

## 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
<b>Identificadores</b>	3886684	33.0	<b>35723272</b>	<b>71.5</b>
<b>Total</b>	<b>11765175</b>	<b>100.0</b>	<b>49978507</b>	<b>100.0</b>

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 [?, ?]. 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 [?, ?] (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 [?, ?, ?].

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 [?] 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 letra<sup>1</sup> (`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 [?].

Los investigadores Feild, Binkley, Lawrie [?, ?, ?], 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`<sup>2</sup>

---

<sup>1</sup>Este nombramiento lo llaman acrónimo algunos autores.

<sup>2</sup>Este 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 [?].

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 [?, ?]:

**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 [?, ?, ?] 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 [?, ?].

Intuitivamente, se necesita que los ids representen bien al concepto, ya que mayor será impacto que tendrá en la interpretación del sistema [?, ?]. 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 [?] 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)<sup>1</sup> construida por Deissenboeck y Pizka [?] 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 [?, 6].

Se implementó como extensión de la IDE Eclipse 3.1<sup>2</sup>. 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).

---

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

<sup>2</sup><http://www.eclipse.org/jdt>

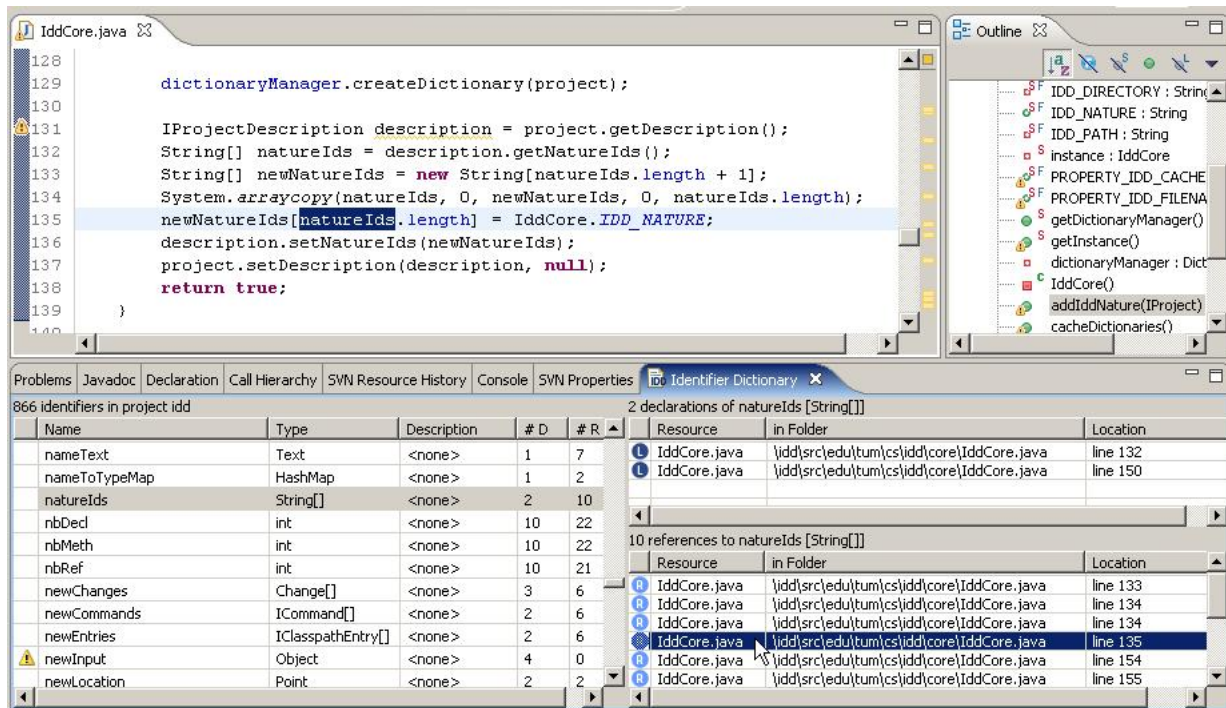


Figura 3.3: Visualización de Identifier Dictionary

**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 referenciado<sup>1</sup>.

**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 IDE<sup>2</sup> 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.

<sup>1</sup>Similar a los warnings de Eclipse

<sup>2</sup>Entornos 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);
```

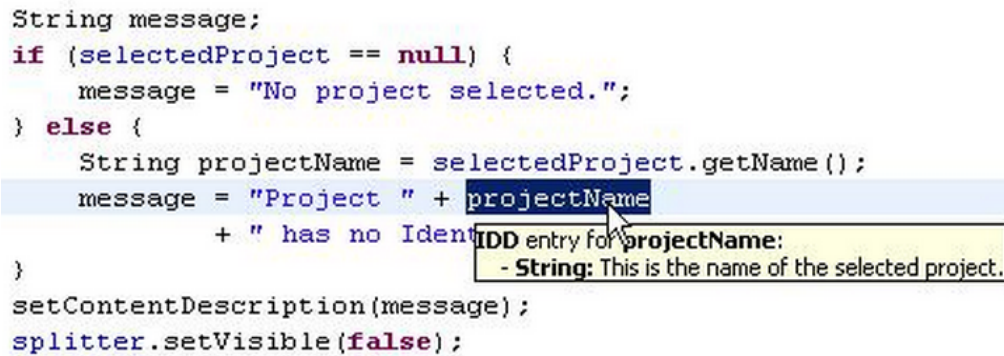


Figura 3.4: Visualización de Identifier Dictionary

**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 [?].

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 [6, ?] 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 [4, 6].

### 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 [6]:

1. **División:** Separar el id en las palabras que lo componen usando algún separador especial<sup>1</sup>. (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*.

---

<sup>1</sup>Siempre 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 [?, 6].

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 [5] 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, contracciones<sup>1</sup> 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.

---

<sup>1</sup>Palabras 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 [?, ?, ?, 6, 4] 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`<sup>1</sup>.

**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

---

<sup>1</sup>Comando 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** buscarPrefijo

---

**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 ( $s + ' ' + \text{buscarPrefijo}(\text{abrevSeparada}, s)$ )// Se extrae y se guarda el último caracter de  $s$ .5 *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  $\text{buscarPrefijo}(s, \text{abrevSeparada})$ 

---

---

**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 ( $\text{buscarSufijo}(\text{abrevSeparada}, s) + ' ' + s$ )// Se extrae y se guarda el primer caracter de  $s$ .5 *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 [4] 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-palabra<sup>1</sup> 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 [5] que se describe en próximas secciones.

La técnica Samurai según los autores [4] 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

---

<sup>1</sup>Que 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  $\text{score}_{sd}$  que corresponde a  $s$ .

La rutina primero examina cada punto posible de división en  $s$  dividiendo en  $\text{split}_{izq}$  y  $\text{split}_{der}$  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 autor<sup>1</sup> (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 ( $\text{score}_{sd}$  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  $\text{score}_{sd}$  (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 [4] 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 [4] 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-

---

<sup>1</sup>Listas 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 muestras<sup>1</sup> [4].

---

<sup>1</sup>Estos 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) [6] 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 [?], el principal recurso son los comentarios y los ids multi-palabras. En este punto se construye un acrónimo<sup>1</sup> 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)<sup>2</sup>. 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.).

---

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

<sup>2</sup>Esta 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`  $\rightarrow$  `horizontal`, `trgn`  $\rightarrow$  `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 [6, 5].

### 3.4.5. Algoritmo de Expansión AMAP

El algoritmo de expansión de abreviaturas que construyó Emily Hill, Zachary Fry, Haley Boyd [5] 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 [5] 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 [5], 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 una 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 [5], 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  $\mathbf{pa} = c_1, c_2, c_3, \dots, c_n$ , 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  $\mathbf{xml} \rightarrow \mathbf{e}xtensible \mathbf{m}arkup \mathbf{l}anguage$ .

**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  $\mathbf{pa} = c_1, c_2, c_3, \dots, c_n$ , 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  $\mathbf{arg} \rightarrow \mathbf{a}ccess \mathbf{r}ights$ , permitiendo más capturas que el patrón anterior.

En el algoritmo 6, se presenta la búsqueda por multi-palabras [5]. 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 &lt;code&gt;dst&lt;/code&gt; is &lt;code&gt;&gt;null&lt;/code&gt;  */ 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 [5].

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 [5] 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 [5] 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 API<sup>1</sup> 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

---

<sup>1</sup>Application 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 [5, ?, ?, 6].

### 3.4.6. Herramienta: Identifier Restructuring

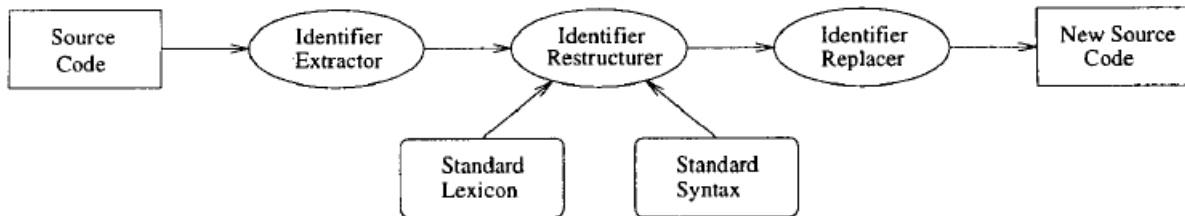


Figura 3.6: Etapas de Restructuring tool

La herramienta Restructuring Tool [?] 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.

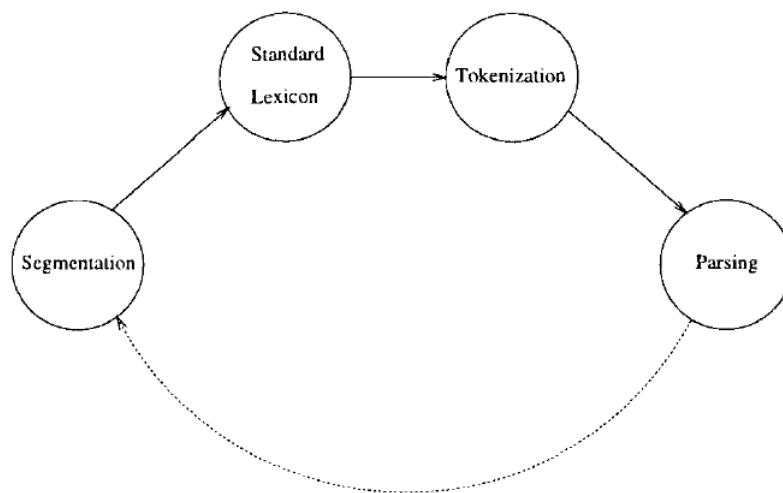


Figura 3.7: Etapas de Identifier Restructurer.

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 [3].

<b>FunctionId</b>	::=	[Context] (Action   PropertyCheck   Transformation)	
<b>Context</b>	::=	Qualifier <noun>	
<b>Qualifier</b>	::=	(<adjective>   <noun>)*	
<b>Action</b>	::=	SimpleAction   ComplexAction	
<b>SimpleAction</b>	::=	DirectAction   IndirectAction	
<b>ComplexAction</b>	::=	ActionOnObject   DoubleAction	
<b>IndirectAction</b>	::=	Qualifier <noun> ActionSpecifier	{Head word = <noun>}
<b>DirectAction</b>	::=	<verb> ActionSpecifier	{Head word = <verb>}
<b>ActionOnObject</b>	::=	<verb> Qualifier <noun> ActionSpecifier	{Head words = <verb>, <noun>}
<b>DoubleAction</b>	::=	(DirectAction   ActionOnObject) <sup>2</sup> {Head words from DirectAction and/or ActionOnObject}	
<b>ActionSpecifier</b>	::=	(<adjective>   <adverb>   <preposition> Qualifier <noun>)*	
<b>PropertyCheck</b>	::=	"is" Qualifier (<adjective>   <noun>) ActionSpecifier	{Head word = <adjective>   <noun>}
<b>Transformation</b>	::=	Source TransformOp Target	{Head words from Source and Target}
<b>Source</b>	::=	Qualifier (<adjective>   <noun>)	{Head word = <adjective>   <noun>}
<b>Target</b>	::=	Qualifier (<adjective>   <noun>)	{Head word = <adjective>   <noun>}
<b>TransformOp</b>	::=	"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 [3] (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 [?] 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 [3]. 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 [?] (figura 3.7).

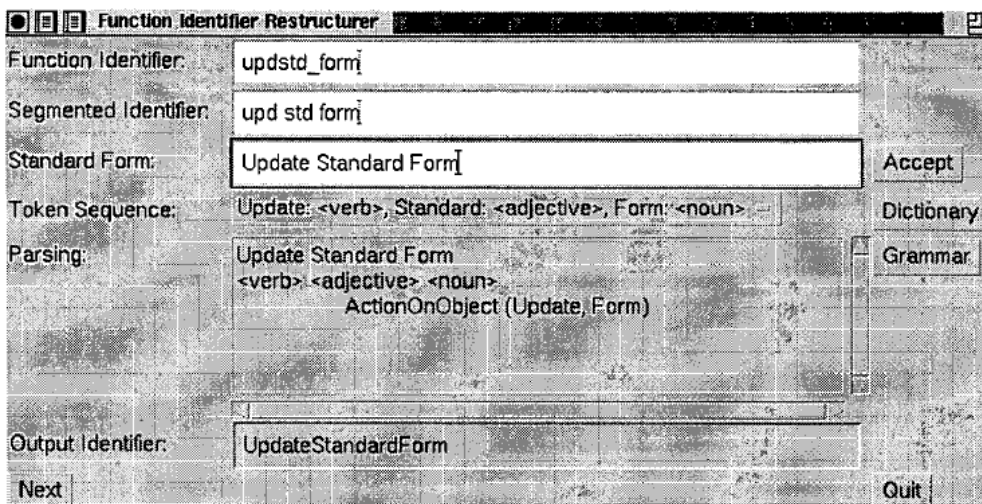


Figura 3.9: Visualización de Restructuring Tool.



### 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)

### 4.1. Introducción

En el capítulo anterior, se explicó la importancia de analizar identificadores (ids) ubicados en el código fuente de un sistema de software. Los ids de un programa normalmente están compuestos por más de una palabra en forma de abreviatura, por ejemplo: `inp_fl.sys`. Detrás de estas abreviaturas, los ids ocultan información es propia del Dominio del Problema [3, 6, 5, 4]. Desafortunadamente, las personas ajenas al código, no comprenden a simple vista la información que los ids poseen en sus abreviaturas e invierten tiempo en entenderlas. Es por esto, que las herramientas automáticas de análisis de ids son bienvenidas en el ámbito de la Comprensión de Programas (CP). Con estas herramientas se logra disminuir los tiempos de comprensión de ids y revelar la información que estos contienen en sus abreviaturas.

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

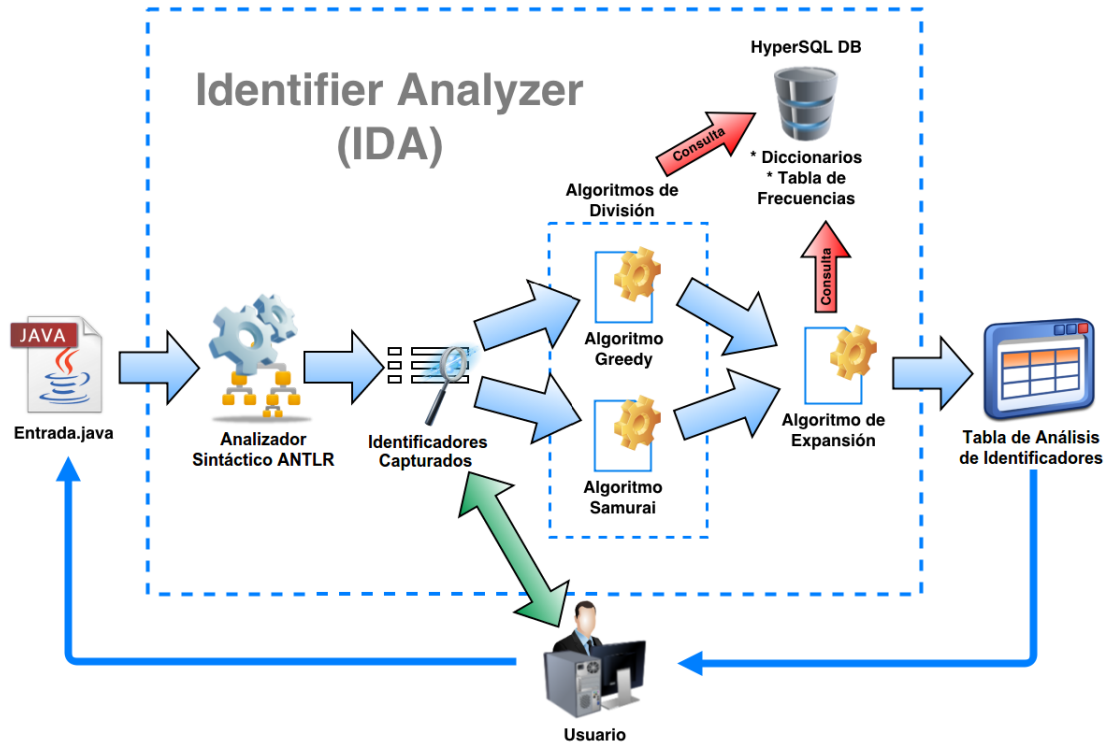


Figura 4.1: Arquitectura de IDA.

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

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

## 4.2. Arquitectura

En la figura 4.1 se puede apreciar la arquitectura de IDA. Esta arquitectura describe tres partes principales, la primera consiste en la *extracción de datos*, la segunda trata sobre la *división de ids* y la tercera sobre *expansión de ids*. A continuación se detallan cada una de ellas.

**Módulo de Extracción de Datos:** Este módulo recibe como entrada un archivo JAVA que ingresa el usuario (ver Figura 4.1 Entrada.java), luego este archivo se procesa por un Analizador Sintáctico (AS) (ver Figura 4.1 Analizado Sintáctico ANTLR). El AS, extrae y almacena, en estructuras internas, la información estática perteneciente al código del archivo ingresado. Esta información, está relacionada con ids, literales y comentarios (ver próxima sección para más detalles). El usuario a través de la interfaz de IDA, puede visualizar esta información capturada del código por medio de tablas claramente definidas (ver figura 4.1 - Flecha Verde).

**Módulo de División de Ids:** Una vez completada la extracción de información, el proceso continua en el módulo de división de ids. Aquí se encuentran implementados dos algoritmos de división; uno es el Algoritmo Greedy y el otro es el Algoritmo Samurai ambos explicados en la sección 3.4.2 y 3.4.3 del capítulo anterior. Estos algoritmos reciben como entrada la información capturada en el módulo de extracción de datos (ids, comentarios, literales), y luego estos algoritmos dividen los ids del archivo JAVA (ver figura 4.1 - Algoritmos de División). Los resultados de las divisiones se almacenan en estructuras internas que serán utilizadas por el módulo de expansión. Cabe recordar que estos algoritmos de división necesitan datos externos para funcionar, uno es el diccionario de palabras (en caso de Greedy) y el otro es lista de frecuencias globales de aparición de palabras (en caso de Samurai). Estos datos externos se encuentran almacenados en una base de datos embebida (ver Figura 4.1 - HyperSQL DB).

**Módulo de Expansión de Ids:** La tercera y última parte, tiene implementado el Algoritmo de Expansión Básico de abreviaturas que fue explicado en la sección 3.4.4 del capítulo anterior. Este algoritmo, toma como

entrada los ids separados en el módulo de división de ids (tanto de Greedy como de Samurai), luego el Algoritmo de Expansión expande las abreviaturas resultantes producto de la división de ids (ver Figura 4.1 - Algoritmo de Expansión). Los resultados de las expansiones se retornan en dos grupos: las expansiones provistas por el Algoritmo de división Greedy y las provistas por el Algoritmo de división Samurai.

El Algoritmo de Expansión también necesita de un diccionario de palabras, por eso se realizan consultas a la base de datos embebida (ver Figura 4.1 - HyperSQL DB). Finalmente, los resultados de las divisiones y las expansiones de los ids, se muestran en una tabla (ver Figura 4.1 - Tabla de análisis de identificadores).

### 4.3. Analizador Sintáctico

Como se explicó en la sección previa, cuando el usuario ingresa un archivo JAVA, IDA examina y extrae información estática presente en el archivo ingresado. Esta información está compuesta por identificadores, comentarios y literales. La manera en que IDA extrae esta información es a través de un Analizador Sintáctico (AS).

La construcción de este AS se llevó a cabo, primero investigando herramientas encargadas de construir AS. Se dio preferencia a aquellas que emplean la teoría asociada a las gramáticas de atributos [1]. De la investigación previamente descrita, se determinó que la herramienta *ANTLR*<sup>1</sup> era la que mejor se ajustaba a las necesidades antes planteadas. Esta herramienta permite agregar acciones semánticas (escritas en JAVA) para el cálculo de los atributos, en una gramática de lenguaje JAVA<sup>2</sup>. Estas acciones semánticas deben estar correctamente insertadas en la gramática para, por ejemplo, implementar estructuras de datos y algoritmos que capturan los ids utilizados en un programa [2]. Una vez insertadas estas acciones, ANTLR lee la gramática y genera el AS adicionando acciones que fueron programadas. De esta manera, se obtiene un AS que recolecta ids mientras examina el código.

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

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

A su vez a estas acciones semánticas, se le agregan otras acciones que extraen comentarios y literales strings. Estos elementos son necesarios ya que sirven para los algoritmos de análisis de ids que serán explicados en próximas secciones.

## 4.4. Base de Datos Embebida

Como se describió en secciones previas, IDA posee una base de datos embebida. Esta base de datos utiliza una tecnología llamada HSQLDB<sup>1</sup>. Dado que HSQLDB esta desarrollada en JAVA, al momento de incorporarla en IDA (que está programada en JAVA) no resulto una tarea difícil. Otra ventaja por la cual se eligió esta tecnología, es que responde rápidamente las consultas. Esto es importante ya que todos los algoritmos de IDA consultan a HSQLDB. Dentro de esta base de datos embebida, se encuentran almacenadas los diccionarios/listas de palabras que IDA necesita para llevar adelante sus tareas. Estos diccionarios/listas se describen a continuación, nombrando también que algoritmo de IDA consulta cada diccionarios/listas.

**Diccionario en Inglés (ispell):** Contiene palabras en Inglés que pertenecen a la lista de Palabras Comando de Linux *Ispell*<sup>2</sup>. Se utiliza en el Algoritmo de Greedy y en el Algoritmo de Expansión (ver capítulo 3).

**Lista de Palabras Excluyentes (stop-list):** Esta compuesta con palabras que son poco importantes o irrelevantes en el análisis de ids<sup>3</sup>. Se utiliza en el Algoritmo de Greedy y en el Algoritmo de Expansión (ver capítulo 3).

**Lista de Abreviaturas y Acrónimos Conocidas:** Contiene abreviaturas comunes del idioma Inglés y Acrónimos conocidos de programación<sup>4</sup> (gif, jpg, txt). Se emplea en el Algoritmo Greedy (ver capítulo 3).

---

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

<sup>2</sup><http://wordlist.aspell.net>

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

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

**Lista de Prefijos y Sufijos Conocidos:** Posee Sufijos y Prefijos conocidos en Inglés<sup>1</sup>, esta lista fue confeccionada por el autor del Algoritmo Samurai (ver capítulo 3). Se consulta solo en dicho algoritmo.

**Frecuencias Globales de Palabras:** Lista de palabras, junto con su frecuencia de aparición. Esta lista fue construida por el autor del Algoritmo Samurai<sup>2</sup>. Se emplea solo en dicho algoritmo, más precisamente en la función de *Scoring* (ver capítulo 3).

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

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

## 4.5. Proceso de Análisis de Identificadores

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

### 4.5.1. Barra de Menú

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

---

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

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



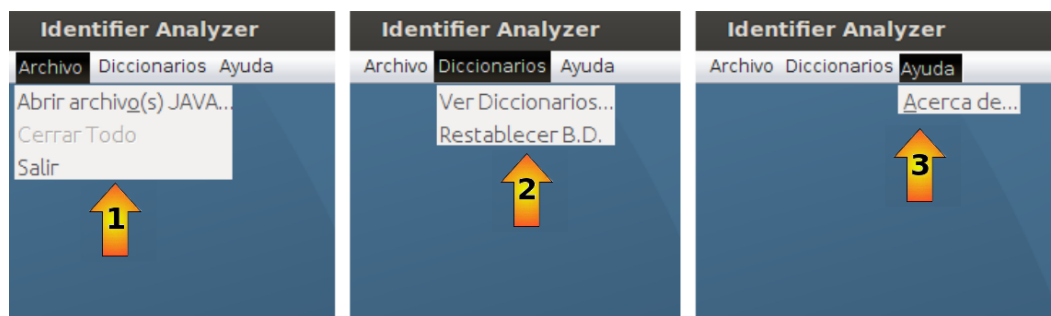


Figura 4.2: Barra de Menú de IDA

Al pulsar<sup>1</sup> en *Archivo* de la barra antedicha, se despliega un menú con las opciones que se describen a continuación (ver figura 4.2 - Flecha 1):

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

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

**Salir:** Cierra la Aplicación.

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

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

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

---

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

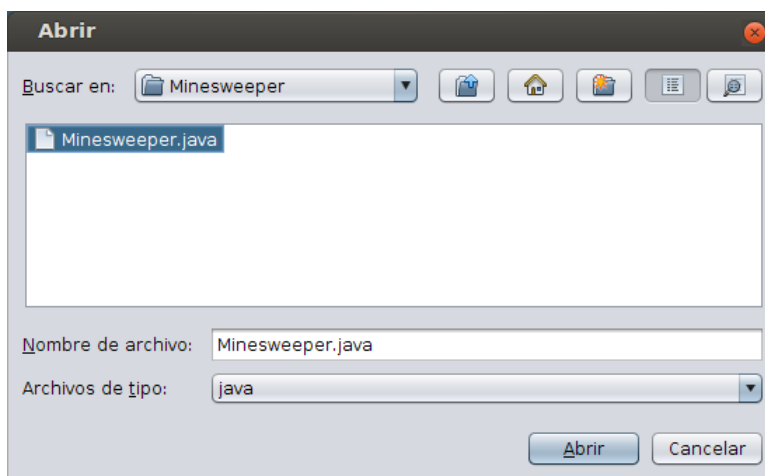


Figura 4.3: Ventana para seleccionar Archivos JAVA.

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

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

#### 4.5.2. Lectura de Archivos JAVA

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

Una vez que el usuario elige el/los archivo/s, IDA utiliza un programa externo llamado JACOB<sup>1</sup>. Este programa JACOB, recibe como entrada un archivo JAVA y embellece el código fuente que esta contenido en el. Este embellecimiento se realiza para facilitar la lectura del código al usuario. En la próxima sección se describe el panel que IDA tiene para visualizar el código leído del archivo.

La herramienta IDA, además realiza un control de los archivos abiertos, impidiendo que se abra el mismo archivo más de una vez. En caso de que esto suceda, se muestra un cartel informando al usuario (ver figura 4.4). Este

---

<sup>1</sup><http://www.tiobe.com/jacobe>

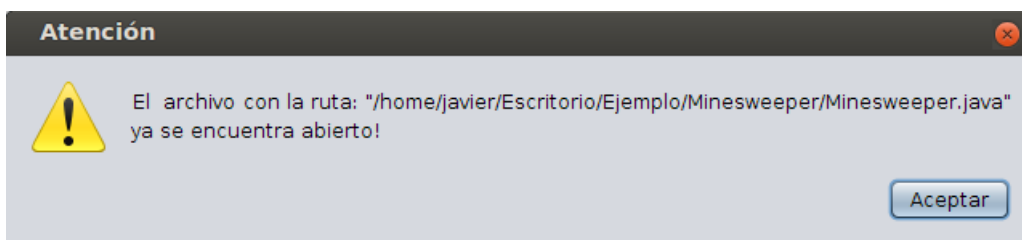


Figura 4.4: Aviso sobre Archivo JAVA ya abierto en IDA.

control se realiza por cuestiones de coherencia al momento de analizar los archivos.

### 4.5.3. Panel de Elementos Capturados

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

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

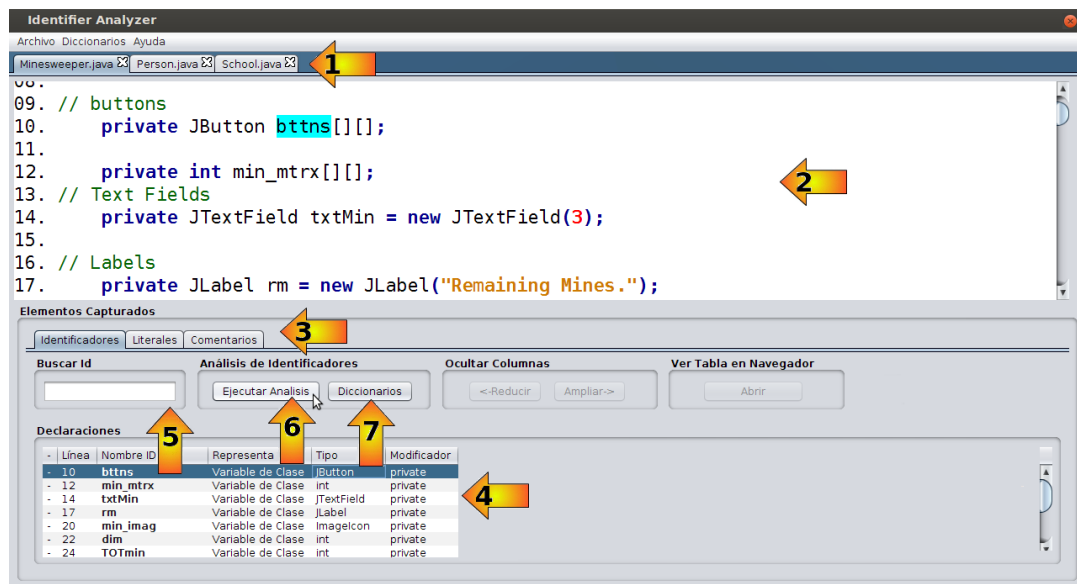


Figura 4.5: Panel de Elementos Capturados

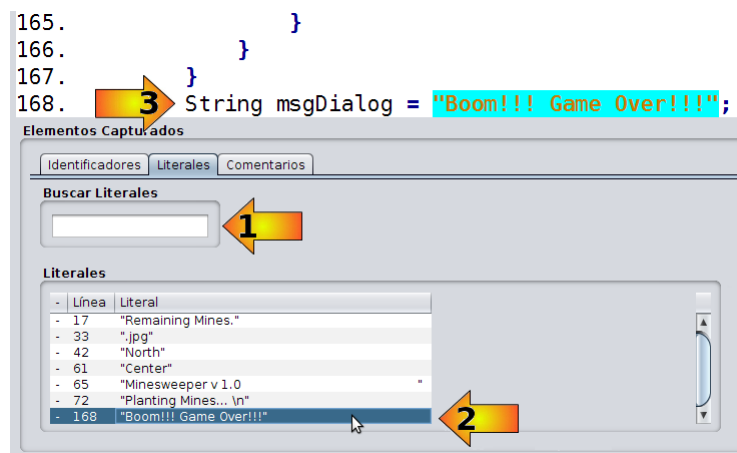


Figura 4.6: Literales Capturados

### Pestaña de Identificadores Capturados

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

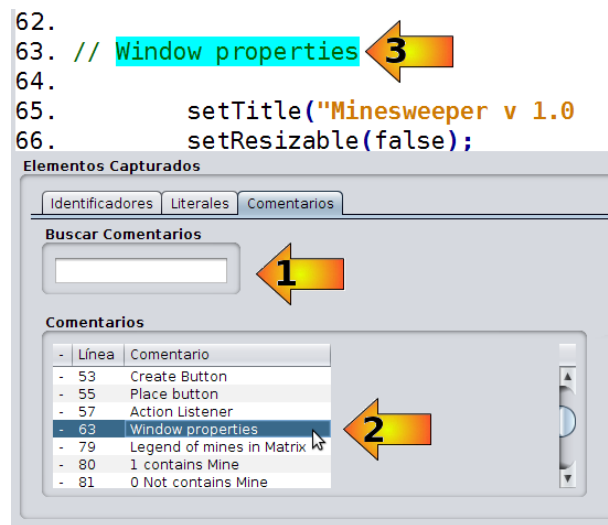


Figura 4.7: Comentarios Capturados

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

### Pestaña de Literales Capturados

Al pulsar la *Pestaña de Literales* (ver figura 4.5 - Flecha 3), se puede apreciar que aparece una *Tabla de Literales* y un práctico buscador en forma de cuadro de texto (ver figura 4.6 - Flecha 1 y 2). Esta tabla contiene dos columnas, número de línea del literal y el literal propiamente dicho (ver figura 4.6 - Flecha 2). De manera similar a la que se describió en el párrafo precedente, el buscador filtra los resultados en la *Tabla de Literales* mientras

se va escribiendo en el. También al pulsar sobre alguna fila, automáticamente el literal correspondiente se resalta en el código que está ubicado en la parte superior (ver figura 4.6 - Flecha 3).

### Pestaña de Comentarios Capturados

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

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

#### 4.5.4. Ventana de Análisis

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

**Parte Izquierda:** Posee un listado con los ids capturados, en el cuadro inferior izquierdo (ver figura 4.8 - Flecha 1). Arriba de estos se encuentran dos botones *Palabras Capturadas* y *Diccionarios* (ver figura 4.8 - Flecha 2), al pulsarlos le brindan información al usuario sobre los datos



Figura 4.8: Ventana de Análisis

que se utilizan para ejecutar los algoritmos de análisis, y ambos botones serán explicados con más detalles en la próxima sección.

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

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

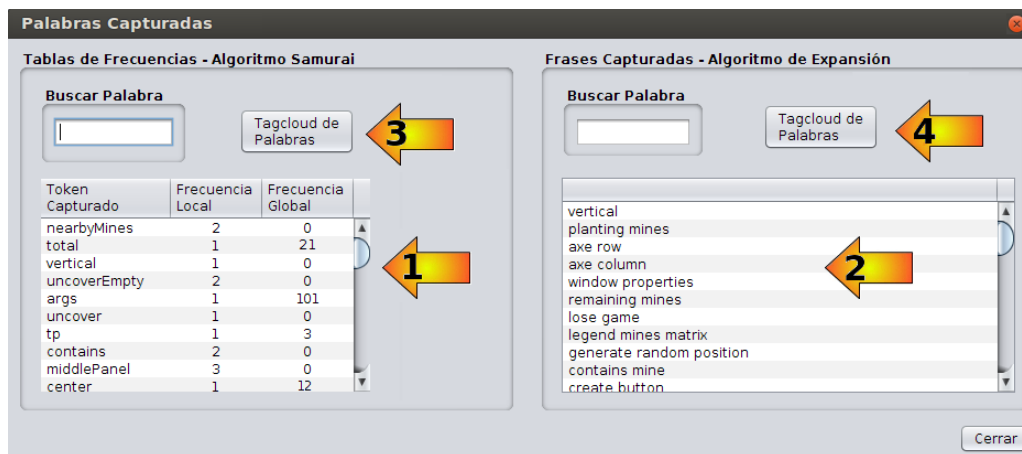


Figura 4.9: Información Utilizada para el Análisis de Ids

#### 4.5.5. Palabras Capturadas y Dicionarios

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

##### Ventana de Palabras Capturadas

Al pulsar el botón *Palabras Capturadas*, se abre una ventana que posee dos cuadros (ver figura 4.9), el cuadro de la izquierda contiene una tabla que muestra las frecuencias correspondiente al Algoritmo Samurai (ver figura 4.9 - Flecha 1). Esta tabla tiene tres columnas, en la primera posee tokens<sup>1</sup>, los mismos fueron capturados por el AS 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 actual. 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).

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

<sup>1</sup>El concepto “token”, en este contexto, significa las palabras que están contenidas en literales, comentarios e ids, este último necesita un proceso especial, para más detalles ver cap. 3 - sección 3.4.3



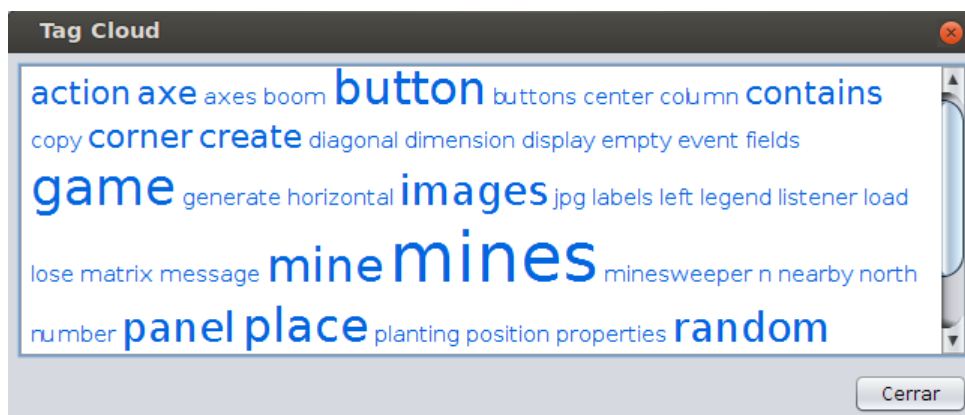


Figura 4.10: TagCloud: Nube de Etiquetas

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 capítulo 3 - sección 3.4.4). Este algoritmo las usa, ya que estas frases son posibles candidatas a expandir las abreviaturas en forma de acrónimo que puede contener un id (Ejemplo: fl  $\rightarrow$  file system).

A modo de agilizar las búsquedas en las tablas descriptas anteriormente (la de frecuencias de Samurai y la de frases), se puede llevar a cabo utilizando los cuadros de texto con rótulo *Buscar Palabra* situados al lado de cada botón *TagCloud de Palabras* (ver figura 4.9 - Flechas 3 y 4).

### Ventana de TagCloud (Nube de Etiquetas)

Como se puede observar en la figura 4.9 - Flechas 3 y 4, existen dos botones con el nombre de *TagCloud de Palabras*, cada uno abre una ventana que contiene una *Nube de Etiquetas* (ver figura 4.10). Esta nube posee palabras y resalta en tamaño más grande aquellas palabras que más frecuencia de aparición tienen (En la figura 4.10 las palabras **mine**, **mines**, **game** etc. son las que mayor aparición tienen). Para generar esta nube se emplea una librería de JAVA llamada OpenCloud<sup>1</sup>. Esta *Nube de Etiquetas* ayuda a ver con más claridad que palabras son más frecuentes en el código.

<sup>1</sup><http://opencloud.mcavallo.org>

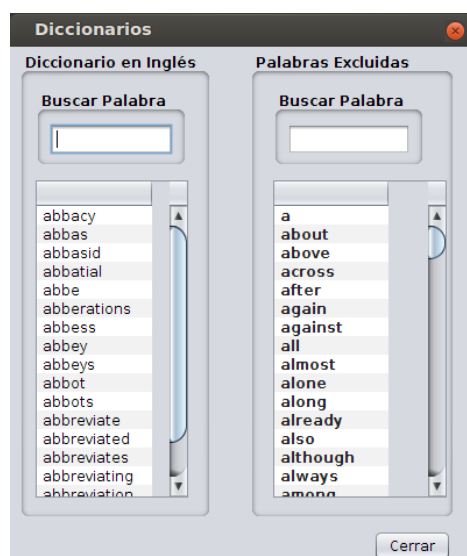


Figura 4.11: Ventana de Diccionarios

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 número de apariciones dentro de esta tabla de frases (ver figura 4.9 - Flecha 2).

### Ventana de Diccionarios

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

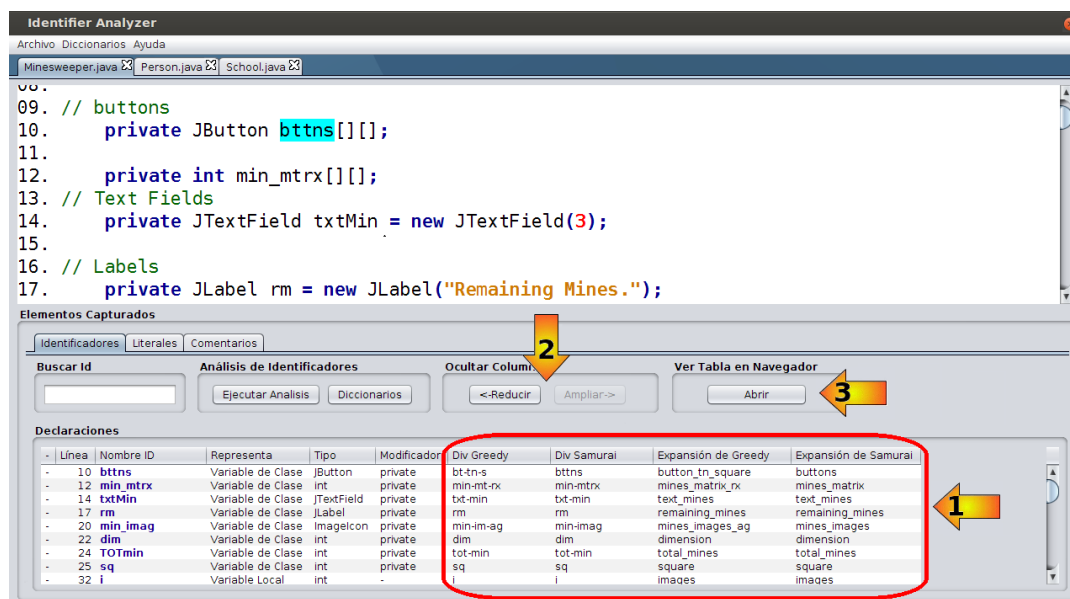


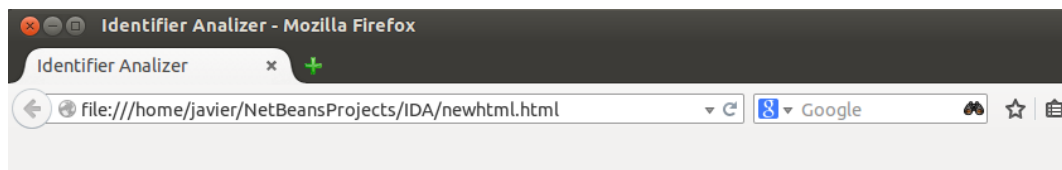
Figura 4.12: Panel de Elementos Capturados

barra de menú (ver Figura 4.2 - Flecha 2). Otro sitio donde puede abrirse, es desde el *Panel de Elementos Capturados*, pulsando el botón que esta al lado de *Ejecutar Análisis* (ver figura 4.5 - Flecha 7).

Todas las ventanas que fueron descriptas previamente (palabras capturadas, tagcloud, diccionarios), en caso de haber sido abiertas por el usuario, el mismo debe cerrarlas si desea continuar con el proceso de análisis de ids.

#### 4.5.6. Nuevamente al Panel de Elementos Capturados

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



## Identifier Analyzer

### Análisis de Identificadores: Minesweeper.java

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

Figura 4.13: Tabla del Análisis de Ids en Navegador Web.

Expansión desde Greedy y Expansión desde Samurai.

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

Luego si el usuario lo decide, puede presionar el botón *Abrir* en el cuadro *Ver Tabla en Navegador* (ver figura 4.12 - Flecha 3). Esta acción abre automáticamente el navegador web por defecto del sistema operativo, y me-

diante una página web en formato html, se muestra una tabla con los resultados obtenidos producto del análisis de ids (ver figura 4.13). De esta manera, el usuario puede visualizar más claramente el análisis realizado, permitiendo también imprimir los resultados en papel (mediante el navegador web), si el usuario lo desea.

Dando como finalizado el proceso necesario que debe realizar el usuario para analizar los ids con la herramienta IDA, en la próxima sección se explican los casos de estudios realizados que demuestran la utilidad de IDA.

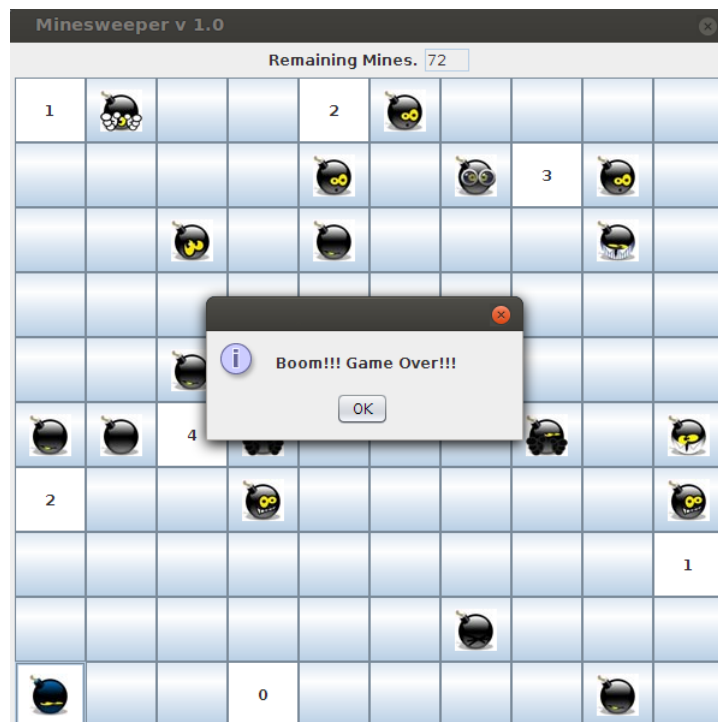


Figura 4.14: Captura del Juego Buscaminas programado en JAVA

## 4.6. Casos de Estudio

En esta sección se presentarán tres casos de estudio realizados con la herramienta IDA. 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 programa JAVA denominado Minesweeper.java, al ejecutarlo posee el clásico y conocido juego llamado Buscaminas (Minesweeper en Inglés - ver figura 4.14). El juego consiste en despejar los casilleros de un tablero que posee minas, estas minas están ocultas y el jugador pierde en caso de detonar alguna. Este programa contiene un módulo de 250 líneas aproximadamente que fueron analizadas por IDA.

#### Captura de Información

Al ingresarse el programa Minesweeper.java a la herramienta IDA, se da comienzo a la fase de extracción de datos y el analizador sintáctico (AS) captura información referente a los ids. Esta información la exhibe IDA al usuario en el *Panel de Elementos Capturados*, en la tabla *Declaraciones* (ver figura 4.5 - flecha 4). En la tabla 4.1 se puede apreciar esta información (por columnas): la línea donde está declarado el id, el nombre, que representa en el código analizado, el tipo del id y que modificador posee.

Para hacer una descripción más detallada sobre los ids capturados, en la tabla 4.1 se puede observar que el archivo Minesweeper.java tiene ids del tipo *hardwords* y *softwords* (ver capítulo 3 - sección 3.3.1). Algunos *hardwords* que se pueden observar son `min_mtx`, `TOTmin`, `topPanel` (entre otros), ya que estos poseen una marca de separación que destacan las palabras que lo componen. Por otro lado, algunos de los *softwords* que se capturaron son `ae`, `plantmines`, `bttns`, `xar` (entre otros). A su vez, se capturaron los comentarios,

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 ambas tablas se muestran las líneas en donde se ubica cada comentario y literal en el código. Tanto los literales y los comentarios se visualizan en IDA a través del *Panel de Elementos Capturados*, eligiendo la pestaña correspondiente (ver figura 4.5 - Flecha 3).

Es importante recordar, que los comentarios y literales de las tablas 4.2 y 4.3 son usados para construir la lista de frases que se muestra en la ventana de *Palabras Capturadas* (ver figura 4.9 - flecha 2). Esta información es útil para el Algoritmo de Expansión. De la misma manera, con los comentarios y literales se construye una parte importante del listado de frecuencias locales de aparición de tokens<sup>1</sup>, que son destinados a ser usados por el Algoritmo de división Samurai. Esta lista se visualiza en IDA también dentro de la ventana de *Palabras Capturadas* (ver figura 4.9 - flecha 1).

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

---

<sup>1</sup>El concepto “token”, en este contexto, significa las palabras que están contenidas en literales, comentarios e ids, este último necesita un proceso especial, para más detalles ver cap. 3 - sección 3.4.3

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

Tabla 4.1: Identificadores extraídos por el AS ANTLR



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

Tabla 4.2: Comentarios extraídos por el AS ANTLR

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

Tabla 4.3: Literales extraídos por el AS ANTLR

## Análisis de Resultados

Cuando el usuario ejecuta las técnicas de análisis de ids, lo lleva a cabo mediante la *Ventana de Análisis* (ver figura 4.8), aquí se aplican las técnicas de división (Greedy y Samurai) y después la técnica de expansión de abreviaturas. Luego, esta *Ventana de Análisis* muestra debajo los resultados obtenidos (ver figura 4.8 - Flechas 4 y 6). Para el caso del programa Minesweeper.java, los resultados que se obtuvieron producto del análisis de ids, se pueden apreciar en la tabla 4.4.

Con respecto a 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 las expansiones realizadas de las distintas partes del id, que fueron efectuadas desde los Algoritmos Greedy y Samurai respectivamente.

Los ids analizados por IDA de Minesweeper.java (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 (ver capítulo 3 - sección 3.4.2), 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 observar como Samurai se da cuenta de donde hacer la división, y no la considera un caso común de camel-case que la separa antes de la mayúscula seguido de minúscula `mins-num` (ver capítulo 3 - sección 3.4.3). No sucediendo lo mismo con Greedy ya supone que es del tipo camel-case, y realiza la separación incorrecta `min-sn-um`.

En lo que respecta a *softwords* se aprecia la presencia de acrónimos como `rm` y `ae` (entre otros); ambos algoritmos de división no los dividen. La razón de esto es porque poseen solo dos caracteres y el algoritmo lo considera acrónimos. A la hora de expandir `rm`, `ae`, el Algoritmo de Expansión consulta la lista de frases (ver figura 4.9 - flecha 2) 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 y procede a separar estos ids siendo que no se deben separar, caso contrario ocurre con Samurai (ver tabla 4.4 - Columnas de División).

Continuado en la tabla 4.4, los ids `i`, `j` son comunes en la mayoría de los códigos, y generalmente se utilizan como contadores en estructuras de control iterativas (`for`, `while`, etc). En estos casos, el Algoritmo de Expansión busca palabras que estén dentro del *Dominio del Problema*. Para lograrlo, recurre a las tablas de Literales y Comentarios (ver tablas 4.2 y 4.3), luego trata de encontrar palabras que comiencen con las letras respectivas, para este caso `image`, `jpg`. De esta manera el algoritmo, trata de darle una traducción válida en este contexto. En caso de no haber coincidencia con ninguna, el algoritmo de expansión buscará en el diccionario de palabras en Inglés.

Para concluir, en las columnas de expansión de la tabla 4.4 se visualizan palabras como: ventana, vacío, plantar, minas, recuadro, botones, panel (entre otras). Estas palabras intuitivamente se consideran que forman parte del *Dominio del Problema* correspondiente al programa Minesweeper.java.

Id	Greedy	Samurai	Exp. desde Greedy	Exp. desde Samurai
Minesweeper	minesweeper	minesweeper	minesweeper	minesweeper
bttns	bt-tn-s	bttns	button tn square	buttons
min mtrx	min-mt-rx	min-mtrx	mines matrix rx	mines matrix
txtMin	txt-min	txt-min	text mines	text mines
rm	rm	rm	remaining mines	remaining mines
min_imag	min-im-ag	min-imag	mines images ag	mines images
dim	dim	dim	dimension	dimension
TOTmin	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
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

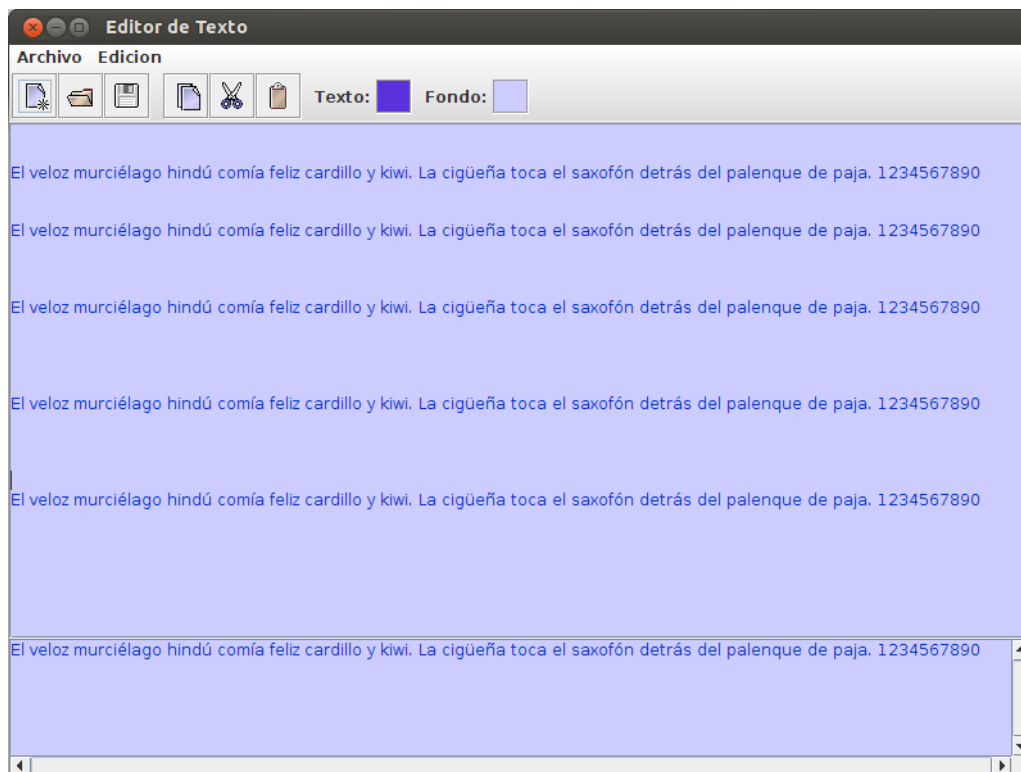


Figura 4.15: Captura del Editor de texto programado en JAVA

#### 4.6.2. Editor de Texto

En el próximo caso de estudio, se seleccionó un programa escrito en JAVA que corresponde a un editor de texto (ver figura 4.15). Este editor posee las herramientas básicas para crear o modificar archivos de textos sin formato, permite cambiar la visualización de colores en las letras y en el fondo, también imprime los archivos que se abran en el. Este editor esta programado con alrededor de 500 líneas de código, las cuales están escritas en un único archivo llamado Editor.java.

Cabe destacar que, a modo de facilitar la lectura, de aquí en adelante, por cada caso de estudio, se mostrará únicamente la tabla con resultados obtenidos producto del análisis de los ids y se excluirán el resto de las tablas que se mostraron en el caso de estudio anterior.

A continuación, se brindan los resultados obtenidos de analizar con la herramienta IDA el editor de textos, el análisis efectuado en los ids se exhibe

en la tabla 4.5. Dado que este programa posee muchos ids, los resultados que se presentan en la tabla 4.5 son los más destacados y no son la totalidad.

### Análisis de Resultados

A simple vista, se puede observar en la tabla 4.5, al igual que el caso de estudio de la sección anterior, que las divisiones entre Greedy y Samurai, las de este último son las que mejor se realizan. Como se explico anteriormente, esto ocurre por que Greedy siempre selecciona la mayor cantidad de divisiones en la palabra como la mejor opción (ver capítulo 3 - sección 3.4.2). Algunos ejemplos de divisiones mal hechas por Greedy son `bckCol` → `b-c-k-col`, `btAccept` → `bt-ace-pt`, `jChoColor` → `j-c-ho-color`, entre otros.

Sin embargo, en este caso de estudio a diferencia del anterior existen algunas divisiones en las que Samurai falla, los ids que se pueden observar son: `flreader`, `ptjob` los cuales directamente no se dividieron como lo hizo Greedy en `fl-reader`, `pt-job`. La primer hipótesis que se maneja sobre este resultado, es que Greedy encuentra en el diccionario en Inglés las palabras `reader` y `job` y con esto basta para llevar a cabo la división. Mientras que Samurai en su función de *Score*, las palabras `fl` con `reader` y `pt` con `job` no representan puntajes (score) altos para ser divididas entre ellas (ver capítulo 3 - sección 3.4.3).

En lo que respecta a la expansión de ids, la mayoría de las abreviaturas resultantes de la división de ids fueron expandidas; algunos ejemplos son `sel` → `select`, `tl` → `tool`, `cl` → `close`, `bck` → `background`, entre otras. Estas palabras son expandidas por el algoritmo de expansión, gracias a los comentarios y literales capturados por el AS. Por otro lado, existen algunas abreviaturas que no se expanden como es el caso de `bt` a `button`, `it` a `item`; que forman parte de los ids `bt-accept` y `men-it-new`. La hipótesis de este comportamiento en el Algoritmo de Expansión, se debe a que estas abreviaturas son tratadas como acrónimos (abreviaturas con más de una palabra) dado que tienen solo dos caracteres y no existe una frase (de comentarios o literales) capturada por el AS, que coincida. Como se consideran abreviaturas con más de una palabra, tampoco se utiliza el diccionario en Inglés para expandir.

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

Tabla 4.5: Resultados destacados del Análisis Realizado a los Ids extraídos de Editor.java

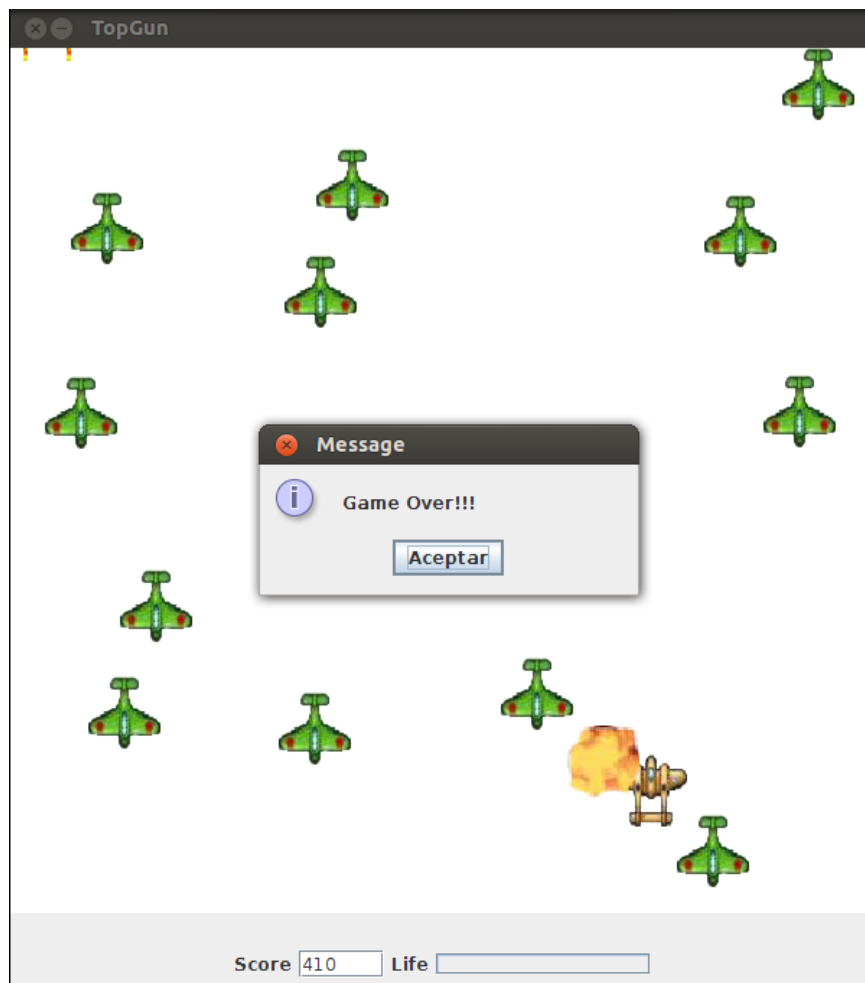


Figura 4.16: Captura del juego Top Gun programado en JAVA

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

La mayoría de los ids analizados (ver tabla 4.5 columnas Expansión desde Greedy y Expansión desde Samurai), se corresponden a distintos elementos de interacción que posee el programa con el usuario. De estos elementos se pueden enumerar: áreas de texto, barras de desplazamiento, menú, copiar, pegar, imprimir, imágenes, colores. Estos elementos sin duda forman parte



del Dominio del Problema en el programa Editor.java.

### 4.6.3. Top Gun

Continuando con los casos de estudio, el próximo es un programa JAVA llamado TopGun.java. Cuando este programa se ejecuta aparece un juego de aviones (ver figura 4.16), en este juego el usuario conduce un avión que dispara. El objetivo consiste en derribar la mayor cantidad de aviones enemigos y un contador suma puntos por cada avión derribado. El juego finaliza cuando al avión se le agota la barra de energía por completo, esta barra disminuye debido a los disparos enemigos que recibe. Este programa, posee un módulo de aproximadamente de 600 líneas que van a ser analizadas por la herramienta IDA. Al igual que el caso de estudio anterior (Editor de texto), debido a que el programa TopGun.java contiene muchos ids, en la tabla 4.6 se lista el análisis de los ids más relevantes.

### Análisis de Resultados

A diferencia de los casos de estudios antes vistos, en la tabla 4.6 se puede observar la presencia de ids capturados conformados por tres palabras; algunos ejemplos de esto son: LIFEprogressBar, hitShotEnemy, hitPlaneEnemy. En la división de ids persiste la tendencia en Greedy con respecto a Samurai, de separar mal algunos ids (al igual que los casos de estudio anteriores). Algunos ejemplos son: lnEne → ln-en-e, enImage → en-e-image, strt\_but → str-t-but (entre otros).

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

Por otro lado, la abreviatura but que representa buttons no se expande, se estima que esto ocurre por que buttons no está en ninguna frase del código

(comentario y literal), y no se expande con palabras del diccionario en Inglés porque **but** es una palabra válida del diccionario.

De los ids analizados (ver tabla 4.6 en las columnas Expansión desde Greedy y Expansión desde Samurai), las palabras que más frecuentemente aparecen están asociados al juego. De estas palabras se pueden nombrar, disparos, aviones, enemigos, imágenes, actualizar pantalla, movimiento. Estas palabras definitivamente forman parte del Dominio del Problema perteneciente al programa TopGun.java.

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

Tabla 4.6: Parte del Análisis Realizado a los Ids de TopGun.java

#### 4.6.4. Conclusiones sobre los casos de estudio

La investigación descrita a lo largo de este trabajo final, tiene como objetivo principal el desarrollo de la herramienta IDA. Con esta herramienta se pretende hacer aportes al área de la CP, dado que no se han construido herramientas automáticas que analicen ids.

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

# 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] 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.
- [4] 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.
- [5] 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.
- [6] 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.