Evaluador Inteligente de Código Java

Franco Madou
AI Group, Universidad de Palermo,
Buenos Aires, Argentina
aigroup@palermo.edu

Martín Agüero
AI Group, Universidad de Palermo,
Buenos Aires, Argentina
aigroup@palermo.edu

Gabriela Esperón
AI Group, Universidad de Palermo,
Buenos Aires, Argentina
aigroup@palermo.edu

Daniela López De Luise
AI Group, Universidad de Palermo,
Buenos Aires, Argentina
aigroup@palermo.edu

Abstract

Actualmente las actividades relacionadas con el desarrollo de software continúan incrementándose, por lo tanto, resulta indispensable y conveniente automatizar parte de las mismas. El presente trabajo estudia y propone una nueva herramienta de soporte a la calidad del software, un evaluador y clasificador de código fuente Java basado en métricas de software e Inteligencia Artificial.
El prototipo desarrollado emplea una red neuronal perceptrón multicapa a fin de reconocer patrones en el código fuente de programas escritos en lenguaje Java. Para ello se emplean métricas de software clásicas y otras diseñadas específicamente para el proyecto. El trabajo parte de una muestra especialmente diseñada con los estilos de codificación y se combina con algoritmos de clustering para crear un set de entrenamiento.
En base a los resultados obtenidos del proceso de clasificación, un sistema experto recomienda a los autores en función de reglas preestablecidas.
La precisión de los resultados confirmaría al prototipo como una alternativa eficaz y automática de soporte al proceso de calidad del software.
Keywords: métricas de software, inteligencia artificial, red neuronal Perceptron, algoritmos de clustering, sistema experto.

Introducción

Para aseverar que un programa sea libre de errores es necesario contar una estrategia que de soporte al proceso. El principal objetivo de un proyecto de software es asegurar que este cumpla con los requerimientos correspondientes y alcance la más alta calidad.
Una acepción de calidad es lograr el cumplimiento de los requisitos. La calidad se define como el conjunto de las características de un programa que cumple con las expectativas del cliente para el cual fue diseñado [1]. Según esta definición resultaría de gran importancia contar con un método que evalúe los posibles errores (desviaciones respecto de lo esperado), de lo contrario no podría afirmarse que el producto cumple con la especificación de requerimientos. A continuación se presenta brevemente las alternativas de manejo de calidad actuales en la comunidad de software.

Estrategias para asegurar el cumplimiento de la calidad:

a) Herramientas test-case: Actualmente la utilización de estas herramientas esta muy difundida. Son capaces de identificar y aislar secuencias que no son ejecutadas o que no devuelven el resultado esperado. Sin embargo se ha demostrado que la mayoría de los errores de un programa son encontrados cuando se realiza la integración de todos los módulos que lo componen por lo que resultaría muy difícil encontrar la mayoría de defectos sólo con testeos unitarios.
b) Lenguajes OO: Los lenguajes orientados a objetos facilitan moduralizar el código. Cada clase tiene sus propios métodos, los cuales deben ser instanciados para realizar una función específica. Esto permite que la ejecución se realice en un orden establecido por lo que el código esta mejor organizado y es más fácil de entender. El paradigma de objetos reduce los niveles de defectos en relación con los lenguajes procedurales. Sin embargo ningún programa esta exento de presentar errores en cualquier etapa del ciclo de vida.
c) Métricas de estimación: A. Albrecht como respuesta a las deficiencias y omisiones de la técnica de estimación por líneas de código, a principios de los años 80 difunde masivamente a través de IEEE el concepto de Puntos de Función (FP) demostrando que no era técnicamente posible medir tasas de producción de software a partir de la métrica líneas de código. Originalmente FP propone métricas para:
1. Características externas al software.
2. Características de interés para el usuario.
3. Aplicación en etapas tempranas del ciclo de vida del producto.
4. Independiente al código fuente o lenguaje.

d) Software de Gestión de la Calidad
La industria del software también gestiona el proceso de calidad mediante diversas herramientas. Bugzilla [2] propone una solución orientada a la comunicación y documentación de bugs encontrados durante la fase de

desarrollo y testing. Sin embargo carece de integración directa con el código fuente. JUnit presenta un conjunto de clases Java que permiten crear casos de prueba (test cases) para evaluar automáticamente las salidas de los métodos de clase. También propone un mecanismo de aserciones y otro de control de excepciones que posibilita automatizar aún más la verificación de algoritmos pero requiere significativas horas de programación para preparar casos de prueba y la reusabilidad del código es limitada [3]. JCosmo [4], se basa en detectores de problemas en el código fuente (code smells). Permite extender la cantidad de detectores agregando módulos estándar o creando propios. Los code smells informan sobre la estructura del programa a nivel paquete, clase, métodos y su interrelación. Depende de Rigi [5], una herramienta externa requerida para visualizar los resultados del análisis, que si bien permite descubrir defectos en el código a partir de la interpretación gráfica, demanda muchas horas de aprendizaje, estudio e interpretación de los gráficos [6].

e) Inteligencia Artificial

La Inteligencia Artificial (IA) es una ciencia que utiliza complejas técnicas para simular de manera artificial el razonamiento humano. IA ofrece muchas ventajas:

1) Mayor velocidad de procesamiento.
2) Posibilidad de que el programa pueda "aprender" y de esta forma lograr una optima performance.
3) Alto grado de eficiencia y correctitud en los resultados.

Propósito de Analizador Java Inteligente (AJI)

El presente trabajo propone una herramienta de soporte a la calidad del software, un evaluador y clasificador de código fuente Java basado en métricas de software e inteligencia artificial. El objetivo es dar soporte automático (sin intervención humana) al desarrollo de código fuente con el propósito de mejorar la calidad del software a través de la inspección, calificación y clasificación de código fuente.

I. ARQUITECTURA

La arquitectura de AJI está dividida en 6 módulos, 4 principales (núcleo) y 2 secundarios (externos). La comunicación entre componentes es únicamente a través de resultados almacenados en la base de datos y las interfaces externas (archivos de código fuente y reportes) son delegadas a módulos específicos e independientes del núcleo (ver Fig. 1).

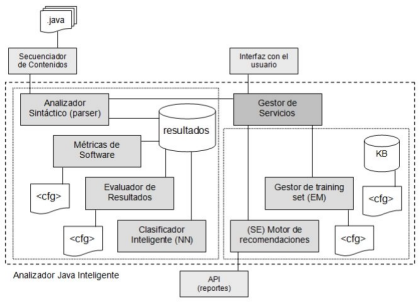

Fig. 1 Diagrama de Arquitectura de AJI

El diseño modular y la parametrización por archivos de configuración hacen de AJI una herramienta flexible y adaptable tanto a los requerimientos científicos como empresariales [7].

Se trata software desarrollado totalmente en lenguaje Java por lo que es ejecutable desde cualquier plataforma JRE 1.6 (Java Runtime Environment versión 1.6) compatible. El diseño de los algoritmos optimiza la utilización de CPU y memoria, posibilitando el análisis de grandes volúmenes de código fuente Java en pocos minutos de ejecución. En las siguientes secciones se describe el diseño y principales aspectos funcionales de cada módulo.

A. Módulo Secuenciador de Contenidos

El módulo Secuenciador de Contenidos es el nexo de AJI con los archivos de código fuente Java (ACF). Esencialmente es la extensión a una implementación de la interfase Collection de la API (Application Program Interface) de Java. Encapsula las palabras y símbolos de los ACF, serializando sincrónicamente y de forma transparente la información a procesar. Es configurable para definir símbolos de separación de palabras y normaliza el código fuera de estándar.

B. Módulo Analizador Sintáctico (Parser)

Como primer componente del núcleo de AJI está el módulo Analizador Sintáctico, cuya función es interpretar y relevar en contexto el código fuente serializado y sincronizado por Secuenciador de Contenidos.

Conceptualmente el algoritmo parser adapta a la sintaxis Java el modelo propuesto por W. A. Woods, Redes de

Transición Aumentadas (ATN) [7]. Básicamente la implementación en Java está definida por los patrones de diseño State y Memento [9]. Éste algoritmo detecta y responde a palabras reservadas, símbolos y estructuras cambiando a un siguiente estado o retornando a uno anterior [10].
El diseño modular de AJI permitirá extender a otros lenguajes todas sus funcionalidades. Seleccionando la implementación del módulo Analizador Sintáctico específico para cada lenguaje de programación ya sea procedural, objetos u orientado a un futuro paradigma de modelado.
Los valores obtenidos a partir del primer relevamiento (resultados elementales), son registrados y referenciados en el repositorio de la aplicación.

C. Módulo Evaluador según Métricas de Software
Es el componente de cálculo y traductor en operaciones algebraicas de las métricas seleccionadas por el analista. Según lo configurado desde un archivo xml, genera métricas de software ejecutando operaciones matemáticas a partir de resultados elementales obtenidos por el módulo antecesor. Por ejemplo, s un resultado denominado v1 como la diferencia entre el total de clases y el número de clases cuyo nombre comienza con una letra en minúscula, es decir, incumpliendo la especificación Java [11]. Otra forma de medir calidad es a partir de una métrica identificada como v12 desde donde AJI pondera el equilibrio entre el total de comentarios relevados en el ACF y los que son de tipo javadoc, estableciendo como de inferior calidad una desproporción entre los mismos. Se trata de la primer fase del análisis donde es ajustable la relevancia de cada operador. Será de gran importancia seleccionar el conjunto de métricas que definan con mayor precisión la identidad (atributos) de cada ACF. Es decir, un tunning10 artesanal de parámetros [12].

D. Módulo Evaluador / Normalizador de Resultados
A partir de valores cuantitativos específicos a cada métrica, el módulo Evaluador de Resultados obtiene calificadores valuados entre -1 y 1. Para representar determinada clasificación, se establecen grupos a partir de ciertos valores de corte (denominados fronteras).
Cada ACF es distribuido en función de la calificación obtenida. Etiquetas flotantes dan referencia al código de identificación (ID) del objeto acumulado en cada serie.
Finalmente el módulo exporta los resultados al formato arff (attribute-relation file format) compatible con el Módulo Clasificador Inteligente.

E. Módulo Clasificador Inteligente por Red Neuronal
El módulo Clasificador Inteligente es básicamente una red neuronal de tipo Perceptrón Multicapas configurada a partir del resultado del preprocesamiento realizado por el algoritmo de clustering Expectation-Maximization (EM) (Fig. 2) y entrenada utilizando el procedimiento de retropropagación (backpropagation) [13].
Cada ACF es distribuido en función de la calificación obtenida. Etiquetas flotantes dan referencia al código de identificación (ID) del objeto acumulado en cada serie.

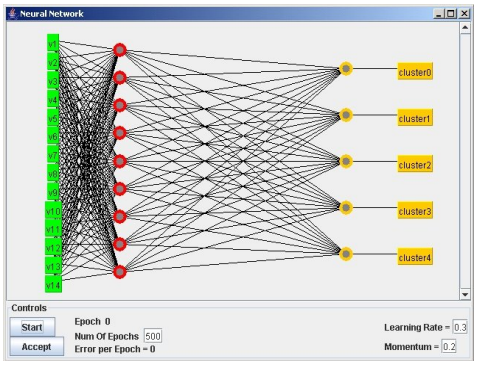

Fig. 2 Red Neuronal de AJI

F. Módulo API (Reportes)
Los resultados del análisis son registrados en la base de datos y están disponibles desde una API en dos modalidades: una gráfica (reportes) y otra accesible desde software Java (API). El estilo de los reportes es configurado desde un archivo jrxml por lo que es posible diseñarlos desde una herramienta gráfica externa [14]. La información relevante a cada análisis puede ser representada visualmente desde gráficos chart o en forma de listado, regulando el grado de detalle a través de consultas predefinidas (drill down / up).

G. Sistema Experto (SE)
El último componente de AJI es un Sistema Experto que realiza recomendaciones sobre los ACF utilizando los resultados obtenidos por la Red Neuronal. Actúa teniendo en cuenta el cluster en el cual se encuentra el archivo y evalúa cada una de las métricas prefijadas en base a la memoria de trabajo definida. A partir de reglas, el sistema emite un informe de sugerencias al programador a modo de guía para perfeccionar el modo de escribir código fuente.

II. CARACTERÍSTICAS DEL EXPERIMENTO

Selección de muestras:
Con el propósito de definir un Universo de Estilos de Codificación (UEC) a partir de un conjunto de muestras heterogéneas, suficientemente representativas y correctas, se recopilaron 1000 ACF con las siguientes características:

Tabla 1
Características de la muestra

Tipo de autor	*Origen*	*Cantidad*	*Tamaño promedio*
Aficionado (poco o medianamente experimentado)	planet-source-code.com	250	7 K
Académico	Patterns In Java Volume I, The Java Tutorials	250	2.7 K
Profesional	Aplicaciones Open Source	250	10.8 K
Sun Microsystems	API J2SE	250	8 K

Ponderación de resultados:
Para establecer valores de corte determinantes de grupos de calificación relativos a cada métrica, previamente se evaluaron resultados de más de 200 ACF. La tarea consistió básicamente en determinar de un valor límite para cada grupo en función del nivel de correctitud cuantificado desde los resultados obtenidos en cada métrica [v1, v2, v3...v14]. En todos los casos, se definen 3 niveles en relación a la concordancia con la especificación del lenguaje Java.

Data Mining:
Previamente a la clasificación automática y con el objetivo de obtener un training set representativo del UEC, en una primera fase se experimentó con el algoritmo de clustering kmeans . En las diferentes configuraciones se buscó minimizar la distancia cuadrática de todos los puntos al centro del cluster. Dado que el resultado de inferior valor era aún superior a 50 se decidió cambiar al algoritmo Expectation-Maximization (EM) para definir clusters [14]. Parametrizado para seleccionar automáticamente por cross validation (validación cruzada, método pesimista) el número de clusters, EM arrojó los siguientes resultados:

Tabla 2
Clusters definidos por EM

cluster #	*instancias*	*%*
0	9	1
1	127	13
2	102	10
3	149	15
4	572	57
5	4	4

Log likelihood: -1.79183. Sabiendo que el algoritmo EM finaliza cuando la fórmula que mide la calidad de clusters no muestra incremento significativo.
Para ello supone:

r1.P(a)+r2.P(b)+r3.P(c)+r4.P(d)+r5.P(e)

Siendo a,b,c,d ,e clusters y r1, r2, r3, r4, r5 los parámetros. El algoritmo usa un registro de las probabilidades de que los parámetros sean los verdaderos. Estas probabilidades van cambiando a medida que procesan datos. A la

medida de bondad o credibilidad de éstas probabilidades lo llama Log likelihood. Se obtiene como la productoria de las probabilidades condicionales sobre toda la muestra.
A partir de éste resultado parcial, el objetivo se enfocó en incrementar el Log likelihood, es decir, la calidad de clusters. Para ello se reconfiguró el algoritmo EM con los siguientes parámetros:
Máximo de iteraciones: 100
Desviación estándar mínima: 1.0 x 10-6
Cantidad de clusters: 5
Cantidad de semillas (seeds): 200
Modo de cluster: evaluar sobre datos de entrenamiento.
Obteniendo el siguiente resultado:

Tabla 3
Instancias por cluster

cluster #	*instancias*	*%*
0	326	33
1	50	5
2	190	19
3	246	25
4	188	19

Log likelihood: -9.06605

Un algoritmo automático, compara resultados y mide distancias respecto del promedio, así determina el grado de relevancia de las métricas asociadas a cada cluster, es el proceso de profiling de clusters **Error! Reference source not found.**.

Red Neuronal:
Para el contexto descripto en la presente sección, una red neuronal perceptrón multicapa resultó ser la estructura neuronal más precisa para clasificar el UEC. Entrenada con el algoritmo backpropagation y configurada con los siguientes parámetros:
Tasa de aprendizaje: 0.3 (La cantidad en que los pesos son actualizados).
Momentum: 0.2 (Énfasis aplicado a los pesos durante la actualización).
Tiempo de entrenamiento: 500 epochs (La cantidad de ciclos requeridos para entrenar la red neuronal).
Umbral de validación: 20 (El valor define la cantidad de veces que puede repetirse un error antes de finalizar el entrenamiento).
Fuente de entrenamiento: training set y cross validation
La red neuronal entrenada tanto utilizando el training set (método optimista) o por cross-validation a 10 folds alcanza marcas de precisión destacables [17].

III. METRICAS DE SOFTWARE

Una métrica se define como una medida que permite valorar objetivamente la calidad de los elementos. Para el caso del código fuente, las métricas referencian distintas características del software a partir de valores numéricos. Generalmente por sí sola una métrica no resulta significativa de modo que resulta conveniente analizar un conjunto de ellas para llegar a una conclusión aceptada.
El presente proyecto estudia la evaluación de código fuente utilizando diversas métricas, algunas clásicas (como por ejemplo: LOC y CP) y utilizadas y otras que fueron especialmente diseñadas para este trabajo [17]. Algunas de éstas últimas son:
V6 (variables constantes/total de variables): Define una proporción entre tipos de variables. Un excesivo uso de variables estáticas puede indicar falta de reasignación de valores a una misma variable.
V10 (Uso de clases internas y anidadas): Es la proporción de clases internas y anidadas sobre el total de clases.
V11 (Equilibrio entre comentarios de línea y comentarios compuestos): El uso exclusivo de uno de los dos modos que permite el lenguaje Java agregar comentarios al código fuente puede indicar desconocimiento del lenguaje o disminuir el índice de mantenibilidad por tratarse de código difícil de leer.
V22 (Cantidad de clases con constructor default): La existencia de una mayoría de constructores default podría indicar desconocimiento de las opciones que ofrece el lenguaje para instanciar con distintos argumentos.

IV. SISTEMA EXPERTO

Considerando las ventajas que aportarían las métricas AJI cuenta con un Sistema Experto que en base a éstas realiza recomendaciones sobre los errores que cometen los programadores para que puedan mejorar su forma de codificar (ver Tabla 4).

Tabla 4
Recomendaciones según métricas y clasificación

Métrica	*Significado*	*Intervalos*	*Recomendaciones*
V1	Líneas por clase	0-100	Sin sugerencia
		101-200	Se recomienda disminuir la cantidad de LOC por clase
		201 o más	Es necesario disminuir la cantidad de LOC por clase
V2	Métodos por clase	0-20	Sin sugerencia
		21-40	Se recomienda disminuir la cantidad de métodos por clase
		41 o más	Es necesario disminuir la cantidad de métodos por clase
V15	Porcentaje de comentarios	entre 0 y 1	Se recomienda emplear mayor número de comentarios
		más de 1 y hasta 60	Sin sugerencia
		más de 60	Se recomienda utilizar menor número de comentarios
V22	Clases con constructor default	0	Sin sugerencia
		1	Se sugiere codificar todos los constructores
		más de 1	Se recomienda codificar el constructor

Un sistema experto o sistema basado en el conocimiento es un sistema informático capaz de emular el comportamiento de un experto en un campo concreto. Está formado por una base de conocimiento, memoria de trabajo y un motor de inferencia que es el que modela el proceso del razonamiento humano.
El sistema experto que se propone utiliza la información aportada por la red neuronal y recomienda en base a reglas predefinidas, las cuales no sólo tienen en cuenta cada una de las métricas sino también el cluster y la importancia de esa métrica en ese cluster. Las reglas están escritas es un archivo de extensión DRL. Un archivo puede contener más de una regla. La sintaxis es la siguiente:

```
When “nombre regla” atributos
   When
      Condición
Then
   Acción
End
```

Un ejemplo es la regla V15 (ver Fig. 3):

```
rule "v15nosignificativo"
    when $archivoFuente : ArchivoFuente ( cluster.ranking >= 1 )
    then $archivoFuente.recomendacion.setTexto("no es significativo en este cluster")
end

rule "v15coincide"
    when $archivoFuente : ArchivoFuente ( cluster.ranking <= 1, medicion.distancia == "COIN" )
    then $archivoFuente.recomendacion.setTexto("es significativo pero no hace falta recomendar")
end

rule "v15derecha1"
    when $archivoFuente : ArchivoFuente ( cluster.ranking <= 1, medicion.distancia == "1DER" )
    then $archivoFuente.recomendacion.setTexto("debe utilizar menos comentarios")
end

rule "v15izquierda1"
    when $archivoFuente : ArchivoFuente ( cluster.ranking <= 1, medicion.distancia == "1IZQ" )
    then $archivoFuente.recomendacion.setTexto("debe utilizar mas comentarios")
end
```

Fig. 3 Regla que es parte de la base de conocimiento del SE de AJI

La métrica V15 mide el porcentaje de líneas de comentarios que hay por clase. En primer lugar la regla analiza si la métrica no fue significativa en el clúster, es decir si el ranking en ese clúster fue mayor a 1. En caso negativo, verifica si su valor coincide con el valor que se espera que ésta tenga en el clúster. Si no hay coincidencia lo próximo que se observa es si el valor esperado es mayor o menor al valor de la métrica. En base a este análisis se realiza la recomendación.

V. CONCLUSION Y TRABAJO FUTURO

El presente proyecto propone, a partir del empleo de métricas y técnicas de Inteligencia Artificial mejorar el nivel de los programadores y de esta forma incrementar la calidad del software. El objetivo final es brindar recomendaciones en base a archivos fuente java procesados y evaluados utilizando las métricas antes mencionadas.

Actualmente se está trabajando en la optimización de las reglas para el sistema experto. Una vez concluido esta fase se planea poner a disposición de los programadores una versión beta del sistema a la cual se podrá acceder a través de la Web. De esta manera los desarrolladores de software Java obtendrán respuesta inmediata la cual les informara los aspectos del código fuente a perfeccionar. A su vez, esto permitirá la retroalimentación del sistema, posibilitando el ajuste automático del mismo.

Referencias

[1] IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology.

[2] Bugzilla: http://www.bugzilla.org/about/

[3] http://www-128.ibm.com/developerworks/java/library/j-junit4.html

[4] Eva van Emden, Leon Moonen, "Java Quality Assurance by Detecting Code Smells".

[5] Rigi: http://www.rigi.csc.uvic.ca/Pages/description/whatitis.html

[6] Neil Walkinshaw, "Partitioning Object-Oriented Source Code for Inspections", University of Strathclyde, Glasgow, 2006.

[7] Agüero M., López De Luise D., "Analizador Java Inteligente", WICC 2007.

[8] W.A. Woods, "Transition Network Grammars for Natural Language Analisis", pp. 591-606, Communications of the ACM, 1970.

[9] Bruce Eckel, "Thinking in Patterns", 2003.

[10] Mark Watson, "Practical Artificial Intelligence Programming in JAVA", 2005.

[11] James Gosling, Bill Joy, Guy Steele, Gilard Bracha "The Java Language Specification 3er Edition", Pretience Hall. 2005.

[12] Daniela López DeLuise, Martín Agüero, "Aplicación de Métricas Categóricas en Sistemas con Lógica Difusa", Revista IEEE América Latina, 2007.

[13] Patrick H. Winston, "Inteligencia Artificial, tercera edición", Addison Wesley Iberoamericana, 1992.

[14] jReport: http://sourceforge.net/projects/jasperreports

[15] Ian H. Witten, Eibe Frank "Data Mining: Practical Machine Learning Tools and Techniques", pp. 265, Morgan Kaufmann, 2005.

[16] Agüero M., Esperón G., Madou F, López De Luise D., "Intelligent Java Analyzer", IEEE CERMA 2008.

[17] Agüero M., Esperón G., Madou F, López De Luise D., AI Group, Facultad de Ingeniería de la Universidad de Palermo, Buenos Aires, Argentina. "Métricas de Software e Inteligencia Artificial", IEEE ANDESCON, 2008.