AID: Una Herramienta para el Análisis de Identificadores en Programas JAVA

Javier Azcurra[1], Mario Berón[2], Pedro Rangel Henriques[3], Maria J. Varanda Pereira[4]

[1]UNSL: javierazcurra_m@yahoo.com.ar - [2]UNSL: mberon@unsl.edu.ar

[3]Universidade do Minho: pedrorangelhenriques@gmail.com

[4]Instituto Politécnico de Bragança: mjoao@ipb.pt

Resumen: *Las demandas actuales en el desarrollo de software implican una evolución y mantenimiento constantes con el menor costo de tiempo y recursos. La Comprensión de Programas (CP) es una disciplina de la Ingeniería de Software (IS) que ofrece Métodos, Técnicas, Estrategias y Herramientas para llevar adelante esas tareas. Generalmente las técnicas de comprensión emplean dos fuentes importantes de información: Estática y Dinámica. En ambas fuentes existe un elemento que brinda información y es muy utilizado: los identificadores (Id). Estudios indican que los Ids contienen indicios sobre las funcionalidades de los sistemas. Por tal motivo, construir herramientas automatizadas de comprensión que puedan extraer y analizar los Ids es un aporte muy importante al área de la CP.*

En este artículo se presenta AId, una herramienta que extrae y analiza Ids con el propósito de encontrar su significado y de esta forma ayudar a comprender el programa de estudio.

Palabras Claves: *Comprensión de Programas, Identificadores, Herramienta de Análisis de Ids.*

Abstract: *The current demands in software development imply a permanent maintenance and evolution with minimal costs (time and other resources). Program Comprehension (PC), a discipline of Software Engineering, provides methods, techniques and strategies to carry out these tasks. The comprehension techniques use two kind of information: Static and Dynamic. Both techniques share a common element that is always present and provides useful information: the Identifiers (Id). Previous research indicates that Ids, yet when they are acronyms or composed, give important clues about the system functionalities. For this reason, to build comprehension tools that automatize the process of extraction and analysis of the identifiers is an important contribution for PC.*

This article presents AId, a tool that extracts and analyzes Ids with the goal of finding their meaning; in that way Aid aims to help understanding the program under study.

Keywords: *Program Comprehension, Identifiers, Tool to analyse Ids.*

INTRODUCCIÓN

La CP es una disciplina de la Ingeniería de Software cuya finalidad es facilitar el entendimiento de los sistemas (Albanes et al., 2011). Una manera, sino la más importante, de lograr este objetivo consiste en relacionar dos dominios muy importantes: el Dominio del Problema y el Dominio del Programa. Para poder hacer esta relación se debe:

i) construir una representación del Dominio del Problema, ii) construir una representación del Dominio del Programa, y iii) elaborar una estrategia de vinculación entre ambos dominios (Storey, 2005).

Tomando en consideración el Dominio del Problema se puede decir que un posible camino para elaborar una representación consiste en: i) extraer los conceptos usados en el programa y ii) establecer relaciones entre estos conceptos para

formar lo que se conoce como un mapa conceptual (Berón, 2009). Una aproximación interesante para recuperar los conceptos consiste en capturar los Ids del programa y realizar un análisis considerando el contexto en donde aparecen. Estudios realizados (Lawrie et al., 2007) indican que gran parte de los programas están constituidos por Identificadores (Ids), los cuales ofrecen abundante información. Generalmente los Ids están compuestos por más de una palabra en forma de abreviatura. Varios autores (Enslen et al., 2009) coinciden que estas abreviaturas pueden contener información oculta que es propia del Dominio del Problema (conceptos). Una manera de extraer esta información de los Ids consiste en expandir estas abreviaturas en sus correspondientes palabras en lenguaje natural. Para conseguir este objetivo se deben realizar los siguientes pasos: i) extraer los Ids del código, ii) ejecutar técnicas de división, donde el Id se descompone en distintas palabras abreviadas que lo componen, iii) emplear estrategias de expansión de abreviaturas para transformar las mismas a palabras completas.

Normalmente los nombres de los Ids siguen los criterios establecidos por el programador (Caprile and Tonella, 2000). Estos criterios representan la principal dificultad para las técnicas de expansión de abreviaturas de Ids. Esto se debe a que no siempre los programadores usan las convenciones adecuadas y en muchos casos los nombres no reflejan la semántica correcta. Una forma de combatir estas dificultades consiste en utilizar fuentes de información informal que se encuentran disponibles en el código fuente. Por información informal se entiende aquella contenida en los comentarios de los módulos, comentarios de las funciones, literales, documentación del sistema y todos lo demás recursos descriptivos del programa que estén escritos en lenguaje natural. Sin duda los comentarios tienen como principal finalidad ayudar a comprender un segmento de código (Freitas et al., 2008). Por esta razón los comentarios son una fuente de información natural para entender el significado de los Ids en el código como así también para extraer conceptos del Dominio del Problema. Por otro lado, para poder entender la semántica de los Ids, se consideran también los literales, los cuales representan un valor constante formado por secuencias de caracteres. Ellos son generalmente utilizados en instrucciones que muestran texto por pantalla o para inicializar variables de tipo string. En el caso que estas fuentes de información informal sean escasas en el código de estudio se puede recurrir a alternativas externas como los diccionarios predefinidos de palabras en lenguaje natural.

En la actualidad existen varias estrategias de análisis de Ids. Sin embargo no todas están implementadas en herramientas automáticas.

Este artículo está organizado de la siguiente manera: Primero se explican los conceptos subyacentes del contexto sobre teoría de análisis de Ids. Luego se presenta AId (Analizador de Identificadores) una herramienta que reúne las características mencionadas en la introducción de este artículo: analizar Ids y expandir abreviaturas. A continuación se muestra el funcionamiento de AId por medio de un caso de estudio. Finalmente se presentan las conclusiones y trabajos futuros.

ANÁLISIS DE IDENTIFICADORES

Para comenzar se dará la definición de identificador, eje central de análisis: “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, esto es, para dar a cada entidad un nombre único que los individualice”.

Cada lenguaje tiene sus propias reglas que definen cómo pueden estar construidos los nombres de sus Ids. Por ejemplo en lenguajes como C y JAVA no está

permitido declarar Ids que coincidan con palabras reservadas o que contengan símbolos especiales (& ! %), a excepción del guión bajo (_).

Un identificador normalmente está asociado a un concepto del Programa: Identificador Concepto.

En otras palabras un Id es un representante de un concepto que pertenece al Dominio del Problema. Por ejemplo el Id userAccount representa el concepto "cuenta de usuario". Con esto se muestra la importancia de analizar los Ids (Binkley et al., 2013).

Para mejorar la CP se requiere que los nombres de los Ids transmitan de manera clara los conceptos que representa (Deibenbock and Pizka, 2006). Lamentablemente en la práctica esto no se tiene en cuenta. 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. Por eso, durante la etapa de mantenimiento de los sistemas, los nombres de los Ids no son muy tenidos en cuenta para comprender el sistema a mantener.

Antes de continuar describiendo la importancia de los nombres de los Ids se van a clasificar las distintas formas en que se los puede nombrar. Estudios realizados con 100 programadores sobre comprensión de Ids indican que existen cuatro formas principales de construir Ids multi-palabra (tomando como ejemplo el concepto File System Input): Palabras completas (fileSystemInput), Abreviaturas (flSysIpt), Acrónimo (fsi), y la combinación de las anteriores (flSystemIpt).

Los resultados del estudio mencionado en el párrafo precedente muestran 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 mismas (Deibenbock and Pizka, 2006).

Binkley y su equipo de investigadores clasifican los nombres de los Ids con dos términos conocidos en la jerga del análisis de Ids: hardwords y softwords (Binkley et al., 2013). Los hardwords destacan la separación de cada palabra que compone al Id multi-palabra a través de una marca específica; algunos ejemplos son: fileSystem (notación camelcase) o fileSYSTEM en donde se marca claramente la separación de cada palabra con el uso de mayúsculas y minúsculas; también es normal dividir las palabras con un símbolo especial, como es el caso del guión bajo: file_system. Los softwords no poseen ningún tipo de separador o marca que dé 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 artículo.

Volviendo a la importancia en los nombres de los Ids se puede decir que existen innumerables convenciones sobre los nombres de los Ids. Algunas de ellas son: en el caso de JAVA los nombres de los paquetes deben ser con minúscula (main.packed) y los nombres de las clases deben tener una mayúscula en la primera letra de cada palabra que compone al nombre (MainClass). En caso de C# las clases se nombran igual que en Java. Pero para los paquetes deben comenzar con mayúscula y el resto minúscula (Main.Packed). Esto indica que se concentra más en los aspectos sintácticos del Id que están asociados al Dominio del Programa y no tanto en los aspectos semánticos asociados al Dominio del Problema. Una evidencia fehaciente de la importancia en la semántica de nombres son las técnicas que se aplican para protección de código. Algunas de ellas se encargan de reemplazar los nombres originales de los Ids por secuencias de caracteres aleatorios y de esta manera se reduce la comprensión. Las técnicas antes mencionadas se conocen con el nombre de Técnicas de Ofuscación de Código.

Cuando los programadores desarrollan sus aplicaciones restan importancia a la semántica de

nombres asignados a los Ids. Existen tres razones destacadas que conllevan a esto: i) los Ids son escogidos por los programadores sin tener en cuenta los conceptos que tienen asociados; ii) los desarrolladores tienen poco conocimiento de los nombres usados en los Ids ubicados en otros sectores del código fuente y iii) durante la evolución del sistema los nombres de los Ids se mantienen y no se adaptan a nuevas funcionalidades (o conceptos) que puedan estar asociados. En este sentido la asignación de nombres a los Ids se combate con la programación "menos egoísta". Ésta consiste en hacer programas más comprensibles para el futuro lector que no está familiarizado. Para lograr este objetivo se deben respetar dos reglas sobre los nombres que se le asignan a los Ids: i) 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. ii) Nombre Consistente: Cada Id debe tener asociado, sí y sólo sí, un único concepto (Deibenbock and Pizka, 2006).

El uso de sinónimos y homónimos perjudica los nombres consistentes en los Ids. Los homónimos son palabras que pueden tener más de un significado. Por tal motivo si un nombre de un Id está asociado a más de un concepto no estará claro a cuál representa. Por otro lado los sinónimos indican que un mismo concepto puede tener asociado diferentes nombres. Por ejemplo los nombres accountBankNumber y accountBankNum son sinónimos porque hacen referencia al mismo concepto "número de cuenta bancaria".

Está demostrado que la utilización de nombres consistentes, tal como fue mencionado antes, es una característica deseada. Esto ocurre porque se hace difícil identificar con claridad los conceptos en el Dominio del Problema, por lo tanto, los esfuerzos para comprender un programa son mayores. Si los Ids tienen nombres consistentes y concisos (es decir identifican bien al concepto) los conceptos del Dominio del Problema pueden ser descubiertos más fácilmente (Binkley et al., 2013). De esta manera durante las etapas de desarrollo y mantenimiento de software es difícil conservar 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.

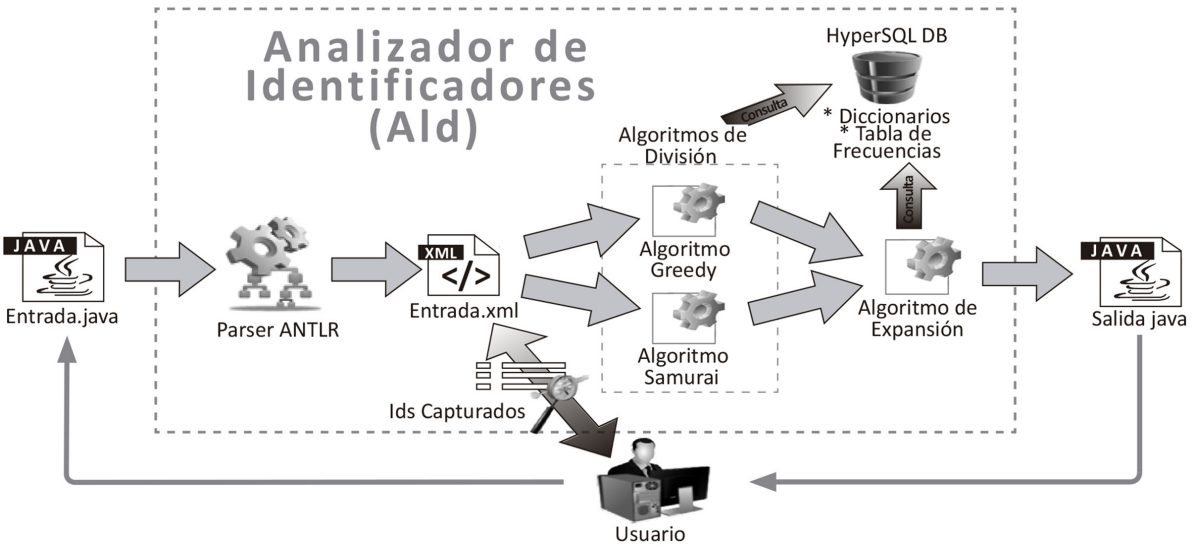

Figura 1 - Arquitectura de AId,

Deibenbock y su equipo de investigadores proponen utilizar una herramienta que resuelva los problemas relacionados a la asignación de nombres que fue explicado en párrafos anteriores. Dada la dificultad inherente a la construcción de una herramienta totalmente automática para nombrar correctamente los Ids elaboraron una herramienta semiautomática 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.

Esto ocurre porque con los diccionarios se describe en forma clara todos los términos utilizados en los grandes sistemas de software (Deibenbock and Pizka, 2006).

Las personas que leen los códigos de programas tienen serios inconvenientes para entender el propósito de los Ids y deben invertir tiempo en analizar el significado de su presencia. Las estrategias automáticas dedicadas a facilitar este análisis son bienvenidas en el contexto de la CP.

AID: UNA HERRAMIENTA PARA EL ANÁLISIS DE IDENTIFICADORES

La literatura dedicada a estrategias de análisis de Ids indica que los Ids ocultan información relevante del Dominio del Problema a través de las abreviaturas (Enslen et al., 2009). Una forma de exhibir esta 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 lleva a cabo para realizar la traducción de Ids consta de dos pasos: i) División: Separar el Id en las palabras que lo componen usando algún separador especial (Ejemplo: flSys ⇒ fl-sys). ii) Expansión: Expandir las abreviaturas que resultaron como producto del paso anterior.

La Figura 1 muestra la arquitectura de AId (Analizador de Identificadores), una herramienta que automatiza el proceso de análisis de Ids. Esta herramienta fue implementada con lenguaje Java y el entorno de desarrollo NetBeans. En esta misma figura se puede observar que AId permite realizar un análisis en tres etapas: la primera consiste en la extracción de datos, la segunda tiene por objetivo realizar la división de Ids y la tercera sobre la expansión de Ids.

La fase de extracción de datos (primer paso) recibe como entrada un programa Java (Entrada. java en Figura 1) que luego será procesada por un analizador sintáctico (parser), el cual fue programado utilizando la herramienta ANTLR (ANother Tool for Language Recognition). El analizador sintáctico extrae y almacena, en estructuras internas, la información estática del código del programa Java. Esta información está compuesta con Ids (como elemento principal), literales y comentarios. Finalmente toda esta información se almacena en un archivo xml (ver Entrada.xml en Figura 1).

Concluida la extracción de información se da inicio a la fase de división de Ids. En esta etapa pueden seleccionarse dos algoritmos: Greedy y Samurai. Ambos reciben como entrada la información de la etapa anterior y divide los Ids del código de entrada. Las divisiones son almacenadas en estructuras internas que serán consultadas en la fase siguiente. Como estos algoritmos de división necesitan datos externos como diccionarios (en el caso Greedy) o tablas de frecuencias de palabras (en el caso Samurai), estos datos se encuentran almacenados en una base de datos HyperSQL DB (ver parte superior de la Figura 1). La tercera y última fase implementa el algoritmo de expansión convencional que será explicado más adelante. Este algoritmo toma los Ids divididos en la etapa anterior y procede a expandirlos. Aquí también se necesitan

los diccionarios. Por este motivo se hacen consultas a HyperSQL DB. Una vez expandidos todos los Ids son sustituidos por los originales en el programa de entrada generando de esta manera un archivo de salida (Salida.java en Figura 1).

ANALIZADOR SINTÁCTICO

Se investigaron herramientas destinadas a construir analizadores léxicos y sintácticos. En este sentido se dio preferencia a aquellas herramientas que utilizan la teoría de gramáticas de atributos (Aho et al., 2006). De la investigación previamente descripta se determinó que la herramienta ANTLR era la que mejor cumplía con los requisitos antes mencionados. Ésta permite adicionar acciones escritas en Java para el cálculo de atributos mediante una gramática libre de contexto que genera lenguaje Java. Estas acciones semánticas deben ser correctamente asociadas a la gramática para, por ejemplo, implementar estructuras de datos y algoritmos que almacenen los Ids utilizados en un programa (Aho et al., 1983). Una vez insertadas estas acciones la herramienta se encarga de leer la gramática y generar el analizador incorporando las acciones que fueran programadas. De esta manera se obtiene un analizador sintáctico que no sólo examina el código Java para detectar errores a nivel sintáctico sino que también extrae y almacena Ids. Además de estas acciones se agregaron otras que extraen comentarios y literales strings. Estos elementos son necesarios ya que sirven como entrada para los algoritmos de análisis de Ids que serán explicados en las próximas secciones.

ALGORITMO GREEDY

Como se mencionó antes el módulo de división de AId consta de dos algoritmos: Greedy y Samurai. En esta sección se explicará el primero.

Como ya fue dicho varias veces el algoritmo de Greedy, inicialmente propuesto por Lawrie, Feild, Binkley, utiliza tres listas, a saber:

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

2. Abreviaturas conocidas: Es una lista que se construyó con abreviaturas extraídas de distintos programas de autores expertos. Incluyen abreviaturas comunes (ejemplo: alt ⇒ altitude) y abreviaturas de programación (ejemplo: txt ⇒ text).

3. Palabras excluyentes (stop words): Posee palabras que son irrelevantes para realizar la división de los Ids. Incluye palabras claves de lenguajes de programación (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 tres listas mencionadas en el párrafo anterior las cuales son globales al algoritmo (Feild et al., 2006). Esto ocurre porque las tres listas son usadas por las sub-rutinas del algoritmo que procede de la siguiente manera (ver Figura 2, algoritmo 1): el Id que recibe como entrada primero se divide con espacios en blanco en los hardwords que lo componen. Si el Id posee caracteres especiales (ejemplo: fileinput ⇒ txt fileinput y txt, ver Figura 2 algoritmo 1 en la línea 2), o si es camelcase (fileinputTxt ⇒ fileinput y txt, en la línea 3).

Cada palabra resultante, en caso que esté en alguna de las 3 listas, se trata 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 ⇒ file - input, línea 5). Para subdividir estas palabras se buscan los prefijos y los sufijos más largos posibles dentro de ellas. Para esta tarea también se han utilizado las tres listas antes mencionadas (líneas 6

y 7). Por un lado se buscan prefijos con un proceso recursivo (ver Figura 2, algoritmo 2). Este proceso comienza analizando toda la palabra por completo.

```
Algoritmo 1: División Greedy
  Entrada: dHardwork // identificador a dividir
  Salida: softwordDiv // Id separado con espacios
  Variables Globales:
    ispellList // Palabras de ispell + Diccionario
    abrevList // Abreviaturas conocidas
    stopList // Palabras Excluyentes
1 softwordDiv← ""
2 softwordDiv← dividirCaracsEspDig(idHardword)
3 softwordDiv← dividirCamelCase(softwordDiv)
4 Para cada substring s en softwordDiv hacer
5 |  Si (s ∉ (stopList ∪ abrevList ∪ ispellList )) ent
6 |  |  sPrefijo ← buscarPrefijo(s,"")
7 |  |  sSufijo ← buscarSufijo(s,"")
  |  |  // Se elige la división que más particiones hizo
8 |  |  s ← maxDivisión(sPrefijo,sSufijo)
9 retornar softwordDiv // Id dividido por espacios.
```

```
Algoritmo 2: buscarPrefijo
Entrada: s // Abreviatura a dividir
Salida: abrevSeparada//Abrev dividida por espacios
    // Punto de parada de la recursión
1 Si (length(s) = 0) ent
2 |  return abrevSeparada
3 Si (s ∈ (stopList ∪ abrevList ∪ ispellList )) ent
4 |  return (s + ' ' + buscarPrefijo(s,""))
  // Se extrae y se guarda el último caracter de s.
5   abrevSeparada ← s[length(s) - 1] + abrevSeparada
  //Llamada recursiva sin el último caracter.
6 s ← s[0,length(s) - 1]
7 retornar buscarPrefijo(s, abrevSeparada )
```

```
Algoritmo 3: buscarSufijo
Entrada: s // Abreviatura a dividir
Salida: abrevSeparada//Abrev dividida por espacios
    // Punto de parada de la recursión
1 Si (length(s) = 0) ent
2 |  return abrevSeparada
3 Si (s ∈ (stopList ∪ abrevList ∪ ispellList )) ent
4 |  return (s + ' ' + buscarSufijo(s,"") + ''+s)
  // Se extrae y se guarda el último caracter de s.
5   abrevSeparada ← abrevSeparada + s[0]
  //Llamada recursiva sin el último caracter.
6 s ← s[1,length(s)]
7 retornar buscarSufijo(s, abrevSeparada )
```

Figura 2 - Algoritmos que implementan la técnica.

Se van extrayendo caracteres del final hasta encontrar el prefijo más largo que sea una palabra válida o no haya más caracteres (líneas 5 - 7 de la función). Cuando una palabra se encuentra en una lista (línea 3 de la función) se coloca en un separador (' '). El resto que fue descartado se procesa de nuevo a través de la función buscarPrefijo para buscar más subdivisiones (línea 4). De manera simétrica otro proceso recursivo se usa para tratar los sufijos (ver Figura 2, algoritmo 3). Este proceso también extrae caracteres, pero en este caso desde la primera posición hasta encontrar el sufijo más largo presente en una lista o hasta que no haya más caracteres (líneas 5 hasta la 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 para buscar más subdivisiones (ver Figura 2, algoritmo 1, línea 8). Finalmente el algoritmo Greedy retorna el Id destacando las palabras que lo componen mediante el caracter de espacio en blanco (fl inp txt). La ventaja de hacer dos búsquedas (prefijo y sufijo) es aumentar las chances de dividir al Id correctamente. A modo de ejemplo suponga que la palabra abreviada fl no se encuentra en ninguna de los tres listas, pero sí las palabras inp y txt. Dada esta situación, si el Id flinptxt se procesa por ambas rutinas, el resultado de invocar a buscarPrefijo será que 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 inp el resto de la cadena no se procesará y tampoco se dividirá entre inp y txt. No obstante,, este inconveniente no lo tendrá buscarSufijo porque al retirar los caracteres del principio de la palabra inp txt será separado. Como inp es una palabra conocida se agregará un espacio entre fl e inp. De esta manera el Id queda correctamente separado en tres partes: fl inp txt.

```
Algoritmo 4: División Samurai
Entrada: token // token a dividir
Salida: tokenSep // token dividido por espacios
1 tokenSep ← divisiónHardWord (token)
2 retornar tokenSep
```

```
Algoritmo 5: DivisiónHardWord
Entrada: token // token a dividir
Salida: tokenSep // token dividido por espacios
1 token ← dividirCaracsEspDig(token)
2 token ← dividirMinusSeguidoMayus(token)
3 tokenSep ← ""
4 Para cada substring s en token hacer
5   Si (∃ {i|esMayus(s[i]) ∧ esMinus(s[i + 1])}) ent
6     n ← length(s) - 1
      // se determina el tipo de la división
7     scoreCamel ← score(s[i,n])
8     scoreAlter ← score(s[i+1,n])
9     Si (scoreCamel > √scoreAlter) ent
10      Si(i > 0) ent
11        s ← s[0,i - 1] + ' ' + s[i,n]
12    en otro caso
13      s ← s [0,i] + ' ' + s[i + 1,n]
14  tokenSep ← tokenSep + ' ' + s
15 token ← tokenSep
16 tokenSep ← ' '
17 Para cada substring s en softwordDiv hacer
18     tokenSep ← tikenSep + ' '
                + divisiónSoftWord (s,score(s))
19 retornar tokenSep
```

Figura 3 - Algoritmo Samurai, divisiónHardword.

ALGORITMO SAMURAI

Esta técnica, propuesta por Eric Enslen, Emily Hill, Lori Pollock, VijayShanker divide a los Ids en secuencias de palabras similar al algoritmo Greedy, con la diferencia que es más efectiva (Enslen et al., 2009). La estrategia utiliza información que está presente en los códigos para llevar a cabo el objetivo. Esto permite que no sea necesario utilizar diccionarios predefinidos. Además las palabras que se obtuvieron 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 en base a una función de puntuación (scoring). Esta función utiliza información que se recupera extrayendo las frecuencias de aparición de palabras dentro del código fuente. Estas palabras pueden estar contenidas en comentarios, literales strings y/o documentación. Esto ocurre porque el objetivo de Samurai es más amplio y no sólo consiste en dividir Ids. Por esta razón el nombre del parámetro de entrada del algoritmo es token y no simplemente Id. El algoritmo primero se encarga de extraer información referida a la frecuencia de los tokens del código fuente. Entonces construye dos tablas de frecuencia de los mismos. Para la construcción de una de las tablas primero ejecuta el algoritmo que extrae del código fuente todos los tokens de tipo hardword. Éstos son agregados a la tabla de frecuencias locales. Una entrada de ella se corresponde a la lista de tokens extraídos del programa bajo análisis (cada token es único en la tabla). La otra entrada se corresponde al número de ocurrencia de cada token.

Por otro lado existe una tabla de frecuencias globales. Ésta contiene las mismas dos columnas que la anterior, tokens y sus frecuencias. La diferencia principal es que la información es capturada de distintos programas de gran envergadura. Durante el proceso de división de token Samurai ejecuta la función scoring que se basa en información de las dos tablas antedichas. El algoritmo Samurai recibe como entrada el token a dividir y retorna como resultado el token separado con espacios. La ejecución se inicia invocando la rutina divisiónHardWord (ver Figura 3, algoritmo 4, líneas 1 y 2). Esta rutina se encarga de dividir las hardwords (palabras que poseen guión bajo o sean camelcase) y entonces cada una de las palabras obtenidas son pasadas a la rutina división SoftWord que será explicada más adelante.

En la rutina divisiónHardWord (ver Figura 3, algoritmo 5) primero se ejecutan dos funciones (líneas 1 y 2). La primera dividirCaracteresEspecialesDigitos sustituye todos los posibles caracteres especiales y números que el token tiene por espacios en blanco.

La segunda dividir MinusSeguidoMayus, al igual que la anterior, agrega un blanco entre dos caracteres que sea una minúscula seguido por una mayúscula. En este punto sólo 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. Las palabras restantes del tipo mayúscula seguido de minúscula (List, ASTVisitor, GPSstate) continúa con el proceso de división. Este caso también es del tipo camelcase donde la mayúscula indica el comienzo de la nueva palabra. Por otro lado el autor, a través de estudios efectuados, se encontró con variantes en donde la mayúscula indica el fin de una palabra. Ejemplos: SQL list, GPS state. El algoritmo decide entre ambas variantes calculando la puntuación (score) de la parte derecha de ambas divisiones (líneas 7 y 8). Aquellas con puntuación más alta entre las dos será la elegida (línea 9). Para aclarar ambas opciones:

1. Realizar la separación entre el cambio de mayúscula a minúscula (ejemplo: GPS state, línea 11).

2. Realizar la separación antes de la primera mayúscula siguiendo la filosofía camelcase (ejemplo: GP Sstate, línea 13).

En este caso la opción correcta debe ser la a) ya que state debe tener una puntuación mayor que Sstate. Por último las partes resultantes de la división son enviadas a divisiónSoftWord (línea 18).

La rutina recursiva divisiónSoftWord (ver Figura 4, algoritmo 6) recibe como entrada un substring s que puede tener tres tipos de variantes:

1. Todos los caracteres en minúscula.

2. Todos los caracteres en mayúscula.

3. El primer caracter con mayúscula seguido por minúsculas (Visitor).

El otro parámetro de entrada es la puntuación original scoresd que corresponde a s. La rutina primero examina cada punto posible de división en s dividiendo en splitizq y splitder respectivamente (líneas 4 y 5). La decisión de cuál es la mejor división se basa en: a) substrings que no tengan prefijos o sufijos conocidos (línea 6), b) que el puntaje de la división elegida sobresalga del resto de los puntajes (líneas 7 hasta 9).

Algoritmo 6: división SoftWord
Entrada: s // softword string
$score_{sd}$ // puntaje de s sin dividir
Salida: token Sep // token dividido por espacios
1 tokenSep ← s,n ← length(s) - 1
2 i ← 0, maxScore ← -1
3 ***mientras*** (i < n) ***hacer***
4 $score_{izq}$ ← score(s[0,i])
5 $score_{der}$ ← score(s[i+1,n])
6 preSuf ← esPrefijo(s[0,i]) V esSufijo(s[i+1,n])
7 $split_{izq}$ ← $\sqrt{score_{izq}}$ > max(score(s),$score_{sd}$)
8 $split_{der}$ ← $\sqrt{score_{der}}$ > max(score(s),$score_{sd}$)
9 ***Si***(!presuf ∧ $split_{izq}$∧ $split_{der}$) ***ent***
10 ***Si*** (($split_{izq}$+ $split_{der}$) > maxScore) ***ent***
11 maxScore ← ($split_{izq}$ +$split_{der}$)
12 tokenSep ← s[0,i] +‘ ’ + s [i+1,n]
13 ***en otro caso, Si*** (!presuf ∧ $split_{izq}$) ***ent***
14 temp ← divisiónSoftWord (s [i+1,n],$score_{sd}$)
15 ***Si*** (temp se dividido?) ***ent***
16 tokenSep ← s[0,i] +‘ ’ + temp
17 i ← i+1
18 ***retornar*** tokenSep

Figura 4 - Algoritmo Samurai, divisiónSoftword.

Para aclarar este punto, para cada partición (izquierda o derecha) obtenida se calcula el score (líneas 4 y 5). Luego éste se compara con el puntaje máximo de la palabra original scoresd y la puntuación de la palabra actual (score(s)). En un principio ambos son iguales, pero a medida que avance la recursión score(s) puede variar relativamente con respecto a scoresd (líneas 7 y 8). En el caso de no tener los prefijos y sufijos comunes se considera

que la parte izquierda es un candidato. Por otro lado la rutina es invocada recursivamente sobre la parte derecha de la cadena para realizar posibles subdivisiones (línea 14).

Si la parte derecha finalmente se divide, luego entre la parte izquierda y la derecha también. Sin embargo cuando la parte derecha no se divide tampoco se debería separar entre ambas partes (la línea 13 controla esto). Los análisis de datos hechos por el autor indican que no se debe separar la actual posición debido a que hay evidencias que esto puede ser un error; un caso erróneo de división encontrado por el propio autor es string-ified. Otro problema ocurre cuando al calcular el puntaje de palabras con pocas letras tienden a ser más altos que el resto y conlleva a malas divisiones. Por tal motivo el autor decidió agregar la raíz cuadrada (líneas 7 y 8) antes de comparar el puntaje. Un ejemplo es per-formed. 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 dos decisiones claves durante el proceso de división: en la rutina divisiónHardWord para determinar si la división del Id es un caso de camelcase o la variante camelcase (líneas 7 y 8) y 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. La fórmula es la siguiente:

Frec(s, p) + (globalFrec(s) / log10 (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 (Enslen et al., 2009).

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 de palabras (en lenguaje natural) que se construye a partir del código fuente; una de frases (en lenguaje natural) presentes también en el código; una de palabras irrelevantes (stop-list); un diccionario de palabras en inglés (Lawrie et al., 2007).

```
Algoritmo 7: expansióndivisión SoftWord
Entrada:abrev        // abreviaturas a expandir
        wordList //Palabras extraídas del código
        phraseList//Frases extraídas del código
        stopList  //Palabras Excluyentes
        dic       //Diccionario en inglés
Salida: expansión // abreviatura expandida, o null
1 Si (abrev ∈ stopList) ent
2 |_retornar null
3 listaExpansión ← [ ]
  // Buscar acrónimo
4 Para cada frase fra en phraseList hacer
5 | Si (abrev es un acrónimo de fra) ent
6 |_|_retornar fra
  // Buscar abreviatura común
7 Para cada palabra w en wordList hacer
8 | Si (abrev es una abreviatura de w) ent
9 |_|_retornar w
  // Buscar diccionario
10  listaCandidatos←buscarDiccionario(abrev,dic)
    listaExpansión.add(listaCandidatos)
11 unicaExp ← null
  //Retorna un único resultado
12 Si (length(listaExpansión) = 1) ent
13 |unicaExp ← listaExpansión [0]
14 retornar unicaExp
```

Figura 5 - Algoritmo de Expansión Básica de Abreviaturas.

La primera lista se confecciona utilizando una herramienta que extrae las palabras de los comentarios que se encuentran y fuera de los métodos incluyendo aquellos que son relativos a la clase de estudio. La lista de frases se construye utilizando una herramienta que las extrae, los principales recursos son los comentarios y los Ids multi-palabras

(Feng and Croft, 2001). En este punto se construye un acrónimo con las palabras de las frases. Si este 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 file status_exists).

La lista de palabras irrelevantes y el diccionario se construyen de la misma manera que lo hace el algoritmo de Greedy (ver sección Algoritmo de Greedy).

Cuando las listas de palabras y frases potenciaAID les están creadas comienza la ejecución del algoritmo. Este algoritmo (ver Figura 5, algoritmo 7) recibe como entrada una abreviatura a expandir y las cuatro listas descriptas previamente (Pakhomov, 2002). El primer paso es ver si una abreviatura forma parte de la lista de palabras irrelevantes (o palabras a ignorar, línea 1). En caso que así sea no se retornan resultados (Larkey et al., 2000). La razón de ésto es que estas palabras no aportan información importante para la comprensión del código y son fácilmente relacionados por los ingenieros del software. Algunos casos son artículos/conectores (the, an, or) y palabras reservadas del lenguaje de programación (while, for, if, etc.). Siguiendo con la ejecución, el algoritmo verifica que las frases extraídas del código se correspondan con alguna abreviatura en forma de acrónimo (línea 5).

En caso de no encontrar una abreviatura intenta buscar que las letras de la misma coincidan en el mismo orden que una palabra presente en la lista de palabras recolectadas del código (línea 8). Ejemplos: horiz ⇒ HORIZontal, trg ⇒ TRianGle.

En caso no tener éxito la búsqueda continúa como último recurso con el diccionario compuesto con palabras en inglés (línea 10). Esta técnica de expansión descripta sólo retorna una única expansión potencial para una abreviatura determinada; caso contrario no retorna ninguna otra cosa (líneas 12 y 13). El motivo de ésto es que el algoritmo no tiene programado cómo decidir una única opción entre varias alternativas de expansión. Esta funcionalidad se deja, por lo autores, como trabajo futuro.

CASO DE ESTUDIO

En esta sección se describe cómo funciona AId a través de un caso de estudio: un programa Java de 600 líneas de código. Al ejecutar la herramienta el primer componente de interacción es una simple barra de menú localizada en el tope de la pantalla

Figura 6 - Captura de pantalla de AId, ventana de selección de archivos.

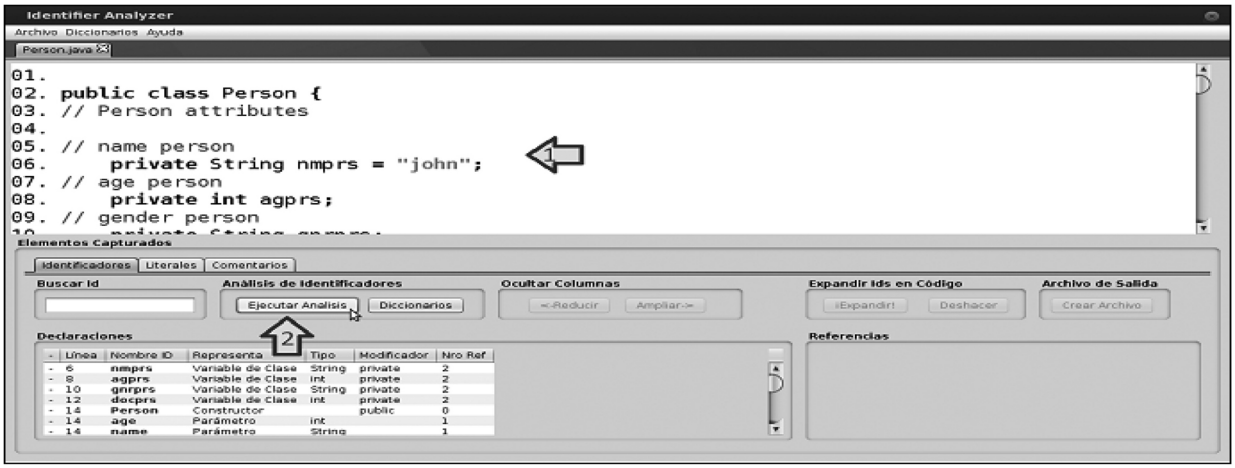

Figura 7 - Captura de pantalla de AId, panel de elementos capturados.

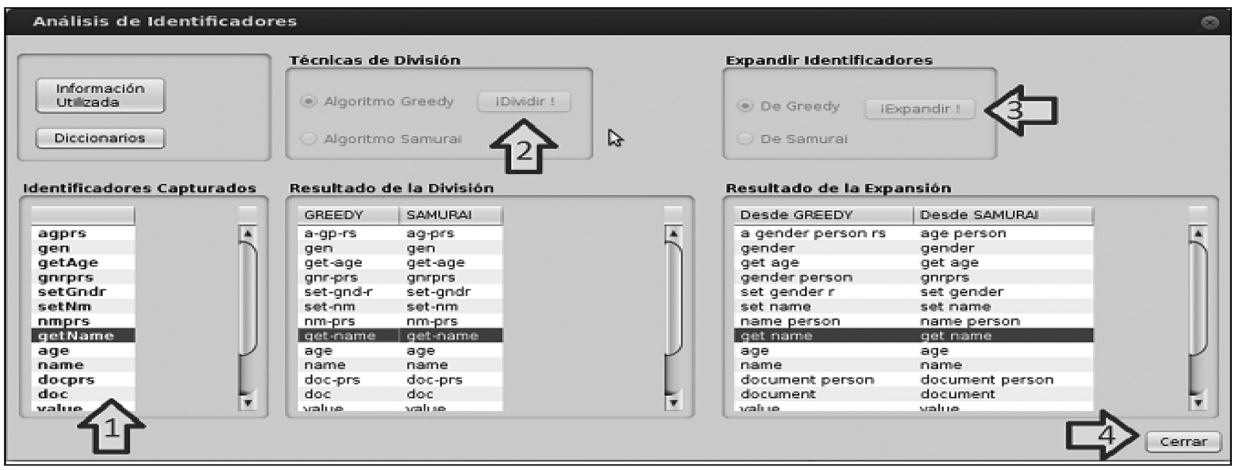

Figura 8 - Captura de pantalla de AId, panel de Análisis de Ids.

(ver Figura 6). Al hacer click en el menú Archivo de la barra antes mencionada y luego en Abrir Archivos(s) Java del menú desplegable se muestra una ventana de selección de archivos. Aquí el usuario puede escoger archivos Java de entrada (en la Figura 6 el menú Archivo está marcado con la flecha 1 y la ventana para seleccionar archivos con la flecha 2).

Una vez seleccionados los archivos Java comienza la ejecución del analizador sintáctico que extrae los Ids, comentarios y literales. Al finalizar la ejecución del analizador aparecerá un panel que contiene secciones. En la parte de arriba está el cóFigura digo leído del archivo Java (ver flecha 1 de la Figura 7). En la parte inferior están los elementos extraídos del código analizado.

Para proceder con el análisis de Ids se debe pulsar el botón Ejecutar Análisis (ver flecha 2 de la Figura 7) ubicado en el panel de Análisis de Identificadores. Una vez realizada esta tarea se abre un nuevo panel; en la parte inferior pueden observarse los Ids capturados (ver flecha 1 en Figura 8).

En el panel central superior se pueden seleccionar los algoritmos de división de Ids (Greedy y Samurai, ver flecha 2 de la Figura 8). El botón Dividir del mismo panel ejecuta la técnica seleccionada mostrando en el panel inferior la tabla con los resultados de división (Resultados de la División en Figura 8). De la misma forma los paneles subsiguientes de

la derecha permiten expandir las abreviaturas obtenidas en el panel de división (ver Figura 8). Pul sando el botón Expandir ejecuta el algoritmo de expansión básico tomando como entrada los Ids divididos por Greedy y/o Samurai, conforme el usuario decida (ver flecha 3 de la Figura 8). Los resultados se muestran en el panel de abajo. Las tres tablas mostradas en los tres paneles inferiores ayudan al usuario a comparar los resultados obtenidos.

Al pulsar en el botón Cerrar (ver flecha 4 de la Figura 8) la ventana del panel de análisis se oculta y los resultados obtenidos son cargados en la tabla

En caso de necesitar deshacer esta acción de sustitución se puede pulsar el botón Deshacer y restablecer el código original. Para finalizar, al pulsar el botón Crear Archivo (ver flecha 3 de la Figura 9) se crea un nuevo archivo con el mismo código pero con todos los Ids expandidos. En el ejemplo mostrado anteriormente AId recibe como entrada un programa Java de 600 líneas del código. En la Tabla 1 se pueden observar algunos Ids analizados. En las columnas de Algoritmo Greedy y Algoritmo Samurai se pueden apreciar los resultados de haber ejecutado las técnicas de división. Luego, en las próximas

Figura 9 - Vista principal de AId, aqui se aprecia la expansión de los Ids.

inferior del panel principal junto a la información que el analizador sintáctico había recuperado (ver flecha 1 de la Figura 9).

En la última columna se pueden apreciar múltiples cajas de selección. Éstas permiten al usuario escoger la expansión producida por Greedy o por Samurai (ver flecha 1 de la Figura 9).

Una vez concluida esta selección se procede a utilizar el panel Expandir Id en Código (ver flecha 2 de la Figura 9). Este panel tiene dos botones: Expandir o Deshacer. El primero lleva a cabo la sustitución de los Ids en el código de acuerdo con lo escogido en las cajas de selección mostrando el código resultante en el panel de arriba.

dos, se muestra el resultado de haber aplicado el algoritmo de expansión básico en cada resultado de los algoritmos de división. Como se observa, la mayoría de los casos de éxito fueron obtenidos de Samurai. Como observación importante se puede decir que Greedy consulta el gran volumen de datos provisto por el diccionario predefinido y que Samurai se basa en las tablas de frecuencias con los datos propios del código lo que indica que este último debe ser más preciso. Siguiendo con el caso de estudio, en la parte superior de la Figura 10 se puede observar una parte del código de estudio (dos métodos) con los Ids sin expandir. En la parte inferior de la Figura 10 se muestra el mismo frag-

mento de programa después de aplicar los algoritmo de expansión, o sea, con los Ids expandidos (resaltados en el código). Es evidente que esta última versión posee mayor información semántica y, por consiguiente, se torna más fácil percibir qué hace esa parte del código.

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

public void setDocPrs(it value)  {
      this.docPrs = value:
}
```

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

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

Figura 10 - Comparación del código de estudio, antes y después de expandir los Ids.

CONCLUSIONES Y TRABAJOS FUTUROS

Una motivación para desarrollar la herramienta AId se basa en la ausencia de herramientas con características similares. No abundan herramientas que posean una interfaz amigable con el usuario y ejecute técnicas de Análisis de Ids de un código recibido como entrada. Tampoco fue posible encontrar muchas implementaciones que integren extracción, división y expansión de abreviaturas de Ids. El uso del parser construido con ANTLR posibilita extraer con facilidad los elementos estáticos presentes en el código que son necesarios para los algoritmos que analizan Ids. Estos elementos son los Ids como objetos principales, comentarios, literales y documentación Java Doc.

Cabe destacar que la herramienta AId tiene implementadas dos técnicas de división que son Greedy y/o Samurai. La primera necesita consultar un diccionario de palabras en Inglés y una lista genérica de abreviaturas conocidas para llevar a cabo sus tareas. Ambas estructuras ocupan mucho espacio de almacenamiento, por eso se utiliza una base de datos para hacer las consultas más eficientes (Lawrie et al., 2006). La segunda alternativa, el algoritmo Samurai, divide los Ids mediante la utilización de recursos específicos extraídos del código. Este recurso son los comentarios, los literales y documentación Java Doc. Con estos recursos se construye una tabla con las frecuencias de aparición de palabras que después se utiliza en la función de scoring (ver Algoritmo Samurai). Sin embargo es posible que estos recursos sean escasos. Por esta razón los autores decidieron construir una lista de palabras perteneciente a un conjunto amplio de programas escritos en Java. Esta tabla no sólo ocupa menos espacio que los diccionarios genéricos de Greedy sino que también es más eficaz dado que está formado por palabras más adecuadas al ámbito de la ingeniería de software. Esto implica que la división sea más eficiente y, por consiguiente, que la expansión sea más precisa. Por otro lado el algoritmo de expansión básico usa los mismos diccionarios de palabras que utiliza Greedy con la diferencia que consulta previamente una lista de frases capturadas del código dando preferencia a esta lista. Es importante mencionar que AId incorpora una lista de frases no sólo de los comentarios sino también de los literales strings siendo ésta una característica que no se encuentra en el estado del arte de las herramientas actuales. Este algoritmo tiene el problema que ante múltiples alternativas de expansión no sabe escoger una única opción.

Como trabajo futuro se plantea implementar nuevas técnicas de análisis de Ids. Una de ellas es AMAP (Automatically Mining Abbreviation Expansions in Programs) (Hill et al., 2008). Esta técnica no necesita de diccionarios con palabras en inglés como es el caso del algoritmo de expansión básico; en su

lugar va observando gradualmente en el código los comentarios y literales presentes partiendo desde el lugar del Id que se quiera expandir. También resuelve el problema que posee el algoritmo básico cuando no conoce qué opción escoger ante muchas opciones de expansión. Para llevar a cabo la tarea anteriormente mencionada el algoritmo prioriza la frecuencia de aparición de las palabras. Esto se hace partiendo del lugar desde donde se encuentra el Id analizado. También AMAP permite entrenarse con un conjunto de programas pasado por entrada para recolectar más palabras y mejorar aún más la precisión de la expansión. Otra mejora futura para la herramienta AId es la posibilidad de construir un plugin para NetBean o Eclipse. Esto permitirá que el usuario abra un proyecto Java e inmediatamente con AId expanda los Ids para ayudar en la comprensión. También se pretende, en los próximos pasos de investigación, estudiar y proponer nuevas técnicas de análisis de Ids y hacer un estudio profundo de sus ventajas con respecto a la comprensión para un número significativo de programas.

AGRADECIMIENTOS

Se agradece el apoyo del Proyecto “Ingeniería de Software: Aspectos de Alta Sensibilidad en el Ejercicio de la Profesión de Ingeniero de Software” - UNSL, al Prof. Germán Montejano y al Mg. Augusto Farnese.

REFERENCIAS

Albanes, Berón, Henriques and Pereira, “Estrategias para relacionar el dominio del problema con el dominio del programa para la comprensión de programas”, XIII Workshop de Investigadores en Ciencias de la Computación, 449 – 453, (2011).

Storey, “Theories, methods and tools in program comprehension: past, present and future”, 13th International Workshop on Program Comprehension, 181–191, (2005).

Berón, “Inspección de Programas para Interconectar las Vistas Comportamental y Operacional para la Comprensión de Programas”, Tesis (Doctoral en Ciencias de la Computación), Universidad Nacional de San Luis, (2009).

Lawrie, Feild, and Binkley, “Extracting meaning from abbreviated identifiers”, Seventh IEEEInternational Working Conference on Source Code Analysis and Manipulation, 213–222, (2007).

Enslen, Hill, Pollock and Vijay-Shanker, “Mining source code to automatically split identifiers for software analysis”, 6th IEEE International Working Conference on Mining Software Repositories, 71–80, (2009).

Caprile and Tonella, “Restructuring program identifier names”, International Conference on Software Maintenance, 97–107, (2000).

Freitas, da Cruz and Rangel Henriques, “The role of comments on program comprehension”, Informatics Department, Universidade do Minho, Braga, Portugal, (2008).

Binkley, Davis, Lawrie, Maletic, Morrell and Sharif, “The impact of identifier style on effort and comprehension”, Empirical Software Engineering, 18(2):219–276, (2013).

Deissenboeck and Pizka, “Concise and consistent naming”, Software Quality Journal, 14(3):261–282, (2006).

Aho, Lam, Sethi and Ullman, “Compilers: Principles, Techniques, and Tools”, Second Edition, Addison Wesley, ISBN: 978-0321486813, (2006).

Aho, Ullman and Hopcroft, “Data structures and algorithms”, Addison-Wesley Reading, Mass, (1983).

Feild, Binkley and Lawrie. “Identifier splitting: A study of two techniques”, Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems, 154–160, (2006).

Feng and Croft, “Probabilistic techniques for phrase extraction”, Information processing & management, 37(2):199–220, (2001).

Pakhomov, "Semi-supervised maximum entropy based approach to acronym and abbreviation normalization in medical texts", Proceedings of the 40th annual meeting on association for computational linguistics, 160-167, (2002).

Larkey, Ogilvie, Price and Tamilio, "Acrophile: an automated acronym extractor and server", Proceedings of the fifth ACM conference on Digital libraries, 205–214, (2000).

Lawrie, Feild and Binkley, "Quantifying identifier quality: an analysis of trends", Empirical Software Engineering, 12(4):359–388, (2006).

Hill, Fry, Boyd, Sridhara, Novikova, Pollock and Vijay-Shanker, "Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tolos", Proceedings of the 5th Int'l Working Conf. on Mining Software Repositories, 79–88, ACM, (2008).

Fecha de Presentación: 15/05/15
Fecha de Aprobación: 08/06/15