

Capítulo 3

Análisis de Identificadores: Estado del Arte

3.1. Introducción

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

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

Teniendo en cuenta que los códigos crecen con los nuevos requerimientos y el frecuente mantenimiento, los sistemas son cada vez más complejos y difíciles de entender. Un camino para lograr un entendimiento ágil y facilitar las arduas tareas de interpretación de códigos, es a través del uso de las herramientas de comprensión.

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 que hay detrás de los componentes visibles en los códigos. Entre los distintos componentes visibles, los lectores de los códigos fijan su atención principalmente en los ids y los comentarios, ambos son una fuente de información importante para la CP. Sin embargo, cuando en el código no abundan los comentarios, el foco de atención se centra en los ids.

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

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

Tabla 3.1: Análisis Léxico de códigos JAVA

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

esta información, mejora la comprensión, aumenta la productividad y agiliza el mantenimiento de los sistemas.

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

3.2. Conceptos claves

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

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

Generalmente, la buena practica de programación recomienda que un id dentro del código este asociado a un concepto del programa [?, ?, ?].

Identificador \Leftrightarrow Concepto

Por ejemplo, el id `openWindow` está asociado al concepto ‘abrir una ventana’.

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

3.3. Nombres de Identificadores

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

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

3.3.1. Clasificación

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

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

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

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

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

¹Que contiene más de una palabra.

²Esta clasificación se la conoce como acrónimo.

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

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

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

3.3.2. Importancia en los Nombres

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

- En el caso de JAVA, los nombres de los paquetes deben ser con minúscula (`main.packed`). Las clases con mayúscula en la primer letra de cada palabra que compone el nombre (`MainClass`).
- En el caso de C#, las clases se nombran igual que JAVA. Pero para el caso de los paquetes deben comenzar con mayúscula y el resto minúscula (`Main.Packed`).

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

Una evidencia fehaciente de la importancia en la semántica de nombres, son las técnicas que se aplican para protección de código. Algunas de ellas

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

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

Figura 3.1: Trozo de Código de un Sistema Comercial

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 a la semántica de nombres asignados a los ids. Existen tres razones destacadas que conllevan a esto:

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

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

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

Nombre Consistente: 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 sería un id nombrado como `file_name`, el cual está incompleto y no representa de forma semánticamente concisa el concepto mencionado.

Los propiedades que violan nombrar a un id de manera *consistente* son conocidas en el lenguaje natural como sinónimos y homónimos. Los homónimos son palabras que pueden tener más de un significado. Por ende, si el nombre de un id 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.

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 nombres consistentes tales como se mencionó anteriormente, hacen que se dificulte identificar con claridad los conceptos en el dominio del problema, lo que hace aumentar los esfuerzos de comprensión del programa.

Por lo tanto, si los ids están contruidos de forma *concisa* (identificando bien al concepto) y la *consistencia* está presente, se pueden descubrir los conceptos que representan en el dominio del problema más fácilmente. De esta manera, se agiliza la comprensión, aumenta la productividad, mejora la calidad durante la etapa de mantenimiento [?, ?].

Intuitivamente, se necesita que los ids representen bien al concepto, ya que mayor será el impacto que tendrá en la interpretación del sistema [?, ?]. Sin

embargo, durante las etapas de desarrollo y mantenimiento de software, es muy difícil mantener una consistencia global de nombres en los ids, sobre todo si el sistema es grande. En este sentido, cada vez que un concepto se modifica el nombre del id asociado debe cambiar y adaptarse a la modificación.

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

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

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

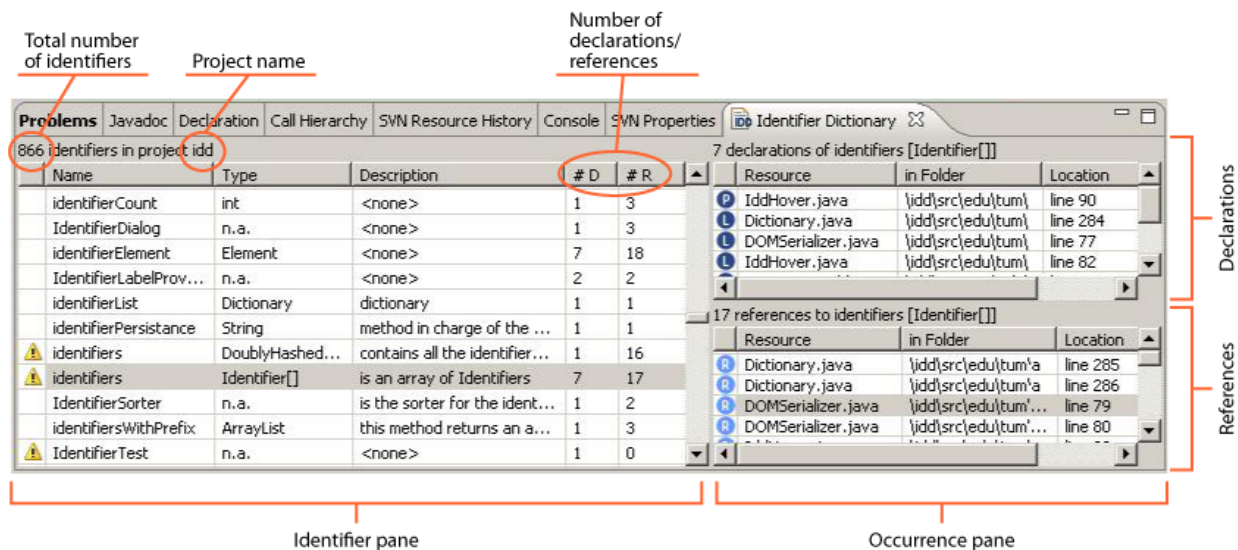


Figura 3.2: Visualización de Identifier Dictionary

3.3.3. Herramienta: Identifier Dictionary

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

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

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

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

²<http://www.eclipse.org/jdt>

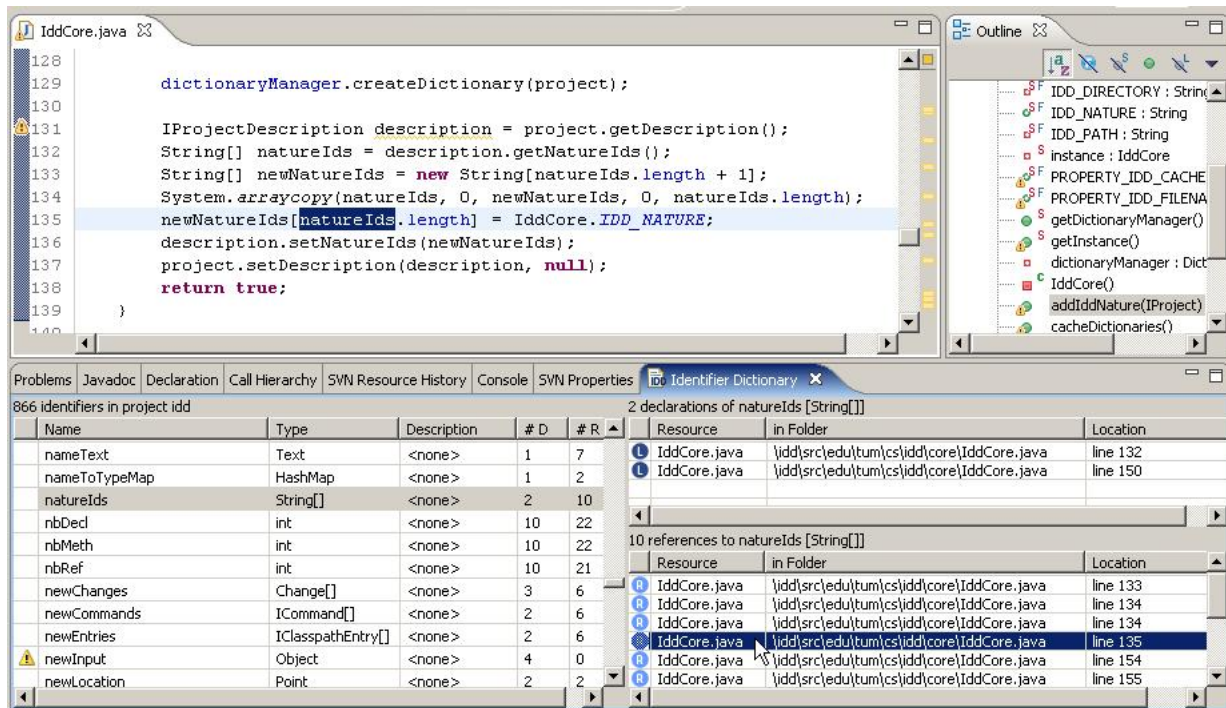


Figura 3.3: Visualización de Identifier Dictionary

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

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

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

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

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

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




Figura 3.4: Visualización de Identifier Dictionary

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

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

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

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

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

¹Similar a los warnings de Eclipse

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

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

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

3.4. Traducción de Identificadores

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

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

3.4.1. Conceptos y Desafíos observados

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.2. Algoritmo de División Greedy

El Algoritmo de división Greedy elaborado por Lawrie, Feild, Binkley [?, ?, ?, ?, ?], emplea tres listas para realizar su trabajo:

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

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

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

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

El algoritmo de Greedy utiliza las 3 listas nombradas al comienzo de la sección en forma de variable global. Esto ocurre porque las 3 listas son usadas por subrutinas más tarde. El algoritmo procede de la siguiente manera (Ver Algoritmo 1), el id que recibe como entrada primero se divide (con espacios en blanco) en las hardwords que lo componen (ejemplo: `fileinput.txt` \Rightarrow `fileinput` y `txt` en la línea 2 - algoritmo 1, si es camel-case `fileinputTxt` \Rightarrow `fileinput` y `txt` en la línea 3 - algoritmo 1). Luego, cada palabra resultante en caso que esté en alguna de las 3 listas, se distingue como un único softword (ejemplo: `txt` pertenece a la lista de abreviaturas conocidas, línea 5). Si alguna palabra no está en alguna lista se considera como múltiples softwords que necesitan subdividirse (ejemplo: `fileinput` \Rightarrow `file` y `input`, línea 5). Para subdividir estas palabras se buscan los prefijos y los sufijos más largos posibles dentro de ellas, para hacer esta búsqueda se utilizan las mismas 3 listas antes mencionadas (líneas 6 y 7).

Por un lado se buscan prefijos con un proceso recursivo (Ver Función **buscarPrefijo**). Este proceso comienza analizando toda la palabra por completo. Se van extrayendo caracteres del final hasta encontrar el prefijo más largo o no haya más caracteres (líneas 5 - 7, **buscarPrefijo**). Cuando una palabra se encuentra en alguna lista (línea 3, **buscarPrefijo**) 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, **buscarSufijo**). De la misma forma que la función de prefijos, cuando encuentra una palabra, se inserta un separador (‘ ’) y el resto se procesa por la función **buscarSufijo** (línea 4).

Una vez que ambos procesos terminaron, los resultados son retornados al algoritmo principal (**resultadoPrefijo**, **resultadoSufijo** líneas 6 y 7, algoritmo 1). Mediante una función de comparación se 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, suponga que la palabra abreviada `fl` no se encuentra en ninguno de los 3 listados y las palabras `input` y `txt` si están. Dada esta situación, si el id `flinputtxt` se procesa por ambas rutinas, el resultado será que **buscarPrefijo** no divida al id. Esto sucede porque al retirar caracteres del último lugar, nunca se encontrará un prefijo conocido. Más precisamente al no dividirse entre `fl` e `input` el resto de la cadena no se procesará y tampoco se dividirá entre `input` y `txt`.

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

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

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

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

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

Los autores de la estrategia Samurai sostienen, que los ids multi-palabras también se encuentran dentro de los comentarios y literales strings del código. Es por esta razón, que el parámetro de entrada del algoritmo se nombra de

¹Que posee más de una palabra.

una manera más genérica: *token*, y no simplemente “identificador”. Por tal motivo, se menciona la expresión “dividir al token”, en lugar del “dividir al identificador”.

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

División Hardword

En la rutina *divisiónHardWord* (Ver Algoritmo 2) primero se ejecutan dos funciones (líneas 1 y 2). La primera *dividirCaracteresEspecialesDigitos*, toma al token y reemplaza todos los posibles caracteres especiales y números por espacio en blanco. La segunda *dividirMinusSeguidoMayus*, agrega un blanco entre dos caracteres que sea una minúscula seguido por una mayúscula. En este punto solo quedan tokens de la forma *softword* o que contengan una mayúscula seguido de minúscula (Ejemplos: List, ASTVisitor, GPSstate, state, finalstate, MAX).

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

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

Ejemplos: List, AST Visitor.

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

El algoritmo para determinar cual es el tipo correcto y no tomar malas decisiones (Ejemplos: SQ Llist o GP Sstate), divide a la palabra por ambas consideraciones (ejemplo: GP Sstate y GPS state), luego calcula el puntaje (score)

Algoritmo 2: divisiónHardWord

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

```

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

```

de la parte derecha de ambas divisiones (líneas 7 y 8). Aquella con puntaje más alto entre las dos será por la cual se decida (línea 9). Tomando como ejemplo el id *GPSstate*, para el caso camel-case calculará $score(Sstate)$ y para la otra variante $score(state)$. Intuitivamente, la función score (a través de las tablas de frecuencias de palabras) elegirá *state* sobre *sstate* ya que esta última debería tener un puntaje inferior (frecuencia menor de aparición). Por ende

Samurai detecta que **GPSstate** es del tipo Variante de Camel-Case. La división elegida se lleva a cabo en las líneas 11 y 13 (según el caso). Finalmente, todas las partes que fueron divididas se envían a *divisiónSoftWord* (línea 17, 18).

División Softword

La rutina recursiva *divisiónSoftWord* (Ver Algoritmo 3) recibe como entrada un substring **s**, el cual puede tener tres tipos de variantes: a) todos los caracteres en minúsculas (**visitor**), b) todos con mayúsculas (**VISITOR**), c) el primer caracter con mayúscula seguido por todas minúsculas (**Visitor**). El otro parámetro de entrada es el puntaje original (**score_{sd}**) correspondiente a **s**.

La rutina primero examina cada punto posible de división en **s** dividiendo en **split_{izq}** y **split_{der}** respectivamente (líneas 4 y 5). La decisión de cual es la mejor división (línea 9), se basa en: a) substrings que no tengan prefijos o sufijos conocidos, los mismos están disponibles en la página web del autor¹ (línea 6), b) que el puntaje de la división elegida sobresalga del resto de los puntajes (líneas 7-8).

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

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

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

Algoritmo 3: divisiónSoftWord

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

```

1  $tokenSep \leftarrow s$ ,  $n \leftarrow \text{length}(s) - 1$ 
2  $i \leftarrow 0$ ,  $maxScore \leftarrow -1$ 
3 mientras ( $i < n$ ) hacer
4      $score_{izq} \leftarrow \text{score}(s[0,i])$ 
5      $score_{der} \leftarrow \text{score}(s[i+1,n])$ 
6      $preSuf \leftarrow \text{esPrefijo}(s[0,i]) \vee \text{esSufijo}(s[i+1,n])$ 
7      $split_{izq} \leftarrow \sqrt{score_{izq}} > \max(\text{score}(s), score_{sd})$ 
8      $split_{der} \leftarrow \sqrt{score_{der}} > \max(\text{score}(s), score_{sd})$ 
9     si ( $\neg preSuf \wedge split_{izq} \wedge split_{der}$ ) 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  $tokenSep$ 

```

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

Si la parte derecha finalmente se divide (línea 15 - el si es verdadero), luego entre la parte izquierda y la derecha también. Por ejemplo, el id *countrownumber* primero se analiza *rownumber* (parte derecha - línea 14) como este finalmente se separará en *row number*, la palabra *count* (parte iz-

quiera) se divide del resto (línea 16), dando como resultado `count row number`. Sin embargo, cuando la parte derecha no se divide tampoco se debería separar entre ambas partes (línea 15 - el si es falso). Los análisis de datos hechos por el autor [?] obligan a hacer este control, ya que se encontraron abundantes casos erróneos de división, uno de ellos es `string ified`. Con este nuevo control al no dividirse `ified` tampoco lo hará del resto dejando la palabra correctamente unida `stringified`.

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

En el algoritmo anterior *divisiónHardWord*, la presencia de la raíz cuadrada (línea 9), cuando se compara el caso camel-case y el caso alternativo también es para solucionar este mismo problema.

Función de Scoring

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

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

Dado un string `s`, la función *score(s)* retorna un valor que indica: i) la frecuencia de aparición de `s` en el programa bajo análisis, y ii) la frecuencia de aparición de `s` en un conjunto grande de programas predefinidos. Las tablas de frecuencias de aparición de palabras local y global le brindan información

a la función score. La inclusión de la tabla de frecuencias global, está dada porque los programas pequeños en general tienen frecuencias bajas en la tabla local, y a veces, la función score no trabaja correctamente a causa de esto.

La fórmula es la siguiente:

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

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

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

3.4.4. Algoritmo de Expansión Básico

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

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

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

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

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

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

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

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

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

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

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

Algoritmo 4: Expansión Básica

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

```

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

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

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

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

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

```

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

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

3.4.5. Algoritmo de Expansión AMAP

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

Las fuentes de palabras que se utilizan son:

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

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

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

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

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

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

El algoritmo AMAP ejecuta los siguientes pasos:

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

A continuación, se explican cada uno de estos pasos.

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

Búsqueda por Palabras Singulares

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

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

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

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

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

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

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

La búsqueda de palabras singulares se presenta en el algoritmo 5. Los parámetros de entrada son: la palabra abreviada a expandir, 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 (del algoritmo 5), se impide básicamente dos cosas:

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

b) En caso de que el patrón sea el *compuesto por letras* (no sea el prefijo), se hacen dos controles más (primer argumento del \wedge en el **si**). Uno es, que la

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

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

abreviatura no posea muchas vocales consecutivas (“ $[\text{^\wedge}aeiou]^+$ ” logra eso) y la otra es que longitud sea mayor a 3. La autora a través del análisis de datos determinó esta restricción [?], ya que el *patrón compuesto por letras* tiene el inconveniente que es muy flexible y tiende a capturar muchas palabras largas incorrectas. Por ejemplo: *lang* \Rightarrow (loading, language), o *br* \Rightarrow (bar, barrier, brown), entre otros.

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

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

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

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

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

Para finalizar, en la línea 10 si la palabra abreviada tiene más de un caracter y el patrón es de tipo prefijo, se busca usando (**patrón**) en los co-

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”

```

mentarios de la clase. En la línea 9 se restringe esta búsqueda, porque la autora sostiene [?], que buscar con un solo caracter en comentarios implica tener muchos resultados y más aun si el patrón es el compuesto por letras.

Búsqueda por Multi-Palabras

El algoritmo de búsqueda por multi-palabras a diferencia del explicado

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

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

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

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

En el algoritmo 6, se presenta la búsqueda por multi-palabras [?]. 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.

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

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

Algoritmo 6: Búsqueda por Multi Palabras

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

```

1 si (patrón acrónimo  $\vee$  length(pa) > 3) entonces
    // Si alguna de las siguientes búsquedas encuentra un
    único resultado, el algoritmo lo retorna
    finalizando la ejecución
2   Buscar en Comentarios JavaDoc con “@param pa patrón”
3   Buscar en Nombres de Tipos y la correspondiente Variable
    declarada con “patrón pa”
4   Buscar en el Nombre del Método con “patrón”
5   Buscar en todos los ids (y sus tipos) dentro del Método con
    “patrón”
6   Buscar en Literales String con “patrón”
    // En este punto se buscó en todos los lugares
    posibles dentro del método
7   Buscar en palabras que están en los Comentarios del Método con
    “patrón”
8   si (patrón acrónimo) entonces
    // Solo se busca con patrones Acrónimos
9   Buscar en palabras que están en los Comentarios de la Clase
    con “patrón”

```

Los patrones de *combinación de palabras* son menos restrictivos que los patrones de *acrónimos* y frecuentemente conllevan a malas expansiones. En caso que no sea acrónimo, la búsqueda se restringe a palabras abreviadas

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

Figura 3.5: Ejemplos de distintas Búsquedas dentro del Código

ingresadas con longitud 4 ó mayor (línea 1, algoritmo 6). Esto genera la sensación de que se pierden casos de 2 ó 3 caracteres pero estudios indican que son la minoría [?].

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

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

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

Luego en la línea 9 se busca en comentarios de la clase con el *patrón acrónimo*. Cabe aclarar que *patrón de combinación de palabras* en este caso no se usa (línea 8) ya que puede tomar palabras largas incorrectas.

Finalmente después de observar cientos de casos de palabras largas, la autora [?] 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 abreviatura posea múltiples alternativas potenciales de expansión, dentro del mismo alcance estático. Por ejemplo, el patrón prefijo para **val** puede coincidir **value** o **valid**. La técnica de elección entre múltiples candidatos procede de la siguiente manera:

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

una sola aparición en los comentarios de JAVA Doc. Para llegar a una decisión, primero se almacenan ambas opciones. Después, prosigue con el resto de las búsquedas (nombres de tipos de ids, literales, comentarios) en cuanto aparezca una de las dos, por ejemplo `file socket` este termina prevaleciendo sobre `file system`.

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

Expansión Más Frecuente (EMF)

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

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

La técnica EMF tiene dos niveles de análisis, el primero es a nivel de pro-

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

Tabla 3.2: Algunas Frecuencias Relativas de Ids en JAVA 5

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

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

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

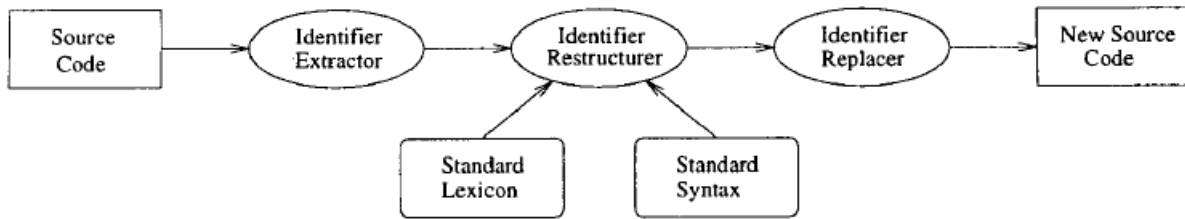


Figura 3.6: Arquitectura de Identifier Restructuring

3.4.6. Herramienta: Identifier Restructuring

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

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

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

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

¹Todas las figuras de esta sección se obtuvieron del artículo correspondiente a la herramienta que se describe [?].

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

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

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

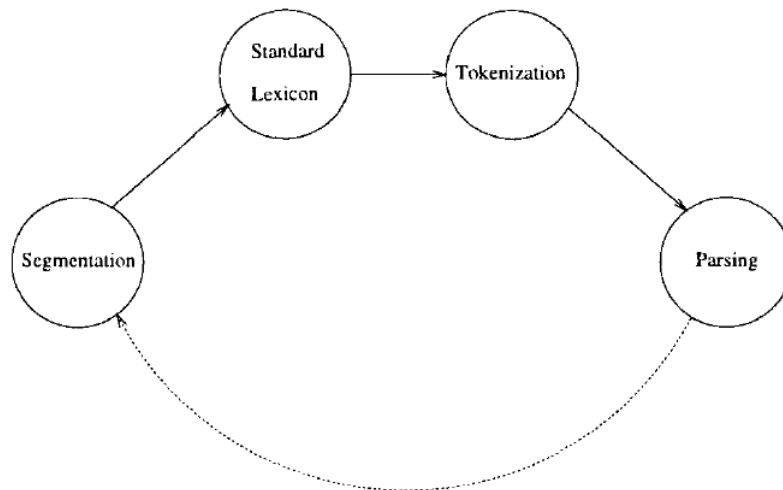


Figura 3.7: Etapas de Identifier Restructurer

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

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

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

Figura 3.8: Gramática que determina la Función de los Ids

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

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

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

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

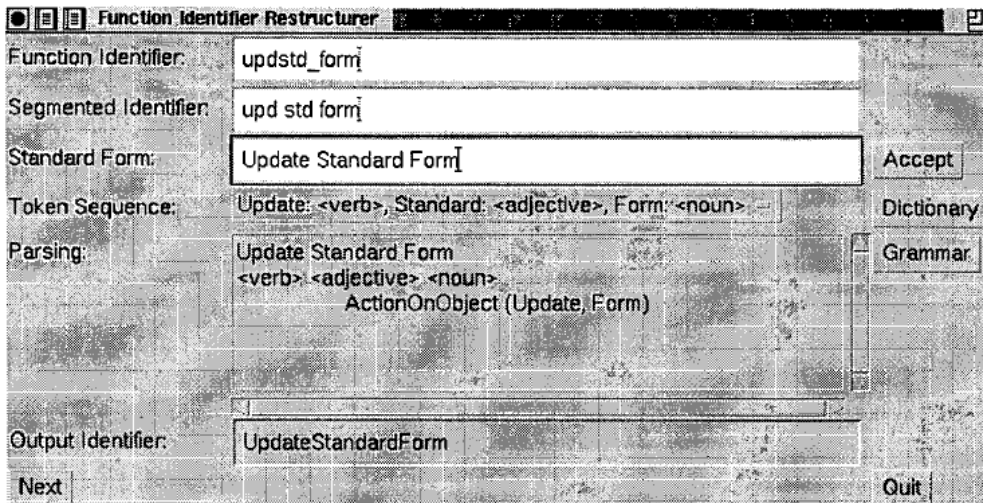


Figura 3.9: Visualización de Identifier Restructuring

Interfaz para el Usuario

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

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

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

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

3.5. Notas y Comentarios

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

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

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

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