AId: Uma Ferramenta para Análise de Identificadores de Programas Java

Javier Azcurra
Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina
javierazcurra_m@yahoo.com.ar

Mario Berón
Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina
mberon@unsl.edu.ar

Germán Montejano
Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina
gmonte@unsl.edu.ar

Augusto Farnese
Departamento de Ciências da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
farnese@dcc.ufmg.br

Pedro Rangel Henriques
Departamento de Informática
Universidade do Minho
Braga, Portugal
pedrorangelhenriques@gmail.com

Maria J. Varanda Pereira
Departamento de Informática
Instituto Politécnico de Bragança
Bragança, Portugal
mjoao@ipb.pt

Resumo

As demandas atuais no desenvolvimento de software implicam uma evolução e manutenção constante do software com menor custo de tempo e recursos [15,16, 17, 18]. A Compreensão de Programas (CP), uma disciplina da Engenharia do Software, fornece os métodos, técnicas e estratégias para levar adiante esta tarefa. Em geral, as técnicas de compreensão fazem uso de duas classes muito importantes de informação: Estática e Dinâmica. Em ambas as classes, há um elemento que é informativo e sempre usado: os identificadores (Id). Estudos indicam que os Ids, mesmo quando abreviados ou compostos, encerram indícios das funcionalidades dos sistemas onde são usados [12, 7, 9, 8]. Por esta razão construir ferramentas de compreensão que automatizem o processo de extração e análise dos identificadores é uma contribuição muito importante para a CP.

Neste artigo apresenta-se a AId uma ferramenta que: i) automatiza a recuperação de identificadores encontrados em programas escritos em Java, e ii) aplica algoritmos de análise de identificares a fim de capturar o seu significado com vista a ajudar a compreender o programa.

1. Introdução

A Compreensão de Programas [20, 21, 31] é uma disciplina da Engenharia de Software cuja finalidade é facilitar o entendimento dos sistemas. Uma forma, se não a mais importante, de atingir este objetivo consiste em relacionar dois domínios muito importantes, como são: o Domínio do Problema e o Domino do Programa. Para poder fazer essa relação deve-se: i) construir uma representação do Domínio do Problema, ii) construir uma representação do Domínio do Programa, e iii) elaborar uma estratégia de vinculação de ambos domínios [19, 22, 23, 30].

Levando em consideração o Domínio do Problema, pode-se dizer que um possível caminho para elaborar uma representação consiste em: i) em extrair os conceitos usados no programa, e ii) estabelecer relações entre eles formando assim aquilo que se designa por mapa conceitual. Uma aproximação interessante para recuperar os conceitos consiste em capturar os identificadores do programa e realizar uma análise considerando o contexto onde aparecem. Os estudos realizados [1, 6, 7, 24] indicam que uma grande parte dos programas têm identificadores (Ids), os quais fornecem abundante informação. Geralmente, os Ids são compostos por mais de uma palavra em forma de abreviatura. Vários autores são de opinião [12, 7, 9, 8] que essas abreviaturas podem conter informação oculta que é própria do Domínio do Problema (conceitos). Uma forma possível de extrair essa informação dos identificadores consiste em expandir essas abreviaturas para as suas correspondentes palavras expressas em linguagem natural. Para atingir esta meta deve-se realizar as seguintes tarefas: i) extrair os identificadores do código, ii) aplicar técnicas de divisão onde o Id se decompõe nas distintas palavras abreviadas que o compõem, e iii) usar estratégias de expansão das abreviaturas para transformar as mesmas em palavras completas.

Normalmente, os nomes dos Ids são escolhidos mediante os critérios do programador [14, 7, 8]. Isto representa a principal dificuldade para as técnicas de expansão de abreviaturas de Ids (porque nem sempre os programadores usam as convenções adequadas e em muitos casos os nomes não refletem a semântica correta). Uma forma de enfrentar esta dificuldade consiste em usar as fontes de informação informal que se encontram disponíveis no código fonte. Por informação informal se entende aquela contida nos comentários dos módulos,

comentários das funções, literais, documentação do sistema e todos os demais recursos descritivos do programa que estejam escritos em linguagem natural. Sem duvida, os comentários têm como principal finalidade ajudar a compreender um segmento do código [13, 25]. Por esta razão, os comentários são uma fonte de informação natural para entender o significado dos Ids no código, como também para extrair conceitos do Domínio do Problema. Por outro lado para poder entender a semântica dos Ids, consideram-se também os literais (strings constantes). Eles representam um valor constante formado por sequencias de caracteres. Eles são geralmente utilizados nas instruções de escrita de mensagens para ficheiro ou écran ou para inicializar variáveis do tipo string. No caso destas fontes de informação informal serem escassas no código da aplicação, pode-se recorrer a alternativas externas como é o caso dos dicionários predefinidos de palavras em linguagem natural.

Na atualidade pode-se classificar as estratégias de análise de identificadores em dois grandes grupos. As que usam dicionários predefinidos e as que não. As primeiras são as mais arcaicas e as segundas as mais recentes. Infelizmente, nem todas as estratégias de análise de identificadores estão programadas em ferramentas automáticas.

O artigo está organizado como se descreve a seguir. Na seção 2, se explicam os conceitos subjacentes ao contexto da teoria de análise de identificadores. Na seção 3, se apresenta o AId uma ferramenta que reúne as características mencionadas na introdução deste artigo e que analisa detalhadamente identificadores e expande as abreviaturas. Na seção 4, se mostra o funcionamento do AId usando um caso de estudo. Finalmente, na seção 5 apresentam-se as conclusões e futuras extensões deste trabalho.

2. Análise de Identificadores

"Um identificador (Id) basicamente se define como uma sequência de letras, dígitos ou caracteres especiais de qualquer comprimento que serve para identificar as entidades do programa, isto é, para dar a cada uma um nome único que a individualize".

Cada linguagem tem as suas próprias regras que definem como podem ser construídos os nomes dos seus Ids. Por exemplo, em linguagens como C e Java não se pode declarar Ids que coincidam com as palavras reservadas ou que contenham alguns símbolos especiais ($,&,#) com exceção dos hifens ou underscore (-_).

No caso geral, um Id num programa está associado com um ou mais conceitos do programa.

$$\text{Identificador} \Leftrightarrow \text{Conceito}$$

Noutras palavras, um Id é um representante dum conceito que pertence ao Domínio do Problema [1, 2, 25]. Por exemplo, o Id *openWindow* está associado ao conceito "abrir uma janela". Daí a importância de analisar os Ids.

Para melhorar a CP se requere que os nomes dos Ids transmitam de maneira clara os conceitos que representam [1, 3, 4]. Contudo, na prática isto não é levado em conta. Durante o desenvolvimento dos sistemas, o critério seguido para atribuir nomes aos Ids está mais orientado para o estilo de escrita no formato do código e sua documentação do que no conceito que o Id representa. Por isso durante a etapa de manutenção de software o nome dos Ids pode não ser útil para compreender o sistema.

Antes de continuar descrevendo a importância dos nomes de identificadores serão classificadas as distintas formas em que se pode atribuir um nome a um Id. Estudos realizados com 100 programadores [4] sobre compreensão de Ids indicam que existem três tipos principais de nomes (tomando como exemplo o conceito *File System Input*): palavras completas (*fileSystemInput*), abreviaturas (*flSysIpt*), uma só letra (*f*). É importante ter em conta que também os nomes dos Ids podem estar compostos por mais de uma palavra como claramente mostra-se acima.

Os resultados do estudo mostram que as palavras completas são as melhor compreendidas, no entanto as estatísticas mostram em alguns casos que as abreviaturas, que se localizam em segundo lugar, não demostram uma diferença notória em relação às palavras completas [4].

Feild e o seu grupo de pesquisa fazem duas distinções conhecidas como *hardwords* e *softwords* [5, 6, 2]. As hardwords destacam a separação de cada palavra que compõe o identificador através duma marca específica; alguns exemplos são: *fileSystem*[1] ou *fileSYSTEM* (marca a separação com o uso duma maiúscula entre minúsculas) assim também como file-*system* ou file_system em que se utiliza um caráter especial, como o hífem ou o underscore. As *softwords* não tem nenhum tipo de separador ou marca que dê indícios das palavras que o compõem. Por exemplo: *textinput* ou *TEXTINPUT* estas palavras estão compostas por *text* e por *input*, mas a palavra composta não tem uma marca que destaque as partes. A nomenclatura das hardwords e softwords será utilizada no resto deste artigo.

Voltando a descrever a importância dos nomes de identificadores pode-se dizer que existem inúmeras convenções de nomes dos Ids, algumas delas são: no caso de Java, os nomes dos pacotes devem começar com minúscula (main.packed); o nome das classes deve ter uma maiúscula na primeira letra de cada palavra componente (MainClass). No caso de C#, as classes são nomeadas da mesma forma que em Java. Porém o nome

[1] Esta notação é conhecida no contexto da programação como camel-case.

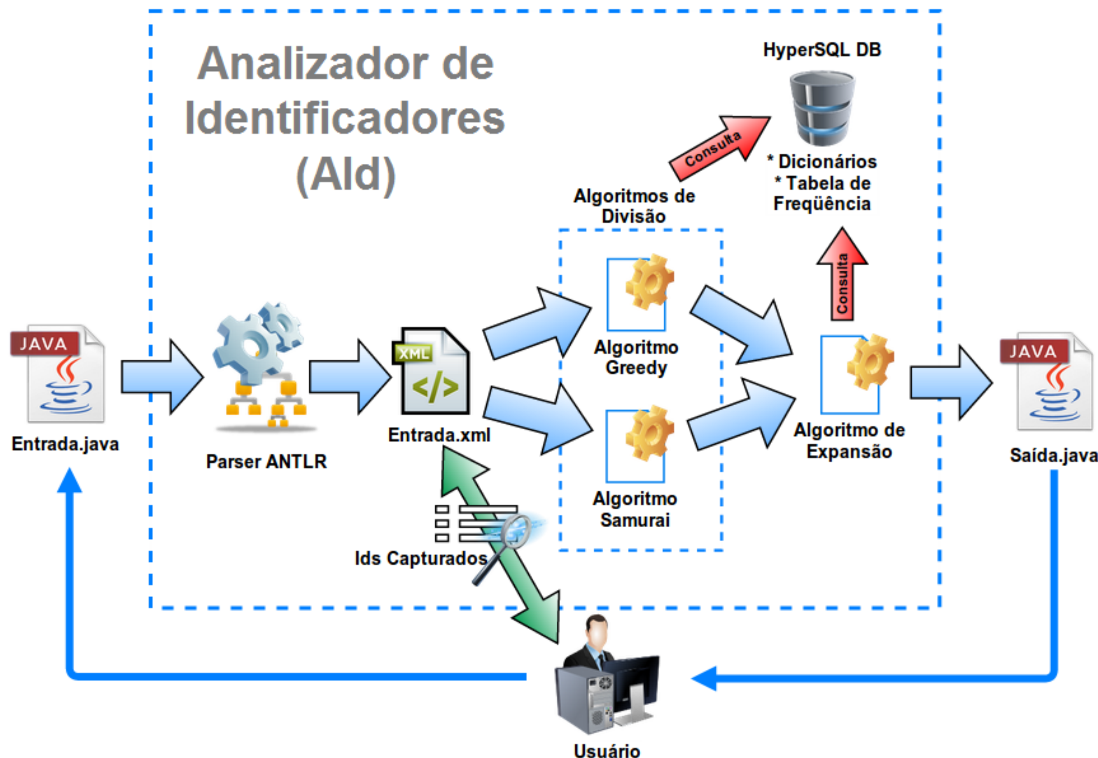

Figura 1 – Arquitetura do AId

dos pacotes em cada nível deve começar com maiúscula e o resto com minúscula (Main.Packed). Isto indica que o critério seguido se concentra mais em aspetos sintáticos do Id, ligados ao domínio da programação e não tanto nos aspetos semânticos ligados aos conceitos do domínio do problema. Uma evidência clara da importância do critério orientado para a semântica são as técnicas que se aplicam na proteção do código. Algumas delas se encarregam de substituir os nomes originais dos Ids por sequências de caracteres aleatórios e dessa forma reduzem a compreensão: técnicas de ofuscação de código.

Quando os programadores desenvolvem as suas aplicações não prestam importância à correta atribuição de nomes com semântica aos Ids. Existem três razões principais para isso: i) Os Ids são escolhidos pelos programadores sem ter em conta os conceitos associados; ii) Os desenvolvedores têm pouco conhecimento dos nomes usados em Ids localizados em outros setores do código fonte, e iii) Durante a evolução do sistema, os nomes dos Ids se mantêm e não se adaptam às novas funcionalidades (ou conceitos) a que possam vir a estar associados. Neste sentido, a má atribuição de nomes aos Ids se combate com as normas da chamada programação "menos egoísta". Esta consiste em fazer programas mais claros e compreensíveis para o futuro leitor que não esteja familiarizado. Para atingir este objetivo se devem respeitar duas regras de atribuição de nomes [1, 3]: i) Atribuição concisa: o nome do Id é conciso se a semântica do nome coincide exatamente com a semântica do conceito que o Id representa, e ii) Atribuição Consistente: cada Id deve ter associado um e só um único conceito.

O uso de *sinónimos* e *homónimos* prejudica a atribuição consistente de nomes aos Ids. Os homónimos são palavras que podem ter mais de um significado. Por este motivo, se o nome dum Id está associado a mais dum conceito, não estará claro que conceito representa.

Por outra parte, os sinónimos indicam que para um mesmo conceito podem ter associados diferentes nomes. Por exemplo, os nomes *accountBankNumber* e *accountBankNum* são sinónimos porque fazem referência ao mesmo conceito "número de conta bancária". Está demonstrado que a utilização de nomes consistentes, tal como foi mencionado atrás, é uma característica desejada. Isto é porque eles fazem que seja difícil identificar com clareza os conceitos no Dominio do Problema e, portanto os esforços para compreender o programa são maiores. Se os Ids tem nomes consistentes e concisos (identificando bem o conceito) os conceitos do Dominio do Problema podem ser descobertos mais facilmente [1, 3]. Intuitivamente, se necessita que os Ids representem bem um conceito, já que maior será o impacto que terá na interpretação do sistema [1, 3]. Contudo, durante as etapas de desenvolvimento e manutenção de software é difícil manter uma consistência global de nome nos Ids, sobretudo se o sistema for grande. Cada vez que um conceito se

modifica, o nome do Id associado deve mudar e adaptar-se à modificação.

Deissenboeck e o seu grupo de pesquisa em [1] propõem utilizar uma ferramenta que resolva os problemas da má atribuição de nomes descritos nos parágrafos precedentes. Dada a dificuldade de construir uma ferramenta totalmente automática para nomear corretamente os Ids, eles fizeram uma ferramenta semiautomática que necessita da intervenção do programador. Esta ferramenta, à medida que o sistema se vai desenvolvendo, constrói e mantém um dicionário de dados contendo informação sobre os Ids. No âmbito da Engenharia do Software o conceito de dicionários de dados é importante. Isto é, porque com os dicionários é possível descrever com clareza todos os termos utilizados nos grandes sistemas de software.

2.1. Tradução de Identificadores

As pessoas perante o código dos programas têm sérias dificuldades em perceber o propósito dos Ids, mas na verdade precisam de tempo a analisar o significado de sua presença. As estratégias automáticas dedicadas a facilitar esta análise são bem-vindas no contexto da CP.

A literatura dedicada a estratégias de análise de Ids indica que os Ids ocultam informação relevante do Domínio do Problema através de abreviaturas [7, 8]. Uma forma de exibir a informação oculta nos Ids é tentar converter estas abreviaturas em palavras completas em linguagem natural. Portanto, o foco da análise de Ids baseia-se na tradução das palavras abreviadas para palavras completas. O processo automático que se leva a cabo para realizar a tradução de Ids consta de dois passos [7]:

1. Divisão: separar o Id nas palavras que o compõem usando algum separador especial (Ejemplo: *flSys* ⇒ *fl-sys*).
2. Expansão: Expandir as abreviaturas que resultaram como produto do passo anterior.

3. AId: uma ferramenta para Análise de Identificadores

A figura 1 mostra a arquitetura do AId uma ferramenta que automatiza o processo de análise de identificadores. Esta ferramenta foi implementada usando a linguagem Java e o ambiente de desenvolvimento Netbeans. Na mesma figura pode-se constatar que AId permite realizar a análise em três etapas, a primeira consiste na extração de dados, a segunda tem por objetivo realizar a divisão dos Ids, e a terceira faz a expansão de Ids.

A fase de extração de dados (primeiro passo) recebe como entrada um ficheiro Java (*Entrada.java* na figura 1) que é processado por um analisador sintático (parser). Este parser foi programado utilizando a ferramenta ANTLR[2]. O analisador sintático extrai e armazena em estruturas internas informação estática sobre o código lido do ficheiro Java. Esta informação é composta por Ids como elemento principal, literais e comentários. Finalmente, toda esta informação recolhida se armazena num ficheiro XML (*Entrada.xml* na figura 1). Concluída a extração de informação se dá inicio à fase de divisão de Ids. Nessa etapa podem ser escolhidos dois algoritmos: *Greedy* e *Samurai*. Ambos os algoritmos recebem como entrada a informação da etapa anterior e dividem os Ids do código de entrada. As divisões são armazenadas em estruturas internas que serão consultadas na fase seguinte. Como estes algoritmos de divisão necessitam de dados externos, como dicionários (no caso de Greedy) ou tabelas de frequência de palavras (no caso de Samurai), estes dados se encontram armazenados na base de dados *HyperSQL DB* (parte superior da figura 1). A terceira e última fase implementa o algoritmo de expansão convencional que será explicado mais adiante. Este algoritmo toma os Ids divididos na etapa anterior e procede à sua expansão. Aqui também se necessita dos dicionários, por isso se fazem consultas à *HyperSQL DB*. Uma vez expandidos todos os Ids, estes são substituídos pelos Ids originais no ficheiro de entrada, gerando desta maneira o ficheiro de saída (*Saída.java* na figura 1).

3.1. Analisador Sintático

Foram pesquisadas ferramentas destinadas a construir analisadores léxicos e sintáticos. Neste sentido se deu preferencia àquelas ferramentas que se baseiam na teoria associada às gramáticas de atributos [28]. Da pesquisa previamente descrita, se determinou que a ferramenta ANTLR era a que melhor cumpria com o requisito antes mencionado. Esta ferramenta permite adicionar regras para o cálculo dos atributos, escritas em Java, à gramática independente de contexto inicial, neste caso também em linguagem Java. Estas regras de cálculo, ou ações semânticas, devem ser corretamente associadas à gramática para, por exemplo, implementar estruturas de dados e algoritmos que recolham os identificadores utilizados num programa [29]. Uma vez inseridas estas ações, a ferramenta se encarrega de ler a gramática e gerar o analisador adicionando as ações que foram programadas. Desta forma, se obtém um analisador sintático que não só examina o código Java para detectar erros ao nível sintático mas que também recolhe e armazena Ids. Além destas ações, se adicionaram outras ações semânticas que extraem comentários e strings literais. Estes elementos são necessários já que servem como entrada para os algoritmos de análise de Ids que serão explicados nas próximas secções.

[2] ANother Tool for Language Recognition - http://www.antlr.org

Algoritmo 1: Divisão Greedy
Entrada: idHarword *// identificador a dividir*
Saída: softwordDiv *// Id separado com espacios*
Variaveis Globais:
ispellList *// Palaras de ispell + Dicionário*
abrevList *// Abrevituras conhecidas*
stopList *// Palavras Excludentes*

```
1 softwordDiv← “”
2 softwordDiv← dividirCaracsEspDig(idHarword )
3 softwordDiv ← dividirCamelCase(softwordDiv)
4 Para cada substring s en softwordDiv fazer
5 | Se (s ∉ (stopList ∪ abrevList ∪ ispellList )) então
6 | | sPrefixo ← procurarPrefixo(s,“”)
7 | | sSufixo ← procurarSufixo(s,“”)
  | | // Se escolhe a divisão que fiz mais partições
8 |_|_s ← maxDivisão(sPrefixo,sSufixo)
9 return softwordDiv // o Id dividido por espaços.
```

Algoritmo 2: procurarPrefixo
Entrada: s // Abreviatura a dividir
Saída: abrevSeparada //Abrev dividido por espaços

```
  // Punto de parada da recursión
1 Se (length(s) = 0) então
2 |_return abrevSeparada
3 Se (s ∈ (stopList ∪ abrevList ∪ ispellList )) então
4 |_return (s + ‘ ’+ procurarPrefixo(s,“”))
  // Se extrae e se almacena o último caracter de s.
5 abrevSeparada ← s[length(s) - 1] + abrevSeparada
  //Chamada recursiva sem o último caracter.
6 s ← s[0 , length(s) - 1]
7 return procurarPrefixo(s,abrevSeparada )
```

Algoritmo 3: procurarSufijo
Entrada: s // Abreviatura a dividir
Saída: abrevSeparada //Abrev dividido por espaços

```
  // Punto de parada da recursión
1 Se (length(s) = 0) então
2 |_return abrevSeparada
3 Se (s ∈ (stopList ∪ abrevList ∪ ispellList )) então
4 |_return (procurarSufixo(s,“”) + ‘ ’+ s)
  // Se extrai e se guarda o primer carater de s.
5 abrevSeparada ← abrevSeparada + s[0]
  //Chamada recursiva sem o primeiro caracter.
6 s ← s[1 , length(s)]
7 return procurarSufixo(s,abrevSeparada)
```

3.2. Módulo de Divisão

O módulo de divisão implementa dois algoritmos, eles são: *Greedy* e o *Samurai*, os quais serão descritos a seguir.

3.2.1. Algoritmo Greedy

Como já foi várias vezes dito, o algoritmo Greedy, inicialmente proposto por o Lawrie, Feild, Binkley [10,5,6,7,8] utiliza três listas, a saber:

1. Palavras de dicionários: contêm palavras extraídas de dicionários públicos e do dicionário que utiliza o comando de Linux ispell.
2. *Abreviaturas conhecidas*: a lista se constrói com abreviaturas extraídas de distintos programas de autores especialistas. Incluem-se abreviaturas comuns (exemplo: *alt* → *altitude*) e abreviaturas de programação (exemplo: *txt* → *text*).
3. *Palavras a ignorar (stop words)*: possui palavras que são irrelevantes para realizar a divisão dos Ids. Inclui palavras chave da linguagem de programação (exemplo: *while*), Ids predefinidos (exemplo: *NULL*), nomes e funções de bibliotecas (exemplo: *strcpy*, *errono*), e todos os Ids que tenham um só caracter. Esta lista é muito grande.

O algoritmo Greedy utiliza as três listas mencionadas acima em forma de variável global. Isto ocorre porque as três listas são usadas por sub-rotinas.
O algoritmo procede da seguinte forma (ver algoritmo 1[3]), o Id que recebe como entrada se divide com espaços em branco, nas hardwords que o compõem (exemplo: *fileinput-txt* → *fileinput* y *txt* na linha 2 se é separada com caracteres especiais, ou, *fileinputTxt* → *fileinput txt* na linha 3) se é camelcase. Cada palavra resultante é pesquisada e caso esteja numa das três listas, é armazenada como uma só softword (exemplo: *txt* pertence à lista de abreviaturas conhecidas-linha 5). Se alguma palavra não está em nenhuma lista se considera como múltiplas softwords que necessitam subdividir-se (exemplo: *fileinput* → *file* e *input* - linha 5). Para subdividir estas palavras se procuram os prefixos e os sufixos mais longos possíveis dentro delas. Esta tarefa também se faz utilizando as três as listas antes mencionadas (linhas 6 e 7). Por um lado se procuram prefixos com um processo recursivo (ver algoritmo 2). Este processo começa analisando toda a palavra por completo. Vão sendo extraídos carateres do final até encontrar o prefixo mais longo que seja uma palavra válida, ou até que não haja mais caracteres (linhas 5 até 7 da função). Quando uma palavra se encontra numa lista (linha 3 da função) se coloca um separador (‘ ’). O resto que foi descartado se processa de novo através da função *procurarPrefixo* para procurar mais subdivisões (linha 4). De forma simétrica, outro processo recursivo se usa para tratar dos sufixos (algoritmo 3). Também extrai caracteres, mas neste caso desde a primeira posição até encontrar o sufixo mais longo presente numa lista ou até que não haja mais caracteres (linhas 5 até 7 da função).

[3] *softwordDiv* se modifica a traves de s na linha 8.

```
Algoritmo 4: Divisão Samurai
Entrada: token // token a dividir
Saída: tokenSep // token dividido por espaços
1 tokenSep ← divisãoHardWord (token)
2 return tokenSep
```

```
Algoritmo 5: divisãoHardWord
Entrada: token // token a dividir
Saída: tokenSep // token dividido por espaços
1 token← dividirCaracsEspDig(token)
2 token← dividirMinusSeguidoMayus(token)
3 tokenSep ← “”
4 Para cada substring s en token fazer
5   Se (∃{i|esMayus(s[i]) ∧ esMinus(s[i + 1])}) então
6     n ← length(s) – 1
      // se determina o tipo de divisão
7     scoreCamel ← score(s[i,n])
8     scoreAlter ← score(s[i+1,n])
9     Se(scoreCamel> √scoreAlter) então
10      Se (i > 0) então
11        s ← s[0,i − 1] + ‘ ’ + s [i,n]
12    em outro 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 fazer
18   tokenSep← tokenSep + ‘ ’
                 + divisãoSoftWord (s,score(s))
19 return tokenSep
```

Da mesma forma que a função de prefixos, quando encontra uma palavra, se insere um separador (‘ ’) e o resto se processa através da função *procurarSufixo* (linha 7). Quando ambos os processos terminarem, os resultados são retornados ao algoritmo principal. Mediante a função de comparação se escolhe o que tem maiores partições (linha 8 do algoritmo 1). Finalmente, o algoritmo Greedy retorna o Id destacando as palavras que o compõem usando como separador o carater espaço em branco (por exemplo *file input txt)* (linha 9 do algoritmo 1). A vantagem de fazer duas procuras (prefixo e sufixo) é aumentar as possibilidades de dividir o Id correctamente. Por exemplo, supondo que a palavra abreviada *fl* não se encontra em nenhum das três listas, mas sim as palavras *input* e *txt*. Dada esta situação, se o Id *flinputtxt* for processado por ambas as rotinas, o resultado de invocar *procurarPrefixo* é que não conseguirá dividir o Id. Isto sucede porque ao retirar caracteres do fim nunca se encontrará um prefixo conhecido. Não havendo divisão entre *fl* e *input* o resto da cadeia não se processará e tão pouco se dividirá em input e txt. Contudo, problema não surge quando se usa *procurarSufixo* porque ao retirar os caracteres do princípio da palavra *input txt* ficarão separados. Como o input é uma palavra conhecida se agregará um espaço em branco entre *fl* e *input*. Desta maneira o Id fica corretamente separado em três partes: *fl input txt*.

3.2.2. Algoritmo Samurai

Esta técnica, proposta por Eric Enslen, Emily Hill, Lori Pollock e VijayShanker [8], divide os Ids em sequências de palavras de forma semelhante ao algoritmo o Greedy mas a separação é mais efetiva. A estratégia utiliza informação presente no código para levar a cabo o objetivo. Isto permite que não seja necessário utilizar dicionários predefinidos. Além disso, as palavras que se obtiverem produto da divisão não estão limitadas pelo conteúdo destes dicionários. Desta forma, a técnica vai evoluindo com o tempo na medida em que novas tecnologias e as novas palavras se incorporem ao vocabulário dos programadores. O algoritmo escolhe a partição mais adequada nos Ids multi-palavra com base em uma função de pontuação (scoring). Esta função utiliza informação que se recolhe extraindo as frequências de aparição das palavras dentro do código fonte. Estas palavras podem estar contidas nos comentários, nas strings literais ou na documentação. Isto ocorre porque o objetivo de Samurai é mais amplo e não consiste somente em dividir Ids. Por esta razão, o nome do parâmetro de entrada do algoritmo é token e não simplesmente Id. O algoritmo primeiro se encarrega de extrair informação respeitante à frequência dos tokens no código fonte. Em seguida, constrói duas tabelas de frequência de tokens. Para a construção duma das tabelas primeiro executa o algoritmo que extrai do código fonte todos os tokens do tipo hardword. Estes tokens são agregados na tabela de frequências específicas. Uma entrada desta tabela corresponde à lista de tokens extraídos do programa em análise (cada token é único na tabela). A outra entrada corresponde ao número de ocorrência de cada token.

Por outro lado, existe a tabela de frequências globais. Esta tabela contém as mesmas duas colunas que a tabela anterior, tokens e suas frequências. A diferença principal é que a informação é recolhida de distintos programas de grande envergadura. Durante o processo de divisão do token, Samurai executa a função de scoring que se baseia na informação das tabelas mencionadas anteriormente. O algoritmo Samurai recebe como entrada o token a dividir e retorna como seu resultado o token separado com espaços. A execução se inicia invocando a rotina *divisãoHardWord* (ver algoritmo 4, linhas 1 e 2). Esta rotina se encarrega de dividir as hardwords (palavras que possuem underscore o são do tipo camel-case), e então cada uma das palavras obtidas são passadas à rotina *divisãoSoftWord* que será explicada mais adiante.

```
Algoritmo 6: divisãoSoftWord
Entrada: s // softword string
         score_sd // pontuação de s sem dividir
Saída: tokenSep // token dividido por espaços
1 tokenSep ← s, n ← length(s) - 1
2 i ← 0, maxScore ← − 1
3 enquanto (i < n) fazer
4  | score_izq ← score(s[0,i])
5  | score_der ← score(s[i+1,n])
6  | preSuf ← esPrefijo(s[0,i]) ∨ esSufijo(s[i+1,n])
7  | split_izq ← √score_izq > max(score(s),score_sd)
8  | split_der ← √score_der > max(score(s),score_sd)
9  | Se (!presuf ∧ split_izq ∧ split_der) então
10 | | Se ((split_izq + split_der) > maxScore) então
11 | | | maxScore ← (split_izq + split_der)
12 | |_|_tokenSep ← s[0,i] + ' ' + s [i+1,n]
13 | em outro caso, Se(!presuf ∧ split_izq) então
14 | | temp ← divisãoSoftWord (s [i+1,n],score_sd)
15 | | Se (temp se dividido?) então
16 | |_ |_tokenSep ← s[0,i] + ' ' + temp
17 |_i ← i+1
18 return tokenSep
```

Na rotina *divisãoHardWord* (ver algoritmo 5) primeiro se executam duas funções (linhas 1 e 2). A primeira *dividirCaracteresEspeciasDigitos*, que substitui todos os possíveis caracteres especiais e números que o token possua por espaços em branco.

A segunda *dividirMinusSeguidoMayus*, da mesma forma que a anterior, agrega um espaço em branco entre dois caracteres que sejam uma minúscula seguido por uma maiúscula. Neste ponto só ficam tokens da forma softword ou que contenham uma maiúscula seguida de uma minúscula (Exemplos: *List*, *ASTVisitor*, *GPSstate*, *state*, *finalstate*, *MAX*). Os casos de softword que se obtiveram (*finalstate*, *MAX*) são passados diretamente à rotina *divisãoSoftWord*. As palavras obtidas que sigam o padrão maiúscula seguida de minúscula (*List*, *ASTVisitor*, *GPSstate*) vão ser submetidas a novo processo de divisão. Este caso é também do tipo camel-case onde a maiúscula indica começo da nova palavra. Contudo, o autor através de estudos que efetuou, encontrou variantes onde minúscula que surge a seguir a uma sequência de maiúsculas indica o fim da palavra (exemplo: SQLlist). O algoritmo decide entre ambas as variantes, calculando a pontuação (score) da parte direita de ambas as divisões (linhas 7 e 8). Aquela com pontuação mais alta entre as duas, será a escolhida (linha 9).

Para esclarecer melhor ambas as opções:

a) realizar a separação entre a mudança de maiúscula para minúscula (exemplo: *GPS state* - linha 11).
b) realizar a separação antes da primeira maiúscula segundo com a filosofia camel-case (exemplo: *GP Sstate* - linha 13)

Neste caso, a opção correta dever ser a a), já que state deve ter uma pontuação maior do que *Sstate*. No final as partes resultantes da divisão são enviadas a *divisãoSoftWord* (linha 18).

A rotina recursiva *divisãoSoftWord* (ver algoritmo 6) recebe como entrada um substring s que pode ter três tipos de variantes:

a) Todos os carateres em minúsculas.
b) Todos os carateres com maiúsculas.
c) O primeiro caráter com maiúscula seguido apenas por minúsculas (Visitor).

O outro parâmetro de entrada é a pontuação original $score_{sd}$ que corresponde a *s*. A rotina primeiro examina cada ponto possível de divisão em *s* dividindo em $score_{izq}$ $score_{der}$ respectivamente (linhas 4 e 5). A decisão de qual é a melhor divisão baseia-se em: a) substrings que não tenham prefixos ou sufixos conhecidos (linha 6), b) a pontuação da divisão escolhida sobressaia do resto das pontuações (linhas 7 até 9).

Para esclarecer este ponto, para cada partição (esquerda ou direita) obtida calcula-se o score (linhas 4 e 5). Depois este é comparado como a pontuação da palavra original ($score_{sd}$) e a pontuação da palavra atual (score(s)). No principio ambas são iguais, mas à medida que avança a

```
Algoritmo 7: expançãoBasica
Entrada: abrev       // abreviaturas ao expandir
         wordList    //Palavras extraídas do código
         phraseList  //Frases extraídas do código
         stopList    //Palavras Excludentes
         dic         //Dicionário em Inglês
Saída: expansão      // abreviatura expandida
1 Se (abrev ∈ stopList) então
2  |_return null
3 listaExpanção ← [ ]
  // Procurar acrónimo
4 Para frase fra em phraseList fazer
5  | Se (abrev é um acrónimo de fra) então
6  |_ |_return fra
  // Procurar abreviatura comum
7 Para palavra w em wordList fazer
8  | Se (abrev é uma abreviatura de w) então
9  |_ |_return w
  // Procurar dicionário
10 listaCandidatos ←buscarDicionário(abrev,dic)
   listaExpanção.add(listaCandidatos)
11 unicaExp ← null
  //Se retorna o único resultado
12 Se (length(listaExpanção)=1) então
13 |_unicaExp ← listaExpanção[0]
14 return unicaExp
```

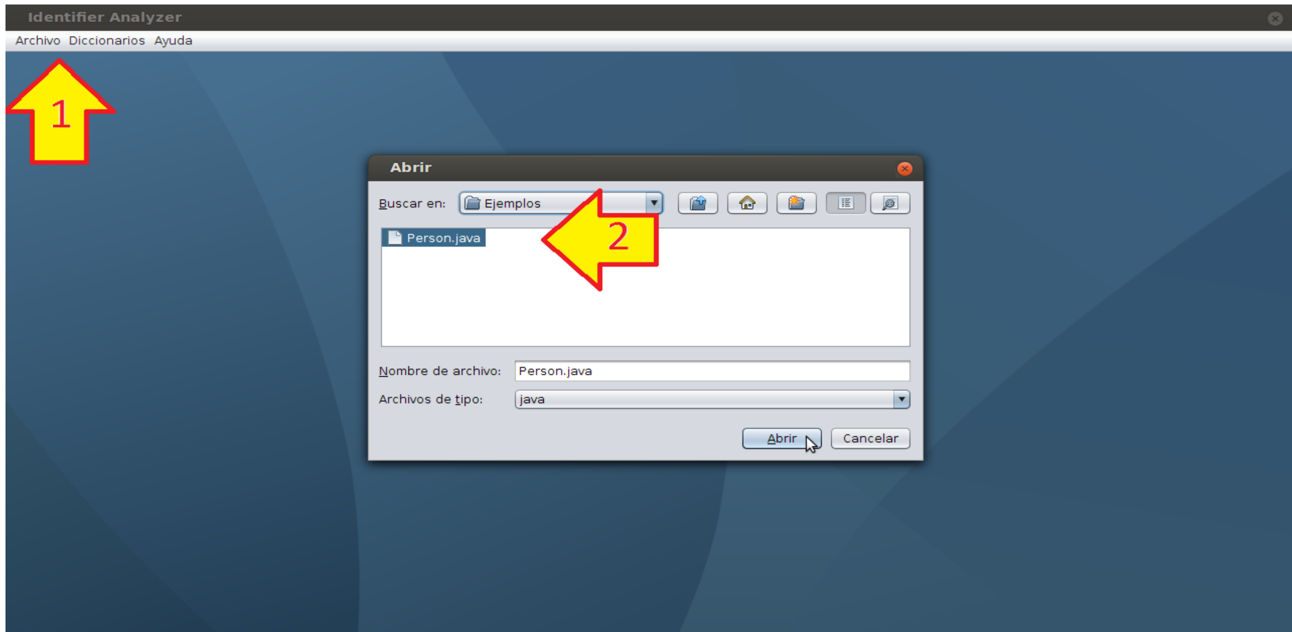

Figura 2 – Captura de uma écran do AId

recursão score(s) varia relativamente ao $score_{sd}$ (linhas 7 e 8). No caso de não ter prefixos e sufixos ordinários, se considera que a parte esquerda é um candidato. Por outro lado, a rotina é invocada recursivamente sobre a parte direita da cadeia para proceder a mais subdivisões (linha 14).

Se a parte direita finalmente se dividir, logo entra a parte esquerda e a direita também. Contudo, no caso que a parte direita não se dividir tão pouco deveria separar entre ambas as partes (o ***Se*** da linha 13 controla esta tarefa). As análises de dados [8] indicam que não se deve separar a atual posição, tendo em conta só a evidência na esquerda já que pode ser um erro; um caso de erro de divisão encontrado pelo próprio autor é string-ifed [8]. Este Id aparece muitas vezes no código fonte dos programas estudados pelo que o seu score é mais alto do que o resto. Por esta razão a raiz quadrada se utiliza no cálculo do resultado de score antes de comparar (linha 7 e 8), senão a divisão frequentemente seria errônea. Um exemplo é a palavra *performed*. Para que a técnica Samurai possa levar a cabo a tarefa de separação de instruções de controlo como é o caso do *se* (*if*) se necessita da função de scoring. Como foi explicado anteriormente esta função participa em duas decisões chave durante o processo de divisão: Na rotina *divisãoHardWord* para determinar se a divisão do Id é um caso camel-case ou não (linhas 7 e 8), na rotina *divisãoSoftWord* para pontuar as diferentes participações de substrings e escolher a melhor separação (linhas 4, 5, 7 e 8). Dada uma string *s*, a função score(s) indica i) a frequência de aparição de *s* no programa em análise e ii) a frequência num conjunto grande de programas predefinidos.

A fórmula é a seguinte:

$$Frec(s, p) + (globalFrec(s) / \log10 (totalFrec(p)))$$

Onde *p* é o programa em estudo, *Frec(s,p)* é a frequência de ocorrência de *s* em *p*. A função *totalFrec(p)* é a frequência total de todas as strings no programa *p*. A função *globalFrec(s)* é a frequência de ocorrência de *s* num conjunto grande de programas tomados como amostra [8].

3.3. Módulo de Expansão

O algoritmo de expansão de abreviaturas, idealizado por Lawrie, Feild, Binkley (os mesmos autores da técnica de separação Greedy) [7], trabalha com quatro listas para realizar a sua tarefa:

i) Uma lista de palavras (em linguagem natural) que se extraem do código fonte de muitos programas,
ii) Uma lista de frases (em linguagem natural) presentes também no código fonte referido,
iii) Uma lista de palavras a ignorar (stop words), e
iv) Uma lista de palavras dum dicionário em inglês.

A lista de palavras se constrói utilizando uma ferramenta que extrai as palavras dos comentários que se encontram dentro e fora dos métodos, incluindo aqueles que são relativos à classe de estudo. A lista de frases se constrói utilizando uma ferramenta que as extrai [11], os principais recursos são os comentários e os Ids multipalavras. Neste ponto se constrói um acrônimo com

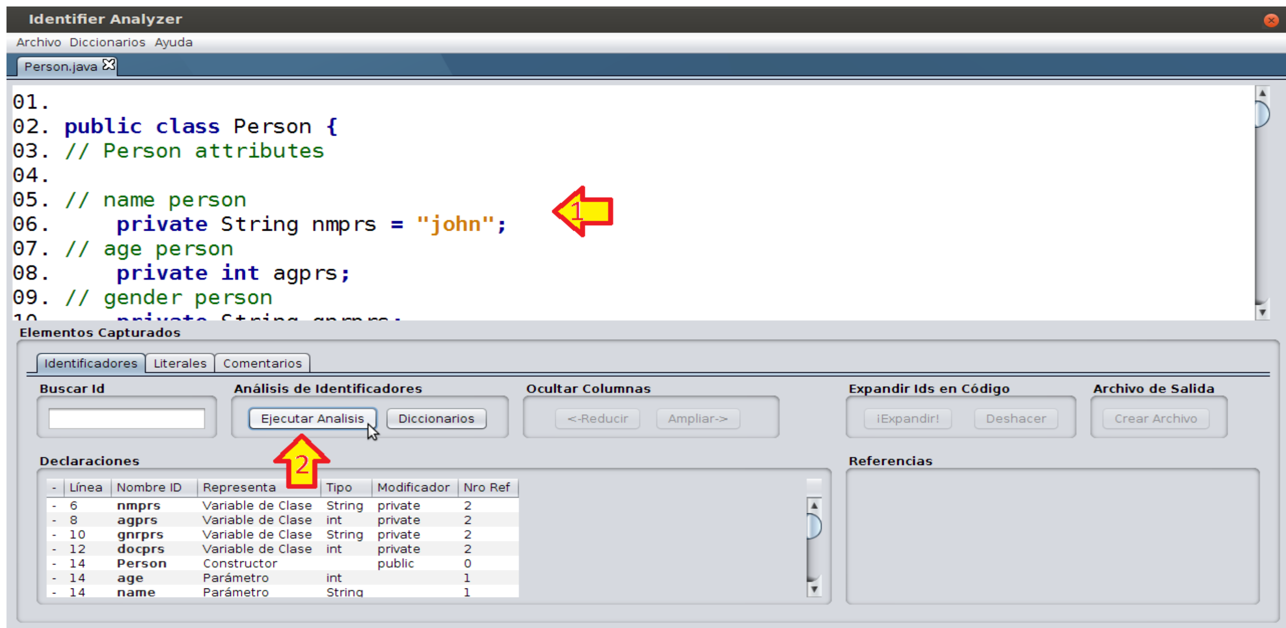

Figura 3 – Captura de um écran do AId

as palavras de alguma frase, se esse acrônimo coincide com algum dos Ids extraídos então essa frase se considera como potencial expansão [26, 27] (exemplo: a frase *file status* é uma expansão possível para o Id *fs_exists* → *file status exists*).

A lista de palavras irrelevantes e a lista de dicionários se constroem da mesma maneira que se faz no algoritmo Greedy (ver seção 3.2.1).

Quando as listas de palavras e frases potenciais estão criadas a execução do algoritmo se inicia. Este algoritmo (ver algoritmo 7) recebe como entrada a abreviatura a expandir e as quatro listas descritas previamente. O primeiro passo é ver se a abreviatura forma parte da lista de palavras irrelevantes (ou palavras a ignorar, linha 1). No caso que assim seja não se retornam resultados. A razão disto é porque estas palavras não aportam informação importante para a compreensão do código e são facilmente reconhecidas pelos engenheiros de software. Alguns casos são artigos/conectores (*the*, *an*, *or*) e palavras reservadas da linguagem de programação (*while*, *for*, *if*, etc.). Seguindo com a execução verifica-se se algumas das frases extraídas do código corresponde à abreviatura em forma de acrônimo (linha 5).

Em caso de não encontrar uma abreviatura se procura se as letras da abreviatura coincidem na mesma ordem com as letras duma palavra presente na lista de palavras recolhidas do código (linha 8). Exemplos: *horiz* → *horizontal*, *trgn* → *triangle*. No caso de não ter êxito, a procura continua, como último recurso, com o dicionário composto com palavras em inglês (linha 10). Esta técnica de expansão descrita só retorna uma única expansão potencial para uma abreviatura determinada; caso contrário não retorna coisa alguma (linhas 12 e 13). O motivo disto é porque não tem programado como decidir uma única opção entre várias alternativas de expansão. Esta funcionalidade é deixada, pelos autores, para trabalho futuro [7, 9].

4. Caso de Estudo

Nesta seção se descreve como funciona o AId aplicado a um caso de estudo, um programa Java de 600 linhas de código. Ao executar a ferramenta, o primeiro componente da interação é uma simples barra de menu localizada no topo da écran (ver figura 2). Ao fazer click no menu *Archivo* (ficheiro em português) da barra antes mencionada e logo em *Abrir Archivos(s) Java* (Abrir Ficheiro(s) Java) do menu suspenso mostra-se uma janela de seleção de ficheiros. Aqui o usuário pode escolher os ficheiros Java de entrada (na figura 2 o menu ficheiro está marcado com uma seta com um 1 e a janela de diálogo abrir com uma seta com um 2).

Uma vez escolhidos os ficheiros Java começa a execução do analisador sintático que extrai os Ids, comentários e literais. Ao finalizar a execução do analisador, aparecerá um painel que contem seções. Na parte de acima o código lido do ficheiro Java (ver seta com um 1 na figura 3). Na parte inferior, os elementos extraídos do código analisado.

Para proceder com a análise de Ids, se deve pulsar o botão *Ejecutar Análisis* (Executar Analise, ver seta com um 2 na figura 3) localizado no painel *Análisis de Identificadores* (Análise de Identificadores). Uma vez realizada essa tarefa se abre um novo painel; na parte inferior pode ver-se dos Ids recuperados (ver seta com um 1 na figura 4).

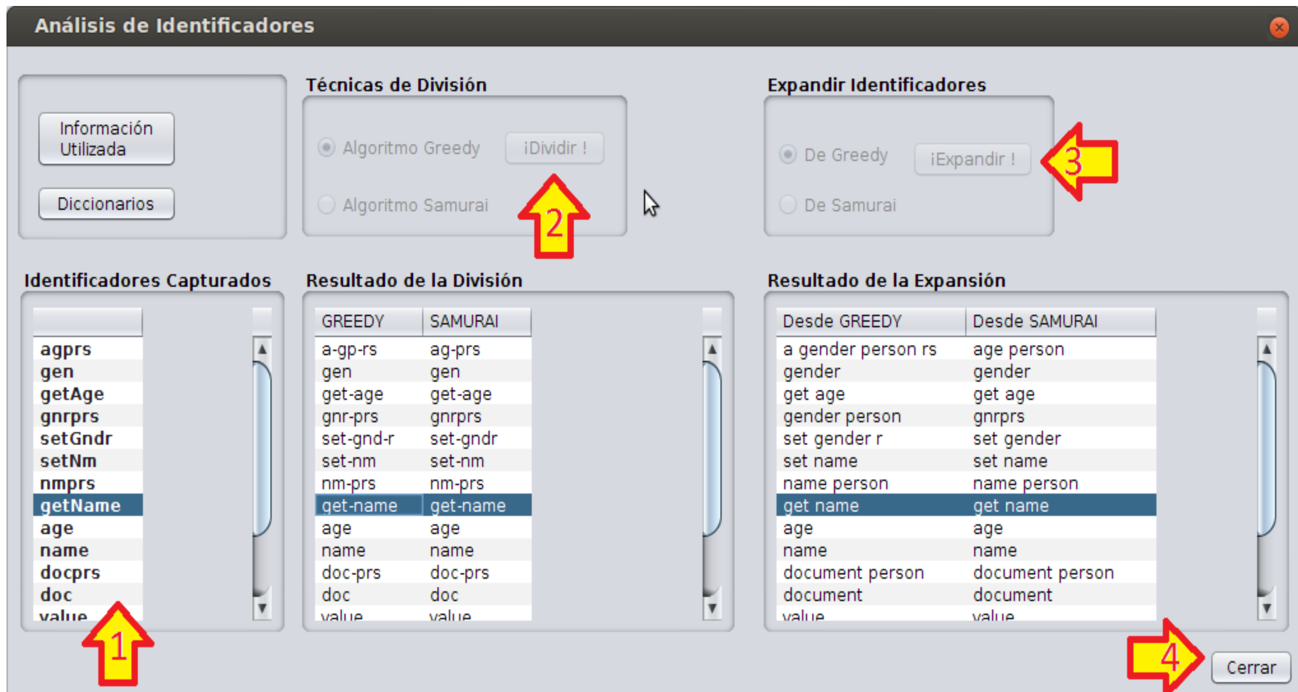

Figura 4 – Painel de Analise

No painel central superior se podem selecionar os algoritmos de divisão de Ids (Greedy e Samurai, ver seta com um 2 na figura 4), o botão *Dividir* do mesmo painel executa a técnica selecionada mostrando no painel inferior a tabela com os resultados da divisão (*Resultados de la División* na figura 4). Da mesma forma, os painéis subsequentes da direita permitem expandir as abreviaturas obtidas no painel de divisão (ver Figura 4). Pulsando o botão *Expandir* executa o algoritmo de expansão básico tomando como entrada os Ids divididos por Greedy ou Samurai, conforme o usuário decida (ver a seta com um 3 na figura 4). Os resultados se mostram no painel de abaixo. As três tabelas mostradas nos três painéis inferiores ajudam o usuário a comparar os resultados obtidos (ver figura 4).

Ao pulsar o botão *Cerrar* (Fechar em português e indicado pela seta com um 4 na figura 4), a janela do painel de análise se oculta e os resultados obtidos são carregados na tabela inferior do painel principal junto à informação que o analisador sintático havia recolhido (seta 1 na figura 5).

Na última coluna, se podem apreciar múltiplas caixas de seleção. Essas caixas permitem ao usuário escolher a expansão produzida pelo Greedy ou pelo Samurai (ver seta com um 1 na figura 5).

Uma vez concluída esta seleção, se procede a utilizar o painel *Expandir Id en Código* (seta 2 na figura 5). Este painel tem dois botões: *Expandir* e *Deshacer*. O primeiro leva a cabo a substituição dos Ids no código de acordo com o escolhido nas caixas de verificação mostrando o código resultante no painel de acima.

No caso de necessitar desfazer esta ação de substituição se pode pulsar o botão *Deshacer* (Desfazer em português) e restabelecer o código original. Para finalizar, ao pulsar o botão *Crear Archivo* (Criar Ficheiro em português indicado na seta 3 da figura 5) cria-se um novo ficheiro com o mesmo código porém com todos os Ids expandidos.

No exemplo mostrado anteriormente, o AId recebeu como entrada um programa Java de 600 linhas de código. Na tabela 1 se podem observar alguns identificadores analisados. Nas colunas *Algoritmo Greedy* e *Algoritmo Samurai* se podem apreciar os resultados de ter executado as técnicas de divisão. Logo nas próximas duas se mostra o resultado de haver aplicado o algoritmo de expansão básico a cada resultado dos algoritmos de divisão. Como se observa, a maioria dos casos de sucesso foram obtidos com o Samurai. Como observação importante se pode dizer que o Greedy consulta um grande volume de dados no dicionário predefinido e que

Identificador	Algoritmo Greedy	Algoritmo Samurai	Expansão do Greedy	Expansão do Samurai
nmprs	nm-prs	nm-prs	name person ✓	name person ✓
agprs	a-gp-rs	ag-prs	a gender person rs ✗	age person ✓
gnrprs	gnr-prs	gnrprs	gender person ✓	gnrprs ✗
docprs	doc-prs	doc-prs	document person ✓	document person ✓
age	age	age	age ✓	age ✓
setDocPrs	set-doc-prs	set-doc-prs	set document person ✓	set document person ✓
setGndr	set-gnd-r	set-gndr	set gender r ✗	set gender ✓

Tabela 1 – Resultado parcial do Analise

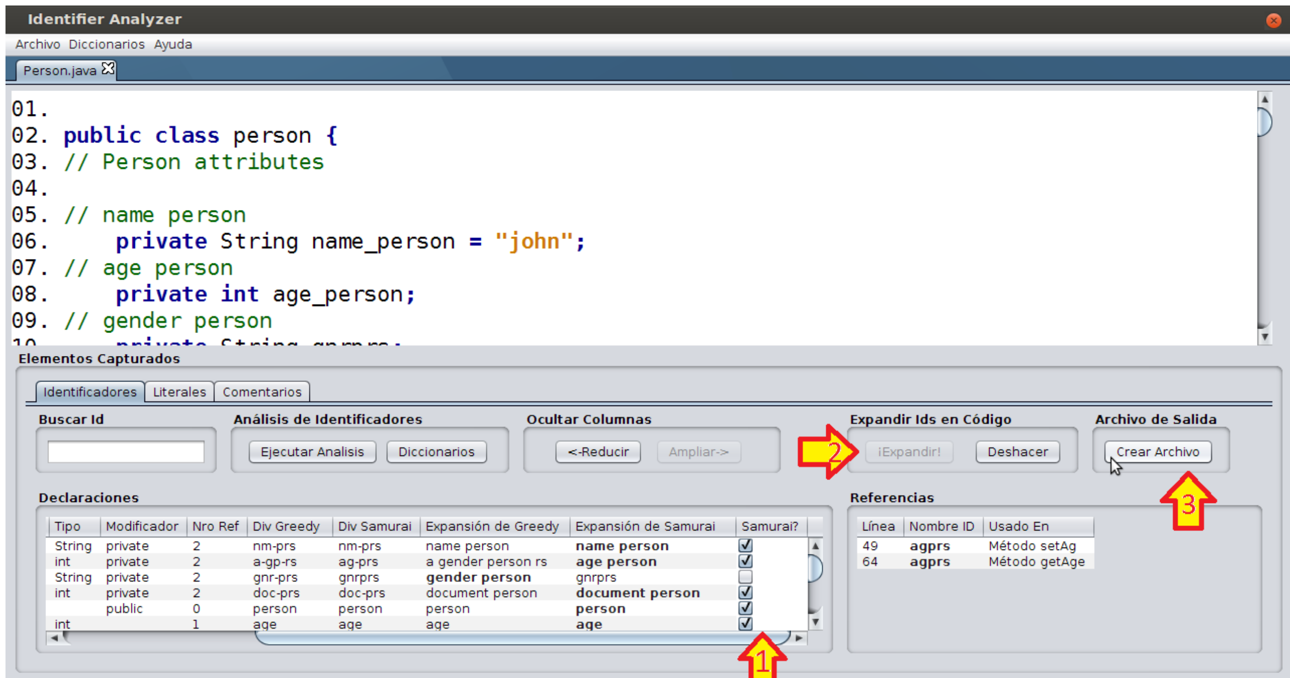

Figura 5 – Vista principal do Aid com foco na expansão de identificadores

o Samurai baseia-se nas tabelas de frequências com os dados próprios do código. O qual indica que este último deve ser mais preciso.

Seguindo com o caso de estudo, na parte superior da figura 6 pode-se observar uma parte do código em análise (dois métodos) com os identificadores não expandidos. Na parte inferior da figura 6, mostra-se o mesmo fragmento do programa depois de aplicar os algoritmos de expansão, ou seja, com os identificadores expandidos (ressaltados com cor amarelo). É evidente que a última versão dá mais informação semântica e, por conseguinte, torna mais fácil perceber o que faz essa parte do código.

5. Conclusão e Futuras Extensões

A motivação para desenvolver a ferramenta AId baseia-se na ausência de ferramentas com características similares. Não abundam ferramentas que possuam uma interface amigável com o usuário e executem técnicas de Análise de Identificadores (Ids) dum código recebido como entrada. Tão pouco foi possível encontrar muitas implementações que integrem extração, divisão, e expansão de abreviaturas de Ids. O uso do parser construído com ANTLR possibilitou extrair com facilidade os elementos estáticos presentes no código que são necessários para os algoritmos que analisam Ids. Estes elementos são os Ids como objetos principais, comentários, literais e documentação Java Doc. Cabe destacar que a ferramenta AId tem implementadas duas técnicas de divisão que são o Greedy e o Samurai. A primeira necessita consultar um dicionário de palavras em inglês e uma lista genérica de abreviaturas conhecidas para levar a cabo as suas tarefas. Ambas as estruturas ocupam muito espaço de armazenamento pelo que se utiliza uma base de dados para fazer as consultas mais eficientes [10, 5, 6, 7]. A segunda alternativa, o algoritmo Samurai [8], divide os Ids mediante a utilização de recursos específicos extraídos do código. Estes recursos são os comentários, os literais e documentação Java Doc. Com estes recursos, se constrói um vetor com as frequências de ocorrência das palavras, vetor esse que é depois usado na função de scoring (ver seção 3.2.2). Todavia é possível que estes recursos sejam escassos. Por esta razão os autores decidiram construir uma lista de palavras pertencente a um conjunto amplo de programas escritos em Java. Este vetor, não só ocupa menos espaço que os dicionários genéricos de Greedy como também é mais eficaz visto ser formado por palavras mais

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

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

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

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

Figura 6 – Comparação do Código em estudo, antes e depois da expansão de Identificadores

adequadas no âmbito da engenharia do software. Isto implica que a divisão seja mais eficiente e, por conseguinte, que a expansão seja mais precisa. Por outro lado, o algoritmo de expansão básico usa os mesmos dicionários de palavras que utiliza o Greedy, mas com a diferença que consulta previamente a lista de frases capturadas do código, dando a preferencia a esta lista. É importante mencionar que AId incorpora na lista de frases não só os comentários como também as strings literais sendo esta uma caraterística não encontrada, no estado da arte, nas ferramentas atuais. Este algoritmo tem o problema que ante as múltiplas alternativas de expansão, não sabe escolher uma única opção. Como trabalho futuro se planeia implementar novas técnicas de análise de Ids. Uma delas é o AMAP (Automatically Mining Abbreviation Expansions in Programs) [9]. Esta técnica não necessita de dicionários com palavras em inglês, como é o caso do algoritmo básico de expansão; os seus proponentes observam gradualmente no código os comentários e literais presentes partindo desde o lugar do Id que se deseja expandir. Também resolve o problema que possui o algoritmo básico quando não conhece que opção escolher ante muitas opções de expansão. Para levar a cabo a tarefa atrás mencionada, o algoritmo prioriza a frequência de aparição das palavras. Isto se faz partindo do lugar onde se encontre o Id analisado. Também AMAP permite treinar com um conjunto de programas recebidos como entrada para recolher mais palavras e melhorar ainda mais a precisão da expansão. Outra melhora futura para a ferramenta AId é a possibilidade de construir um plugin para o NetBeans ou Eclipse. Isto permitirá que o usuário abra um projeto Java e imediatamente com Aid expanda os Ids para melhorar a compreensão. Também se pretende, nos próximos passos da investigação, estudar e propor novas técnicas de análise de identificadores e fazer um estudo profundo das suas vantagens, no que respeita à compreensão, para um conjunto significativo de sistemas.

6. Referências

[1] Florian DeiBenbock and Markus Pizka. Concise and Consistent Naming. In IWPC, page 97-106. IEEE Computer Society, 2005.

[2] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. Empirical Software Engineering, 18(2):219-276, 2013.

[3] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic Identifier Conciseness and Consistency. In SCAM, page 139-148. IEEE Computer Society, 2006.

[4] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. In Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, page 3-12, June 2006.

[5] Henry Feild, David Binkley, and Dawn Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In Proceedings of IASTED International Conference on Software Engineering and Applications, 2006.

[6] Binkley D. Feild, H. and D. Lawrie. Identifier splitting: A study of two techniques. In Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems, 2006.

[7] D. Lawrie, H. Feild, and D. Binkley. Extracting Meaning from Abbreviated Identifiers. In Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on, page 213-222, Sept 2007.

[8] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In 6th IEEE International Working Conference on Mining Software Repositories, page 71-80. IEEE, may. 2009.

[9] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. In Proceedings of the 5th Int'l Working Conf. on Mining Software Repositories., page 79-88. ACM, 2008.

[10] Henry Field, Dawn Lawrie and David Binkley. Quantifying identifier quality: an analysis of trends. Kluwer Academic Publishers-Plenum Publishers. 2006.

[11] Fangfang Feng and W. Bruce Croft. Probabilistic techniques for phrase extraction. Inf. Process. Manage., 37(2):199-220, 2001.

[12] Bruno Caprile and Paolo Tonella. Nomen est omen: analyzing the language of function identifiers. In Proc. Sixth Working Conf. on Reverse Engineering, page 112-122. IEEE, October 1999.

[13] José Luís Freitas, Daniela da Cruz, and Pedro Rangel Henriques. The role of comments on program comprehension. In INForum.

[14] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In Proc. Int'l Conf. on Software Maintenance, page 97-107. IEEE, 2000.

[15] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In ICSE - Future of SE Track, page 73-87, 2000.

[16] Roger S. Pressman. Software Engineering - A Practitioner's Approach. McGraw-Hill, 5 edition, 2001.

[17] P.F. Tiako. Maintenance in joint software development. In Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International, page 1077-1080, 2002.

[18] A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. Computer, 28(8):44-55, 1995.

[19] M. Beron, P. Henriques, and R. Uzal. Inspección de Programas para Interconectar las Vistas Comportamental y Operacional para la Comprensión de Programas. PhD thesis, Universidade do Minho, Braga. Portugal, 2010.

[20] M. Storey. Theories, methods and tools in program comprehension: past, present and future. In Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on, page 181-191, May 2005.

[21] Michael P. O'Brien. Software comprehension - a review and research direction. 2003.

[22] Mario Beron, Pedro R. Henriques, Maria J. Varanda, and Roberto Uzal. Program inspection to inter-connect the operational and behavioral views for program comprehension. 2007.

[23] David W Embley. Toward semantic understanding: an approach based on information extraction ontologies. In Proceedings of the 15th Australasian database conference-Volume 27, pages 3-12. Australian Computer Society, Inc., 2004.

[24] Dawn Lawrie, Henry Feild, and David Binkley. An empirical study of rules for well-formed identifiers. Journal of Software Maintenance and Evolution: Research and Practice, 19(4):205-229, 2007.

[25] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. IEEE Trans. Soft. Eng., 28(10):970–983, 2002.

[26] L. S. Larkey, P. Ogilvie, M. A. Price, and B. Tamilio. Acrophile: an automated acronym extractor and server. In Proc. Conf. Digital Libraries, 2000.

[27] S. Pakhomov. Semi-supervised maximum entropy based approach to acronym and abbreviation normalization in medical texts. In Proc. Association for Computational Linguistics, 2001.

[28] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers principles, techniques and tools. 2006.

[29] Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. Data structures and algorithms / Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Addison-Wesley Reading, Mass, 1983.

[30] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In ICSM, page 602-611, 2001.

[31] Albanes, J.L., Berón M., Henriques P. and Pereira MJ. In WICC 2011. Estrategias para relacionar el dominio del problema con el dominio del programa para la CP, 2011.