UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MAPEO DE EJEMPLOS DE CÓDIGO FUENTE PARA DAR APOYO EN EL USO DE APIS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN
TECNOLOGÍAS DE LA INFORMACIÓN

YASETT GISELA ACURANA FLORES

PROFESOR GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
ÉRIC TANTER
DIEGO ARROYUELO BILLIARDI

SANTIAGO DE CHILE
2017

RESUMEN

Los desarrolladores de software con frecuencia recurren a Interfaces de Programación de Aplicaciones (APIs) para extender la funcionalidad de sus programas. El uso de APIs, que son un conjunto de reglas y convenciones mediante las que un programa puede comunicarse con otro, puede ocasionar defectos en el código fuente, como por ejemplo: defectos funcionales y/o de rendimiento.

Cuando un desarrollador desconoce el uso de una API, desea aprender más sobre su uso, o su código fuente no funciona como espera, busca manualmente ejemplos de la funcionalidad provista por la API. Esta tarea puede consumir mucho tiempo y ser propensa a errores. Por ejemplo, cuando inserta en su código la invocación a un método sin conocer bien los parámetros que debe enviar y luego el método no retorna el resultado esperado.

En la presente tesis se propone apoyar a los desarrolladores mediante la sugerencia de ejemplos de buen uso de las APIs. Los ejemplos son presentados en orden de relevancia, de acuerdo al código que están escribiendo los desarrolladores que usan la API. La implementación de esta solución consiste en la construcción y uso de un repositorio de ejemplos de código fuente, junto a un plug-in creado para el entorno de desarrollo Eclipse. El plug-in realiza la búsqueda de ejemplos del repositorio, muestra los ejemplos por orden de mayor a menor similitud y permite integrar el código fuente de un ejemplo en el editor de código fuente.

La utilidad de la herramienta ha sido validada por medio de un estudio con usuarios, donde se evaluó que el plug-in ayuda a desarrolladores con poco conocimiento de una API a hacer un mejor y más rápido uso de éstas. A los desarrolladores se les asignó dos tareas de programación para completar, una de ellas utilizando el plug-in y la otra mediante la búsqueda de ejemplos en Internet. Como resultado, se encontró que los desarrolladores terminaron las tareas hasta un 66% más rápido cuando usaron el plug-in, respecto de la búsqueda en Internet, y en su código fuente no se encontraron casos de mal uso de las APIs.

En base a los resultados obtenidos, se concluye que, pese a que la técnica planteada tiene sus limitaciones, se pueden obtener buenos resultados con la solución propuesta en la medida que el repositorio contenga los ejemplos que el desarrollador necesita. Como uno de los siguientes pasos se considera importante lograr una mejor precisión en los resultados de la búsqueda de los ejemplos, para que esta solución sea de mayor utilidad a los desarrolladores.

Dedicado a mi familia

Por apoyarme en todas mis aventuras,
especialmente ésta.

Agradecimientos

En este camino de dos años me han acompañado varias personas, gracias a ellos el Magíster en Tecnologías de Información ha sido una de las mejores experiencias de mi vida. Intentaré nombrar a la mayoría, sin extenderme demasiado.

Primero, quiero agradecer a mis padres y mi hermano porque, aun estando lejos, han estado conmigo en espíritu. Su fe en mí ha sido determinante para poder lograr mis metas.

A Juan Pablo Sandoval, por ser mi inspiración, mi compañero y principal apoyo. Cuando pensé que estaba sola, él estaba ahí, por eso mis éxitos son también suyos.

A Jocelyn Simmonds, por sus valiosos consejos y guía en éste y todos los proyectos que he emprendido. Le agradezco por su dedicación en el proyecto, que sin su participación no podría haberse completado. Es una de las personas que más admiro, ha sido un honor poder intercambiar ideas y aprender de ella en esta etapa.

A los miembros de mi Comité Académico, por el tiempo y esfuerzo invertidos en la revisión de este proyecto.

A Pere Casafont, por su apoyo en la implementación de la parte técnica. La experiencia y habilidad que puso fueron un aporte importante para el proyecto. Agradezco también a la empresa Toptal por permitirme contar con su ayuda.

A Leslie Toloza y Alejandro Gamboa por el apoyo moral. Ellos me dieron el impulso necesario para concluir la última parte del Magíster y además me enseñaron a ser una mejor persona.

Tabla de Contenido

Índice de Ilustraciones

Índice de Listados

Capítulo 1- Introducción

1.1 Contexto

Los desarrolladores de software normalmente utilizan Interfaces de Programación de Aplicaciones (APIs) durante la construcción de sus programas. La cantidad de las APIs se va incrementando a medida que pasa el tiempo y evolucionan las tecnologías, por lo que los desarrolladores se ven con frecuencia en la necesidad de tener que aprender a utilizar APIs que no conocen [1].

La complejidad y tamaño de las APIs también ha ido incrementando, lo que hace más difícil el aprender a utilizarlas apropiadamente. Dado que al utilizar una API se busca mejorar la calidad del código y agilizar el desarrollo de software, no es deseable que estas ventajas se vean anuladas por las dificultades que se presentan al utilizar las APIs.

Distintos estudios se han llevado a cabo para investigar qué dificultades tienen los desarrolladores al utilizar APIs que no conocen y cuáles son las causas de esas dificultades. Entre ellos se encuentra el estudio sobre los problemas que enfrentan los desarrolladores al aprender a usar las APIs [2]. Este estudio fue conducido por Robillard, que reunió un grupo de 83 desarrolladores de software que estaban trabajando en Microsoft. El estudio fue realizado a partir de una encuesta sobre los obstáculos que los desarrolladores enfrentaron al aprender a utilizar las APIs y las estrategias que siguieron para aprender. Los resultados de la encuesta mostraron que el 74% de los desarrolladores mencionaron haber tenido algún obstáculo con el aprendizaje de la API. Los obstáculos fueron categorizados y se identificó que los de mayor ocurrencia son: ejemplos de código insuficientes o inadecuados, el diseño de la estructura de la API y la falta de documentación. En cuanto a las estrategias, las más aplicadas por los encuestados fueron la lectura de documentación (78%) y el uso de ejemplos de código fuente (55%). Otras estrategias mencionadas en menor proporción son: la experimentación directa con la API (34%), la lectura de artículos (30%) y consulta a compañeros de trabajo (29%).

Estas observaciones tienen relevancia para los desarrolladores de APIs pero también para los creadores de herramientas de desarrollo, ya que demuestran que existe una brecha entre los usuarios de las APIs y las fuentes de información sobre las mismas, como ejemplos de código fuente o documentación.

Otro estudio [1], realizado por Duala-Ekoko y Robillard, fue realizado con 20 personas que tenían entre 1 y 6 años de experiencia en Java. A estas personas se les pidió completar dos tareas para las que se requería utilizar APIs en las que no tenían experiencia previa. Como resultado del estudio, se registraron las preguntas que los desarrolladores se plantearon al utilizar las APIs, por ejemplo: *¿Qué paquetes o espacios de nombre de la API proveen tipos que son relevantes para la tarea que estoy realizando?* Entre estas

preguntas, los desarrolladores encontraron mayor dificultad para responder las siguientes:

- ¿Qué palabras describen mejor una funcionalidad que provee la API?
- ¿Cómo se relaciona el tipo X con el tipo Y?
- ¿Provee la API un tipo específico para manipular objetos del tipo X?
- ¿Cómo puedo crear un objeto de un determinado tipo sin un constructor público?
- ¿Cómo determinar el resultado de la llamada a un método?

Por lo tanto, como conclusión del estudio se recomienda que las dificultades encontradas sirvan como base para evaluar las herramientas que existen en la actualidad e identificar áreas donde falta apoyo.

1.2 Problema a resolver

Un desarrollador que no está familiarizado con una API, desconoce o ha olvidado la funcionalidad correcta de los métodos, puede experimentar dificultades para programar. Además, un desarrollador que va a utilizar una API después de un tiempo, puede encontrar que en la versión más reciente se han realizado cambios y el uso recomendado ya no es el mismo que el de la versión que conocía.

Como consecuencia, el desarrollador que desconoce el uso de una API, tiende a buscar ejemplos en Internet. Esta tarea significa a los desarrolladores un gasto de tiempo considerable, dado que típicamente el desarrollador se toma el tiempo de revisar varios de los resultados de su búsqueda y juzgar la calidad de los ejemplos encontrados [3].

Esta situación no sólo se produce con desarrolladores que utilizan una API por primera vez. Desarrolladores que se consideran expertos también hacen uso de los ejemplos de código que encuentran en Internet, pues aunque hayan programado un bloque de código cientos de veces y entienden su funcionamiento, prefieren copiarlo y pegarlo antes que aprenderlo de memoria [4].

Al integrar ejemplos en su código fuente, el desarrollador puede introducir defectos funcionales [5] y de rendimiento [6], en especial si no comprende por completo la funcionalidad de los métodos que está llamando de la API o los parámetros que debe enviar.

Este problema es relevante porque, según reportan varios estudios empíricos [2][7], la falta de ejemplos es uno de los mayores obstáculos que enfrentan los desarrolladores al utilizar APIs.

1.3 Descripción de la solución

Como solución al problema, se propone construir un repositorio con ejemplos de buen uso de las APIs y un plug-in para el entorno de desarrollo Eclipse, que tome como base de conocimiento el repositorio y busque en éste los ejemplos más parecidos al código que está

escribiendo el desarrollador. La búsqueda de ejemplos se realiza mediante la comparación entre el código fuente del desarrollador con los ejemplos de uso de APIs. La comparación entre el código del desarrollador y un ejemplo del repositorio se realiza a partir de sus árboles de sintaxis abstracta (AST)[8]. En caso de que existan similitudes entre ambos árboles, se busca si en el código del desarrollador falta una invocación a un método, un parámetro en la instanciación de un objeto o una estructura de control. Como resultado, el plug-in muestra al desarrollador el(los) ejemplo(s) similares al código del desarrollador y las invocaciones a métodos, parámetros o estructuras de control que probablemente faltan en su código.

Parte de la tesis es la construcción de ejemplos de uso de APIs en un repositorio. Pese a que existen varios repositorios de ejemplos de código fuente, se propone construir un repositorio propio del proyecto, donde los ejemplos de código tengan un formato específico, adecuado para que sirvan de entrada al algoritmo de comparación entre ejemplos de código y el código del desarrollador. Por esta razón, antes de construir los ejemplos, se ha definido el formato que deben seguir. El repositorio es de acceso público y está documentado para que otros desarrolladores puedan utilizarlo o contribuir con nuevos ejemplos.

Tanto las APIs como el plug-in están desarrollados en el lenguaje de programación Java. Esto debido a que continúa siendo uno de los lenguajes más populares [9], lo que implica que la solución será de potencial beneficio para más desarrolladores.

El plug-in será desarrollado para el entorno de desarrollo Eclipse. Se ha elegido este entorno de desarrollo porque su arquitectura facilita el desarrollo de plug-ins y es uno de los entornos más conocidos para programar en el lenguaje Java [10].

Dentro del alcance de este proyecto, se agregó al repositorio una muestra representativa de ejemplos de código de dos APIs: Java NIO [11] y Google Guava [12]. Java NIO es una API que ofrece funcionalidad para operaciones intensivas de entrada y salida, mientras que Google Guava es una API con varios métodos que realizan tareas comúnmente útiles como: manejar colecciones, almacenar objetos en caché, hashing, entre otras.

1.4 Objetivos

El objetivo general es mejorar el uso de las APIs que hacen desarrolladores que no están familiarizados con ellas, mediante sugerencias de ejemplos de uso adecuado, logrando entregar software de mejor calidad.

Los objetivos específicos que permitirán cumplir con el objetivo general son:

- Construir un repositorio de ejemplos de uso de APIs, validado por expertos.
- Implementar un plug-in para el entorno de desarrollo de software Eclipse. El plug-in mostrará los ejemplos de código que sean aplicables a los métodos que está construyendo el desarrollador y facilitará la inserción de estos ejemplos en su código fuente.

- Validar la utilidad del plug-in, en términos de reducción de tiempo y cantidad de defectos, mediante la ejecución de experimentos con un grupo de desarrolladores de software.

1.5 Metodología

El desarrollo de la tesis inicia con un estudio del estado del arte sobre las herramientas, técnicas y métodos que han sido empleados en la sugerencia de código fuente al desarrollador. El objetivo de este estudio es conocer las estrategias utilizadas por otras herramientas y diferenciar el enfoque planteado en la presente propuesta.

En base a la estrategia que se ha definido para obtener y mostrar los ejemplos de código, se procede a diseñar un plug-in que realiza la comparación entre el código del desarrollador y los ejemplos, muestra los ejemplos más parecidos al código del desarrollador y facilita la inserción de los ejemplos en el editor de código fuente.

La implementación del plug-in se ha realizado en iteraciones, cuyos entregables son: el plug-in implementado y el repositorio de ejemplos, junto con su documentación.

Una vez implementado el plug-in, la validación se realiza mediante un experimento controlado con un grupo de desarrolladores de software, a quienes se ha instruido en el uso del plug-in implementado y se ha evaluado su desempeño, en comparación con la búsqueda de ejemplos en Internet. La comparación del rendimiento de ambos casos se realiza en función del tiempo que les tomó completar tareas de programación y la cantidad de potenciales defectos presentes en su código.

A partir de los resultados de la validación, se plantea una discusión sobre la utilidad de la herramienta, los casos donde es más útil, las limitaciones y las mejoras que se pueden realizar a futuro.

1.6 Estructura de la Tesis

En el siguiente capítulo se presenta el marco teórico y los trabajos relacionados. El Capítulo 3 describe el diseño de la solución propuesta. Luego, el Capítulo 4 explica detalles sobre la implementación. El Capítulo 5 expone la estrategia de validación de la solución y el análisis de los resultados obtenidos. Finalmente el Capítulo 6 lista las conclusiones del trabajo y abre una discusión sobre el trabajo futuro.

Capítulo 2 - Marco Teórico y Trabajos Relacionados

En este capítulo se presenta el marco teórico y trabajos relacionados con la tesis. La finalidad es lograr un mejor entendimiento de los conceptos necesarios para entender la solución propuesta.

Primero se describen los conceptos relacionados con programación: API, snippet y árbol de sintaxis abstracta (AST). A continuación se explican conceptos relacionados con el despliegue y almacenamiento de aplicaciones: repositorio de ejemplos de código fuente y plug-in.

Finalmente se presenta una revisión del trabajo relacionado, con otros enfoques para mostrar sugerencias de código fuente al desarrollador. En esta parte se hace énfasis en las diferencias que tienen estos enfoques con el propuesto y sus limitaciones, mismas que el presente proyecto pretende superar.

2.1 Conceptos de programación

2.1.1 Interfaz de Programación de Aplicación (API)

Una Interfaz de Programación de Aplicaciones (API) es un conjunto de reglas y convenciones mediante las que un programa puede comunicarse con otro [13]. Sirve como una interfaz entre programas y permite la interacción entre ellos.

La API está definida a nivel de código fuente y provee un nivel de abstracción a sus consumidores. La especificación de una API puede tomar varias formas, pero a menudo incluye rutinas, estructuras de datos, clases, variables o llamadas remotas.

El propósito de una API es facilitar al desarrollador el uso de ciertas tecnologías en la construcción de aplicaciones. Al abstraer detalles de implementación y exponer sólo los componentes que el desarrollador necesita, las APIs reducen la carga del desarrollador.

La documentación de la API es un recurso crucial para desarrollar aplicaciones que utilicen la API. Típicamente contiene especificaciones de los tipos de datos y funciones, ejemplos de cómo usar las funcionalidades y las restricciones propias de la API.

Escribir una buena documentación es una tarea que requiere tiempo y esfuerzo. Asimismo, es importante mantenerla actualizada. Muchos equipos de desarrollo cometen el error de esperar a después de que se ha lanzado una nueva versión de la aplicación para actualizar la documentación.

2.1.2 Ejemplo de código fuente (Snippet)

Es un bloque de código fuente reutilizable, diseñado para ilustrar una tarea específica. Generalmente contienen bloques de código de uso común, como los bloques if-else o try-catch-finally, pero también pueden contener clases o métodos completos. Por ejemplo, el siguiente snippet [14] muestra el uso del objeto `ObjectOutputStream`:

```
1  FileOutputStream fileOutputStream = new FileOutputStream(file);
2
3  ObjectOutputStream objectOutputStream = new
4  ObjectOutputStream(fileOutputStream);
5
6  try {
7        objectOutputStream.write("Test");
8  } finally {
9        objectOutputStream.close();
10 }
```

Listado 1: Ejemplo de Uso del Objeto ObjectOutputStream

Este ejemplo ilustra a los desarrolladores sobre la creación del objeto: qué métodos pueden ser invocados y en qué orden se sugiere la invocación, de manera que se garantice un uso adecuado.

Los snippets proveen una manera sencilla de insertar bloques de código comúnmente usados dentro de bloques más grandes. En lugar de escribir el mismo código una y otra vez, el desarrollador puede guardar un snippet y reutilizarlo cuando lo necesite.

2.1.3 Árboles de sintaxis abstracta

Un árbol de sintaxis abstracta (AST) es la representación a manera de árbol de la estructura sintáctica del código fuente. Cada uno de los nodos del árbol denota una construcción en el código fuente. La sintaxis se denomina abstracta porque no representa cada detalle presente en la sintaxis real; por ejemplo, elementos como llaves o comas son considerados de menor importancia y por lo tanto no están en el árbol.

El AST es la estructura utilizada por el compilador, que realiza un número de verificaciones y luego genera la tabla de símbolos. En muchos lenguajes, primero el analizador léxico se encarga de obtener una lista de tokens del código fuente, luego el intérprete añade las relaciones entre los tokens, dando lugar al árbol. Por ejemplo, el código siguiente:

```
1  public void sum()
2     int result = 0;
3     while (result < 10){
4        result = result + 1;
5     }
6  }
```

Listado 2: Ejemplo de Código Fuente para Generar AST

Resulta en el árbol de la Figura 1:

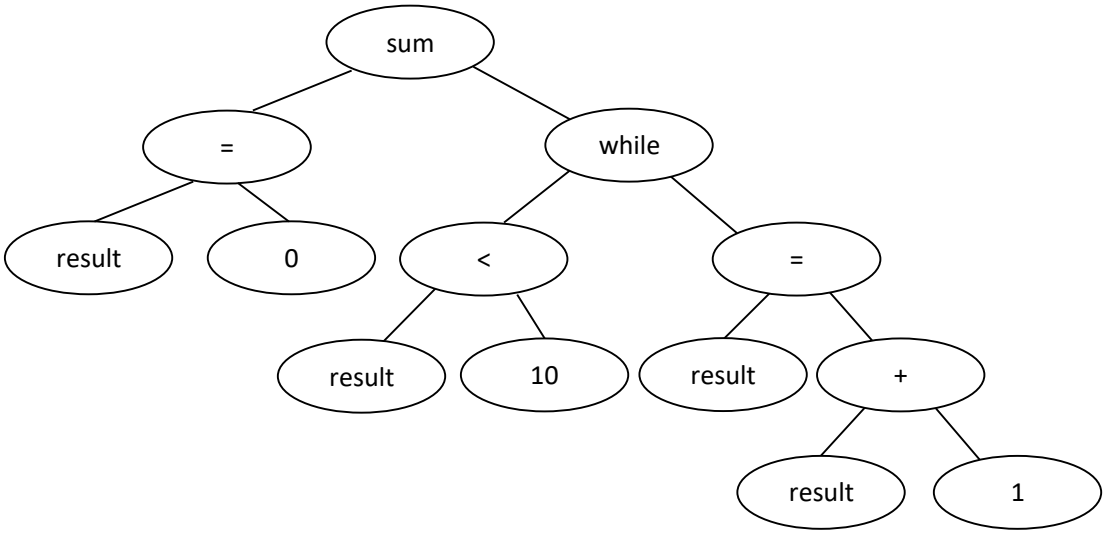

Figura 1: Árbol de Sintaxis Abstracta.

El AST siempre tendrá un único nodo raíz. Todos los nodos, excepto éste, tienen un sólo nodo padre. Además, todo nodo tiene una secuencia de hijos.

2.2 Conceptos de despliegue y almacenamiento de aplicaciones

2.2.1 Repositorios de ejemplos de código fuente

En el campo de desarrollo de software, un repositorio de ejemplos de código fuente es una ubicación donde se almacenan snippets de código, de manera que puedan ser gestionados desde un solo lugar y permitan un fácil acceso. Estos repositorios favorecen la colaboración entre desarrolladores, ya que les permite guardar código fuente que otros pueden reutilizar.

En la actualidad existen varias plataformas en línea donde es posible alojar código fuente y repositorios de ejemplos del uso de las APIs, por ejemplo, el repositorio de ejemplos de código que utiliza el plug-in de Eclipse, Snipmatch, que está alojado en la plataforma GitHub [15]. Otros repositorios son exclusivos para ejemplos de

código fuente, como los repositorios en línea Snippet Repo [16] y Smipple [17], ambos con miles de ejemplos en distintos lenguajes.

2.2.2 Plug-in

Un plug-in es un componente de software que añade una funcionalidad específica a un programa informático existente. Por ejemplo, los navegadores de Internet, que permiten a los usuarios instalar plug-ins para realizar tareas para las que por defecto no están programados, como mostrar el contenido de un archivo PDF o gestionar las contraseñas de los sitios a los que se accede.

Un programa puede soportar la instalación de plug-ins por diferentes motivos: para ampliar las capacidades de la aplicación, para reducir el tamaño inicial (de distribución), para permitir que otros desarrolladores creen sus propios plug-ins y puedan extender la utilidad de la aplicación.

Además de los navegadores, existen muchos tipos de programas que típicamente soportan plug-ins, como: clientes de email, reproductores de multimedia, editores de audio o editores de código fuente. En el caso de los editores de código fuente, los plug-ins permiten mejorar el desarrollo de software o trabajar con distintos lenguajes de programación. Tal es el caso de Eclipse, que tiene una amplia galería de plug-ins, para realizar tareas como: verificar los estándares de codificación, elaborar diagramas UML, conectarse a repositorios de código fuente, entre otras.

2.3 Trabajo relacionado: herramientas de sugerencia de código fuente

Code completion es una característica de los entornos de desarrollo que consiste en la predicción del resto de la palabra que el desarrollador está escribiendo. Por ejemplo, si el desarrollador escribe `Sys` en el editor de texto, el entorno de desarrollo presenta una lista de sugerencias como: `System`, `SystemColor`, `SystemException`, `systrace`, entre muchas otras. Si la palabra a escribir está en la lista de sugerencias, el desarrollador puede seleccionarla para que se inserte en el código fuente.

El uso de las herramientas de *Code completion* en los entornos de desarrollo se ha vuelto muy popular entre los desarrolladores, llegando a ser utilizada incluso varias veces por minuto [18]. Esto se debe a que, al predecir correctamente la palabra que el desarrollador intenta escribir y completarla en el código, estas herramientas evitan errores de tipeo, ahorran el esfuerzo de memorizar todos los nombres de métodos o variables y aceleran el desarrollo de software. Además, motiva a los desarrolladores a elegir nombres más largos y descriptivos para sus variables y métodos, dado que no hará falta escribirlos por completo, sino que la herramienta se encargará de completarlos.

El conjunto de sugerencias a presentar depende del contexto en que se está programando y está compuesto generalmente por nombres de variables, métodos y estructuras de control. Pese a que este conjunto es finito y limitado por las dependencias del proyecto,

normalmente las listas de sugerencias contienen un gran número de ítems, muchos de ellos sin relevancia para el desarrollador [19]. Además, en la mayoría de los casos, las sugerencias vienen ordenadas de forma alfabética, sin considerar la importancia desde el punto de vista del desarrollador.

El tema de asistencia a los desarrolladores para sugerir nuevas instrucciones o corregir las existentes en el código fuente ha sido ampliamente tratado. Las soluciones propuestas apuntan a sugerir bloques completos de código fuente en lugar de sólo el nombre de un método o variable, como hacen actualmente la mayoría de entornos de desarrollo. Se han planteado varios enfoques para realizar esta tarea, por ejemplo, usando el contexto del código fuente para sugerir ejemplos [20] [21]; identificando secuencias comunes en las invocaciones a métodos [23]; o realizando minería de datos sobre el código fuente [24]. Estas técnicas se describen en las siguientes subsecciones.

2.3.1 Sugerencia en base a métodos y tipos de datos en común

Un enfoque similar al que plantea la presente tesis es el que proponen Holmes y Murphy [20], sobre comparar la estructura del código del desarrollador con la de ejemplos generados a partir del código de proyectos existentes. A diferencia del enfoque propuesto en la presente tesis, esta técnica no analiza los ASTs, las partes relevantes de la estructura de los ejemplos que se considera durante la comparación son: nombres de clases, métodos, variables, herencia, tipos instanciados en la clase y llamadas entre clases. La comparación se realiza a partir de los tipos, llamadas a métodos y las superclases (se verifica si las clases tienen el mismo padre).

Este enfoque fue implementado en la herramienta Strathcona. Strathcona toma como entrada proyectos de software, después de analizarlos, almacena en una base de datos la estructura de los ejemplos de código que va a sugerir al usuario. Para usar la herramienta, el desarrollador selecciona la parte del código fuente de la que quiere obtener ejemplos e invoca a la herramienta. A continuación, la herramienta extrae los elementos relevantes de la estructura y busca los ejemplos de código que los incluyen.

La desventaja de este enfoque es que el desarrollador de antemano debe tener una idea del nombre de método o el tipo de dato que está buscando. Además, no se puede garantizar la calidad del código que propone al desarrollador.

2.3.2 Code completion basado en snippets

Esta técnica, propuesta por Wightman et al. [21], se basa en dos afirmaciones (según cita de Wightman et al. traducida al español):

"Si se tiene metadatos (por ejemplo, nombres de variables) sobre cómo integrar el snippet en el código fuente, es posible implementar una herramienta poderosa para presentar sugerencias al desarrollador e integrarlas en su código."

“Cuando un desarrollador elabora un snippet está dispuesto a invertir tiempo en hacerlo configurable y adaptable.”

Los ejemplos que se presentan al desarrollador son almacenados en un repositorio de código fuente. Estos ejemplos deben ser creados por el desarrollador, con la sintaxis propia de JFace [22].

La búsqueda de ejemplos se realiza mediante palabras clave. El desarrollador ingresa una o más palabras que representan la funcionalidad que realiza el snippet. Luego, el motor de búsqueda de snippets se encarga de buscar aquellos que mejor responden a la consulta y los muestra en orden de mayor a menor coincidencia.

Si el desarrollador elige uno de los snippets, éste se inserta en su código fuente, adaptando automáticamente los nombres de las variables y métodos a los que ya existen en su código.

Esta estrategia de búsqueda e integración de snippets está implementada en la herramienta Snipmatch [21]. Snipmatch es un plug-in para Eclipse cuyo funcionamiento se basa en snippets. Provee el lenguaje JFace, un lenguaje de marcado para implementar los snippets, de manera que sea fácil integrarlos al código fuente. Su sintaxis facilita completar los nombres o tipos de variables que tiene el snippet, de acuerdo a lo que se está programando.

Una limitante que se presenta al momento de usar esta herramienta es que la búsqueda depende de las palabras clave que describen el snippet y no considera sinónimos. Si por ejemplo, el desarrollador requiere un snippet cuya descripción es “*Buscar en directorio*” e ingresa las palabras clave “*Encontrar archivo en carpeta*”, la herramienta no le devolverá el snippet deseado. Además, para elaborar un snippet se debe seguir la sintaxis utilizada propia de JFace, lo que requiere invertir tiempo. La efectividad de esta herramienta depende en gran medida de que contenga los ejemplos que el desarrollador necesita, si al desarrollador le toma trabajo mantener su repositorio actualizado, podría preferir el uso de otras herramientas.

2.3.3 Búsqueda de anomalías en la interacción entre objetos

Es común que cuando dos objetos interactúan sigan un determinado protocolo, por ejemplo, seguir una determinada secuencia en la invocación de mensajes. Si el desarrollador no sigue esta secuencia, puede dar lugar a errores en tiempo de ejecución.

La técnica presentada por Wasylkowski et al. [23] propone tomar el código fuente de proyectos existentes, extraer las secuencias de llamadas a métodos y crear modelos de uso de una API. A partir de estos modelos, buscar en el código fuente bloques de código donde las llamadas a métodos no están siguiendo la secuencia y marcar estos casos como anomalías.

Este enfoque es limitado porque asiste al usuario solamente con la secuencia de los métodos a invocar; sin tomar en cuenta otros elementos como estructuras de control (`if`, `for`, `while`, etc.) y parámetros. Además, al considerar el orden de la invocación a métodos como único criterio para identificar problemas, las advertencias que genera resultan ser con frecuencia falsos positivos. Por otra parte, está enfocado en sugerir posibles problemas en la secuencia de llamadas a métodos; sin embargo, esta información se puede utilizar para proponer sugerencias al desarrollador sobre cómo aplicar correcciones.

2.3.4 Obtención de sugerencias aplicando minería de datos sobre el código

La minería de datos provee un conjunto de técnicas para obtener patrones a partir de grandes volúmenes de datos. Aplicando estas técnicas a la obtención de sugerencias de código fuente, es posible encontrar bloques de código que los desarrolladores programan con frecuencia. Cada bloque encontrado sugiere un conjunto de llamadas a métodos que típicamente son invocados juntos y además provee un ejemplo de la secuencia que se podría seguir al invocar estos métodos.

Un ejemplo del uso de técnicas de minería de datos para extraer el conjunto de ejemplos a sugerir es la herramienta MAPO [24]. Esta herramienta considera a las secuencias de invocaciones a métodos o similitud en los nombres de métodos y clases, como patrones de uso de la API. La herramienta toma como entrada el código fuente de proyectos existentes y aplica técnicas de minería para extraer estos patrones de uso, que no están previamente definidos, sino que van surgiendo de los proyectos procesados. Luego considera los patrones como índices para buscar en el código fuente del desarrollador los ejemplos de código que va a sugerir al desarrollador. A continuación, analiza el código que está escribiendo el desarrollador y muestra los ejemplos de código que están relacionados.

Una potencial desventaja de este enfoque es que MAPO establece el orden de sugerencia o ranking de los ejemplos de código comparando los nombres de método y clase del ejemplo con los del código del desarrollador, lo que puede ocasionar que ejemplos útiles para el desarrollador sean catalogados con baja similitud, sólo porque el método que los contiene tiene un nombre distinto. Por otra parte, debido a que obtiene los patrones de uso de las APIs a partir de código de proyectos existentes, la funcionalidad de la API que no fue utilizada en estos proyectos carecerá de ejemplos.

2.3.5 Minería de sugerencias relevantes sobre una fuente de conocimiento en línea (Stack Overflow)

Esta técnica propone tomar el contexto de lo que el desarrollador está programando y buscar discusiones relacionadas en Stack Overflow [25]. Toma el

código fuente y extrae las palabras, filtrando palabras muy comunes (stop words) para hacer más precisa la consulta.

Las discusiones encontradas son evaluadas en base a distintos criterios, como: similitud de los términos presentes en la descripción de la discusión (no incluye código fuente), similitud de tipos y métodos, puntuación de la discusión en Stack Overflow, puntuación de la respuesta aceptada en Stack Overflow, entre otros. Después de evaluar las discusiones, se presentan al desarrollador en orden descendente de acuerdo a la calificación obtenida.

Una desventaja de este enfoque es que no considera las fechas ni versiones de las APIs, por lo que las discusiones encontradas podrían estar ser muy antiguas y no ser de utilidad para la versión de la API que el desarrollador está utilizando. Otra limitación está en el hecho de que el desarrollador no puede almacenar respuestas que le hayan sido de utilidad para uso futuro.

2.3.6 Discusión

Tal como se expone en las secciones previas, las soluciones propuestas para la obtención y presentación de ejemplos de código fuente al desarrollador todavía tienen limitaciones y aspectos que se pueden mejorar. Si bien todas las técnicas descritas se aplicaron sobre proyectos en lenguaje Java, la solución propuesta en la presente tesis pretende diferenciarse de los enfoques existentes, abordando algunas mejoras.

A diferencia de MAPO [24] y Strathcona [20], la solución propuesta requiere que los ejemplos sean creados manualmente. Esto permite al desarrollador personalizar los ejemplos y añadir nuevos, según sus necesidades. Por otra parte, el formato de los ejemplos es simple y no requiere conocer un nuevo lenguaje, como es el caso de la herramienta Snipmatch [21], cuyos ejemplos deben ser elaborados en el lenguaje de plantilla JFace.

La solución propuesta facilita la detección de posibles defectos en el código fuente, de manera similar a lo que hace la técnica presentada por Wasylkowski et al. [23]. Lo que diferencia a este proyecto es que no verifica solamente las llamadas a métodos, sino también las estructuras de control y parámetros en la inicialización de objetos. Además, la técnica de Wasylkowski et al. se basa en el orden de las llamadas a métodos para indicar si existe un posible defecto en el código fuente. Esto puede resultar en una alta cantidad de falsos positivos, tal como lo reportan sus resultados. En cambio, la solución propuesta verifica la existencia de los elementos (llamadas a métodos, estructuras de control, parámetros), alertando al desarrollador sólo si encuentra alguno que no esté presente en su código.

Respecto de la búsqueda de sugerencias en Stack Overflow [25], la comparación que hace del código fuente que aparece en las respuestas de la discusión y el código del desarrollador se realiza utilizando el intérprete JDT de Eclipse, de manera

similar a lo propuesto por este proyecto. Sin embargo, una diferencia significativa es que la solución propuesta incluye un repositorio donde es posible actualizar o mejorar los ejemplos existentes y añadir otros nuevos.

Capítulo 3 - Diseño de la solución

En este capítulo, se describen paso a paso las decisiones de diseño tomadas para la solución propuesta por la presente tesis. Cada uno de los objetivos específicos se ve cumplido por los componentes que se describen en este capítulo.

En los párrafos siguientes se lista los componentes de la solución y se explica de manera general su función. A continuación, se presenta el algoritmo para la generación y comparación de árboles de sintaxis abstracta (AST). Luego se explica el cálculo del Grado de Similitud entre ASTs. Finalmente se define el algoritmo para identificar elementos (llamadas a métodos, parámetros, estructuras de control) faltantes en el código del desarrollador.

La solución está compuesta por:

- **Repositorio de ejemplos:** conjunto de ejemplos de buen uso de las APIs Java NIO y Google Guava, alojado en el repositorio en línea GitHub.
- **Plug-in:** componente de software que se encarga de comparar el código de los ejemplos del repositorio con el código del desarrollador, muestra los resultados ordenados por similitud, ubicando primero los más similares y permite al desarrollador integrar los ejemplos en su código fuente.

Estos componentes están representados en la Figura 2. A lo largo del capítulo se describen los componentes presentados en el párrafo anterior. El repositorio de ejemplos se explica en la Sección 3.1 y el plug-in en la Sección 3.2.

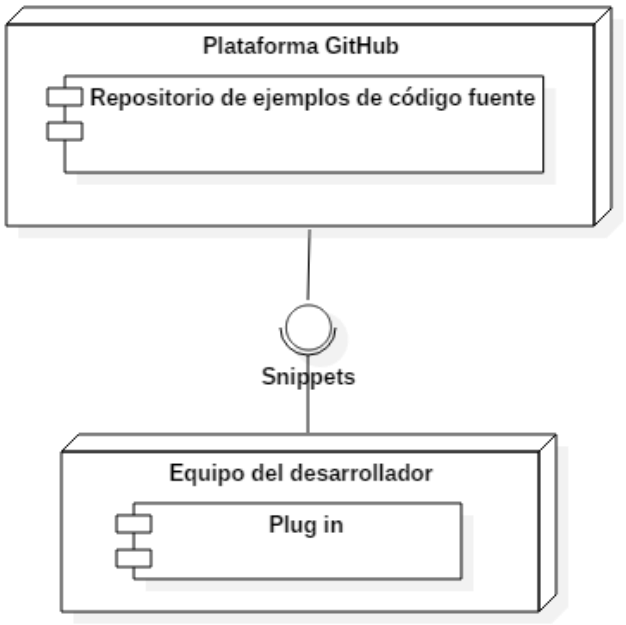

Figura 2. Diagrama de Componentes de la Solución en Etapa de Diseño.

3.1 Diseño del repositorio de ejemplos de código fuente

Los ejemplos de buen uso de las APIs son uno de los componentes principales de la solución. Están compuestos por las siguientes partes:

- Descripción del ejemplo (línea 4), es un resumen de la función que realiza el ejemplo. Debe ser lo más descriptiva posible, porque es el único metadato que se guarda del ejemplo.
- Código fuente del ejemplo (línea 6), es el cuerpo del ejemplo. Está contenido dentro de un método (líneas 5 y 7).

```
1  import java.util.*;
2
3  public class API {
4     /*Descripción del ejemplo*/
5     public void ejemplo() {
6        //Código fuente del ejemplo
7     }
8  }
```

Listado 3. Partes de un ejemplo de código fuente

Para mantener un repositorio ordenado y uniforme, se ha definido un formato específico para los ejemplos:

- Las dependencias del ejemplo de código deben estar entre las sentencias `import` de la cabecera de la clase (líneas 1 - 3).
- Los métodos deben estar incluidos en una clase (líneas 5 y 23). Una clase puede contener múltiples ejemplos.
- El nombre de la clase (línea 5) no afecta el procesamiento de los ejemplos; sin embargo, para mantener organizado el repositorio, se recomienda que los ejemplos contenidos en la clase sean de la misma API y la clase lleve el nombre de ésta. De manera similar, el nombre del método que contiene el ejemplo (línea 7) no se muestra ni afecta al ejemplo, pero es recomendable que describa su propósito.
- Deben llevar un comentario que indique su propósito (línea 6). Este texto es el título del ejemplo que se muestra en la lista de resultados del plug-in (Ver Figura 8).
- Deben estar al interior de un método (líneas 7 y 22). Sólo debe haber un ejemplo por método.
- El ejemplo puede tener una longitud variable, desde una línea a cientos de líneas. Sin embargo, para mejores resultados, es recomendable que realice una sola tarea.
- El código fuente del ejemplo no debe contener errores de compilación. Esto se debe a que el AST se genera a partir de código compilable.

A continuación se muestra un ejemplo del repositorio, con el formato descrito en el párrafo anterior.

```
1  import java.nio.ByteBuffer;
2  import java.nio.CharBuffer;
3  import java.nio.charset.Charset;
4
5  public class JavaNIO {
6     /*Code and decode charset*/
7     public void charsetEncodingDecoding() {
8        Charset charset = Charset.forName("US-ASCII");
9        System.out.println(charset.displayName());
10       System.out.println(charset.canEncode());
11       String texto = "Charset example.";
12       //Convert the byte buffer into the input charset of a unicode char buffer
13       ByteBuffer buffer = ByteBuffer.wrap(texto.getBytes());
14       CharBuffer charBuffer = charset.decode(buffer);
15       //Convert the unicode char buffer into a byte buffer in the input charset
16       ByteBuffer newBuffer = charset.encode(charBuffer);
17       while (newBuffer.hasRemaining()) {
18           char chars = (char) newBuffer.get();
19           System.out.print(chars);
20       }
21       newBuffer.clear();
22    }
23 }
```

Listado 4. Formato de un ejemplo de código fuente

3.2 Diseño del plug-in

Para acceder a los ejemplos del repositorio y poder insertarlos en el editor de código fuente, se ha diseñado un plug-in para el entorno de desarrollo Eclipse. Este plug-in se encarga de buscar los ejemplos, mostrarlos al desarrollador y le permitirle integrarlos con su código fuente.

En las secciones siguientes, se describe en un alto nivel de detalle, la manera en que se realiza la búsqueda de ejemplos de código fuente del repositorio.

3.2.1 Comparación entre árboles de sintaxis

La búsqueda de ejemplos de código fuente del repositorio comienza con la comparación entre ASTs, para encontrar los ejemplos más similares al código del desarrollador. La comparación entre ASTs considera los nodos del árbol, no así la estructura propia del árbol. Esta comparación se basa en la edición de un árbol para transformarlo en otro mediante acciones [26]. Esta secuencia de acciones es denominada script de edición.

Dado que la comparación se realiza entre nodos del AST, esta se efectúa a nivel de la estructura del código, sin verse afectada por la sintaxis o convenciones de programación propias del lenguaje.

Puede haber más de una forma de transformar un árbol en otro. En estos casos, el algoritmo de comparación entre ASTs considera la secuencia de acciones con el número mínimo de pasos.

Las acciones tienen un tipo, que puede ser:

- *Insert*: añade un nuevo nodo al árbol. Si el nodo padre está definido, el nodo a añadir será el i-ésimo hijo. Caso contrario, el nodo a añadir será el nuevo nodo raíz y el anterior nodo raíz se volverá el hijo único del nodo a añadir.

  Ejemplo:

Código del desarrollador	Código del ejemplo del repositorio

`public void getSales(){`
`  int result = 0;`
 	
`}`
	`public void calculateFactor(){`
	`  int result = 0;`
	`  int factor = 0.5;`
	`}`

Listado 5. Ejemplo de código fuente para una Acción de tipo *Insert*

Estos bloques de código dan lugar a los ASTs de la Figura 3. El AST de la izquierda corresponde al código del desarrollador y el de la derecha a un ejemplo del repositorio. Los nodos resaltados en tono gris en el AST del ejemplo del repositorio son los que se debería insertar en el AST del código del desarrollador para que ambos ASTs sean iguales. Por lo tanto, al efectuar la comparación entre estos ASTs, el resultado es una acción de tipo *Insert*.

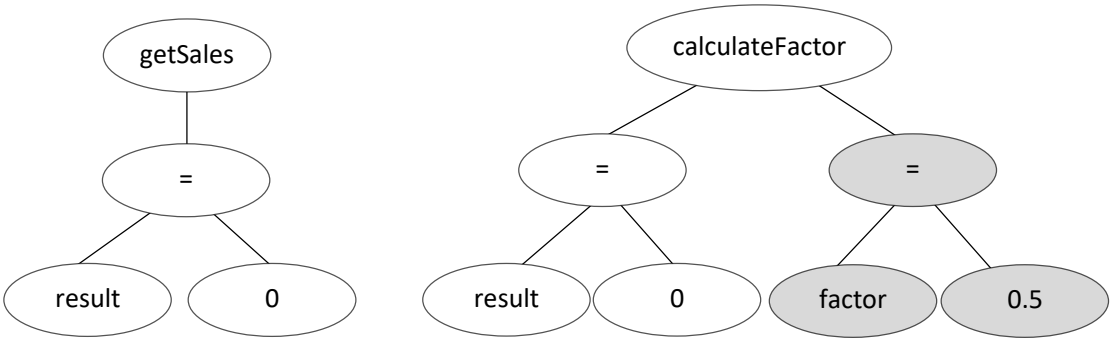

Figura 3. Ejemplo de comparación entre ASTs que resulta en una acción *Insert*.

- *Update*: reemplaza el valor antiguo del nodo por un valor nuevo.
  Ejemplo:

Código del desarrollador	Código del ejemplo del repositorio

`public void getSales()`
`  int amount = 0;`
`}`
	`public void calculateFactor()`
	`  int result = 0;`
	`}`

Listado 6. Ejemplo de código fuente para una Acción de tipo *Update*

Estos bloques de código dan lugar a los ASTs de la Figura 4. Para pasar del AST del código del desarrollador al AST del ejemplo del repositorio se debe modificar el nombre de la variable `amount` por `result`. Por lo tanto, la comparación entre estos ASTs da como resultado una acción de tipo *Update*.

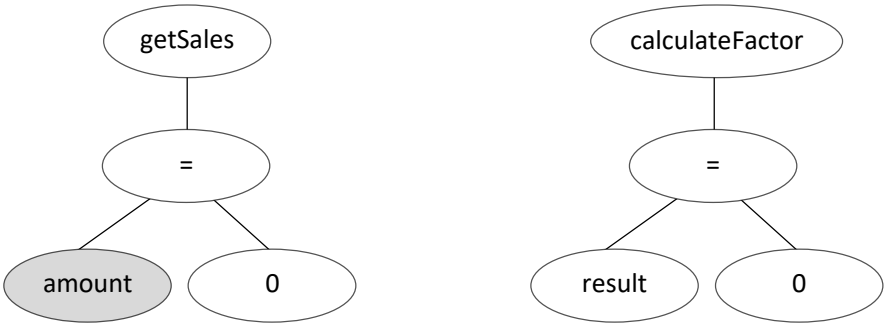

Figura 4. Ejemplo de comparación entre ASTs que resulta en una acción *Update*.

Es importante resaltar que el nombre de los métodos que contienen al código fuente (`getSales`, `calculateFactor`) no se considera en la comparación de ASTs; sólo se toma en cuenta el código que está al interior del método.

- *Delete*: borra un nodo del AST.

  Ejemplo:

Código del desarrollador	Código del ejemplo del repositorio

`public void getSales(int factor)`
`  int result = 100 * factor;`
`}`
	`public void getFactor()`
	`  int result = 100;`
	`}`

Listado 7. Ejemplo de código fuente para una Acción de tipo *Delete*

Estos bloques de código dan lugar a los ASTs de la Figura 5. En el AST del código del desarrollador están resaltados en tono gris los nodos que se debe borrar para que ambos ASTs sean iguales. Por lo tanto, al efectuar la comparación entre estos ASTs, el resultado son acciones de tipo *Delete* por

cada nodo que se debe eliminar.

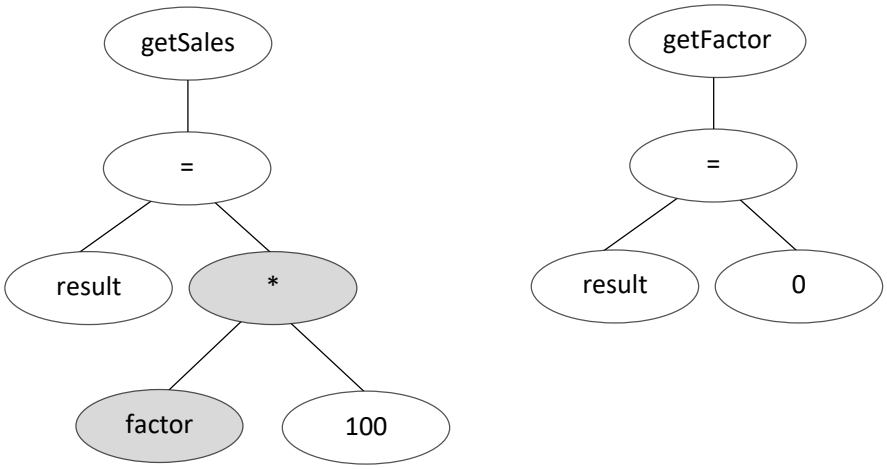

Figura 5. Ejemplo de comparación entre ASTs que resulta en acciones *Delete*.

- *Move*: mueve el nodo, de su posición a la posición i-ésima de su nuevo nodo padre. Todos los nodos hijos del nodo a mover también serán trasladados, por lo que se considera un movimiento de subárboles.

  Ejemplo:

Código del desarrollador	Código del ejemplo del repositorio

`public void getSales()`
`  int result = 100;`
`  int factor = 0.5;`
`}`
	`public void getFactor()`
	`  int factor = 0.5;`
	`  int result = 100;`
	`}`

Listado 8. Ejemplo de código fuente para una Acción de tipo *Move*

Los bloques de código del Listado 8 dan lugar a los ASTs de la Figura 6. Para llegar del AST del código del desarrollador al AST del ejemplo del repositorio se debe mover de posición la declaración de las variables cuyos nodos están resaltados en tono gris; por lo tanto, el resultado de la comparación de estos dos ASTs es una acción de tipo *Move*.

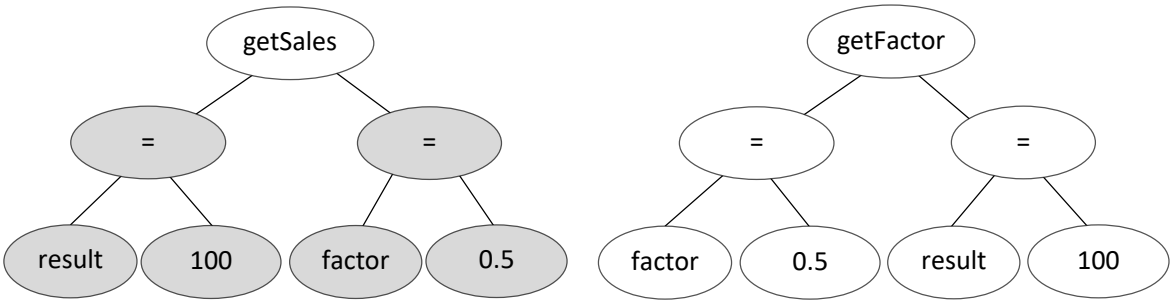

Figura 6. Ejemplo de comparación entre ASTs que resulta en acciones *Move*.

El resultado de la comparación entre el AST del código del desarrollador y el AST del ejemplo del repositorio está expresado en forma de una secuencia de acciones:

$A = \{a_1, a_2, a_3, \ldots, a_k\}$ = secuencia de acciones necesarias para llegar de un AST al otro

En la comparación se considera el tipo de nodo y también la etiqueta. Por ejemplo, si el código del desarrollador no tiene una declaración de variable, esto da lugar a un nodo *Insert*, pero si la variable existe con otro nombre, se genera una acción de tipo *Update*.

3.2.2 Cálculo del Grado de Similitud

El siguiente paso es calcular el Grado de Similitud a partir de la lista de acciones que resultan de la comparación entre ASTs.
La fórmula para el cálculo del Grado de Similitud fue desarrollada específicamente para el trabajo y se explica a continuación.

Sea:

$D = \{d_1, d_2, d_3, \ldots, d_i\}$ AST del código fuente del desarrollador.

$E = \{e_1, e_2, e_3, \ldots, e_j\}$ AST del ejemplo del repositorio.

$A = \{a_1, a_2, a_3, \ldots, a_k\}$ = secuencia de acciones necesarias para llegar de un AST al otro.

El Grado de Similitud se define como:

$$Gd(D, E) = 1 - \sum a_i / j, \forall\, a_i \in A$$

Por ejemplo, si el ejemplo del repositorio de código fuente tiene cinco nodos y en la comparación con el AST del código del desarrollador se obtiene una lista con cinco acciones, el Grado de Similitud tendrá un valor de:

$$Gd(D, E) = 1 - 5/5 = 0$$

Esto indica que ambos ASTs son totalmente diferentes. Por el contrario, si la lista de acciones tiene un valor igual a 0, el Grado de Similitud será igual a:

$$Gd(D, E) = 1 - 0/5 = 1$$

Siendo 1 el máximo valor al que puede llegar el Grado de Similitud.

3.2.3 Búsqueda de los elementos faltantes en el código del desarrollador

El objetivo de la solución es presentar al desarrollador los ejemplos más parecidos a su código en las primeras posiciones de los resultados de la búsqueda. Por esta razón, luego de haber calculado el Grado de Similitud, se ordena los ejemplos que serán mostrados al desarrollador, en orden descendente de acuerdo a la similitud.

Al mostrar los ejemplos al código del desarrollador, se resaltan los elementos del código (estructuras de control, parámetros, llamadas a métodos) que están presentes en cada ejemplo del repositorio y no se encuentran en su código. Esta función permite al desarrollador notar con mayor facilidad elementos que podrían faltarle y agregarlos.

Por ejemplo, en los bloques de código del Listado 9, se verifica que de todas las llamadas a métodos, estructuras de control y parámetros que tiene el ejemplo del repositorio, al código del desarrollador le falta una estructura de control `for` y llamadas a los métodos `add` e `intersection`. Por lo tanto, cuando se muestre el ejemplo al desarrollador, se resaltarán en rojo el ciclo `for` y los nombres de los métodos mencionados.

Código del desarrollador	Código del ejemplo del repositorio

`public void joinSets(){`
 	
`  Set<Integer> numbers = ImmutableSet.of(2, 5);`
 	
`  Set<Integer> evenNumbers = ImmutableSet.of(2, 4);`
 	
`}`
	`public void setIntersection() {`
	
	`  Set<Integer> oddNumbers = new TreeSet <Integer>();`
	
	`  for(int i = 1; i <= 9; i += 2){`
	
	`  oddNumbers.add(i);`
	
	`  }`
	
	`  Set<Integer> primeNumbers = ImmutableSet.of(2,3, 5, 7);`
	
	`  SetView<Integer> intersection Sets.intersection(primeNumbers, oddNumbers);`
	
	`}`

Listado 9. Búsqueda de elementos faltantes en el código del desarrollador

Dado que no todos los ejemplos son estrechamente similares al código del desarrollador, esta búsqueda de elementos faltantes se realiza sólo con los ejemplos de mayor similitud. La elección se realiza comparando el Grado de Similitud obtenido con un *threshold*. Los ejemplos de código cuyo Grado de Similitud sea menor al *threshold* son seleccionados y continúan con el paso siguiente: buscar qué

elementos del ejemplo de código (estructuras de control, parámetros, llamadas a métodos) no aparecen en el código del desarrollador.

El *threshold* es configurable y el valor por defecto ha sido definido en base a experimentación con distintos valores.

3.2.4 Interfaz gráfica

En cuanto a la parte gráfica del plug-in, se ha definido que estará ubicado en el panel inferior de *Views* o vistas de Eclipse, sector que se resalta en la Figura 7. Se ha elegido situar el plug-in en una vista para que los ejemplos estén disponibles al desarrollador en todo momento, no sólo cuando invoca al menú contextual, como la herramienta de sugerencia de ejemplos que Eclipse tiene por defecto.

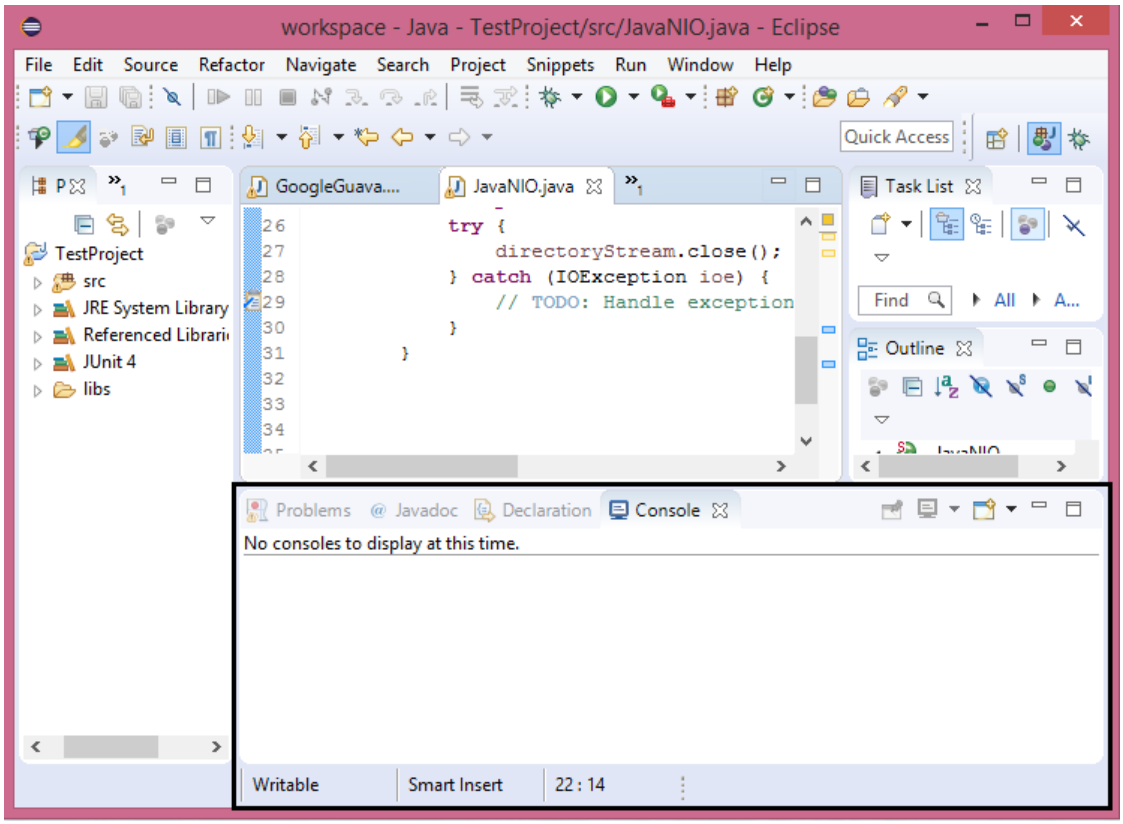

Figura 7. Disposición por Defecto de las Vistas y Editor de Código en Eclipse.

Capítulo 4 - Implementación de la solución

En este capítulo se presentan detalles de la construcción de los componentes de la solución: el repositorio de ejemplos y el plug-in. Se inicia explicando las clases que intervienen en la generación y comparación de árboles de sintaxis abstracta (ASTs). Dado que durante la implementación, se encontraron varios casos especiales, se describen en este capítulo y se indica la manera en que han sido tratados.

La solución tiene los componentes presentados en la Figura 8:

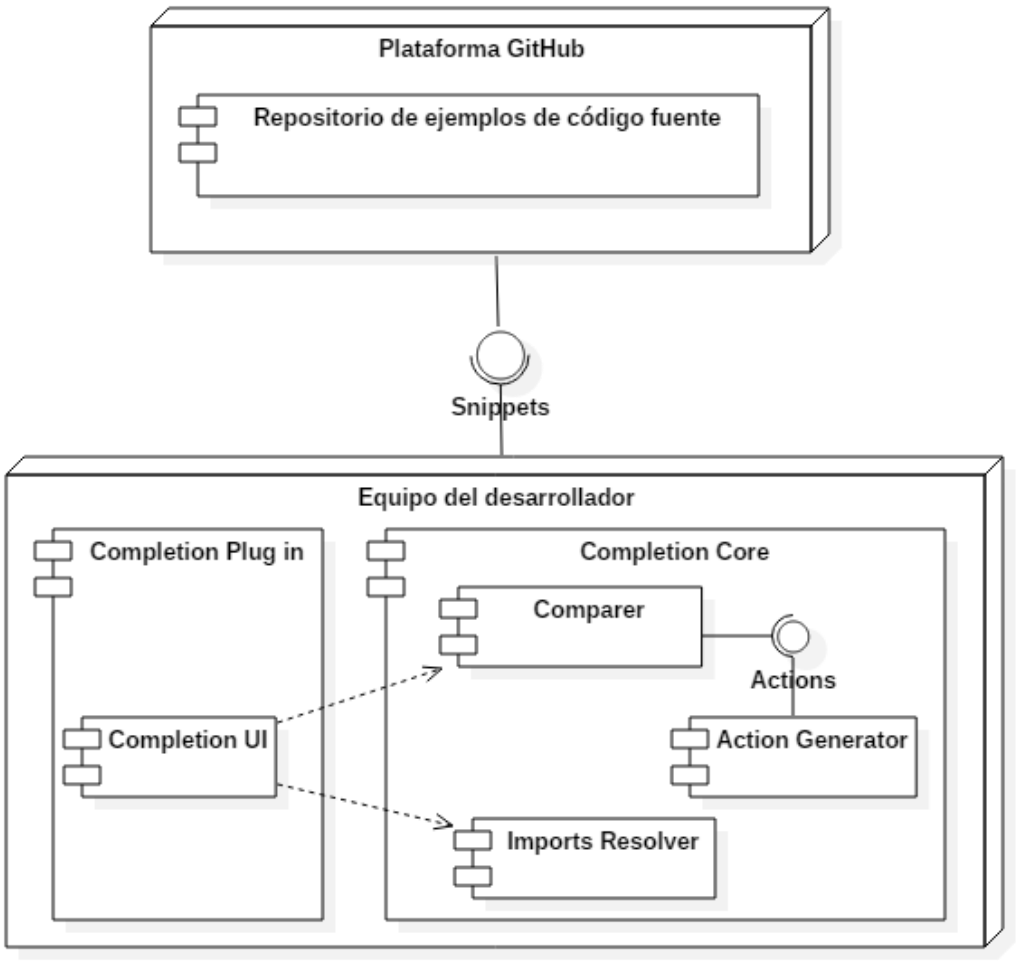

Figura 8. Diagrama de los Componentes de la solución en Etapa de Implementación.

A continuación se describe brevemente las funciones de los componentes del plug-in:

- *Comparer*, que se encarga de la comparación y verificación de elementos faltantes (llamadas a métodos, estructuras de control o parámetros) en el código del desarrollador.
- *Actions Generator*, que obtiene los ASTs del ejemplo de código fuente y el código

del desarrollador, además calcula la secuencia de acciones necesarias para pasar de un AST al otro.
- *Imports Resolver*, que se encarga de identificar las dependencias del proyecto.
- *Completion UI*, que realiza la interacción con el usuario, extendiendo la funcionalidad que provee el entorno de desarrollo Eclipse.

El primer componente en ser implementado es el repositorio de ejemplos de código fuente, que se explica en la siguiente sección.

4.1 Construcción del repositorio de ejemplos de código fuente

El repositorio de ejemplos está alojado en la plataforma GitHub[*]. Esta ubicación, que es de acceso público, permite a otros desarrolladores añadir sus propios ejemplos.

Para agregar un nuevo ejemplo, basta con seguir el formato definido en la Sección 3.6 del Capítulo 3 de este documento. El código fuente y el comentario en la cabecera del método que describe su propósito son la única información que se requiere del ejemplo.

En el marco de la presente tesis se ha poblado el repositorio con ejemplos de dos APIs: Java NIO y Google Guava. Se han implementado 25 ejemplos por cada API. Los ejemplos contienen combinaciones de llamadas a los métodos propios de las APIs, además de creación de objetos cuyos tipos son específicos de las APIs.

No existe un tamaño máximo para los ejemplos; sin embargo, como referencia, en la implementación actual el promedio de líneas de código para los ejemplos de Java NIO es de 15.24 y el promedio para los ejemplos de Google Guava es de 6.8 líneas.

Cada ejemplo está diseñado para cumplir una tarea, a continuación se muestran algunos ejemplos que están almacenados en el repositorio:

- Utilizando la API Google Guava, verificar si una lista está ordenada de acuerdo con un determinado criterio:

```
1  /*Verify if a list is ordered*/
2        public void verifyIsOrdered(){
3              List<Integer> numeros = new ArrayList<Integer>();
4              numeros.add(new Integer(5));
5              numeros.add(new Integer(2));
6              numeros.add(new Integer(15));
7              Ordering<Integer> orden = Ordering.natural();
8              boolean isOrdered = orden.isOrdered(numeros);
9              System.out.println(isOrdered);
10 }
11
```

Listado 10. Ejemplo de buen uso de la API Google Guava

[*] https://github.com/Yasett/code-completion-snippets

- Utilizando la API Java NIO, escribir en un archivo:

```
1  /*Write in a file*/
2  public void writeFile() {
3      List<String> lines =
4  Arrays.asList(String.valueOf(Calendar.getInstance().
5                           getTimeInMillis()),"test line", "test line 2");
6
7      try {
8          Path path = Paths.get("file_path.ext");
9          OutputStream outputStream = Files.newOutputStream (path,
10                                      StandardOpenOption.APPEND);
11         for (String line : lines) {
12               outputStream.write((line + System.lineSeparator()).
13               getBytes(StandardCharsets.UTF_8));
14         }
15
16        } catch (Exception e) {
17           //TODO: Handle exception
18        }
}
```

Listado 11. Ejemplo de buen uso de la API Java NIO

Los ejemplos del repositorio fueron validados por dos programadores con experiencia en Java, que trabajan como desarrolladores senior en la empresa Toptal[†]. A ambos se les pidió especificar su experiencia e indicaron tener al menos cinco años de experiencia en el lenguaje Java. También declararon tener 6 meses de manejo de las APIs Java NIO y Google Guava y haber trabajado en al menos 2 proyectos que utilizan estas APIs.

A los programadores expertos se les proporcionó acceso al repositorio y se les pidió juzgar la correctitud de los ejemplos. Como resultado, se corrigieron los ejemplos observados y se reemplazaron cinco de los ejemplos por ser considerados no comprensibles o de poca utilidad. Uno de los ejemplos corregidos fue `CopyFile`, que estaba programado de la siguiente forma:

[†] Toptal es una empresa de tecnología con base en Silicon Valley, USA. Ofrece el servicio de desarrolladores y diseñadores a empresas que necesiten contratar personal en modalidad freelance.

```
1   /* Copy the contents of a file */
2   public void CopyFile() {
3         File sourceFile = new File("file_path.ext");
4         File destFile = new File("file_path.ext");
5         if (!sourceFile.exists() || !destFile.exists()) {
6               // Source or destination file doesn't exist
7               return;
8         }
9
10        try {
11              FileInputStream fis = new FileInputStream(sourceFile);
12              FileOutputStream fos = new FileOutputStream(destFile);
13              FileChannel srcChannel = fis.getChannel();
14              FileChannel sinkChanel = fos.getChannel();
15              try {
16                    srcChannel.transferTo(0, srcChannel.size(),
17                    sinkChanel);
18              } catch (Exception e) {
19                    // TODO: Handle exception
20              }
21              fis.close();
22              fos.close();
23        } catch (FileNotFoundException fnfe) {
24              // TODO: Handle exception
25        }catch (IOException ioe) {
26              // TODO: Handle exception
27        }
28  }
```

Listado 12. Ejemplo del repositorio objetado por los expertos

Ambos expertos observaron que podía escribirse de manera más simple y uno de ellos sugirió el utilizar un bloque *try-with-resources*[‡] (líneas 11 - 14). Gracias a su recomendación, el ejemplo fue reformulado:

```
1   /*Copy the contents of a file*/
2   public void CopyFile() {
3       File sourceFile = new File("file_path.ext");
4       File destFile = new File("file_path.ext");
5
6       if (!sourceFile.exists() || !destFile.exists()) {
7       // Source or destination file doesn't exist
```

[‡] Sentencia `try` que declara uno o más recursos. Un recurso es un objeto que debe ser cerrado después de que el programa haya terminado de usarlo. La sentencia *try-with-resources* se asegura de que cada recurso sea cerrado al terminar de ejecutarse.

```
8           return;
9       }
10
11      try (FileInputStream fis = new FileInputStream(sourceFile);
12      FileOutputStream fos = new FileOutputStream(destFile);
13      FileChannel srcChannel = fis.getChannel();
14      FileChannel sinkChanel = fos.getChannel();) {
15
16      srcChannel.transferTo(0, srcChannel.size(), sinkChanel);
17
18      } catch (Exception e) {
19      //TODO: Handle exception
20      }
21  }
```

Listado 13. Correcciones implementadas en ejemplo del repositorio

Una vez construido y validado el repositorio, se procede a la implementación del plug-in. Para explicar la implementación del plug-in y su interacción con el repositorio, se repasan los pasos que ocurren cuando el desarrollador de software utiliza la solución:

Paso 1. El desarrollador escribe código fuente e invoca al Completion plug-in.

Paso 2. El plug-in filtra ejemplos del repositorio, en base a las librerías que tenga importadas el desarrollador en su proyecto.

Paso 3. El plug-in genera los ASTs y calcula el Grado de Similitud.

Paso 4. El plug-in muestra la lista de ejemplos encontrados y el desarrollador revisa los ejemplos, verificando qué partes del ejemplo no están presentes en su código.

Paso 5. El desarrollador integra el código fuente del ejemplo en su propio código.

La secuencia de llamadas entre los componentes que forman parte del plug-in: *Completion Core* y *Completion plug-in* se detalla en el diagrama de secuencia de la Figura 9. La implementación de las tareas que realizan, se describe paso por paso a lo largo de este capítulo.

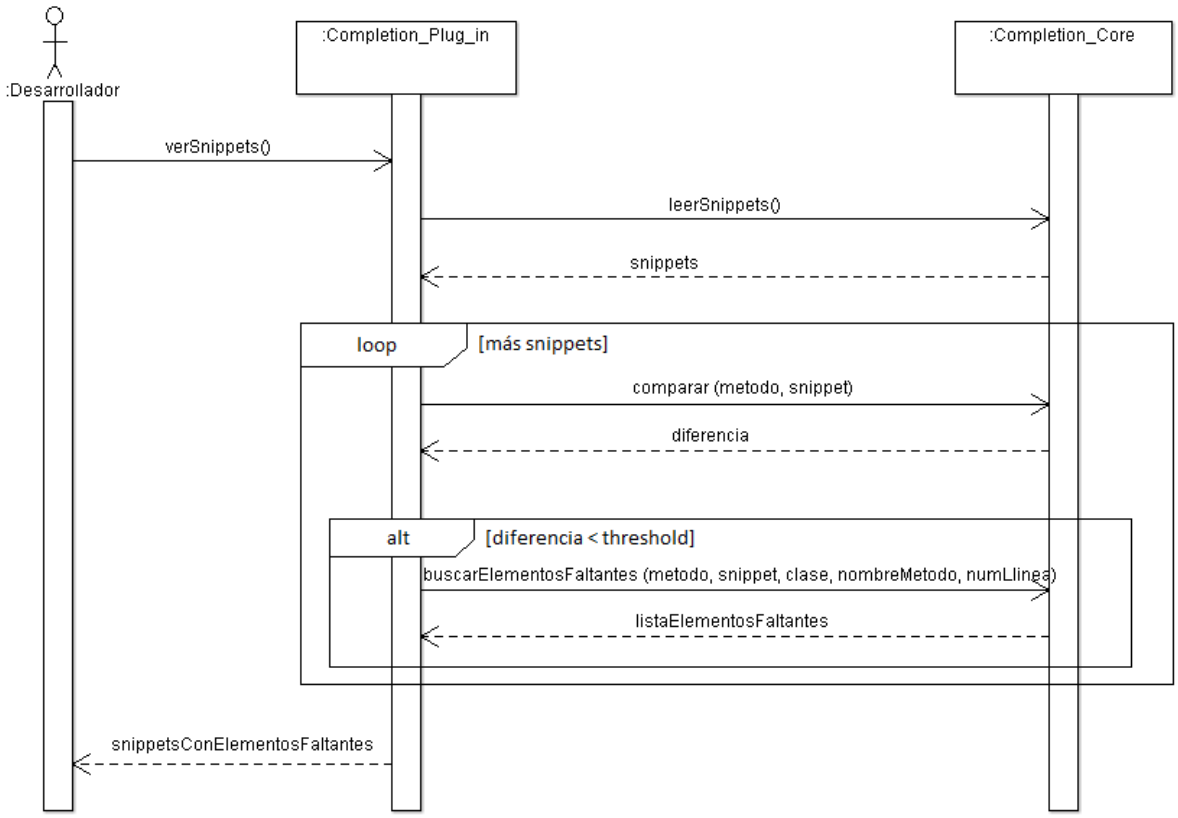

Figura 9. Diagrama de secuencia de la búsqueda de ejemplos.

4.2 Invocación del plug-in

El plug-in se invoca a partir de un sub menú contextual en el editor de código fuente del entorno de desarrollo Eclipse o mediante un atajo de teclado (Ctrl + Alt + S). El plug-in funciona a nivel de método, es decir, al momento de llamar al plug-in, se toma como entrada el código fuente que se encuentra al interior del método donde está el cursor. Luego, el plug-in llama al proyecto *Completion Core* para pedir los ejemplos relevantes.

4.3 Filtro de librerías

A medida que se aumentan ejemplos sobre distintas APIs en el repositorio, la búsqueda toma más tiempo. Por esta razón, antes de comenzar la búsqueda, se verifica cuáles APIs están importadas en el proyecto y sólo se realiza la búsqueda entre los ejemplos de código fuente del repositorio que corresponden a esas APIs.

4.4 Generación de ASTs y cálculo del Grado de Similitud

4.4.1 Implementación de la generación de ASTs

El primer paso para el algoritmo de comparación entre ASTs es la generación de árboles de sintaxis abstracta (AST). En el Capítulo 3 de este proyecto se describió la generación de un AST simple. Sin embargo, en la implementación del proyecto se generan los ASTs mediante el intérprete de Eclipse [27], que agrega nodos adicionales.

El intérprete soporta archivos completos así como bloques sueltos de código fuente. El siguiente es un ejemplo de cómo se utiliza la clase `ASTParser` para generar un árbol de sintaxis abstracta:

```
1   protected TreeContext parse(String source) {
2         ASTParser parser = ASTParser.newParser(AST.JLS3);
3         Map options = JavaCore.getOptions();
4          options.put(JavaCore.COMPILER_COMPLIANCE,JavaCore.VERSION_1_8);
5          options.put(JavaCore.COMPILER_SOURCE, JavaCore.VERSION_1_8);
6          options.put(JavaCore.COMPILER_CODEGEN_TARGET_PLATFORM,
7         JavaCore.VERSION_1_8);
8         parser.setKind(ASTParser.K_COMPILATION_UNIT);
9         parser.setCompilerOptions(options);
10         parser.setSource(source.toCharArray());
11         CompilationUnit root = (CompilationUnit)parser.createAST(null);
12         AbstractJdtVisitor visitor = createVisitor();
13         root.accept(visitor);
14         return visitor.getTreeContext();
15  }
```

Listado 14. Código fuente que realiza la generación de ASTs

En el ejemplo, la constante `AST.JLS3` (línea 2) indica que se utilizan las especificaciones de lenguaje de Java. El parámetro `ASTParser.K_COMPILATION_UNIT` (línea 8) indica que la salida de generar el árbol es un objeto de tipo `ICompilationUnit`. El parámetro `JavaCore.VERSION_1_8` (líneas 4, 5 y 6), indica la versión de Java con que el código será interpretado por el compilador.

En la línea 10 se pasa al intérprete el código fuente que dará lugar al AST y en la línea 11 se crea el nodo raíz del AST. Al pasar el parámetro con un valor nulo se indica que no será necesario utilizar un objeto de tipo `ProgressMonitor` para hacer seguimiento al proceso. En la línea siguiente (línea 12), se crea el objeto `AbstractJdtVisitor`, que recorre el código fuente. Este objeto forma los nodos y llena el AST, almacenado en su propiedad `context`.

A diferencia del AST simple explicado en el Capítulo 3, cada nodo del AST generado tiene una etiqueta, que hace referencia a un valor o nombre del elemento

estructural del código fuente, por ejemplo, un nodo de tipo *NumberLiteral* puede tener el valor 1. Algunos valores no representan información relevante y por lo tanto son descartados, por ejemplo el caso de los nodos de tipo *Block*, que no tienen una etiqueta asociada y por lo tanto no tienen valor.

Por ejemplo, considerando el bloque de código siguiente:

```
1   import java.util.Set;
2   import com.google.common.collect.ImmutableSet;
3
4   public class Test {
5
6         public void test(){
7               Set<Integer> numbers = ImmutableSet.of(2, 5);
8               Set<Integer> evenNumbers = ImmutableSet.of(2, 4);
9
10        }
    }
```

Listado 15. Código de ejemplo para la generación de ASTs

El AST que se obtiene es el de la Figura 10. Algunas propiedades relevantes que tiene cada nodo del AST resultante son:

- `Id`: identificador único del nodo.
- `Type`: código numérico que indica el tipo de nodo, por ejemplo, el código 59 corresponde a una declaración de variable. En el Anexo A se encuentra una lista con todos los posibles códigos. La lista completa de los valores que puede tomar el atributo `Type` se encuentra en el Anexo A. Todos los valores de la lista están dentro del alcance del presente proyecto.
- `Label`: se utiliza en el caso de clases, variables y métodos, lleva el nombre de éstos.
- `LineNumber`: es el número de línea del archivo de código fuente donde aparece el nodo.
- `Children`: contiene la lista de nodos hijos.

Bajo la implementación propuesta, los nodos del árbol objetos son de la clase `Tree`, excepto las llamadas a método, que son objetos de la Clase `TreeMethodInvocation`. La clase `TreeMethodInvocation` extiende de `Tree` y le añade el atributo `methodName`, donde almacena el nombre del método.

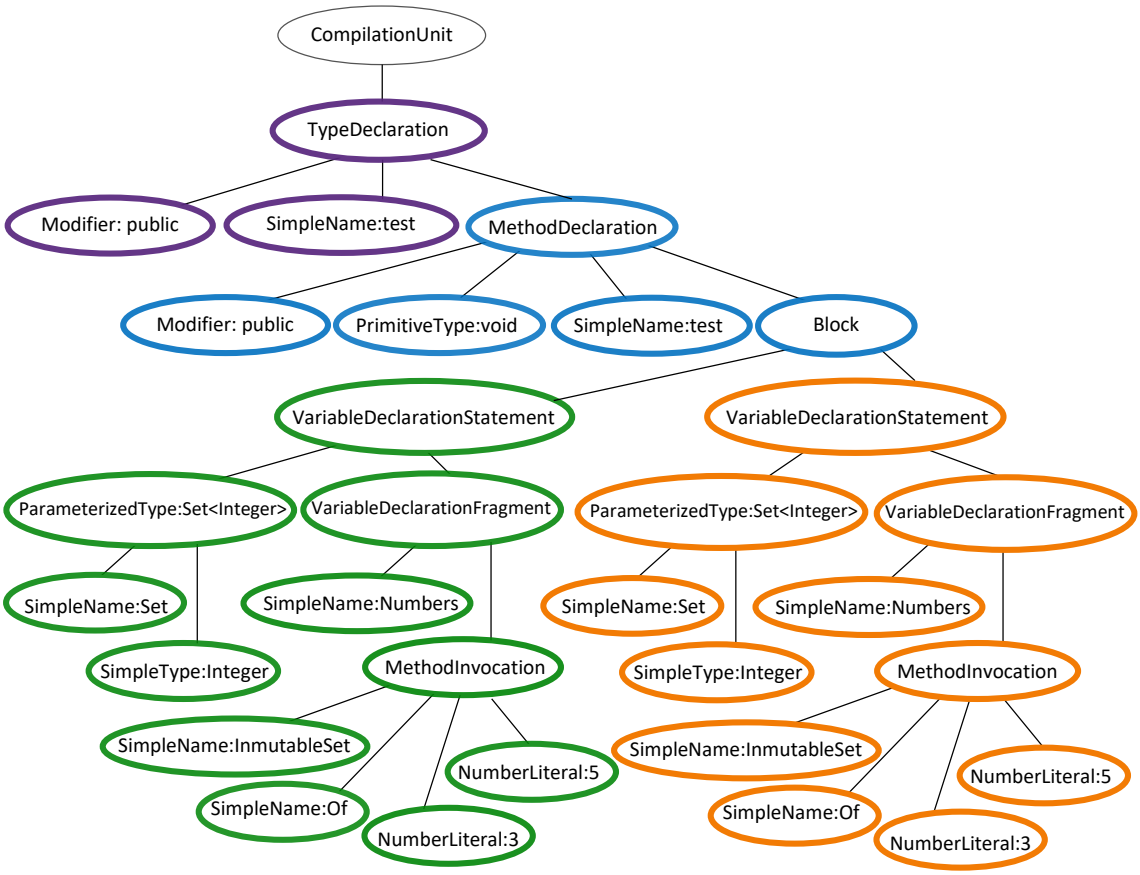

Figura 10. AST resultado del algoritmo de generación.

4.4.2 Implementación de la comparación entre ASTs

La comparación entre ASTs es realizada mediante la API de la librería *GumTree* [26]. Esta librería recibe los AST del código del desarrollador y el ejemplo de código fuente y retorna una lista de acciones que representan la secuencia de pasos para pasar de un árbol al otro. Contienen una referencia al nodo que se debe insertar, eliminar, mover o actualizar. Por ejemplo, considerando los siguientes bloques de código:

Código del desarrollador	Código del ejemplo del repositorio

`public void getCommonNumbers(){`
`  Set<Integer> numbers = ImmutableSet.of(2, 5);`
`  Set<Integer> evenNumbers = ImmutableSet.of(2, 4);`
 	
`}`
	`public void setIntersection() {`
	`  Set<Integer> primeNumbers = ImmutableSet.of(2, 3, 5, 7);`
	`  Set<Integer> oddNumbers = ImmutableSet.of(1, 3, 5, 7);`
	`  SetView<Integer> intersection= Sets.intersection(primeNumbers, oddNumbers);`
	`}`

Listado 16. Ejemplo del repositorio para demostrar la generación de acciones

Al realizar la comparación se obtiene una secuencia de las acciones necesarias para igualar el código del desarrollador al código del ejemplo del repositorio. La secuencia de acciones está compuesta por: una acción de tipo *Insert* para agregar la declaración de variable `SetView<Integer> intersection`, otra acción de tipo *Update* para modificar el nombre de la variable `numbers` por `primeNumbers`, una acción para agregar los parámetros `3` y `7` a la creación de la variable `numbers`, entre otras.

4.4.3 Implementación del cálculo del Grado de Similitud

Una vez obtenidos los resultados de la comparación entre árboles, en términos de la secuencia de acciones necesarias para llegar de un árbol a otro, se realiza el cálculo del Grado de Similitud entre el ejemplo y el código del desarrollador.

Debido a que los ASTs generados en la actual implementación incluyen más nodos que un AST simple, al cálculo base explicado en el Capítulo 3 se le añade una función de verificación v, que evalúa si la acción es o no redundante y por lo tanto si debe ser considerada para el cálculo. La función de verificación v se realiza de la siguiente forma:

Si t_k = *Insert*

Si na_k = `Block`

Si los nodos hijos de $na_k \notin NA$ (Para evitar redundancia)

Considerar acción para el cálculo

Si no

Si $na_k \neq$ `PrimitiveType` y $na_k \neq$ `SimpleName`

Considerar acción para el cálculo

Los nodos de tipo `PrimitiveType` y `SimpleName` no suman al Grado de Similitud porque son redundantes, es decir, existen otros nodos que representan la misma acción. Por lo tanto, para hacer más breve el proceso y evitar redundancia, se eliminan de los ASTs y de la lista de acciones.

Las acciones de tipo *Update* no suman al Grado de Similitud porque son actualizaciones de nombres o valores de variables. Las acciones de tipo *Delete* y *Move* tampoco suman al Grado de Similitud porque la diferencia se calcula en base a los elementos del AST del ejemplo del repositorio que le faltan al AST del desarrollador y estas acciones reflejan los elementos que están de sobra. De hecho, se espera que existan muchas acciones de tipo *Delete*, ya que el código del desarrollador normalmente tendrá más sentencias que el ejemplo.

Por lo tanto, la fórmula del Grado de Similitud se modifica a:

$$Gd(D, E) = 1 - \sum v(a_i) / j, \forall\, a_i \in D$$

Como una mejora a este cálculo, además de contabilizar los elementos que diferencian al AST del código del desarrollador del AST del ejemplo del repositorio, se busca si existen llamadas al mismo método en ambos ASTs. Esto se debe a que encontrar llamadas en común, puede indicar que existe una fuerte similitud. En este mismo sentido, si existen tipos en común (aparte de los tipos que provee el marco de trabajo de Java por defecto) también se considera que hay mayor similitud.

Por esta razón, se hace un recuento de las llamadas a métodos y tipos en común y se restan a la cantidad de acciones, multiplicando el valor por dos, dado que se asigna mayor importancia a la ocurrencia de esta similitud.

Entonces, sea *C(D, E)* = cantidad de llamadas a métodos y tipos en común:

$$Gd(D, E) = 1 - (\sum v(a_i) - 2 * C(D, E)) / j, \forall a_i \in D$$

Luego de hacer la comparación, los ejemplos se ordenan de forma descendente por el Grado de Similitud, de manera que el desarrollador vea primero los ejemplos más similares a su código fuente.

La generación y comparación entre ASTs, además del cálculo del Grado de Similitud se realizan en el proyecto denominado *Completion Core*, que es parte del plug-in, de acuerdo al algoritmo explicado en secciones anteriores.

Las clases principales que realizan la comparación e identifican los elementos de código que pueden faltar al desarrollador se muestran en la Figura 11.

A continuación se describe las responsabilidades de estas clases.

- `CBComparer`: clase que organiza el proceso de búsqueda de ejemplos de código fuente. Llama a las clases que generan los ASTs, a las que realizan la comparación y a las que buscan elementos faltantes en el código del desarrollador. Uno de los métodos más importantes de esta clase es `check`, que realiza la comparación entre ASTs. El método `check` está sobrecargado, para ser utilizado con código compilable y código no compilable, este último se encuentra almacenado en el parámetro `incompleteLines`.
- Otro método destacable es `checkWarnings`, que identifica los elementos (llamadas a métodos, parámetros o estructuras de control) que están en el ejemplo de buen uso de la API, pero no se encuentran en el código del desarrollador.
- `ICBHeuristicChecker`: es la interfaz que implementan las clases que revisan los elementos faltantes en el código del desarrollador. Define el comportamiento que tendrán estas clases.
- `CBAbstractChecker`: clase que se encarga de obtener el resultado de la comparación de árboles de sintaxis abstracta. Este comportamiento es común entre las clases

`CBSnippetChecker`, `CBMissingMessageChecker`, `CBMissingControlStatementChecker` y `CBMissingInitializationParamChecker`; por esta razón fue separado en una clase abstracta.

- `CBSnippetChecker`: clase que realiza el cálculo del Grado de Similitud.
- `CBMissingMessageChecker`: clase cuya tarea principal es verificar si en el código del desarrollador faltan llamadas a métodos que se encuentran en los ejemplos.
- `CBMissingControlStatementChecker`: clase que se encarga de identificar las estructuras de control que faltan en el código del desarrollador pero están presentes en los ejemplos.
- `CBMissingInitializationParamChecker`: clase que busca si existen parámetros que estén en los ejemplos y no en el código del desarrollador.

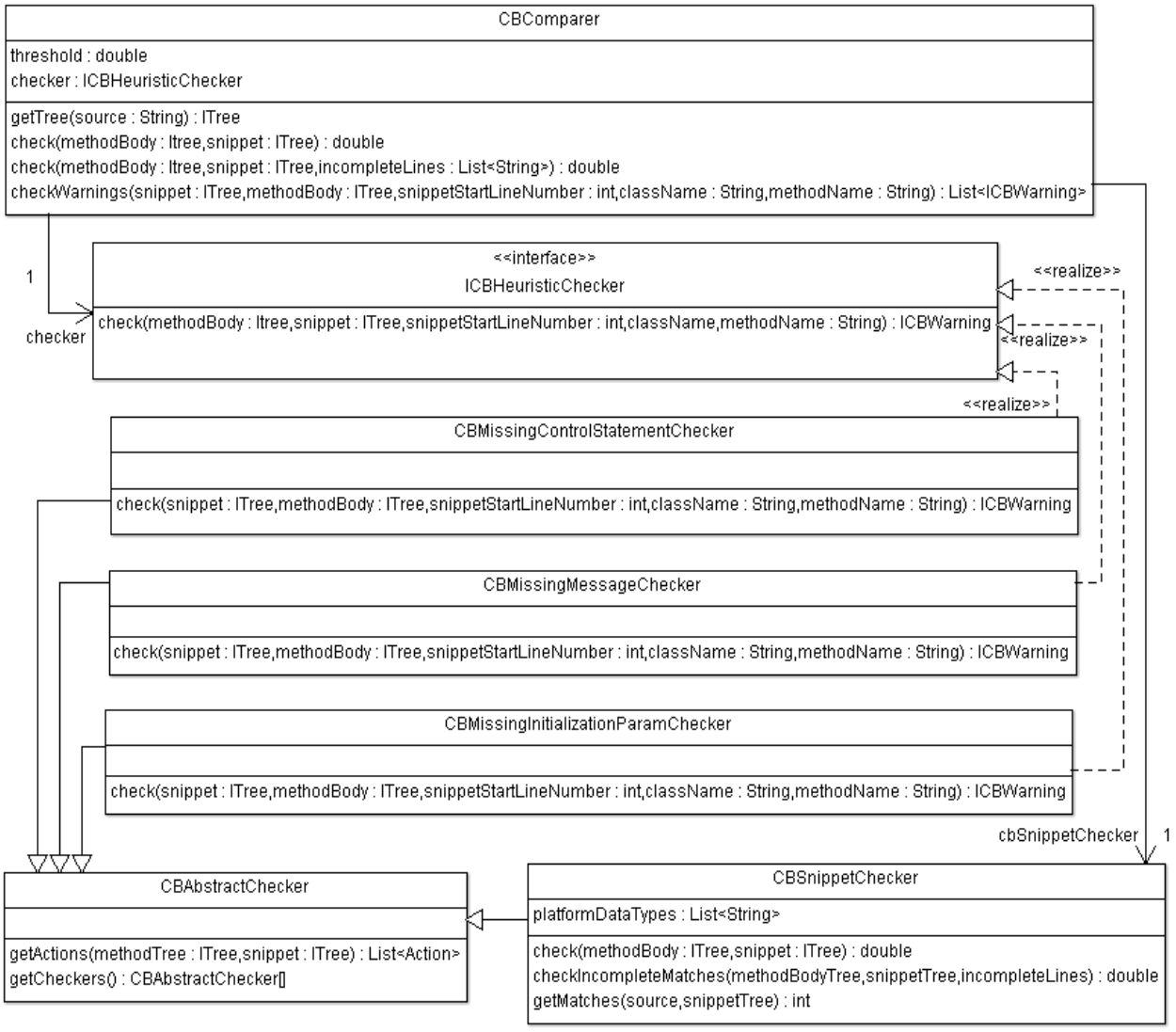

Figura 11. Diagrama de las clases del proyecto *Completion Core* que se encargan de comparar AST e identificar elementos faltantes.

El usuario del plug-in tiene la opción de insertar el ejemplo en su código fuente. Para evitar errores al insertar el ejemplo, se verifica que las dependencias a las que hace referencia se inserten también en el código del desarrollador. Las clases que se encargan de esta tarea se muestran en la Figura 12.

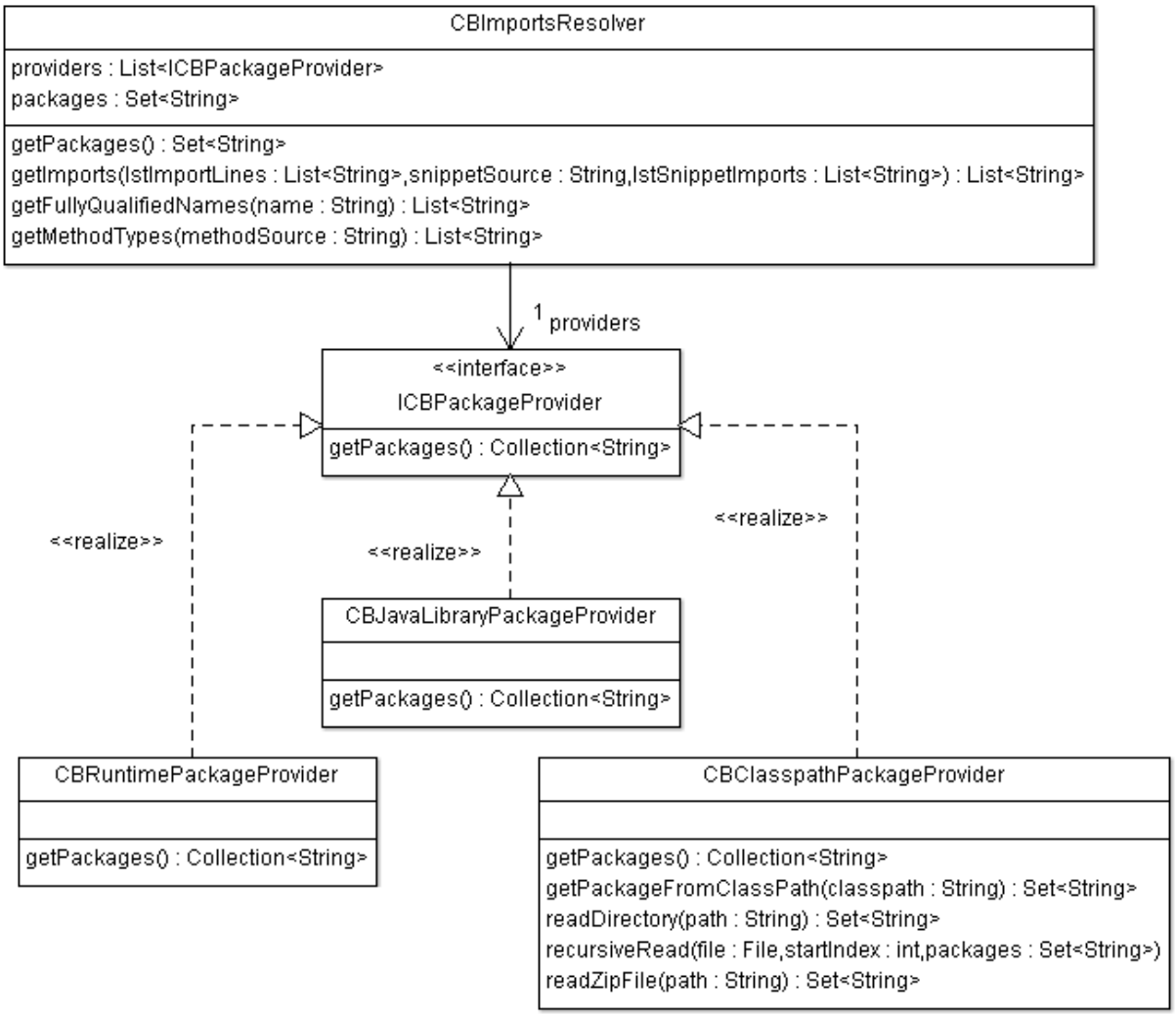

Figura 12. Diagrama de las clases del proyecto *Completion Core* que se encargan de resolver las dependencias.

Las responsabilidades de estas clases son:

- `CBImportsResolver`: Clase que se encarga de buscar los nombres de todos los paquetes que utiliza el proyecto del desarrollador. El método más importante de esta clase es `getPackages`. Se encarga de llamar al resto de las clases para cumplir la tarea principal de esta clase: obtener los nombres de las APIs importadas en el proyecto del desarrollador.
- `ICBPackageProvider`: Interfaz que implementan las clases que obtienen los paquetes que usa el proyecto del desarrollador.
- `CBRuntimePackageProvider`: Devuelve los nombres completos de los paquetes

que se resuelven en tiempo de ejecución.

- `CBJavaLibraryPackageProvider`: Devuelve los nombres completos de los paquetes propios de las APIs de Java.
- `CBClasspathPackageProvider`: Devuelve los nombres completos de los paquetes que son referenciados en los archivos .jar y folders del classpath.

4.4.4 Casos especiales en la comparación entre ASTs

Existen tres casos especiales que, por la forma en que está implementado el algoritmo de comparación, es preciso considerar. Si no se realizan las primeras dos verificaciones, esos casos serían ignorados y no sumarían al Grado de Similitud, porque normalmente las acciones *Update* no son consideradas para el cálculo. En el tercer caso se corrige la cantidad de cambios que el algoritmo indica que existe entre dos ASTs en particular.

1. Cuando se trata de invocaciones a métodos, el algoritmo considera como una acción de tipo *Update* de nombres de método lo que debería ser un *Insert* de un método nuevo. Por ejemplo, dados los siguientes bloques de código:

`InitCommand init = Git.init();`
`init.setDirectory(basedir);`
`Git git = Git.open(gitFile);`
	`InitCommand init = Git.init();`
	`init.setBare(false);`
	`init.setDirectory(basedir);`

Listado 17. Ejemplo del caso especial 1 en el cálculo del Grado de Similitud

   El algoritmo indica que para llegar de un árbol al otro, se debe actualizar el nombre del método `open` por `setBare`. Sin embargo, este no es un cambio de nombres, sino una eliminación del método e inserción de un nuevo método.

   Para que la solución considere este caso, se realiza una verificación en los nodos *Update*, de manera que si el nodo indica actualización de nombre de método, se verifique si se trata en realidad de una eliminación/inserción de llamada a método.

2. Cuando entre el código del desarrollador y el ejemplo del repositorio existe una declaración de objeto similar, el algoritmo de comparación considera que la diferencia es una actualización simple y sólo genera un nodo *Update*. Por ejemplo:

`Path directory =` `Paths.get("C:/");`	`File fileFromPath =` `Paths.get("test.txt").toFile();`

Listado 18. Ejemplo del caso especial 2 en el cálculo del Grado de Similitud

El algoritmo de comparación indica que para pasar del árbol de la izquierda al de la derecha bastan dos acciones: actualizar el tipo `Path` al tipo *File* e insertar la llamada al método `toFile`. Sin embargo, como en el caso anterior lo más indicado sería decir que hay que es necesario eliminar la variable de tipo `Path` e insertar una variable de tipo *File*.

La manera de manejar este caso es: identificar las acciones *Update* cuyo nodo asociado sea de tipo `SimpleType`, luego obtener el nombre del tipo que hay que actualizar y buscar ese tipo en árbol del código fuente del ejemplo. Si el tipo existe, se añade una unidad al Grado de Similitud.

3. Cuando se tiene una acción *Insert* de tipo `ExpressionStatement` en ambos árboles, estas acciones entre sus nodos hijos incluyen una invocación al mismo método y las instancias que lo invocan tienen distintos nombres, el algoritmo considera que se debe eliminar y volver a insertar la llamada al método, pese a que es el mismo. Por ejemplo:

`FileInputStream fis = new FileInputStream(sourceFile);`	`FileInputStream fis = new FileInputStream(sourceFile);`
`FileOutputStream fos = new FileOutputStream(destFile);`	`FileOutputStream fos = new FileOutputStream(destFile);`
`FileChannel srcChannel = **fis.getChannel();**`	`FileChannel srcChannel = **fos.getChannel();**`

Listado 19. Ejemplo del caso especial 3 en el cálculo del Grado de Similitud

En este caso, el algoritmo de comparación indica que para pasar del árbol de la izquierda al árbol de la derecha se debe eliminar la invocación al método `getChannel`, actualizar el nombre de la variable e insertar una llamada al método `getChannel` para esa variable. Por lo tanto, suma 2 unidades al Grado de Similitud, por la inserción y eliminación. Esto no es correcto, pues se trata simplemente de una actualización de nombre de variable y no debería sumar unidad alguna.

Como solución a estos casos, se busca si alguno de los hijos del nodo `ExpressionStatement` es una invocación a método. De ser así, se busca entre las acciones *Delete* e *Insert* un par de acciones que indiquen que el nodo con el nombre de variable debe primero eliminarse y luego volver a insertarse. Si existen estas acciones, no se toma en cuenta la acción para el cálculo de Grado de Similitud, caso contrario, se suma una unidad, como normalmente se hace con las acciones *Insert*.

4.4.5 Procesamiento de código no compilable

Una línea de código que no compila no puede convertirse correctamente en nodos del AST. Estas líneas también deben ser consideradas en el cálculo del Grado de Similitud, de hecho, es muy probable que tengan alta relevancia, al ser el código que el desarrollador no termina de escribir y necesita ayuda para completar.

Definimos que la manera de procesar estas líneas es separarlas del resto del código compilable, tomar los términos que las componen (ya sean nombres de tipos, variables, estructuras, etc.) y buscar estos términos en los snippets. Cada vez que se encuentra una coincidencia, se contabiliza y resta al Grado de Similitud.

Considerando estos casos, la fórmula se modifica a:

$$Gd(D, E) = 1 - (\sum v(a_i) - 2 * C(D, E) - 2 * cnc) / j, \forall a_i \in D$$

Donde:

cnc = Cantidad de veces en que los nodos formados a partir del código no compilable del desarrollador se encuentran en el ejemplo del repositorio.

4.5 Presentación de los resultados de búsqueda al desarrollador

Luego de haber realizado la búsqueda de ejemplos del repositorio y antes de mostrar los ejemplos al desarrollador, se toma los ejemplos con mayor similitud y se busca los elementos (estructuras de control, llamadas a métodos, parámetros en la creación de objetos), según se explica en la Sección 3.2.3 del Capítulo 3 de este documento. La implementación de esta verificación se realiza de la siguiente forma:

- **Estructuras de control:** la búsqueda de estructuras de control faltantes se realiza a partir de la secuencia de acciones que resulta de la comparación entre un ejemplo del repositorio y el código del desarrollador. Se seleccionan todos los nodos *Insert* y se examinan uno a uno. Si el tipo del nodo es una estructura de control (`while`, `for`, `foreach`, etc.), se lleva a una lista de pares clave-valor el nombre de la estructura de control y la línea en que aparece.
- **Llamadas a métodos:** la búsqueda de llamadas a método faltantes se realiza también a partir de la secuencia de acciones que resulta de la comparación entre un ejemplo del repositorio y el código del desarrollador. Primero se seleccionan los nodos *Insert* y se examinan uno a uno. Si el tipo del nodo es `MethodInvocation`, se lleva a una lista de pares clave-valor el nombre de la estructura de control y la línea en que aparece. A continuación se seleccionan los nodos *Update*, para abarcar los casos especiales y se siguen los pasos siguientes:
    1. Verificar que el tipo de nodo sea `SimpleName`, lo que significa un cambio de nombre, si es así continuar con el paso siguiente
    2. Obtener el nombre del método

3. Recorrer los nodos *Insert* buscando otro nodo que haga referencia al método cuyo nombre se guardó en el paso 2. En caso de que no exista, seguir con el paso 4
4. Buscar si existe un nodo de tipo `MethodInvocation` que tenga el nombre obtenido en el paso 2, de ser así, se confirma que este método falta en el código del desarrollador. Entonces, se lleva a una lista de pares clave-valor el nombre del método y la línea en que aparece

- **Parámetros en la creación de objetos:** la búsqueda de parámetros faltantes se realiza también a partir de la secuencia de acciones que resulta de la comparación entre un ejemplo del repositorio y el código del desarrollador. Se examinan todos los nodos *Insert*, buscando aquellos cuyo tipo sea `SimpleName`. Por cada nodo encontrado se verifica que su nodo padre sea de tipo `ClassInstanceCreation`. Si esta condición se cumple, se lleva a una lista de pares clave-valor el nombre del parámetro y la línea en que parece.

Luego de realizar esta verificación, los ejemplos del repositorio son enviados al plug-in para que se encargue de mostrarlos al desarrollador.

4.5.1 Implementación de la interfaz gráfica del plug-in

La técnica propuesta por la presente tesis se implementa en un plug-in para el entorno de desarrollo Eclipse. La plataforma de Eclipse permite a los desarrolladores extender su funcionalidad mediante plug-ins. Mediante los plug-ins es posible contribuir a funcionalidades existentes, por ejemplo, crear una nueva entrada en el menú; o crear entornos de desarrollo totalmente nuevos.

El plug-in es denominado *Completion plug-in*. Tiene las funciones de: mostrar los ejemplos de código, resaltar los elementos faltantes e insertar el código del ejemplo en el editor.

Los ejemplos que retorna el proyecto *Completion Core* se muestran en el visor de snippets (Figura 13). En el panel izquierdo está una lista con los snippets encontrados, en orden descendente de similitud. Al hacer clic en uno de los snippets, en el panel de la parte derecha se muestra el código, donde los elementos que podrían faltar en el código del desarrollador están resaltados en rojo.

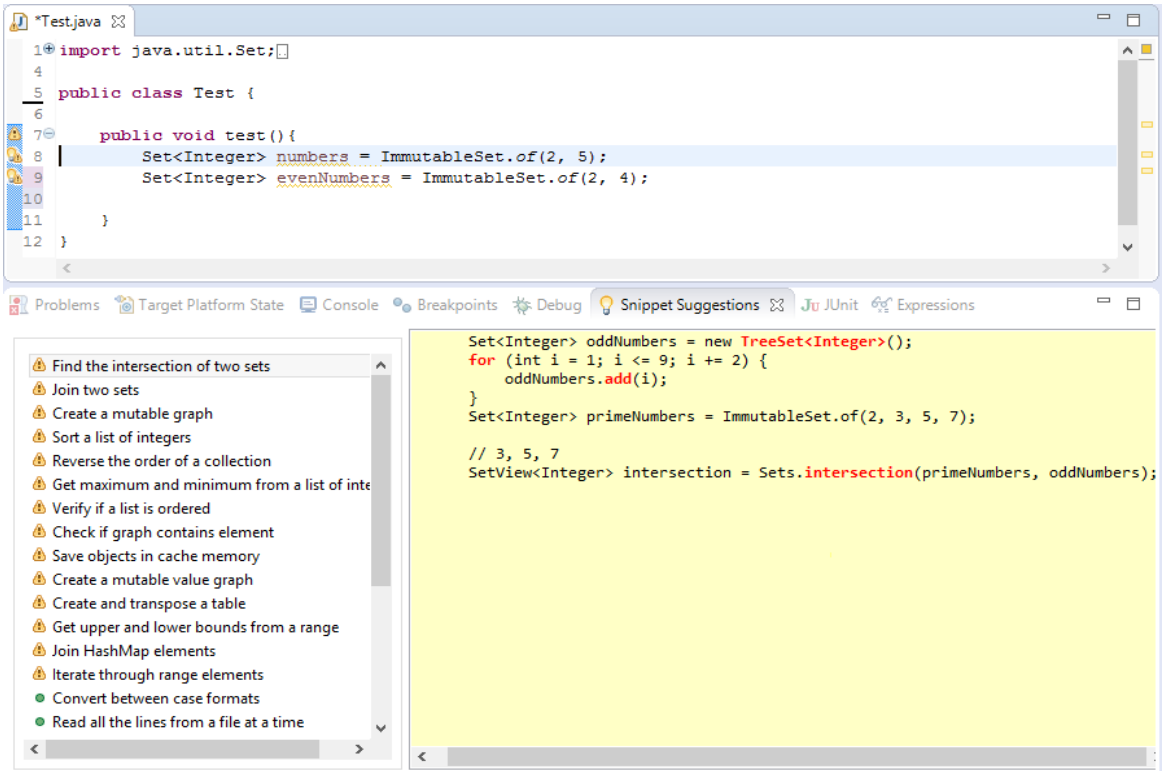

Figura 13. Visor de snippets.

El plug-in muestra un marcador de alerta en el editor de código fuente si se encontraron elementos que pueden faltar en el código del desarrollador (Figura 14). Al pasar el ratón por el marcador de alerta, éste indica en un mensaje si se encontraron estructuras de control, parámetros o llamadas a métodos que no aparecen en el código del desarrollador.

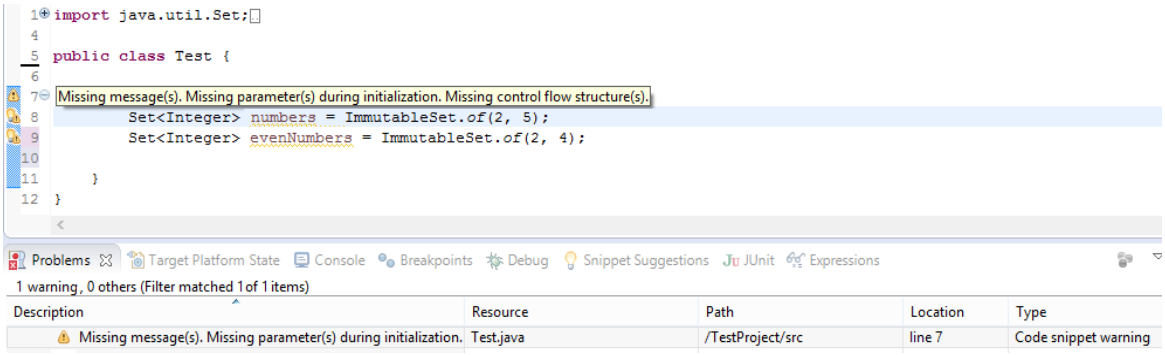

Figura 14. Editor de código fuente.

4.6 Integración de un ejemplo del repositorio en el código del desarrollador

Aparte de consultar los ejemplos que muestra el plug-in, el desarrollador puede insertarlos en su código fuente. El plug-in también es responsable de insertar los ejemplos en el código del desarrollador. Al hacer doble clic sobre el nombre de un ejemplo en el visor de snippets se inserta en la posición actual donde se dejó el cursor. Además del código propio del ejemplo del repositorio, se insertan las sentencias import necesarias para que el ejemplo compile correctamente al ser integrado.

Capítulo 5 - Validación de la solución propuesta

En este capítulo se describe la metodología empleada para la validación de la utilidad de la solución desarrollada, que se realizó mediante un experimento con desarrolladores de software.

En la primera sección se describe de manera general el objetivo del estudio para validar el plug-in, los participantes y las tareas que se les pidió realizar. A continuación se explica las características del estudio piloto que se utilizó para revisar el diseño de la validación. Finalmente se detalla la manera en que fue conducido el experimento.

5.1 Diseño de la validación

5.1.1 Objetivo

Analizar el uso de la solución que hacen los desarrolladores que necesitan usar una API que no conocen con exactitud. Además, compararlo con la búsqueda de ejemplos en Internet. Esto con la finalidad de estudiar la utilidad, en términos de tiempo y cantidad de defectos, de la solución propuesta.

5.1.2 Participantes

Se realizó una invitación abierta a desarrolladores de software a participar del experimento. Se definió el perfil requerido para participar como:

- Un mínimo de 3 años de experiencia en Java: para asegurar que los voluntarios entiendan los ejemplos y puedan integrarlos en su código sin dificultad.
- Conocimiento del manejo del entorno de desarrollo Eclipse: dado que el plug-in fue desarrollado para este entorno de desarrollo.
- No conocer o no haber utilizado las APIs Java NIO o Google Guava dentro del último año, este requerimiento está directamente relacionado con el objetivo de la solución, que es ayudar a desarrolladores que no conocen o no recuerdan el uso de una API.

5.1.3 Materiales

A los participantes se les distribuyó un cuestionario con dos secciones. La primera contiene preguntas generales sobre su experiencia con Java y las APIs a utilizar y la segunda sección tiene preguntas sobre la experiencia y comentarios luego de haber utilizado el plug-in. La lista de preguntas del cuestionario se encuentra en el Anexo B.

Además del cuestionario, los participantes recibieron una copia del plug-in, para instalar en sus versiones de Eclipse, además de un proyecto con dos clases y dos casos de prueba en JUnit. Cada clase tiene un método en blanco para completar durante el experimento y los casos de prueba ayudan al participante a comprobar si logró el resultado esperado.

5.1.4 Tareas

Se definió que el experimento tiene dos partes y se elaboraron dos tareas, una para cada parte:

Tarea 1: Obtener una lista con las rutas de los archivos que están en la carpeta principal de documentos del usuario (Por ejemplo: " Documentos" para sistemas operativos Windows). Ejemplo:

C:\Users\Usuario\.RData

C:\Users\ Usuario \.RHistory

C:\Users\ Usuario \All

C:\Users\ Usuario \CSVer.dll

C:\Users\ Usuario \Default.rdp

C:\Users\ Usuario \desktop.ini

Luego, crear un archivo con el nombre "test.txt" en la raíz de una de las unidades del equipo, por ejemplo, la unidad "D://" y escribir en el archivo esta lista.

Tarea 2: Crear dos sets de números, uno con los primeros diez pares y el segundo con los primeros diez factoriales. Luego, encontrar la intersección de estos conjuntos y mostrar sus elementos en pantalla, separados por comas. Ejemplo:

Set 1: {2,4,6,8,10}

Set 2: {1,2,6,24,120,720,5040,40320,362880,3628800}

Set intersección de 1 y 2: {2,6}

Mostrar en pantalla: "2,6"

Ambas tareas requieren combinar llamadas a las APIs, en el caso de la tarea 1 a Java NIO y en el caso de la tarea 2 a Google Guava. Las llamadas necesarias para completar las tareas estaban contenidas en 4 ejemplos del repositorio (por cada tarea), por lo tanto los participantes tuvieron a disposición el código necesario para realizarlas, combinando varios ejemplos.

5.1.5 Hipótesis

Las hipótesis que planteamos probar con el experimento están estrechamente relacionadas con el objetivo de este proyecto y consisten en que la utilización de un repositorio de ejemplos de buen uso de las APIs:

H1: Permite al desarrollador que no conoce o no recuerda cómo usar una API completar una tarea de programación más rápido que si se recurre a Internet para buscar los ejemplos.

H2: Ayuda al desarrollador que no conoce o no recuerda cómo usar una API a evitar defectos en su código fuente.

5.1.6 Estudio piloto

El estudio piloto es el primer paso para la validación del plug-in implementado. Es una versión pequeña del experimento que se realizará posteriormente. Este estudio fue planificado una vez que se tuvo una clara visión del objetivo y el método para validar el plug-in.

5.1.6.1 Objetivos

El objetivo general del estudio piloto es proveer información y retroalimentación que contribuya al éxito del experimento. Es decir, ahorrar el tiempo y esfuerzo que implicaría un experimento fallido por defectos en los instrumentos o el método.

Los objetivos específicos de realizar este estudio son:

- Evaluar la dificultad de las tareas del experimento: observar si los participantes logran terminar las tareas y medir el tiempo que les toma.
- Detectar posibles fallas en el método: probar que las instrucciones son suficientemente claras para los participantes y que el tiempo destinado a las tareas es el apropiado.
- Identificar preguntas ambiguas en el cuestionario: verificar si los participantes tienen dificultad en comprender alguna de las preguntas.

Estos objetivos ayudan a disminuir el riesgo de falla en el experimento.

5.1.6.2 Desarrollo

Invitamos a dos voluntarios que cumplen los requisitos descritos en la Sección 5.1.2. Aleatoriamente se determinó que el primer voluntario (V1) usaría el plug-in para la tarea 1 y la búsqueda online para la tarea 2, mientras que el segundo voluntario (V2) lo haría de manera inversa.

Se explicó a ambos voluntarios el propósito del plug-in y se les permitió usarlo durante unos minutos antes de comenzar. Ambos se familiarizaron rápidamente con el uso y pudieron completar las tareas sin mayores problemas. Además se les pidió llenar un cuestionario con dos secciones, la primera para ser llenada antes de

comenzar y la segunda para después de haber finalizado con las tareas de programación.

5.1.6.3 Resultados

A partir de los cuestionarios que se les pidió llenar y los comentarios recibidos de los voluntarios al terminar el piloto, obtuvimos las siguientes conclusiones:

- El tiempo establecido para las tareas fue de 20 minutos. V1 terminó ambas tareas en aproximadamente un tercio del tiempo; sin embargo, V2 lo hizo en el tiempo exacto.
- El tiempo de respuesta del plug-in es aceptable, al consultar a los voluntarios, ninguno reportó lentitud al invocarlo.
- Los voluntarios invirtieron tiempo en crear un proyecto Java, que incluyera las APIs a utilizar. Además V2 necesitó una aclaración sobre la estructura deseable para el proyecto y si era posible realizar tests. Estos dos aspectos no fueron considerados antes del estudio.
- Durante el estudio piloto fue evidente que los voluntarios pudieron completar las tareas rápidamente con el plug-in; sin embargo, se observó que las sugerencias presentadas primero no siempre fueron las que realmente el usuario necesita.

5.1.6.4 Aplicación de los resultados

A continuación se describen los ajustes realizados a la metodología e instrumentos, como consecuencia del análisis de la información recibida durante el estudio piloto.

- El tiempo máximo establecido para completar las tareas se amplía a 30 minutos por tarea.
- En lugar de pedir a los voluntarios que creen el proyecto durante el experimento, se crea un sólo proyecto Java que es distribuido junto a los demás materiales. Este proyecto incluye las APIs Java NIO y Google Guava, además de tests unitarios para que los voluntarios tengan claro que han completado la tarea cuando los tests se ejecutan correctamente.
- Además de medir el tiempo y la correctitud del código fuente desarrollado, se evalúa la relevancia en el orden de las sugerencias que presenta el plug-in. Esto mediante la grabación de la sesión de cada voluntario, donde se verifica cuál de los ejemplos elige luego de haber invocado al plug-in.

Adicionalmente, se hicieron leves ajustes a la interfaz de usuario del plug-in y a la descripción de un ejemplo.

5.3 Desarrollo del experimento

Seis desarrolladores con el perfil requerido respondieron a la convocatoria. La Figura 15 muestra un resumen de las características de los participantes. Como indica la figura, la mayoría de los desarrolladores tienen entre 3 a 5 años de experiencia programando en Java y durante su vida profesional han desarrollado más de 6 proyectos. Por otra parte, la mayoría reportó no tener experiencia con las APIs Java NIO y Google Guava y aquellos que tienen poca experiencia no la habían utilizado dentro del último año.

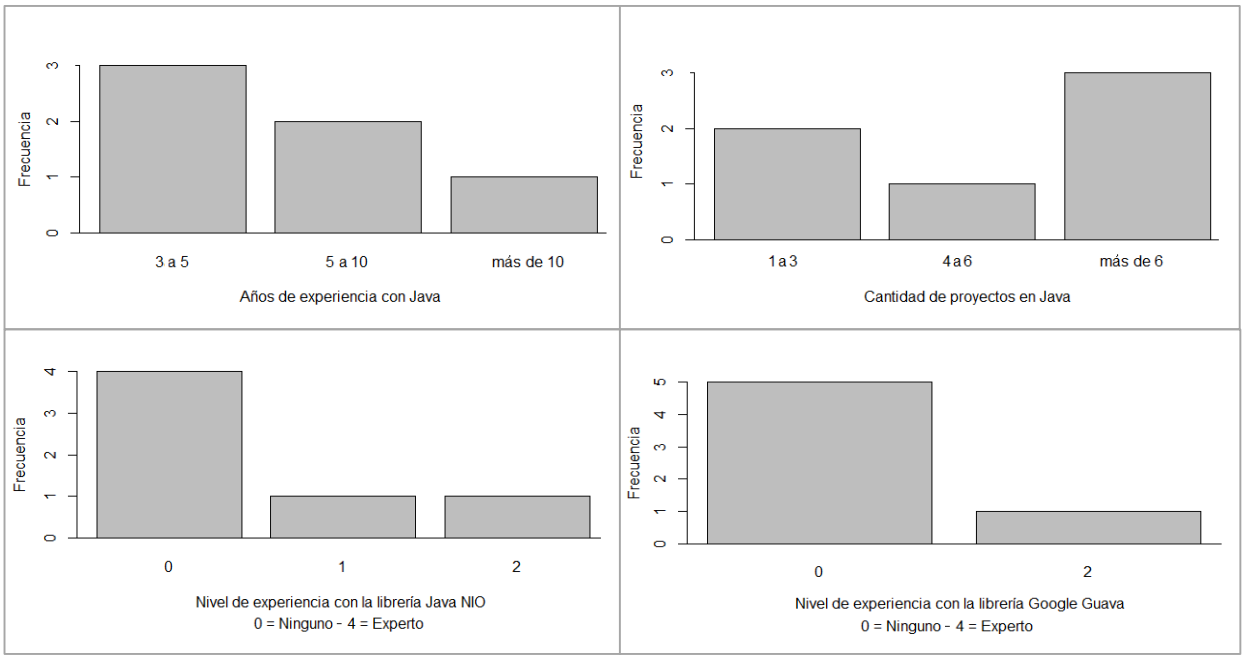

Figura 15. Características de los participantes del experimento.

El experimento se condujo de manera individual. Se asignó aleatoriamente a tres de los desarrolladores realizar la tarea 1 utilizando el plug-in, mientras que los otros 3 usaron el plug-in para resolver la tarea 2. En la primera parte de cada sesión con los distintos desarrolladores se dio una breve explicación de la tarea 1, para asegurar que cada participante entendió perfectamente lo que tenía que hacer. Se instruyó además sobre el uso del plug-in, sólo a los desarrolladores que les correspondía usarlo para la tarea 1 y se pidió que la utilicen durante unos minutos para familiarizarse. En cambio, a los participantes que les correspondía usar el plug-in en la Tarea 2, se les pidió resolver la Tarea 1 buscando ejemplos en Internet, en cualquier página que consideren útil.

Antes de comenzar con la tarea 1, se tomó nota de la hora de inicio y se activó la grabación de la sesión; mediante una aplicación que captura la pantalla del computador.

Una vez que un participante terminó con la tarea 1, se tomó nota de la hora de finalización y se continuó con la tarea 2. En esta parte, aquellos participantes que debían utilizar el

plug-in para realizar la tarea recibieron las instrucciones de uso y se les pidió que prueben utilizarlo.

Tal como ocurrió con la tarea 1, la tarea 2 fue grabada y se tomó nota de la hora de inicio y finalización, indicando a los voluntarios que para ambos casos el tiempo máximo era de 30 minutos. Sin embargo, cuando un participante no terminó en el tiempo establecido, se le permitió continuar hasta concluir con la tarea, con el objetivo de medir su tiempo final y no carecer de ese dato.

Después de resolver las tareas de programación, los participantes llenaron el cuestionario del Anexo B. Las respuestas a estos cuestionarios, junto a las grabaciones de las sesiones se guardaron para analizar y extraer conclusiones.

5.4 Análisis de los resultados

5.4.1 Resultados de la Tarea 1

Los participantes que utilizaron la solución completaron la tarea exitosamente, dentro del tiempo establecido. El participante P6, que utilizó la búsqueda de ejemplos en Internet, no logró completar la tarea a tiempo.

En promedio, demoraron 14 minutos con 13 segundos en realizar la Tarea 1 utilizando la solución y 31 minutos con 20 segundos realizando la búsqueda de ejemplos en Internet, como se observa en la Figura 16.

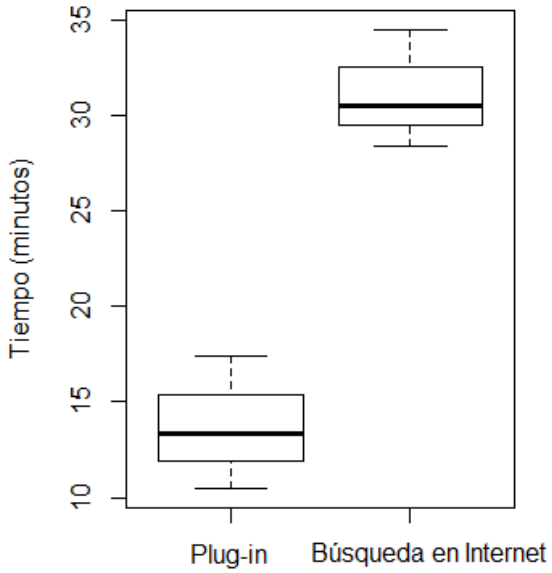

Figura 16. Tiempos de los desarrolladores al completar la Tarea 1.

Luego de revisar el código fuente de todos los participantes, no se encontró potenciales defectos ni bloques que hagan mal uso de la API.

Los participantes invocaron al plug-in un promedio de 2.67 veces para la Tarea 1.

Al navegar en Internet para buscar ejemplos, los participantes visitaron un promedio de 8.3 sitios para esta tarea. Los sitios más comunes que visitaron, tanto para esta tarea como para la siguiente, fueron: el sitio de documentación oficial de las APIs y el foro dedicado a temas de programación Stack Overflow.

5.4.2 Resultados de la Tarea 2

Todos los participantes que utilizaron la solución completaron la tarea a tiempo. El participante P1, que usó la búsqueda de ejemplos en Internet no logró completar la tarea en el tiempo establecido. En promedio, los participantes demoraron 11 minutos con 39 en realizar la tarea utilizando la solución y 21 minutos con 16 segundos buscando ejemplos en Internet, tal como se observa en la Figura 17.

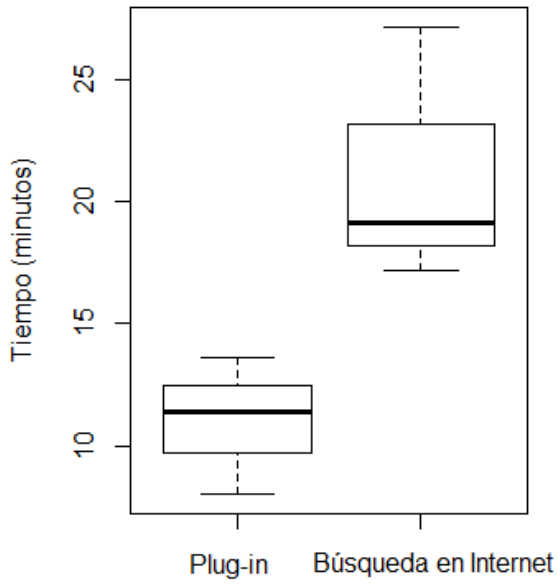

Figura 17. Tiempos de los desarrolladores al completar la Tarea 2.

Los participantes invocaron al plug-in un promedio de 3 veces para la Tarea 2.

Al navegar en Internet para buscar ejemplos, los participantes visitaron un promedio de 10.3 sitios para la Tarea 2 (Ver Figura 19).

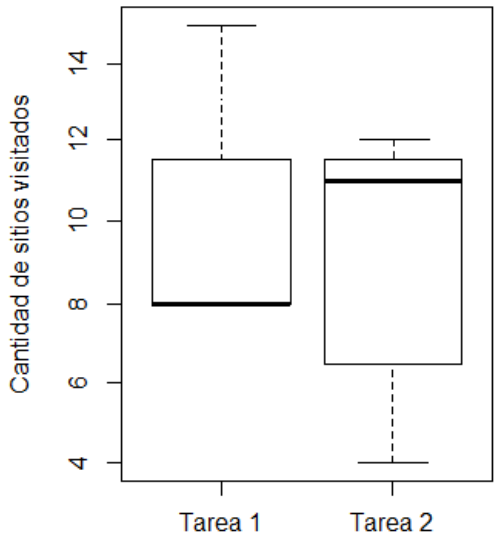

Figura 18. Número de sitios visitados durante la búsqueda de ejemplos.

Se revisó el código fuente entregado por los participantes, encontrando en dos ocasiones un indicador de mal uso de la API:

```
1       Set<BigInteger> evens = Sets.newHashSet();
2       int number = 2;
3
4       while(evens.size() <10){
5             if (number % 2 == 0) {
6                   BigInteger bigint = BigInteger.valueOf(number);
7                   evens.add(bigint);
8             }
9             number++;
10      }
```

Listado 20. Código fuente que contiene una mala práctica de programación

En un bloque similar al del ejemplo, dos participantes realizaron una conversión forzada al tipo *BigInteger*. Esto fue causado porque el ejemplo que encontraron en Internet para obtener el factorial de un número utilizaba el tipo *BigInteger* en lugar de *Integer*.

5.4.3 Resultados relativos a ambas tareas

Al navegar por los ejemplos de ambas tareas, el ejemplo deseado apareció entre los primeros 5 el 71% de las veces, siendo la precisión más alta en la primera invocación al plug-in. Como muestra la Figura 18, en la primera invocación al plug-in que hizo cada participante, el ejemplo buscado apareció en la primera posición en el 83% de las veces. En la segunda invocación, apareció entre los primeros 5 el 67% de las

veces y en la tercera invocación, apareció entre los primeros 6 ejemplos el 60% de las veces.

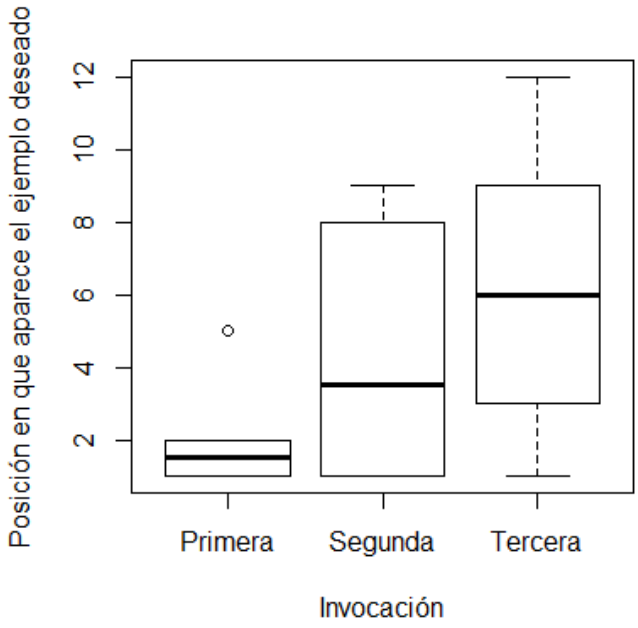

Figura 19. Relevancia de los ejemplos presentados por el plug-in.

En el cuestionario que se proporcionó a los participantes se consultó su opinión sobre el plug-in: 5 de 6 desarrolladores afirmaron estar de acuerdo con que el plug-in es fácil de usar e intuitivo, todos estuvieron de acuerdo con que es rápido, 3 estuvieron de acuerdo con que los ejemplos más relevantes aparecen primero y 5 manifestaron interés en construir sus propios ejemplos.

Por otra parte, los participantes P1, P4 y P5 sugirieron incorporar el visor de ejemplos al menú contextual del editor de código fuente, de manera que el funcionamiento sea lo más parecido al menú de sugerencias de código que viene incorporado en el entorno de desarrollo. El participante P2 hizo notar que los ejemplos siguen estándares de codificación relativamente distintos a los que él utiliza en su equipo de desarrollo.

Por lo tanto se verifica que H1 y H2 son ciertas, ya que los participantes que programaron con ayuda de la solución completaron las tareas más rápido y tuvieron menos defectos en su código fuente, comparados con los participantes que resolvieron las tareas buscando ejemplos en Internet.

5.5 Discusión

Una razón por la que los desarrolladores completaron las tareas más rápido usando la solución es que el repositorio tenía todos los ejemplos necesarios para completar las tareas de programación. Por lo tanto, la efectividad de esta herramienta depende en gran medida de que el repositorio se mantenga actualizado con los ejemplos de uso de las APIs que el

desarrollador necesita. Para alentar a los desarrolladores a contribuir con el repositorio, se ha procurado que los pasos a seguir para agregar un nuevo ejemplo sean sencillos y no requieran conocimientos adicionales. Por otra parte, según manifestaron los voluntarios del experimento, existe una inquietud general de elaborar un conjunto de ejemplos con las funciones más comunes que cada uno utiliza.

El aspecto más importante donde queda trabajo por hacer es en el orden que son presentados los ejemplos. Cuando los métodos tienen pocas líneas de código, resulta sencillo realizar la comparación y búsqueda de ejemplos; sin embargo, al tener métodos más extensos, se dificulta determinar qué ejemplo es el que el desarrollador desea. La versión actual del plug-in ya propone una solución a este problema durante la comparación entre el código fuente del desarrollador y el del ejemplo, buscando los métodos y tipos de datos en común y asignándoles mayor preferencia que al resto de elementos. De esta manera, aunque el código del desarrollador sea extenso, los elementos con preferencia facilitan encontrar el ejemplo deseado. Afinando este grado de preferencia, se estima mejorar la relevancia de los ejemplos que se presentan primero al desarrollador.

Existen casos que el enfoque propuesto no maneja adecuadamente. Por ejemplo, cuando el método que está programando el desarrollador realiza múltiples tareas. El plug-in al realizar la comparación podría encontrar similitud con varios ejemplos no relacionados. Una posible solución a este problema es permitir al desarrollador seleccionar las líneas en su código fuente sobre las que desea ver ejemplos.

Otro escenario donde existen limitaciones es en el manejo de tipos dinámicos. Esto se debe a que la comparación entre ASTs se realiza compilando el código fuente, pero sin ejecutarlo.

5.6 Amenazas a la validez

Validez interna: la validez interna se refiere a la validez de las conclusiones en un entorno experimental. La amenaza principal que encontramos a la validez interna está relacionada con el efecto que tuvo la habilidad técnica de los desarrolladores de software sobre su comportamiento durante el experimento. Los resultados obtenidos miden el desempeño de los participantes en el uso del plug-in. Al margen de la herramienta en sí, su rendimiento dependió también de la experiencia y pericia con el lenguaje y las APIs utilizadas, lo que pudo influenciar a que los resultados sean más o menos favorables. Para reducir el impacto de este factor, se definió un perfil para los voluntarios: al menos 3 años de experiencia con Java, conocimiento del manejo de Eclipse y no haber utilizado recientemente (al menos en el último año) las APIs Java NIO ni Google Guava. En una extensión de este trabajo, se plantea realizar múltiples validaciones con desarrolladores de distinto nivel de experiencia, para analizar el impacto que tiene en los resultados.

Validez externa: la validez externa está relacionada con cuán generalizables son los resultados más allá del contexto donde se aplicó la solución. Nuestro estudio se limitó a

usar dos APIs: Java NIO y Google Guava. Éstas son muy populares y ampliamente utilizadas por los desarrolladores de software. Sin embargo, aunque los resultados son optimistas, no es posible generalizarlos a otras APIs. Esta amenaza se puede reducir como trabajo futuro con nuevos experimentos en que se valide el plug-in con ejemplos de más APIs.

Capítulo 6 – Conclusiones

En este capítulo se presenta un resumen del proyecto y sus contribuciones, se realiza una discusión sobre las lecciones aprendidas y se describe mejoras a ser tomadas en cuenta para futuras fases del proyecto.

6.1 Resumen del proyecto

El repositorio de ejemplos y el plug-in han sido creados para ayudar a los desarrolladores que se enfrentan a APIs que no conocen o no han usado recientemente. Al crear un repositorio con ejemplos y un mecanismo para buscar, navegar e integrar los ejemplos en el código fuente, proveemos al desarrollador de una herramienta que puede ayudarle a aprender el uso de la API, aliviando la carga que significa buscar ejemplos en otras fuentes de información.

Por otra parte, se ha construido un repositorio con un total de 25 ejemplos de buen uso de la API Java NIO y 25 más para la API Google Guava. Al poner a disposición de la comunidad de desarrolladores el repositorio junto con la documentación correspondiente, se apunta a que el repositorio se mantenga actualizado y en crecimiento.

Nuestra evaluación inicial, realizada con 6 desarrolladores de software, reporta resultados favorables, en cuanto al ahorro de tiempo que significa a los desarrolladores usar el plug-in y la reducción en la cantidad de defectos, en comparación con la búsqueda ejemplos en Internet.

Quedan aspectos por mejorar, particularmente en cuanto al orden en que se presentan los ejemplos al desarrollador. Avanzando en esta dirección se espera entregar a la comunidad de desarrolladores una herramienta fácil de usar y rápida, que sea de utilidad al momento de programar usando APIs específicas.

6.2 Contribuciones

Durante la ejecución del proyecto, se lograron las contribuciones que se explican a continuación.

Se ha tomado una técnica de comparación de ASTs existente [26] y adaptado para realizar el cálculo del denominado Grado de Similitud entre un ejemplo del repositorio y el código que está programando el desarrollador. Parte de esta contribución está en la identificación de casos especiales, donde la comparación existente no contabiliza correctamente la diferencia entre los ASTs, y el procesamiento de código no compilable, que por defecto no era considerado en la comparación.

Como contribución técnica, se provee la implementación de un plug-in para el entorno de desarrollo Eclipse. El plug-in realiza la búsqueda en un repositorio de los ejemplos que son más similares al código que el desarrollador está programando, resalta los elementos (llamadas a métodos, estructuras de control, parámetros) que no están en el código del desarrollador y facilita la integración de los ejemplos en el código fuente.

Finalmente, se ha construido un repositorio de ejemplos de buen uso de las APIs, alojado en la plataforma GitHub. Se ha definido un formato para los ejemplos, documentado y publicado junto a instrucciones para agregar ejemplos al repositorio. Además, la implementación inicial del repositorio cuenta con 50 ejemplos, donde 25 ejemplifican el uso de la API Java NIO y 25 corresponden a Google Guava.

6.3 Lecciones aprendidas

En las primeras fases del proyecto se tomaron varias decisiones, cuyos efectos se vieron en fases posteriores. Esto permitió aprender sobre distintos aspectos, los más importantes se describen en los siguientes párrafos.

Respecto del diseño de la solución, se tomó la decisión de separar en proyectos distintos la lógica de la búsqueda de ejemplos y las clases que manejan la interfaz de usuario. Esta consideración técnica favorecerá a que en un futuro se pueda portar la solución a otros entornos de desarrollo que tienen distintos componentes para la interfaz de usuario.

Una decisión que llevó a enfrentar ciertas limitaciones fue definir que la sugerencia de ejemplos sea a partir de la comparación entre los ASTs de los ejemplos de un repositorio y el código fuente del desarrollador. En esa etapa no se conocía las limitaciones que tienen los ASTs, por ejemplo que no pueden ser generados a partir de código no compilable o que no es fácil determinar la similitud entre dos expresiones que son semánticamente equivalentes a partir de sus AST. Las alternativas de solución que se implementaron para superar estas limitaciones se describen en el capítulo de implementación de este documento.

6.4 Trabajo a futuro

Uno de los aspectos más importantes en que hay trabajo por hacer es la precisión en el orden de los resultados de la búsqueda. Se observó en el estudio con usuarios que la precisión en los resultados de la búsqueda de ejemplos disminuye gradualmente en función a la cantidad de líneas de código existentes en el método donde se invoca al plug-in. En una siguiente etapa del proyecto, se propone mejorar el algoritmo de búsqueda de ejemplos enfocándose en la precisión de los resultados.

Por otra parte, según reportan los resultados del estudio realizado, 5 de los 6 desarrolladores están a favor de elaborar sus propios ejemplos y manifestaron la intención de armar repositorios locales en sus computadores. En lugar de tener que descargar todos

los ejemplos del repositorio, es deseable que cada desarrollador pueda guardar en su equipo sólo el conjunto de ejemplos que considere necesarios.

Otra mejora que tiene relación con el repositorio es la integración del repositorio con otros repositorios externos existentes. Esta implementación se propone para una próxima fase del proyecto, dado que la utilidad de la solución propuesta depende en gran medida de que el repositorio tenga una variedad de ejemplos de uso de las APIs.

En cuanto a la validación, en esta fase del proyecto se realizó con seis desarrolladores. Sin embargo, para poder probar de manera más concluyente las hipótesis, en una siguiente fase se planea contar con una muestra más grande de usuarios.

Finalmente, debido a que el plug-in está implementado exclusivamente para el entorno de desarrollo Eclipse, para llegar a un mayor número de usuarios, se plantea adaptarlo a otros entornos como IntelliJ IDEA o NetBeans.

Bibliografía

[1] Ekwa Duala-Ekoko and Martin P. Robillard, “Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study”, en *Proceedings of the 34th International Conference on Software Engineering*, 2012, págs. 266-276.

[2] Martin P. Robillard, “What makes APIs hard to learn? Answers from developers”, *Journal IEEE Software*, vol. 26, issue 6, 2009, págs. 27–34.

[3] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, Scott R. Klemmer, “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code”, en Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2009, págs. 1589-1598.

[4] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, Scott R. Klemmer, “Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover”, *Journal IEEE Software*, vol. 26, issue 5, 2009, págs. 18-24.

[5] Kai Pan, Sunghun Kim, E. James Whitehead Jr., “Toward an understanding of bug fix patterns”, *Empirical Software Engineering*, vol. 14, issue 3, 2009, págs. 286-315.

[6] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, Shan Lu, “Understanding and Detecting Real-World Performance Bugs”, en *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, págs. 77-88.

[7] Martin P. Robillard and Robert DeLine, “A field study of API learning obstacles”, *Journal Empirical Software Engineering*, vol. 16, issue 6, 2011, págs. 703–732.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques, and Tools”, Addison Weasley – Person Education Inc., Second Edition, 2007, págs. 69-70.

[9] TIOBE Software BV, “TIOBE Index for April 2016”, fecha de consulta: 4 de mayo de 2016, disponible en: http://www.tiobe.com/tiobe_index

[10] David Geer, “Eclipse Becomes the Dominant Java IDE”, *Journal Computer*, vol. 38, issue 7, 2005, págs. 16-18.

[11] Oracle, “New I/O APIs”, fecha de consulta: 4 de mayo de 2016, disponible en: https://docs.oracle.com/javase/8/docs/api/

[12] Google, “Guava: Google Core Libraries for Java”, fecha de consulta: 4 de mayo de 2016, disponible en https://github.com/google/guava

[13] Daniel Jacobson, Greg Brail, Dan Woods, “APIs: A Strategy Guide”, O’Reilly Media Inc., First Edition, págs. 6- 7, 99.

[14] Eclipse Foundation, Snippet “Write Object to File", fecha de consulta: 4 de mayo de 2016, disponible en:

http://git.eclipse.org/c/recommenders/org.eclipse.recommenders.snipmatch.snippets.git/tree/snippets/a26cc852-37c4-46f4-8e60-11c6eb6f6bf9.json

[15] Eclipse Foundation, Snipmatch Snippets Repository, fecha de consulta: 31 de marzo de 2017, disponible en: http://git.eclipse.org/c/recommenders/org.eclipse.recommenders.snipmatch.snippets.git/

[16] Repositorio de ejemplos de código fuente "Snippet Repo", fecha de consulta: 31 de marzo de 2017, disponible en: https://snippetrepo.com/

[17] Repositorio de ejemplos de código fuente "Smipple", fecha de consulta: 31 de marzo de 2017, disponible en: http://www.smipple.net/

[18] Marcel Bruch, Martin Monperrus, Mira Mezini, "Learning from Examples to Improve Code Completion Systems" en *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, 213, 222.

[19] Romain Robbes, Michele Lanza, "How Program History Can Improve Code Completion", en *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, págs. 317-326.

[20] Reid Holmes and Gail C. Murphy, "Using Structural Context to Recommend Source Code Examples", en *Proceedings of the 27th international conference on Software engineering*, 2005, págs. 117-125.

[21] Doug Wightman, Zi Ye, Joel Brandt, Roel Vertegaal, "SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization", en *Proceedings of the 25th annual ACM symposium on User interface software and technology*, 2012, págs. 219-228.

[22] JFace – Eclipsepedia, fecha de consulta: 11 de abril de 2017, disponible en: https://wiki.eclipse.org/JFace

[23] Andrzej Wasylkowski, Andreas Zeller, Christian Lindig, "Detecting Object Usage Anomalies", en *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, págs. 35-44.

[24] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, Hong Mei, "MAPO: Mining and Recommending API Usage Patterns", en *Proceedings of the 23rd European Conference on ECOOP*, 2009, págs. 318 - 343.

[25] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Michele Lanza, "Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter", en *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, págs. 102 – 111.

[26] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, Martin

Monperrus, “Fine-grained and accurate source code differencing” en *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, págs. 313 - 324.

[27] Thomas Kuhn, Olivier Thomann, “Abstract Syntax Tree”, fecha de consulta: 3 de abril de 2017, disponible en: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html#sec-parsing-a-source-file

Anexos

Anexo A - Lista de Tipos de Nodo de un Árbol de Sintaxis Abstracta

Constante	**Valor**
ANNOTATION_TYPE_DECLARATION	81
ANNOTATION_TYPE_MEMBER_DECLARATION	82
ANONYMOUS_CLASS_DECLARATION	1
ARRAY_ACCESS	2
ARRAY_CREATION	3
ARRAY_INITIALIZER	4
ARRAY_TYPE	5
ASSERT_STATEMENT	6
ASSIGNMENT	7
BLOCK	8
BLOCK_COMMENT	64
BOOLEAN_LITERAL	9
BREAK_STATEMENT	10
CAST_EXPRESSION	11
CATCH_CLAUSE	12
CHARACTER_LITERAL	13
CLASS_INSTANCE_CREATION	14

COMPILATION_UNIT	15
CONDITIONAL_EXPRESSION	16
CONSTRUCTOR_INVOCATION	17
CONTINUE_STATEMENT	18
CREATION_REFERENCE	89
DIMENSION	85
DO_STATEMENT	19
EMPTY_STATEMENT	20
ENHANCED_FOR_STATEMENT	70
ENUM_CONSTANT_DECLARATION	72
ENUM_DECLARATION	71
EXPRESSION_METHOD_REFERENCE	90
EXPRESSION_STATEMENT	21
FIELD_ACCESS	22
FIELD_DECLARATION	23
FOR_STATEMENT	24
IF_STATEMENT	25
IMPORT_DECLARATION	26
INFIX_EXPRESSION	27
INITIALIZER	28
INSTANCEOF_EXPRESSION	62

INTERSECTION_TYPE	87
JAVADOC	29
LABELED_STATEMENT	30
LAMBDA_EXPRESSION	86
LINE_COMMENT	63
MALFORMED	1
MARKER_ANNOTATION	78
MEMBER_REF	67
MEMBER_VALUE_PAIR	80
METHOD_DECLARATION	31
METHOD_INVOCATION	32
METHOD_REF	68
METHOD_REF_PARAMETER	69
MODIFIER	83
NAME_QUALIFIED_TYPE	88
NORMAL_ANNOTATION	77
NULL_LITERAL	33
NUMBER_LITERAL	34
ORIGINAL	2
PACKAGE_DECLARATION	35
PARAMETERIZED_TYPE	74

PARENTHESIZED_EXPRESSION	36
POSTFIX_EXPRESSION	37
PREFIX_EXPRESSION	38
PRIMITIVE_TYPE	39
PROTECT	4
QUALIFIED_NAME	40
QUALIFIED_TYPE	75
RECOVERED	8
RETURN_STATEMENT	41
SIMPLE_NAME	42
SIMPLE_TYPE	43
SINGLE_MEMBER_ANNOTATION	79
SINGLE_VARIABLE_DECLARATION	44
STRING_LITERAL	45
SUPER_CONSTRUCTOR_INVOCATION	46
SUPER_FIELD_ACCESS	47
SUPER_METHOD_INVOCATION	48
SUPER_METHOD_REFERENCE	91
SWITCH_CASE	49
SWITCH_STATEMENT	50
SYNCHRONIZED_STATEMENT	51

TAG_ELEMENT	65
TEXT_ELEMENT	66
THIS_EXPRESSION	52
THROW_STATEMENT	53
TRY_STATEMENT	54
TYPE_DECLARATION	55
TYPE_DECLARATION_STATEMENT	56
TYPE_LITERAL	57
TYPE_METHOD_REFERENCE	92
TYPE_PARAMETER	73
UNION_TYPE	84
VARIABLE_DECLARATION_EXPRESSION	58
VARIABLE_DECLARATION_FRAGMENT	59
VARIABLE_DECLARATION_STATEMENT	60
WHILE_STATEMENT	61
WILDCARD_TYPE	76

Anexo B - Cuestionario sobre la experiencia del desarrollador con el plug-in

Questionnaire

We would like you to work with the following sets of questions. Your information will be used in an anonymous way.
Thank you for your cooperation.

* Required

Age *

Choose

Country of origin *

Your answer

How many years of experience do you have with Java? *

Choose

How many projects have you worked on that required you to write Java code? *

Choose

Please judge your expertise with the APIs Java NIO and Google Guava, using a scale from 0="None" to 4="Expert" *

	0	1	2	3	4
Java NIO	○	○	○	○	○
Google Guava	○	○	○	○	○

About the Code Completion Tool

Using a rate from 1 = "Strongly disagree" to 5 = "Completely agree", please indicate your level of agreement with the following statements *

	1	2	3	4	5
The tool was intuitive and easy to use.	○	○	○	○	○
The snippets presented by the tool were relevant to what I was programming.	○	○	○	○	○
It was faster to invoke the tool than searching for snippets online.	○	○	○	○	○
I would like to construct my own snippets and upload them to the repository.	○	○	○	○	○

What did you like the most about the tool?

Your answer

What did you dislike the most about the tool?

Your answer

Do you have any suggestions for improvement?

Your answer