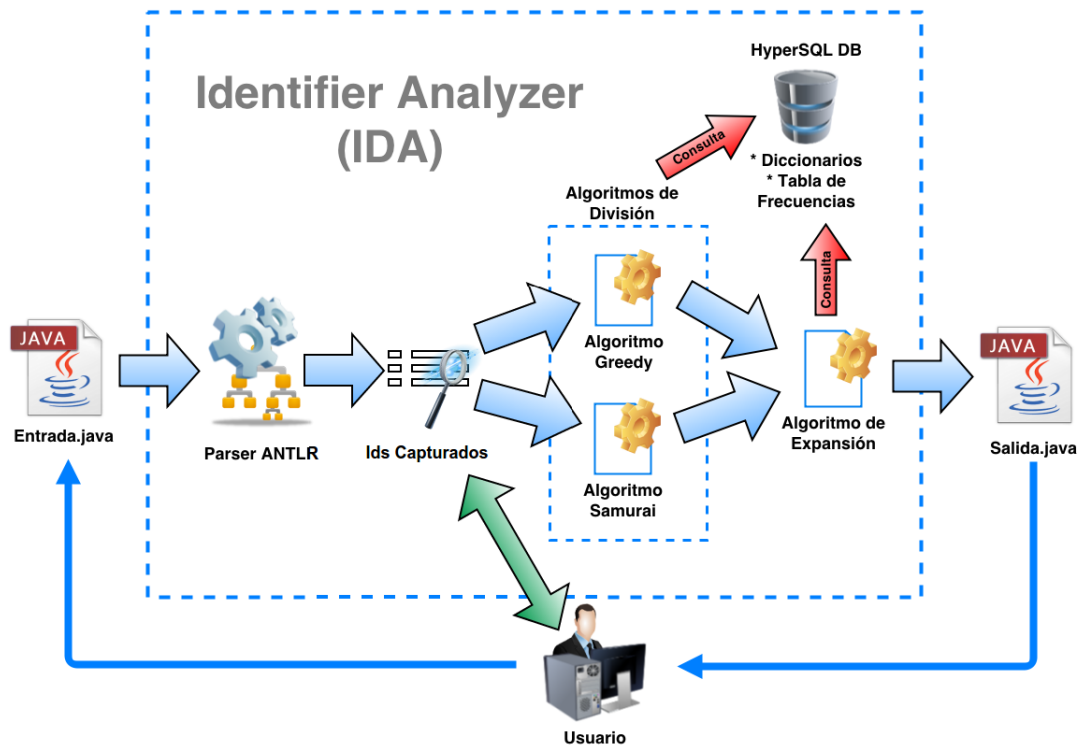


Figura 4.1: Arquitectura de IDA.

4.2. Arquitectura

La herramienta IDA se implementó empleando lenguaje JAVA y se programó usando el entorno de desarrollo NetBeans¹.

En la figura 4.1 se puede apreciar la arquitectura de IDA. Esta arquitectura describe tres etapas, 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.

Fase de Extracción de Datos: en el primer paso, IDA recibe como entrada un archivo JAVA que es pasado por el usuario (ver Figura 4.1 Entrada.java), luego este archivo se procesa por un Analizador Sintáctico (ver Figura 4.1 Parser ANTLR). Este parser, extrae y almacena en estructuras internas la información estática perteneciente al código del archivo

¹<https://www.netbeans.org>

ingresado (el parser se explica con más detalles en la próxima sección). Esta información extraída, está relacionada con ids, literales y comentarios. El usuario a través de la interfaz de IDA, puede visualizar esta información capturada del código por medio de tablas claramente definidas (ver figura 4.1 - Flecha Verde).

Fase de División de Ids: Una vez completada la extracción de información, se da comienzo a la fase de división de ids. La misma tiene implementada dos algoritmos de división descritos en el capítulo 3; uno es el Algoritmo Greedy (ver sección 3.4.2) y el otro es el Algoritmo Samurai (ver sección 3.4.3). Ambos algoritmos reciben como entrada la información capturada de la etapa anterior, luego estos algoritmos se encargan de dividir los ids del archivo JAVA (ver figura 4.1 Algoritmos de División). Las divisiones se almacenan en estructuras internas que serán consultadas en la próxima etapa. 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 sección 4.4 para más detalles), y son consultados por ambos algoritmos de división (parte superior de la figura 4.1 - Flechas Rojas).

Fase de Expansión de Ids: La tercera y última etapa tiene implementado el Algoritmo de Expansión Básico de abreviaturas (ver sección 3.4.4). Este algoritmo toma como entrada los ids divididos en la etapa anterior, tanto de Greedy como de Samurai. Luego, el Algoritmo de Expansión procede a expandir los ids obtenidos por estos 2 algoritmos, dando como resultado ids expandidos desde Greedy y desde Samurai (ver figura 4.1 Algoritmo de Expansión). En este punto, el usuario podrá elegir que expansión es la más adecuada y reemplazar los ids originales en el archivo de entrada, generando de esta manera un nuevo archivo de salida (ver figura 4.1 Salida.java). El Algoritmo de Expansión también necesita de un diccionario de palabras, por eso se realizan consultas a la base de datos embebida (parte superior de la figura 4.1 - Flechas Rojas).

4.3. Analizador Sintáctico (Parser)

Como se explicó en la sección previa, cuando el usuario ingresa un archivo JAVA, IDA utiliza un Analizador Sintáctico (Parser) que examina y extrae información estática presente en el archivo ingresado. Esta información está compuesta por identificadores, comentarios y literales.

La construcción de este Parser se llevó a cabo, primero investigando herramientas encargadas de construir Analizadores Sintácticos. 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. La herramienta *ANTLR* genera un Analizador Sintáctico JAVA en función de la gramática JAVA² que es pasada como entrada. En esta gramática se agregaron acciones semánticas que capturan información relacionada a ids, comentarios y literales. Estas acciones no alteran la estructura original de la gramática. Una vez insertadas estas acciones y si están correctamente colocadas, *ANTLR* se encarga de leer la gramática y generar el Parser adicionando las acciones que fueron insertadas. De esta manera, se obtiene un Parser que examina código JAVA y recolecta datos necesarios (ids, comentarios, literales) que serán utilizados en próximas etapas de la herramienta IDA.

4.4. Base de Datos Embebida

Dado que IDA necesita diccionarios y listas de palabras para poder funcionar, se utilizan casi los mismos que emplean los autores de las técnicas de análisis de ids explicadas en la sección 3.4. Estos diccionarios/listas están almacenadas dentro de IDA en una base de datos embebida. Esta base de datos utiliza una tecnología llamada HSQLDB³ y al estar desarrollada en JAVA permite una correcta integración con IDA. Otra ventaja que tiene esta tecnología es que responde rápidamente las consultas.

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

²Especificaciones de la gramática JAVA en <http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>

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

A continuación, se describen los diccionarios/listas de palabras que están alojadas en la base de datos explicada en el párrafo anterior. También se nombran los algoritmos implementados en IDA que consultan cada lista.

Diccionario en Inglés (ispell): Contiene palabras en Inglés que pertenecen a la lista de Palabras de *Ispell*⁴. Se utiliza en el Algoritmo de Greedy (ver sección 3.4.2) y en el Algoritmo de Expansión (ver sección 3.4.4).

Lista de Palabras Excluyentes (stop-list): Esta compuesta con palabras que son poco importantes o irrelevantes en el análisis de ids. Se utiliza en el Algoritmo de Greedy (ver sección 3.4.2) y en el Algoritmo de Expansión (ver sección 3.4.4).

Lista de Abreviaturas y Acrónimos Conocidas: Contiene abreviaturas comunes del idioma Inglés y Acrónimos conocidos de programación (gif, jpg, txt). Se emplea en el Algoritmo Greedy (ver sección 3.4.2).

Lista de Prefijos y Sufijos Conocidos: Posee Sufijos y Prefijos conocidos en Inglés, esta lista fue confeccionada por el autor del Algoritmo Samurai (ver sección 3.4.3). Se consulta solo en dicho algoritmo.

Frecuencias Globales de Palabras: Lista de palabras, junto con su frecuencia de aparición. Esta lista fue armada por el autor del Algoritmo Samurai. Se emplea solo en dicho algoritmo, más precisamente en la función de *Scoring* (ver sección 3.4.3).

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 recomienda ingresar en IDA archivos JAVA con comentarios, literales e ids acordes a la lengua Inglesa.

Habiendo descripto a grandes rasgos 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.

⁴Comando de Linux. <http://wordlist.aspell.net>



Figura 4.2: Barra de Menú de IDA

4.5. Proceso de Análisis de Identificadores

En esta sección, se describe el flujo de ejecución que debe seguir el usuario con la herramienta IDA para llevar a cabo el análisis de los ids. Se explicarán en detalles la función de cada elemento de IDA y de como cada uno de estos ayuda al usuario a analizar los ids presentes en los archivo JAVA.

4.5.1. Barra de Menús

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

Al pulsar¹ en *Archivo* de la barra antedicha, se despliega un menú con los siguientes ítems (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 (más detalles en la próxima sección).

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 menús, se despliega otro menú con con los siguientes ítems (ver figura 4.2 - Flecha 2):

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

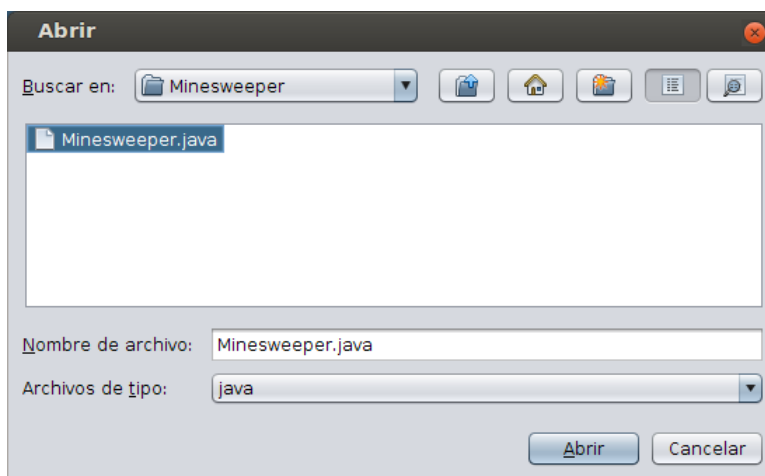


Figura 4.3: Ventana para seleccionar Archivos JAVA.

Ver Diccionarios: Abre una ventana, que muestra un listado de palabras en Inglés que pertenece al comando de Linux *ispell*. La ventana antedicha, también muestra un listado de palabras irrelevantes o stoplist (ver sección 3.4). Esta ventana, se explica con más detalles en la sección 4.5.5.

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

Finalmente al presionar *Ayuda* de la barra de menús, se despliega un solo botón (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 *Abrir archivo(s) JAVA* explicado en la sección anterior (ver figura 4.2 - Flecha 1), se abre 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

¹<http://www.tiobe.com/jacobe>

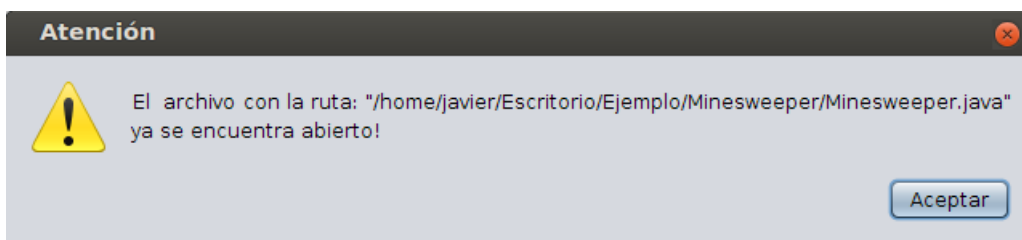


Figura 4.4: Aviso sobre Archivo JAVA ya abierto en IDA.

un archivo JAVA y embellece el código fuente contenido en el archivo. 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, se muestra un cartel informando al 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 (Parte 1)

Una vez que el usuario selecciona el/los archivo/s JAVA, y JACOBÉ embellece el código de cada uno, los mismos son procesados por el parser construido con ANTLR. Al finalizar el procesamiento del parser, el *Panel de Elementos Capturados* aparece (ver figura 4.5). Este panel en la parte superior posee pestañas, cada pestaña está rotulada con el nombre del archivo abierto correspondiente (ver figura 4.5 - Flecha 1). En caso de querer cerrar un archivo particular, esto se logra pulsando en la cruz ubicada al lado del rótulo de cada pestaña (ver figura 4.5 - Flecha 1). Es posible abrir varios archivos 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.

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 parser elaborado con ANTLR

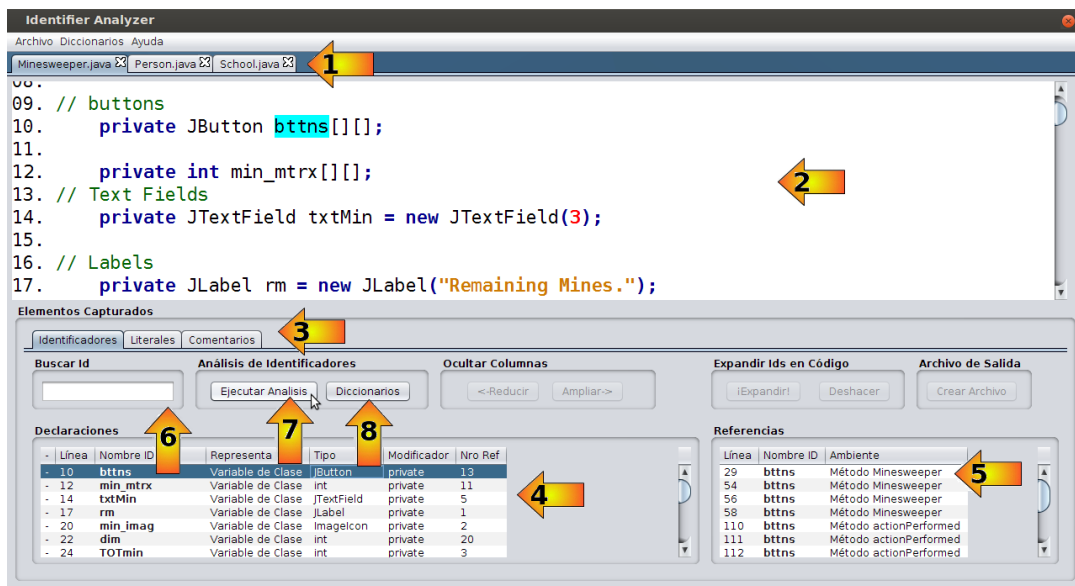


Figura 4.5: Panel de Elementos Capturados

referente a ids, literales y comentarios. Estos últimos tres poseen una pestaña que clasifican a cada uno (ver figura 4.5 - Flecha 3).

Al pulsar la *Pestaña de Literales* (ver figura 4.5 - Flecha 3), se puede apreciar que contiene una tabla y un practico buscador para agilizar la búsqueda de literales en la tabla (ver figura 4.6 - Flecha 1). Esta tabla contiene 3 columnas, número de línea del literal, el literal propiamente dicho y la eventual asociación que pueda tener el literal con algún id (ver figura 4.6 - Flecha 2). Al pulsar sobre alguna fila de la tabla antedicha, automáticamente el literal correspondiente se resalta en el código (del archivo JAVA) ubicado en la parte superior del *Panel de Análisis* (ver figura 4.6 - Flecha 3).

Al presionar la *Pestaña de Comentarios* (ver figura 4.5 - Flecha 3), se visualiza una tabla listando los comentarios encontrados en el archivo y un buscador (ver figura 4.7 - Flecha 1) de comentarios en la tabla. Esta tabla (ver figura 4.7 - Flecha 2) contiene 2 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 inmediatamente se resalta en el código mostrando la ubicación del comentario seleccionado

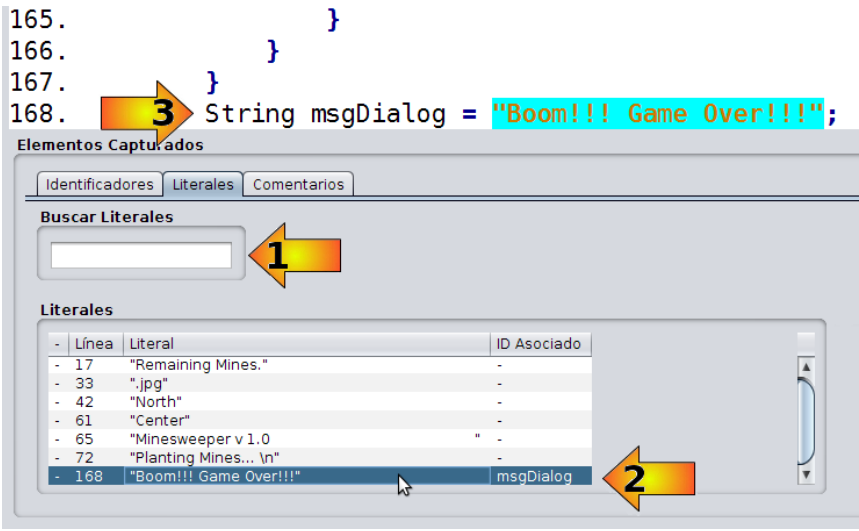


Figura 4.6: Literales Capturados

(ver figura 4.7 - Flecha 3).

Al pulsar la *Pestaña de Identificadores* (ver figura 4.5 - Flecha 3), se muestra la *Tabla de Declaraciones* (ver figura 4.5 - Flecha 4), aquí no solo se detalla el nombre del id, sino que también el número de línea donde esta declarado, el tipo (int, char, etc.), el modificador (publico, privado, protegido), lo que el id representa (variable de clase, constructor, método de clase, etc.) y cuantas referencias tiene (cantidad de lugares desde donde se utiliza como referencia, sin contar la declaración). A su vez, estas referencias son listadas en una tabla contigua llamada *Tabla de Referencias* (ver figura 4.5 - Flecha 5), la misma indica el número de línea y ambiente (ubicación de cada referencia).

Tanto la *Tabla de Declaraciones*, como la *Tabla de Referencias*, al presionar en una fila, inmediatamente se resalta con color en el código del panel superior la declaración del id o la referencia, según corresponda (ver figura 4.5 - Flecha 2). Es el mismo comportamiento que tienen las tablas de literales y comentarios explicados anteriormente. Cabe mencionar, que la *Tabla de Declaraciones* también posee un buscador, que realiza búsquedas por el nombre del id (ver figura 4.5 - Flecha 6).

Hasta aquí, solo se ha descripto como IDA exhibe la información cap-

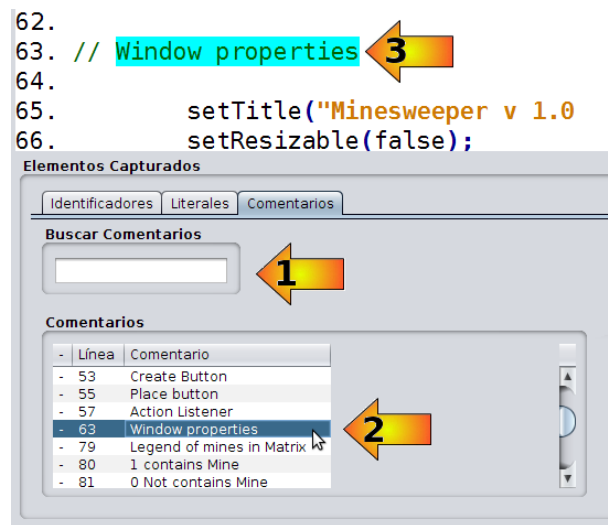


Figura 4.7: Comentarios Capturados

turada del código. A continuación, se explicará como se emplea IDA para analizar los ids. Para comenzar, se debe pulsar en el botón *Ejecutar Análisis* ubicado dentro del cuadro *Análisis de Identificadores* (ver figura 4.5 - Flecha 7), al hacerlo se abrirá el *Panel de Análisis* que será explicado en la próxima sección.

4.5.4. Panel de Análisis

El *Panel de Análisis* (ver figura 4.8) contiene 3 partes principales, (de izquierda a derecha) la primera consiste en los ids capturados, los mismos están listados 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), estos brindan información al usuario sobre los datos que se utilizan para ejecutar los algoritmos de análisis. Los mismos serán explicados en la próxima sección.

Siguiendo con el *Panel de Análisis*, en el cuadro central superior (ver figura 4.8 - Flecha 3) se pueden seleccionar los dos algoritmos de división de ids (Greedy y Samurai), el botón *Divir* del mismo cuadro ejecuta la técnica seleccionada mostrando en el cuadro inferior la tabla con los resultados (ver figura 4.8 - Flecha 4). De la misma manera, en los 2 paneles subsiguientes



Figura 4.8: Panel de Análisis

de la derecha, permiten expandir las palabras y mostrar los resultados en el cuadro de abajo. 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 de la expansión se muestran en la tabla ubicada en el cuadro inferior derecho (ver figura 4.8 - Flecha 6).

4.5.5. Palabras Capturadas y Diccionarios

En la sección anterior, se describieron dos botones *Palabras Capturadas* y *Diccionarios*, ubicados en el *Panel de Análisis* (ver figura 4.8 - Flecha 2). Al pulsar el primero, 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 3 columnas, en la primera posee tokens¹, los mismos fueron capturados por el Parser ANTLR. La segunda columna, contiene la frecuencia local de cada token, cabe recordar que la frecuencia local se construye en función de la frecuencia absoluta de aparición de los tokens en el código del archivo

¹Genérico utilizado por el autor, para denotar ids, palabras de comentarios y literales.

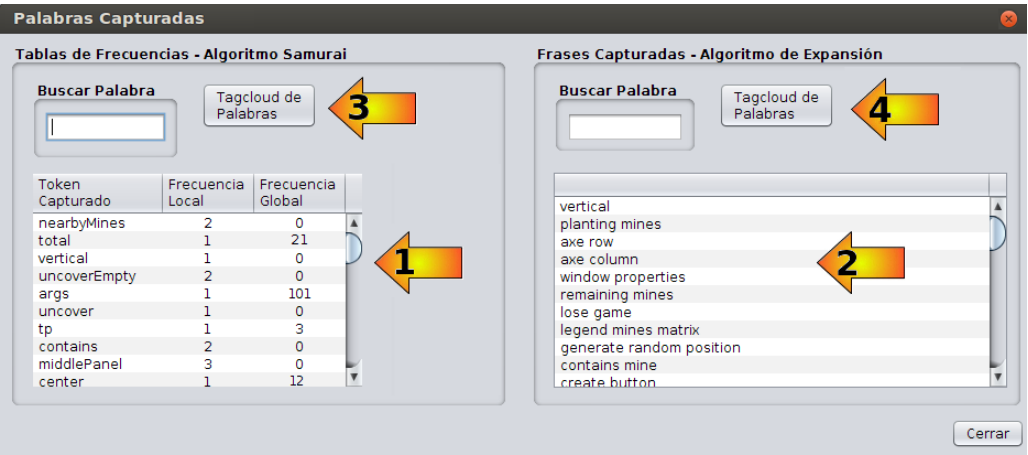


Figura 4.9: Información Utilizada para el Análisis de Ids



Figura 4.10: TagCloud: Nube de Etiquetas

actual (ver sección 3.4.3). La tercera y última columna de la tabla denota la frecuencia global de cada token, la misma esta predefinida en la base de datos HSQLDB (ver sección 4.4) y se armó mediante el análisis hecho por los autores del Algoritmo Samurai (ver sección 3.4.3).

El cuadro de la derecha, lista en una tabla las frases capturadas (ver figura 4.9 - Flecha 2), estas frases se obtienen de los comentarios y los literales strings, el Algoritmo de Expansión es el encargado de utilizarlas (ver sección 3.4.4).

Cada uno de los botones con el nombre *TagCloud de Palabras* (ver figura 4.9 - Flechas 3 y 4) abre una ventana conteniendo una *Nube de Etiquetas*

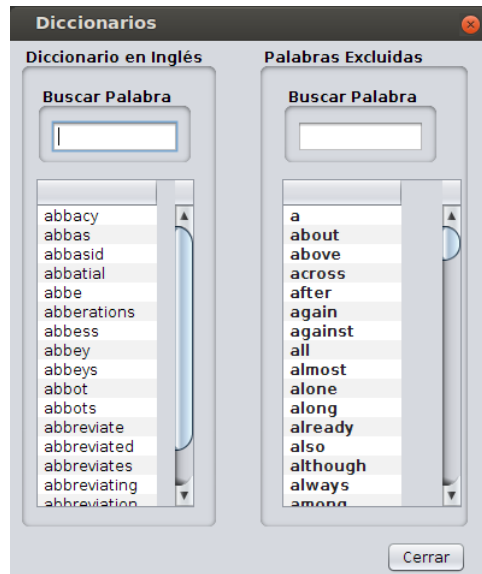


Figura 4.11: Panel de Diccionarios

(ver figura 4.10). Esta nube de palabras resalta en tamaño más grande las palabras que más frecuencia de aparición tienen. Para generar esta nube se emplea una librería de JAVA llamada OpenCloud¹. Esta nube de palabras ayuda a ver con mas claridad que palabras son más frecuentes en el código. Para el caso de la nube de frecuencias de Samurai (ver figura 4.9 - Flecha 3), el tamaño de cada palabra depende de la Frecuencia Local de cada token (ver figura 4.9 - Flecha 1). En el caso de la nube de las frases capturadas (ver figura 4.9 - Flecha 4), el tamaño de las palabras esta dado por el numero de apariciones dentro de esta tabla de frases (ver figura 4.9 - Flecha 2).

A modo de agilizar la búsqueda de alguna palabra y/o token, la misma puede ser hecha en los correspondientes cuadros de texto con rótulo *Buscar Palabra* situados al lado de cada botón *TagCloud de Palabras* (ver figura 4.9 - Flechas 3 y 4).

Volviendo al *Panel de Análisis* si se pulsa el botón *Diccionarios* (ver figura 4.8 - Flecha 2), se abre el *Panel de Diccionarios* (ver figura 4.11). Este panel posee dos tablas, la tabla de la izquierda lista todas las palabras en ingles que tiene el diccionario del comando de Linux *ispell*, esta lista de pala-

¹<http://opencloud.mcavallo.org>

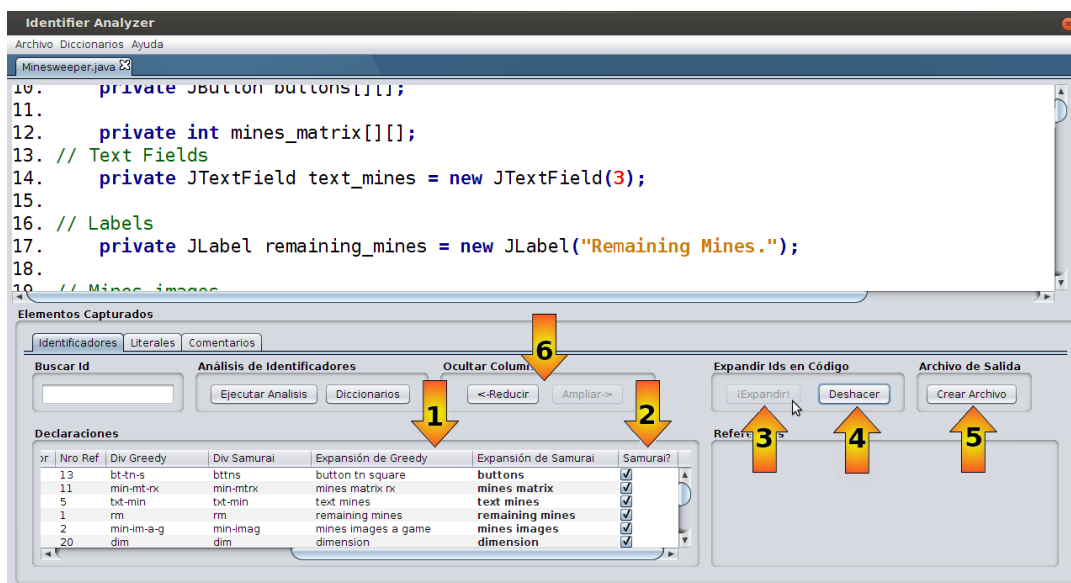
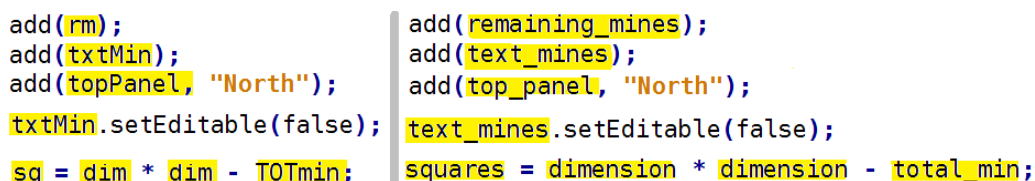


Figura 4.12: Panel de Elementos Capturados

bras es utilizada por el Algoritmo Greedy (ver sección 3.4.2) y el Algoritmo Expansión Básica (ver sección 3.4.4). 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). Este *Panel de Diccionarios* puede ser invocado 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 está al lado de *Ejecutar Análisis* (ver figura 4.5 - Flecha 8).

4.5.6. Panel de Elementos Capturados (Parte 2)

Una vez que los ids fueron analizados (Divididos y Expandidos) mediante el *Panel de Análisis*, el mismo debe ser cerrado 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



```
add(rm);
add(txtMin);
add(topPanel, "North");
txtMin.setEditable(false);
sq = dim * dim - TOTmin;

add(remaining_mines);
add(text_mines);
add(top_panel, "North");
text_mines.setEditable(false);
squares = dimension * dimension - total_min;
```

Figura 4.13: Comparación entre dos trozos de código

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 el *Panel de Análisis*. En estas nuevas columnas, también se muestran al final casillas de selección (ver figura 4.12 - Flecha 2). Estas casillas permiten al usuario elegir, la mejor expansión realizada desde Greedy o Samurai o ninguna de las 2 dejando al id en su formato original.

Al agregar las columnas nuevas se habilita el cuadro *Ocultar Columnas* (ver figura 4.12 - Flecha 6). En el se encuentran dos botones *Reducir* y *Ampliar*. 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, número de referencias y 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 (ver figura 4.12 - Flecha 6).

Cuando el usuario termina de seleccionar la mejor expansión de cada id, se procede al cuadro *Expandir Ids en Código*. Este cuadro contiene dos botones, al presionar *Expandir* (ver figura 4.12 - Flecha 3), la herramienta IDA reemplaza los ids del código de acuerdo a lo seleccionado en las casillas de selección en la tabla de *Declaraciones* que fue explicado al principio de esta sección. Una vez realizado esto, el código es más comprensivo para el usuario, en la figura 4.13 se puede observar dos trozos de códigos, el de la izquierda se observa el código normal, mientras que el de la derecha posee los ids expandidos. En caso de querer retrotraer la acción de reemplazo de ids, se puede pulsar en el botón *Deshacer* ubicado también en el cuadro *Expandir Ids en Código* (ver figura 4.12 - Flecha 4) y restablecer el código original.

Luego si el usuario lo decide, al presionar el botón *Crear Archivo* en el cuadro *Archivo de Salida* (ver figura 4.12 - Flecha 5), crea un nuevo archivo igual que el pasado como entrada, nada más que con los nuevos ids expandidos reemplazando los originales. Una vez realizado esto, se abre una ventana informando al usuario la creación del nuevo archivo.

4.6. Casos de Estudio

En esta sección se presentarán 3 casos de estudios realizados con la herramienta IDA. El primero será sencillo, mientras que los demás contendrán más ids para analizar. En cada uno de estos casos, se examinan los ids de un único archivo JAVA. A través de tablas, se irán mostrando los resultados parciales que se van obteniendo durante el proceso de análisis de los ids. Con estos casos, se pretende ostentar la utilidad de la herramienta IDA en lo que respecta al análisis de ids y mostrar que es un aporte al área de la CP.

4.6.1. Buscaminas (Minesweeper)

El archivo JAVA denominado Minesweeper.java, al ejecutarlo posee el clásico y conocido juego llamado Buscaminas (Minesweeper en Inglés - ver figura 4.14). Este archivo contiene 223 líneas que serán analizadas por IDA.

Al ingresar el archivo Minesweeper.java a la herramienta IDA, comienza la fase de extracción de datos y el analizador sintáctico captura información referente a los ids. Esta información la exhibe IDA al usuario en el *Panel de Elementos Capturados*, en la tabla *Declaraciones* (ver figura 4.5 - flecha 4). En la tabla 4.1 se aprecia esta información: la línea donde está declarado el id, que representa en el código analizado, el tipo, el modificador y la cantidad de referencias que tiene la declaración del id en el código.

Para hacer una descripción más detallada sobre los ids capturados, en la tabla 4.1 se exhibe que el archivo Minesweeper.java tiene ids del tipo *hardwords* y *softwords* (ver sección 3.3.1). Algunos *hardwords* que se pueden observar son `min_mtx`, `TOTmin`, `topPanel` (entre otros), ya que estos poseen una marca de separación que destacan las palabras que lo componen. Por

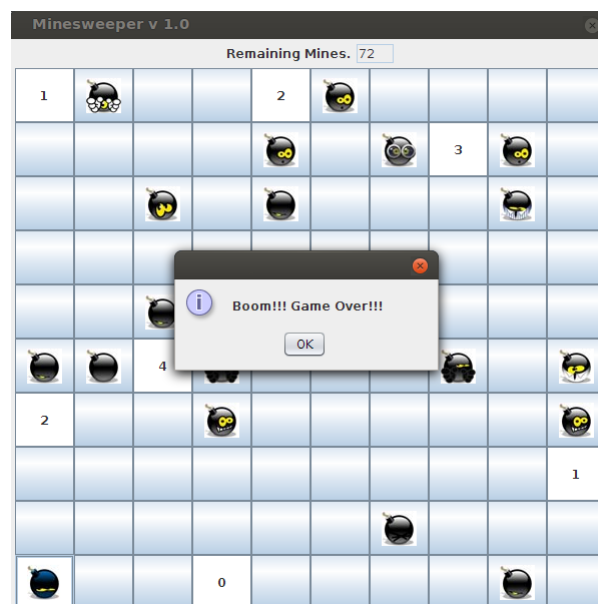


Figura 4.14: Captura del Juego Buscaminas programado en JAVA

otro lado, algunos de los *softwords* que se capturaron son `ae`, `plantmines`, `bttns` (entre otros).

A su vez, se capturaron los comentarios junto a la línea del código donde se ubican cada uno, los mismos se muestran en tabla 4.2. De la misma forma, los literales Strings que se extrajeron se exhiben en la tabla 4.3. En esta tabla, se puede observar que además del número de línea donde se ubica el literal, también muestra un eventual asociación que puede tener un literal con un id (Ejemplo: `msgDialog = "Message"`). Tanto los literales y los comentarios se visualizan en IDA a través del *Panel de Elementos Capturados*, eligiendo la pestaña correspondiente (ver figura 4.5 - Flecha 3).

Es importante recordar, que los comentarios y literales de las tablas 4.2 y 4.3 son usados para armar la lista de frases que se muestra en la ventana de *Palabras Capturadas* en la figura 4.9 - flecha 4. Esta información es útil para el Algoritmo de Expansión.

A continuación, se procede a analizar los ids, el usuario realiza esto con el *Panel de Análisis* (ver figura 4.8) aplicando las técnicas de división y después la técnica de expansión de abreviaturas. Los resultados más destacados se pueden apreciar en la tabla 4.4.

Análisis de Resultados

La información mostrada en la tabla 4.4, 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 los resultados de haber expandido las distintas partes del id, que resultaron desde los Algoritmos Greedy y Samurai respectivamente.

Los ids analizados por la herramienta IDA (ver tabla 4.4) en lo que respecta a *hardwords*, se pueden encontrar con guión bajo *min_imag*, *min_mtrx* para el tipo camel-case *uncoverEmpty*, *msgDialog* y para el caso especial *TOTmin*, *MINSnum* (variante camel-case), entre otros. El algoritmo Greedy manifiesta irregularidades a la hora de dividir ya que siempre considera que la mayor cantidad de divisiones es la mejor opción, esto puede observarse en casos como *min-im-ag*, *bt-tn-s*, *min-sn-um* (ver tabla 4.4 - columna Greedy). Para el caso especial *MINSnum* es interesante ver como Samurai se da cuenta de ello y lo separa correctamente (ver sección 3.4.3), mientras que Greedy supone que es del tipo camel-case haciendo la separación incorrecta *min-sn-um*.

En lo que respecta a *softwords* se aprecia la presencia de acrónimos como *rm* y *ae* (entre otros). El Algoritmo de Expansión consulta la lista de frases (ver figura 4.9 - flecha 4) conformada por comentarios y los literales capturados (ver tablas 4.2 y 4.1), 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 4.4 - Columnas de Expansión). El resto de los *softwords* se puede considerar a *mins*, *bttns*, *dim*, aquí Greedy también acusa inconvenientes separando los ids, mientras que Samurai no los divide (ver tabla 4.4).

Para finalizar, los ids *i*, *j* son comunes en la mayoría de los códigos, y son difíciles de traducir. Por ende, el Algoritmo de Expansión busca en las tablas de Literales y Comentarios (ver tablas 4.2 y 4.3), palabras que estén dentro del *Dominio del Problema* y de esta manera tratar de darle una traducción válida en este contexto.

Línea	Nombre ID	Representa	Tipo	Modificador	Nro. de Ref.
7	Minesweeper	Clase	–	public	–
10	bbtns	Variable de Clase	JButton	private	13
12	min_mtrx	Variable de Clase	int	private	11
14	txtMin	Variable de Clase	JTextField	private	5
17	rm	Variable de Clase	JLabel	private	1
20	min_imag	Variable de Clase	ImageIcon	private	2
22	dim	Variable de Clase	int	private	2
24	TOTmin	Variable de Clase	int	private	3
25	sq	Variable de Clase	int	private	6
37	topPanel	Variable Local	JPanel	–	3
48	middlePanel	Variable Local	JPanel	–	2
71	plantmines	Método de Clase	void	private	2
71	mins	Parámetro	int	–	1
102	main	Método de Clase	void	public	–
102	args	Parámetro	String[]	–	–
107	actionPerformed	Método de Clase	void	public	–
107	ae	Parámetro	ActionEvent	–	1
124	uncoverEmpty	Método de Clase	void	private	1
124	j	Parámetro	int	–	1
124	i	Parámetro	int	–	1
150	win	Método de Clase	void	private	1
159	boom	Método de Clase	void	private	1
172	msgDialog	Variable Local	String	–	1
179	nearbyMines	Método de Clase	int	private	1
188	MINSnum	Variable Local	int	–	5
179	xar	Parámetro	int	–	2
179	yac	Parámetro	int	–	2

Tabla 4.1: Identificadores extraídos por el Parser 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 4.2: Comentarios extraídos por el Parser ANTLR

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

Tabla 4.3: Literales extraídos por el Parser ANTLR

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	tot-min	tot-min	total mines	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
main	main	main	main	main
args	args	args	args	args
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 4.4: Análisis Realizado a los Ids extraídos de Minesweeper.java

4.6.2. Ejemplo 2

....

Capítulo 5

Conclusiones

La motivación de desarrollar la herramienta IDA, esta dada por la ausencia de herramientas con similares características. No abundan herramientas que posean interfaz amigable con el usuario y ejecuten técnicas que analicen ids de un código pasado entrada. Tampoco existen muchas implementaciones que extraigan ids, dividan ids y expandan abreviaturas en nombres de ids.

El uso del parser construido con ANTLR, permite extraer con facilidad los elementos estáticos presentes en el código que son necesarios para los algoritmos que analizan de ids. Estos elementos son, los ids como objetos principales y luego los comentarios, literales y documentación JAVA doc.

Cabe destacar que la herramienta IDA tiene implementada dos técnicas de división, Greedy y Samurai. La primera necesita consultar un diccionario de palabras en Inglés y un listado genérico de abreviaciones conocidas para llevar a cabo sus tareas. Ambas listas ocupan mucho espacio de almacenamiento y se utiliza una base de datos para hacer las consultas más eficientes.

En cambio, el algoritmo Samurai divide los ids mediante la utilización de recursos propios del código. Estos recursos son, los comentarios, los literales y documentación JAVA Doc que son extraídos mediante el parser antes mencionado. Con estos recursos, se arma un listado de frecuencias de aparición de palabras que son usadas en la función de scoring (ver sección nn). Por otro lado, suele ocurrir que estos recursos son escasos, por ende los autores decidieron armar un listado de palabras perteneciente a un conjunto amplio

de programas escritos en JAVA. Este listado, no solo ocupa menos espacio que los diccionarios de Greedy sino que están constituidos con palabras más adecuadas al ámbito de las ciencias de la computación. Esto implica que la división sea más eficiente y por ende que después la expansión sea más precisa.

Por otro lado, el algoritmo de expansión básico emplea los mismos diccionarios de palabras que utiliza Greedy, pero con la diferencia que consulta previamente la lista de frases capturadas del código, dando la preferencia a esta lista primero. La lista de frases se arma en función de los comentarios, literales y documentación JAVA Doc extraídos con el parser explicado al principio. Este algoritmo tiene el problema que ante múltiples alternativas de expansión, no sabe elegir una única opción.

Como trabajo futuro se planea implementar nuevas técnicas de análisis de ids en IDA. Una de ellas es AMAP (Automatically Mining Abbreviation Expansions in Programs). Esta técnica, no necesita de diccionarios con palabras en Inglés como el caso de el algoritmo básico de expansión y observa gradualmente en el código los comentarios y literales presentes partiendo desde el lugar del id que se desea expandir. También, resuelve el problema que posee el algoritmo básico cuando no sabe que opción elegir ante muchas opciones de expansión. Para lograrlo esto, el algoritmo prioriza la frecuencia de aparición de las palabras por cercanía de alcance estático partiendo del lugar donde se encuentre el id analizado. También AMAP permite entrenarse con con conjunto de programas pasado como entrada para recopilar más palabras y mejorar aún más la precisión de la expansión.

Otra mejora futura para la herramienta IDA es la posibilidad de acoplarla como plugin a un entorno de desarrollo como es el caso de NetBeans o Eclipse. Esto permitiría que el usuario abra un proyecto JAVA e inmediatamente con IDA expanda los ids para mejorar la comprensión.

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.
- [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.
- [3] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, page 216–234. Springer, 1999.
- [4] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE - Future of SE Track*, page 73–87, 2000.
- [5] M. Beron, P. Henriques, and R. Uzal. *Inspección de Programas para Interconectar las Vistas Comportamentaly Operacional para la Comprensión de Programas*. PhD thesis, Universidade do Minho, Braga, Portugal, 2010.
- [6] Mario Berón, Pedro Henriques, Maria João Pereira, and Roberto Uzal. Program inspection to interconnect behavioral and operational view for program comprehension. University of York, 2007.
- [7] Mario Berón, Daniel Eduardo Riesco, Germán Antonio 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.

-
- [8] 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. (early access).
 - [9] R. 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.
 - [10] 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.
 - [11] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proc. Int’l Conf. on Software Maintenance*, page 97–107. IEEE, 2000.
 - [12] Florian DeiBenbock and Markus Pizka. Concise and consistent naming. In *IWPC*, page 97–106. IEEE Computer Society, 2005.
 - [13] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM*, pages 602–611, 2001.
 - [14] Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.
 - [15] 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.
 - [16] E. Enslen, E. Hill, L. 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.

-
- [17] 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.
 - [18] 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.
 - [19] Fangfang Feng and W. Bruce Croft. Probabilistic techniques for phrase extraction. *Inf. Process. Manage.*, 37(2):199–220, 2001.
 - [20] José Luís Freitas, Daniela da Cruz, and Pedro Rangel Henriques. The role of comments on program comprehension. In *INForum*, 2008.
 - [21] Wilhelm Hasselbring, Andreas Fuhr, and Volker Riediger. First international workshop on model-driven software migration (mdsm 2011). In *CSMR '11 Proceedings of the 2011 15th European Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 299–300, Washington, DC, USA, März 2011. IEEE Computer Society.
 - [22] 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.
 - [23] IEEE. Ieee standard for software maintenance. *IEEE Std 1219-1998*, page i–, 1998.
 - [24] IEEE. *Standard Glossary of Software Engineering Terminology 610.12-1990.*, volume 1. IEEE Press, 1999.
 - [25] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Source Code Analysis and Manipulation*, 2007.

- SCAM 2007. Seventh IEEE International Working Conference on*, page 213–222, Sept 2007.
- [26] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What’s in a name? a study of identifiers. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on.*, page 3–12, June 2006.
- [27] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *SCAM*, page 139–148. IEEE Computer Society, 2006.
- [28] Dawn Lawrie, Henry Feild, and David Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388, 2007.
- [29] Michael P O’Brien. Software comprehension—a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report.*, 2003.
- [30] Roger S. Pressman. *Software Engineering - A Practitioner’s Approach*. McGraw-Hill, 5 edition, 2001.
- [31] T.A. Standish. *Data structure techniques*. Computer Sciences. Addison-Wesley, 1980.
- [32] M. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on.*, page 181–191, May 2005.
- [33] M. A. Storey, F. D. Fracchia, and H. 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.
- [34] P.F. Tiako. Maintenance in joint software development. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International.*, page 1077–1080, 2002.
- [35] Tim Tiemens. Cognitive models of program comprehension, 1989.

-
- [36] 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 *ASE Workshops.*, page 35–42. IEEE, 2008.
 - [37] A von Mayrhauser and AM. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995.