Mapeo de Atributos a partir de Guías de Estilo de Programación y Automatización de sus Métricas

Dianela Sosa, María Fernanda Papa, Pablo Becker, Luis Olsina

Universidad Nacional de La Pampa, Facultad de Ingeniería, La Pampa
{dianela.sosa,pmfer,beckerp,olsinal}@ing.unlpam.edu.ar

Resumen. Actualmente, es común que en las organizaciones de desarrollo de software intervengan equipos numerosos y descentralizados, lo que puede dificultar que el software sea fácilmente entendido y mantenido. Esta situación pone de manifiesto la necesidad de codificar programas de software siguiendo guías de estilo para el lenguaje usado, que sean claras y conocidas por los desarrolladores. De allí es que surgen diferentes guías de codificación, pero su utilización puede resultar tediosa para desarrolladores juniors y cuando el tiempo de entrega apremia. En este sentido, es importante contar no solo con un enfoque que permita mapear las guías a atributos y estos a sus métricas, sino también con una herramienta que chequee y recomiende mejoras cuando el código no adhiera a dichas guías. Este artículo ejemplifica el uso de un enfoque sistemático que permite mapear guías de estilo de programación a atributos y a sus métricas que los cuantifican. Además, se muestra el empleo de la herramienta JavaStyleInspector que se ha desarrollado para analizar código Java y generar reportes que permiten la mejora rápida del código en favor de cumplir con la Google Java Style Guide. Su uso puede influir positivamente tanto en la enseñanza de las guías de estilo en carreras relacionadas a informática como en el trabajo diario de un profesional de la industria de software.

Palabras claves: Adherencia–Código fuente– Google Java Style Guide–Automatización

1 Introducción

La calidad del software es un concepto complejo que no se puede definir de una forma simple [1], lo que conduce a que en la literatura existan diversas definiciones. Por ejemplo, en [2] el autor define calidad de software como "*el cumplimiento de los requisitos de funcionalidad y desempeño explícitamente establecidos, de los estándares de desarrollo explícitamente documentados y de las características implícitas que se esperan de todo software desarrollado profesionalmente*". En cambio, para el estándar ISO/IEC 9126 [3], la calidad es "*la totalidad de las características de una entidad que se relacionan con su capacidad para satisfacer las necesidades declaradas e implícitas de los usuarios*". A su vez, en el estándar ISO/IEC 25010 [4], el cual reemplazó a [3], se define como "*el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor*". Notar que a fines del 2023 fue publicada la nueva edición de ISO/IEC 25010, con semejante definición.

Dentro de los modelos jerárquicos propuestos por los estándares mencionados se encuentra **mantenibilidad** como una de las características de calidad relevante al momento de evaluar calidad externa e interna de un producto de software, junto con

otras características tales como funcionalidad, eficiencia, fiabilidad, entre otras. Si bien ambos estándares ISO poseen la característica mantenibilidad la cual definen como "*capacidad del producto para ser modificado, corregido, mejorado y adaptado*" difieren en las subcaracterísticas que poseen. Mientras que el estándar ISO/IEC 9126 tiene analizabilidad, modificabilidad, estabilidad, capacidad de ser probado y conformidad de mantenibilidad (maintainability compliance), ISO/IEC 25010 está conformada por modularidad, reusabilidad, analizabilidad, modificabilidad y capacidad de ser probado.

Este trabajo se centrará en la característica mantenibilidad y su susbcaracterística Adherencia (Compliance) por lo que se utilizará como base el estándar ISO/IEC 9126 dado que su sucesor considera dicha susbcaracterística de un modo implícito. Compliance está definida como "*el grado en que el producto de software adhiere a estándares o convenciones relacionados con la mantenibilidad*". En el caso particular de esta investigación se focaliza en el código fuente como producto de software. En este sentido uno de los puntos que hace mantenible un código fuente y por lo tanto de calidad, es la adherencia a guías o estándares de codificación. Las guías de codificación aseguran que los desarrolladores dentro de una organización adopten las mismas prácticas generando códigos fuente más comprensibles y fácil de mantener, lo que reduce tiempo y esfuerzo de compresión cuando un nuevo desarrollador se integra al equipo de trabajo o se debe modificar el producto de software.

Lamentablemente, en la actualidad, muchos desarrolladores muestran una falta de motivación a la hora de adherirse a los estándares de codificación [5], ya que consideran esta tarea aburrida o que cuando los tiempos de entrega apremian lo importante es que el producto funcione. Si bien el código fuente puede seguir ejecutándose cuando no está creado según las guías, la calidad disminuye en términos de mantenibilidad. Como se mencionó, la adopción de una guía de estilo trae múltiples beneficios, pero su aplicación lleva tiempo y práctica. Por un lado, las guías pueden ser complejas o difíciles de entender, ambiguas y generalmente escritas en idioma inglés, lo cual puede ser un problema especialmente para desarrolladores menos experimentados. Esto trae como consecuencia que la aplicación de las directrices en todo el código no sea consistente. Hasta que el equipo se consolida en el uso de las guías es necesario contar con enfoques que faciliten la evaluación de la adherencia del código generado a las guías elegidas.

Contar con un enfoque que permita mapear cada una de las guías (ítems) a uno o más atributos que sean medibles de forma objetiva y/o automática puede ser de mucha utilidad a la hora de validar la adherencia del código fuente, permitiendo un aprendizaje guiado por la práctica y sin requerir un mayor esfuerzo posterior al de la definición de los atributos a evaluar. Otra forma de incentivar el uso de estas guías podría ser mediante el empleo de herramientas que verifiquen la adherencia a ellas y corrijan cualquier incumplimiento detectado. En este sentido, este trabajo persigue el objetivo de aplicar un enfoque que mapea guías de estilo de codificación a atributos y sus métricas correspondientes, junto al uso de la herramienta JavaStyleInspector, la cual fue diseñada y construida para facilitar a los desarrolladores el mantenimiento del código fuente Java de acuerdo a Google Java Style Guide [6] a partir de la evaluación del nivel de cumplimiento de los ítems de dicha guía.

A continuación, el artículo presenta los Trabajos Relacionados (Sección 2) donde se abordan las guías de estilo de codificación y las herramientas que automatizan la

revisión del código fuente en busca de no adherencias a dichas guías. En la Sección 3 se documenta y ejemplifica el enfoque con base ontológica, que permite mapear ítems a atributos y sus métricas. La Sección 4 trata sobre JavaStyleInspector, describiendo la herramienta y explicando su funcionamiento a partir de un ejemplo. Finalmente, en la Sección 5 se presentan las conclusiones y trabajos futuros.

2 Trabajos Relacionados

Los trabajos relacionados a esta investigación van en tres direcciones. Por un lado, lo referido a las guías de estilo, por el otro, a los enfoques que permiten mapear ítems de dichas guías a atributos y sus métricas, y finalmente, a las herramientas cuyo objetivo es la mejora de la mantenibilidad a partir del cumplimiento de las guías de estilo.

Es importante destacar que una guía de estilo de codificación, o también conocida como estándar de codificación, es un conjunto de normas que se utiliza para escribir y formatear código fuente en un determinado lenguaje de programación, y así facilitar la legibilidad y mantenibilidad de dicho código [7]. En la actualidad, se han definido guías de estilo de codificación para la mayoría de los lenguajes de programación.

La necesidad de contar con este tipo de guías surge desde mediados de la década de 1970. En 1974, Kernighan y Plauger [8] presentaron sugerencias sobre cómo escribir código legible en lenguaje C utilizando ejemplos de software reales. Tres años después, Sun Microsystems impulsó pautas de estilo de codificación para el lenguaje Java [9], las cuales fueron extendidas y refinadas por Reddy en 2000 [10] enfatizando la importancia de la legibilidad del código y la facilidad de mantenimiento. En 2004 Google desarrolla Google Java Style Guide [6] para ser empleada en su empresa, pero rápidamente, fue adoptada por desarrolladores independientes. Actualmente, es una guía ampliamente utilizada y que ha sido actualizada periódicamente.

En general, las guías de estilo de los distintos lenguajes de programación abordan aspectos similares, por ejemplo, reglas para la nomenclatura de variables y otros elementos del código, convenciones para la escritura de comentarios y estructuración del código. En este trabajo se decidió utilizar Google Java Style Guide dada su amplia aceptación en la industria, su énfasis en la legibilidad y mantenibilidad del código y sus actualizaciones frecuentes. Esto se puede evidenciar en varios sitios web que hacen referencia a ella, e incluso el sitio de guías de desarrollo de la Universidad de los Andes (Colombia) menciona que "*todo código escrito en Java debería adherirse a las guías de estilo de código de Google*" [11].

El estándar propuesto por Google comienza con una introducción y en su Sección 2 trata sobre cuestiones básicas de los archivos fuentes (nombre y codificación). En la Sección 3 menciona aspectos relacionados con la estructura y el orden de los elementos en el código fuente. La Sección 4 incluye todas las reglas referidas al formato (pautas para la indentación, el espaciado, el uso de llaves, tamaño máximo de línea, formato de bloques de código, declaración de variables y anotaciones). Luego, la Sección 5, define las reglas para el nombramiento de paquetes, clases, métodos, variables y constantes. Las Secciones 6 y 7, recomiendan prácticas de programación y cómo utilizar Javadoc de forma correcta, respectivamente.

Este trabajo se puede considerar una profundización y continuación de lo realizado en [12], donde se mostró de forma resumida en la actividad de definición de requisitos

no funcionales cómo mapear las guías de estilo de programación a atributos y sus métricas. También se aplicó la estrategia GOCAMEC (*Goal-Oriented Context-Aware Measurement, Evaluation, and Change*) para evaluar y mejorar un código fuente Java a partir del incumplimiento de algunos ítems de la guía de estilo de Google. En este trabajo a diferencia de [12], se profundiza sobre el enfoque que permite el mapeo de un ítem o elemento de la guía a uno o más atributos, se amplía la cantidad de ítems de la guía considerados y se realiza un análisis de los resultados obtenidos a partir de la herramienta JavaStyleInspector, cuyo desarrollo se había indicado como un trabajo futuro en [12].

Respecto al enfoque, que mapea requisitos no funcionales en forma de dimensiones e ítems de guías de estilo de codificación con características y atributos para construir un modelo de calidad soportado por conceptos y relaciones ontológicas (como se ilustra en la Sección 3), hasta el presente no se han encontrado trabajos relacionados que vayan en la misma dirección. Si bien el uso del enfoque propuesto requiere una preparación que lleva más tiempo de especificación que si se utiliza la guía de estilos como un checklist, donde sólo se indican con un tilde aquellos ítems que cumplen con los requisitos, los resultados obtenidos a partir de la aplicación del enfoque tiene varios beneficios, a saber: resultados objetivos, reproducibles y comparables, mayor granularidad de análisis, especificación de dónde se requiere el cambio para cumplir con el ítem, entre otras cuestiones que se mencionan en más detalle en la Sección 3.

En cuanto a la herramienta JavaStyleInspector, la cual automatiza el proceso de medición de los ítems mapeados a atributos de la guía, se clasifica dentro de la categoría de análisis estático de código fuente. Este tipo de análisis surge del proceso de examinar el código fuente sin tener que ejecutarlo y tiene distintos objetivos, como analizar el cumplimiento de estándares de codificación, detectar problemas (por ejemplo: división por cero, bucles infinitos, ramas de expresiones lógicas sin utilizar), detectar código duplicado, vulnerabilidades y fallas de seguridad en el código. En esta categoría se encuentran las herramientas SonarQube, Semmle, GitLab Code Quality, Codacy, JetBrains Codana, PMD, entre otras. A diferencia de ellas, JavaStyleInspector no buscará errores de programación, ya que necesita que el archivo donde se encuentra el código fuente compile correctamente para generar los resultados de la medición con el objetivo explícito y único de analizar el incumplimiento a los estándares de codificación. Por lo tanto, se puede indicar que JavaStyleInspector beneficia al desarrollador en comprender la adherencia del código generado y aprender de los incumplimientos a las guías de codificación sin poner el foco en otros aspectos que también podrían requerir cambio.

Los entornos integrados de desarrollo (IDE, por sus siglas en inglés), como lo son IntelliJ, Eclipse, Visual Studio Code, etc., ofrecen la funcionalidad de dar formato al código fuente según las reglas que se hayan configurado a partir de plugins como google-java-format [13]. Además, google-java-format se puede incluir como librería en la configuración de Maven o Gradle. A diferencia de estos plugins, JavaStyleInspector actúa como un asistente que marca las partes donde se presenta el no cumplimiento de la guía o ítem, dando la oportunidad al usuario de ir internalizando las guías a medida que se desarrolla y se van realizando las evaluaciones de adherencia. Además, su uso puede influir positivamente en la enseñanza de las guías en carreras relacionadas a informática, el cual es uno de los objetivos de esta investigación.

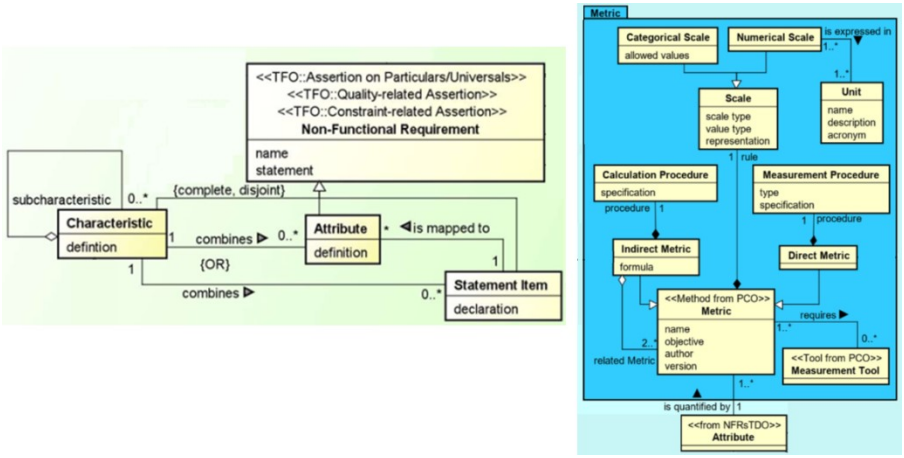
3 Enfoque para Mapear Ítems de Guías de Estilo a Atributos

El enfoque aquí detallado se basa en la arquitectura ontológica FCD-OntoArch (*Foundational, Core, Domain, and instance Ontological Architecture for sciences*) [14] de cinco niveles. Donde los conceptos en los niveles fundacional y central (core) son independientes del dominio y los tres niveles restantes son dependientes del dominio (nivel alto de dominio, bajo de dominio y de instancia). Las dos ontologías que se requieren para explicar el enfoque son NFRsTDO [15] y MetricsLDO [16]. Ambas se encuentran a nivel de dominio en FCD-OntoArch. Mientras que de NFRsTDO se toman los conceptos de Requisito no funcional, Característica, Atributo e Ítem, de MetricsLDO son útiles los conceptos de Métrica, Métrica directa e indirecta, Procedimiento de medición y de cálculo, Escala (Numérica y Categórica) y Unidad. Todos estos conceptos junto a sus atributos y relaciones guían el mapeo realizado en este trabajo brindando un marco de referencia conceptual donde se sabe a qué se refiere cada concepto y qué aspectos son necesarios definir al momento de cumplir con el objetivo. La Tabla 1 presenta la definición de cada uno de los conceptos mencionados, y la Fig. 1 muestra un extracto del modelo de las ontologías exponiendo las relaciones entre los conceptos.

Tabla 1. Conceptos y definiciones útiles en el enfoque de mapeo de ítems a atributos. Definición en inglés obtenidas de los documentos originales [15, 16].

Concepto	Definición	Ontología
Non-Functional Requirement (Requisito no funcional)	It is a Quality-, Constraint-related Assertion that specifies an aspect in the form of a Characteristic, Attribute, or Statement Item to be evaluated on how or how well an Evaluable Entity performs or shall perform.	NFRsTDO
Characteristic (Característica)	It is an NFR that represents an evaluable, non-elementary aspect attributed to a particular entity or its category by a human agent.	
Attribute (Atributo)	It is an NFR that represents a measurable and evaluable physical or abstract aspect attributed to a particular entity or its category by a human agent.	
Statement Item (Ítem)	It is an NFR that represents a declared textual expression of an evaluable physical or abstract aspect asserted for a particular entity or its category by a human agent.	
Metric (Métrica)	The defined Measurement or Calculation Procedure and Scale necessary to correctly quantify an Attribute.	MetricsLDO
Direct Metric (Métrica directa)	It is a Metric that does not depend on other Metrics of any other Attribute.	
Indirect Metric (Métrica indirecta)	It is a Metric that depends on other Metrics of any other Attribute.	
Measurement Procedure (Procedimiento de medición)	Arranged set of instructions or operations of a Direct Metric, which specifies how the steps in the Implement Direct Measurement task should be performed.	
Calculation Procedure (Procedimiento de cálculo)	Arranged set of instructions or operations of an IndirectMetric, which specifies how the steps in the Implement Indirect Measurement task should be performed.	
Scale (Escala)	A set of values with defined properties.	
Unit (Unidad)	Particular quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitude relative to that quantity.	

Con el objetivo de explicar cómo trabajar con el enfoque se presentará el ejemplo del ítem “3.3.3. Ordering and spacing” (Fig. 2) de la Google Java Style Guide a partir del cual se mapearon tres atributos. Notar que, con el fin de resaltar el uso de los conceptos mostrados en la Fig. 1, a medida que se describa el ejemplo se subrayarán los términos de las ontologías y las relaciones además se pondrán en cursiva. Por otro lado, cabe mencionar que algunas partes del ejemplo están redactadas en inglés ya que el mismo está tomado del caso presentado en [12], el cual fue escrito en inglés.



(a) (b)

Fig. 1**.** Extracto de las ontologías de NFRsTDO (a) y MetricsLDO (b) donde se muestran las relaciones entre conceptos.

3.3.3 Ordering and spacing

Imports are ordered as follows:

1. All static imports in a single block.
2. All non-static imports in a single block.

If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.

Fig. 2. Definición del ítem o guía “3.3.3 Ordering and spacing” de Google Java Style Guide [6].

Se parte de que los ítems presentes en la Google Java Style Guide son un tipo de requerimiento no funcional que pueden ser *mapeados* a atributos, los cuales son *combinados* en características y *subcaracterísticas*. El modelo jerárquico está formado por la característica “1. Mantenibilidad” que se relaciona con la *subcaracterística* “1.1. Adherencia”. Esta jerarquía está dada por el objetivo del trabajo que es mejorar la mantenibilidad de un código fuente a partir de su adherencia a ciertos ítems de la Google Java Style Guide. Del análisis de la guía se puede apreciar que el título donde se encuentra ubicado el ítem 3.3.3 puede convertirse en *subcaracterística* de 1.1. En consecuencia, en la Tabla 4 se puede apreciar la nueva *subcaracterística* denominada “1.1.1. Source file structure compliance” que surge del título “Source file structure” presente en la guía. De todos los ítems de ese apartado de la guía se focalizará en el ítem 3.3.3. A partir de la lectura de su definición (Fig. 2) se observa que: 1) las sentencias import estáticas y no estáticas deben estar ordenadas en dos bloques; 2) entre esos dos bloques se debe colocar una línea en blanco a modo de separación; 3) cada sentencia import debe estar en distintas líneas sin que haya líneas en blanco adicionales. De estas tres recomendaciones dadas en el ítem 3.3.3, se

mapearon los atributos *"1.1.1.1. Compliance with the ordering of types of imports"*, "*1.1.1.2. Spacing compliance between static and non-static import blocks"* y "*1.1.1.3. Spacing compliance between import sentences"*. Cada atributo fue definido, por ejemplo, 1.1.1.1 se definió como "the source code file adheres to the ordering of types of imports that is specified in the coding style guide".

Una vez analizados los ítems de la guía y mapeados a atributos, se deben especificar las métricas que *cuantifican* a cada atributo (ver Fig. 1.b). Siguiendo con el ejemplo del atributo 1.1.1.1, este se *cuantificó* con la métrica indirecta "Percentage of compliance with the ordering of type of imports" especificada en la Tabla 2.a. Como se puede apreciar en dicha tabla la definición de la métrica indirecta cuenta con un nombre, objetivo, autor y versión, todas son propiedades de una Métrica presentes en la Fig. 1.b. A lo cual se le suma el procedimiento de cálculo, la escala numérica y su unidad. Al ser una métrica indirecta está relacionada con una métrica directa, la cual se definió en la Tabla 2.b y a diferencia de la primera posee un procedimiento de medición en vez de un procedimiento de cálculo. De este modo se fueron definiendo todas las métricas que *cuantifican* los atributos del modelo en base al enfoque aquí presentado.

Tabla 2. Especificación de la Métrica indirecta que cuantifica el atributo "*1.1.1.1. Compliance with the ordering of types of imports"* y su Métrica directa. Especificación adaptada de [12].

Indirect Metric
Quantified Attribute name: Compliance ordering of types of imports
Metric Name: Percentage of compliance ordering of types of imports (**%COTI**)
Objective: Determine the percentage of Java have the imports ordered by type with resp total Java files to be measured.
Author: P. Becker and L. Olsina **Ve**
Calculation Procedure:
Formula: $\%COTI = \left(\frac{\sum_{i=1}^{\#JF} AOTI_i}{\#JF}\right) * 10$
Scale: Numeric
Scale Type name: Ratio
Value Type: Real
Representation: Continuous

(a)

Direct Metric
Quantified Attribute name: Location of impo file
Metric Name: Availability of ordering of types (**AOTI**)
Objective: Determine the correct location of ir Java file
Author: P. Becker and L. Olsina **Versi**
Measurement Procedure:
Type: Objective
Specification:
AOTI = SI = NSI = 0
if (there are static imports sentences) S
if (there are non-static imports sentence
if (SI = 1 and NSI = 1){
if (all static imports are declared pr all non-static imports)
AOTI = 1
}else
AOTI = 1

(b)

Google Java Style Guide consta de 87 ítems, y el enfoque se aplicó a 37 ítems (ver Apéndice I-Tabla 1 en http://surl.li/sjwmg). Los ítems seleccionados fueron aquellos que establecían una regla y no se limitaban sólo a definiciones (como en "5.3 Camel case: defined"), recomendaciones (como en "4.8.3.1 Array initializers: can be 'block-like'") o agrupaciones de otros ítems (transformados en subcaracterísticas). Por ejemplo, los ítems de la guía agrupados en las secciones Introduction, Programming Practices y Javadoc no se consideraron debido a que la primera sección es

informativa, y las otras dos requieren para su medición interpretar código fuente.

Por otro lado, al igual que el ítem 1.1.1.1, los ítems "4.8.5.2. Class annotations", "4.8.5.3. Method and constructor annotations" y "4.8.6.1 Block comment style" fueron mapeados a más de un atributo. Además, hubo casos en los que más de un ítem fueron mapeados al mismo atributo, por ejemplo, los ítems "2.1. File Name" y "3.4.1. Exactly one top-level class declaration" fueron mapeados al atributo "*1.1.2.4. Adherencia al número de declaraciones de clase de nivel superior por archivo fuente"*. En resumen, los 37 ítems seleccionados se mapearon a 42 atributos (ver Apéndice I-Tabla 3), los cuales fueron combinados en características y subcaracterísticas para formar un árbol de requisitos no funcionales (Apéndice I-Tabla 2). Una vez diseñado el árbol, se definieron todos sus componentes (Apéndice I-Tabla 3) y se especificaron las métricas indirectas, y sus directas, para cuantificar cada atributo. Luego, de los 42 atributos diseñados se seleccionaron 22 (resaltados con color verde en Apéndice I-Tabla 2) y se automatizaron las métricas correspondientes en JavaStyleInspector implementando más de 60 procedimientos de cálculo y de medición.

Es importante destacar que aplicar este enfoque puede insumir más tiempo que el requerido para aplicar las guías de estilo. Es decir, para llevar adelante el enfoque es necesario el análisis de los ítems de la guía, el mapeo de cada ítem a uno o más atributos, y luego para cada atributo se debe proveer su definición y la especificación de una o más métricas. Sin embargo, todo el tiempo invertido en esta primera etapa de especificación redunda en beneficios a la hora de realizar los análisis de resultados. Si se comparan los resultados obtenidos a partir de un checklist de la guía -donde el evaluador indica si el ítem se cumple o no, sin brindar mayores detalles-, con los valores obtenidos al medir según el enfoque se tiene como beneficios: 1) Ofrecer mayor granularidad de información. Al poder mapear un ítem a varios atributos el nivel de granularidad de análisis es mayor, y se puede determinar cuál es la parte del ítem que no fue cumplida en el código fuente facilitando de este modo su posterior reparación. 2) Al indicar cómo debe realizarse la medición a partir de la especificación de métricas que cuantifican de manera objetiva cada atributo los resultados obtenidos pueden ser reproducidos y comparados. Esto permite el análisis temporal de cómo se van aplicando las guías a lo largo de todo el ciclo de desarrollo. Además, esto permite focalizar esfuerzos en aquellas etapas donde se presentan evidencias de que el requisito de aplicar las guías de estilo decae.

Finalmente, el hecho de que el enfoque este soportado por ontologías trae el beneficio de que todos los integrantes del equipo de desarrollo se comuniquen de manera efectiva, con un lenguaje común, conociendo exactamente a qué se refieren sus colaboradores al mencionar conceptos como métrica, ítem, atributo, característica, entre otros.

4 Automatización de Métricas en JavaStyleInspector

4.1 Importancia de Automatizar las Métricas

Contar con una herramienta que automatice el proceso de medición requerido para analizar si un código fuente Java cumple con las guías de estilo de Google garantiza que cada línea del código será revisada en pos de identificar algún incumplimiento y

que se reducirá tiempo y esfuerzo comparado con la misma tarea realizada manualmente. Sin contar que, si se realiza manualmente, el resultado es propenso a un mayor riesgo de errores humanos. Por otro lado, no es necesario esperar a que el código fuente esté finalizado para analizarlo con la herramienta. Desde una etapa temprana de codificación, el código puede ser analizado para evitar que los incumplimientos se propaguen a etapas posteriores. Además, los mismos incumplimientos detectados por la herramienta pueden ser conocidos por un desarrollador que previamente no estaba al tanto de ellos, lo que le permite corregirlos y evitar su repetición en el futuro.

4.2 Tecnologías utilizadas en el desarrollo de JavaStyleInspector

JavaStyleInspector es una aplicación web cuya arquitectura se divide en un frontend desarrollado con Angular (versión 16.2) y Tailwind CSS (versión 3.3.5) en Visual Studio Code, y un backend API REST con Spring Boot (versión 3.2.2) en el IDE IntelliJ IDEA. Al momento de crear el proyecto con Spring Boot se eligió la versión 17 de Java, ya que ofrece soporte a largo plazo. La comunicación entre el frontend y el backend se realizó a través de solicitudes HTTP y respuestas JSON siguiendo el estándar REST. La API REST ha sido documentada usando OpenAPI. El código del frontend y del backend se implementaron en App Services de Microsoft Azure con GitHub Actions, ya que ofrecen un plan gratuito que resulta adecuado para el tamaño actual de la herramienta.

Para realizar la persistencia, se utilizó una base de datos MySQL (versión 8), la cual está alojada en la nube, específicamente en un servidor de Microsoft Azure. En esta base de datos se incluyó el árbol de requisitos no funcionales completo, que contiene 22 atributos relacionados con sus respectivas métricas indirectas y directas. Los resultados de las mediciones también son respaldados en la base de datos. En la Fig. 3 se puede observar el stack tecnológico empleado en el desarrollo.

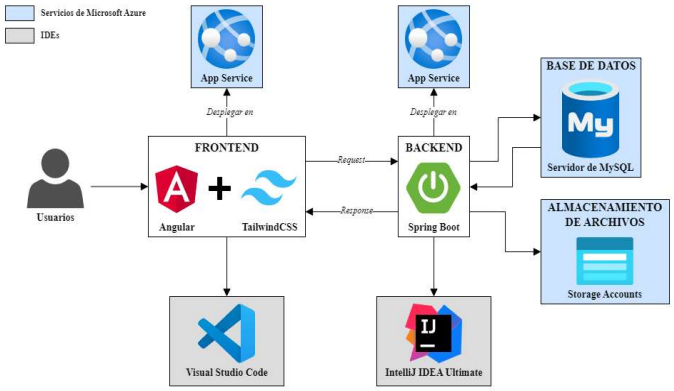


Fig. 3. Stack tecnológico utilizado en el desarrollo de JavaStyleInspector.

Si bien JavaStyleInspector mide la adherencia a la Google Java Style Guide, es lo suficientemente flexible como para incorporar la medición de otros estándares de codificación Java que utilicen el enfoque de mapeo de ítems a atributos y sus métricas. Esto es posible porque Java, el lenguaje utilizado para implementar el

proceso de medición en la herramienta, posee librerías y proyectos de código abierto como, por ejemplo, JavaParser, que facilitan acceder a los elementos de un archivo fuente Java (variables, clases, métodos, etc).

4.3 Ejemplo de uso de JavaStyleInspector

Como primera acción de uso, el usuario debe indicar cuál es el archivo .java a medir, arrastrando el mismo al sector de la interfaz que permite su carga (ver Fig. 4 punto 1). En este caso es "GUICalculator.java" (Apéndice III), el cual fue utilizado en [12]. Usar el mismo código fuente permitió validar los resultados obtenidos por JavaStyleInspector con los realizados manualmente en dicho estudio. A continuación, se selecciona del árbol de requisitos no funcionales los atributos a medir. Para el ejemplo, se eligieron los once que participaron del trabajo expuesto en [12].

En la Fig. 5 se muestran algunos de los atributos escogidos que pertenecen a la subcaracterística Adherencia a la estructura de un archivo fuente que se corresponde a la subcaracterística "1.1.1. Source file structure compliance" de la Tabla 4. Solo queda presionar el botón medir para que los resultados se presenten en pantalla (Fig. 4 punto 3). Estos también pueden ser descargados en formato pdf.

Una vez obtenidos los resultados que se presentan en la cuarta columna de la Tabla 4 y en la Fig. 4 se efectuó el análisis de los mismos. Para ello, estos valores se compararon con los obtenidos de forma manual en [12]. Como se puede apreciar en la Tabla 4 (columnas 3 y 4), hubo tres valores diferentes. Dos de los cuales son cuestiones de redondeo (valores resaltados en naranja en la Tabla 4) y un tercero que tiene una diferencia de 6,4 puntos (resaltado en rojo en la Tabla 4).

Al analizar la causa de esta última diferencia se notó que hubo una interpretación distinta del ítem "4.8.2.1. One variable per declaration" de la Google Java Style Guide mapeado al atributo "*1.1.2.2. Compliance with the number of variables per declaration"*. Esta interpretación dispar recayó en un diseño distinto de la métrica directa Number of individual declarations of variables (#IDV). En el Apéndice II de [12] donde se especifican las métricas utilizadas, se agrega una nota aclarando que los parámetros son un tipo de variable. Por lo tanto, los parámetros son tenidos en cuenta cuando se realiza el cálculo en [12]. La diferencia mencionada se puede observar en la Fig. 6 donde se muestran los valores medidos para los atributos "Número de declaraciones individuales de variables" y "Número de declaraciones de variables". Al volver a revisar la Google Java Style Guide [6] se constató que solo deben considerarse las declaraciones de variables locales y atributos mientras que los parámetros no deben ser tenidos en cuenta. En la implementación de la métrica utilizada en este trabajo se optó por seguir lo que pauta la guía, por eso la diferencia de resultados para el atributo 1.1.2.2.

Durante la etapa de pruebas de usabilidad, JavaStyleInspector fue utilizada por tres de los autores de este artículo, los cuales poseen cierta experiencia en la evaluación de usabilidad de aplicaciones web. Como resultado preliminar, los tres coincidieron en que su uso es sencillo e intuitivo. Actualmente, al momento de la escritura de este artículo, la herramienta está pasando por una evaluación sistemática de calidad en uso. En particular, JavaStyleInspector está siendo utilizada por estudiantes de la carrera de Ingeniería de Sistemas de la Facultad de Ingeniería de la UNLPam en la asignatura Ingeniería de Software II, de donde estamos recolectando datos de calidad en uso para eficacia, eficiencia y satisfacción, tal como lo realizamos en [17].

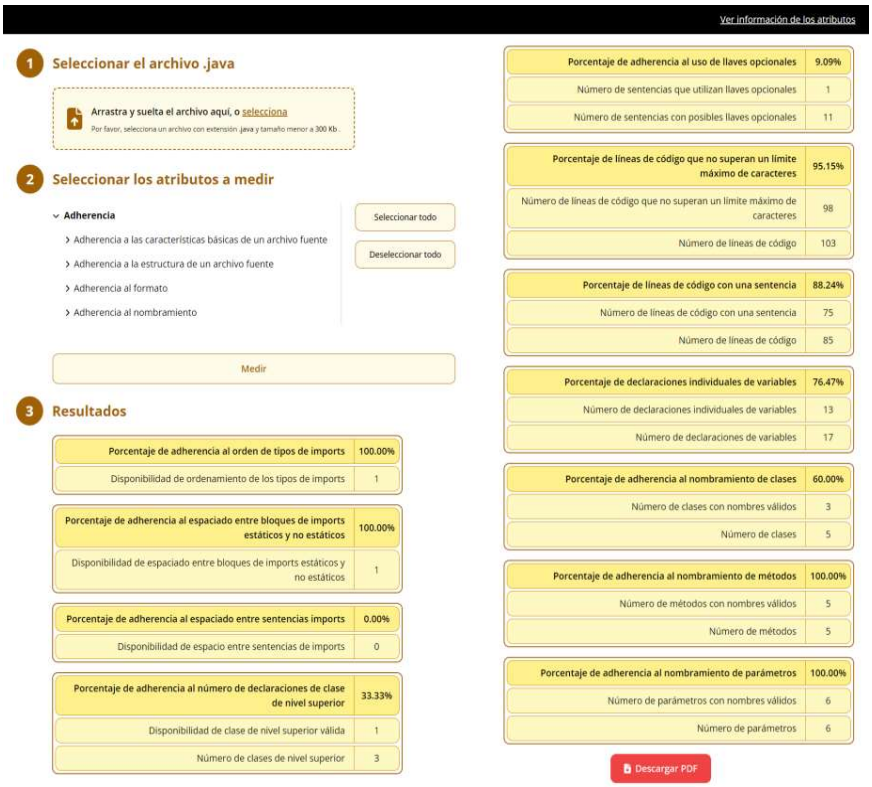
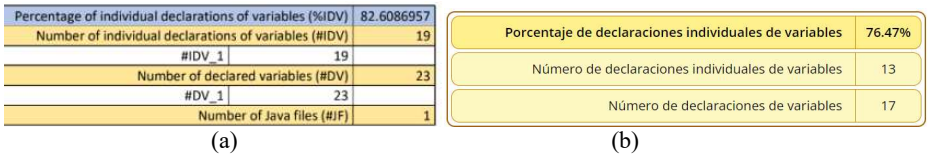


Fig. 4. Captura de pantalla de JavaStyleInspector donde se muestran los pasos para medir y sus resultados.

Fig. 5. Captura de pantalla de JavaStyleInspector donde se seleccionan los atributos a medir.

Tabla 4. Comparativa de las mediciones realizadas en [12] y los valores obtenidos por JavaStyleInspector. Además de la correspondencia entre ítems de la guía de estilo Google y atributos presentes en [12].

Ítems de Google Java Style Guide	Características y *atributos* (*en itálica*)	Valores de medición	
		Manual	Herramienta
	1. Maintainability		
	1.1. Compliance		
3. Source file structure	**1.1.1. Source file structure compliance**		
3.3.3. Ordering and spacing	*1.1.1.1. Compliance with the ordering of types of imports*	100	100
	1.1.1.2. Spacing compliance between static and non-static import blocks	100	100
	1.1.1.3. Spacing compliance between import sentences	0	0
3.4.1. Exactly one top-level class declaration	*1.1.1.4. Compliance with the number of top-level class declarations per source file*	33,33	33,33
4. Formatting	**1.1.2. Formatting compliance**		
4.1.1. Use of optional braces	*1.1.2.1. Compliance with the use of optional braces*	9,09	9,09
4.8.2.1. One variable per declaration	*1.1.2.2. Compliance with the number of variables per declaration*	82,61	76,47
4.3. One statement per line	*1.1.2.3. Compliance with the number of statements per line*	88,23	88,24
4.4. Column limit: 100	*1.1.2.4. Compliance with the maximum line size*	95,14	95,15
5. Naming	**1.1.3. Naming compliance**		
5.2.2. Class names	*1.1.3.1. Class naming compliance*	60	60
5.2.3. Method names	*1.1.3.2. Method naming compliance*	100	100
5.2.6. Parameter names	*1.1.3.3 Parameter naming compliance*	100	100



(a) (b)

Fig. 6. (a) Valores medidos manualmente en [12] – (b) Valores medidos con JavaStyleInspector. Diferencia ocasionada por la interpretación del ítem "4.8.2.1. One variable per declaration" de la Google Java Style Guide.

5 Conclusiones

En este trabajo se presentó un enfoque, basado en conceptos y relaciones ontológicas, que permite mapear requisitos no funcionales en forma de dimensiones e ítems de guías de estilo de codificación con características y atributos para construir un modelo de calidad. En particular, en este trabajo se ilustró un enfoque para mapear ítems de la Google Java Style Guide a un modelo de evaluación jerárquico cuyo foco es mantenibilidad. Contar con este tipo de enfoque representa una guía práctica, que no sólo permite que los desarrolladores hablen un mismo idioma, sino que también permite mapear ítems (en este caso pertenecientes a guías de estilo de codificación) en

atributos cuantificables de manera objetiva y robusta por métricas.

También se presentó la herramienta web JavaStyleInspector, la cual a partir de las métricas surgidas de aplicar el enfoque automatizó la medición para conocer los valores que son necesarios al momento de evaluar el grado de adherencia de un código Java a 37 ítems de la Google Java Style Guide. Esta herramienta es el resultado del avance de la investigación presentada en [12].

Todo desarrollador con experiencia conoce la presión que conlleva cumplir con los tiempos de entrega, en consecuencia, cualquier herramienta ofrece una ventaja significativa en relación a estos tiempos. Conforme a una validación inicial realizada por tres integrantes de este trabajo, el empleo de JavaStyleInspector contribuyó en la mejora de la velocidad de generación de resultados en comparación con el enfoque manual que se siguió en [12]. Además de brindar mayor flexibilidad en el proceso, dado que se pueden medir ciertos atributos utilizando un código fuente específico, luego realizar cambios en el mismo y volver a ejecutar la herramienta con la posibilidad de seleccionar los mismos atributos u otros, y así sucesivamente. Esto último no es factible con la medición manual, ya que, debido a su naturaleza, es probable que no se quiera repetir el proceso muchas veces, lo que puede retrasar el análisis hasta el momento en que se disponga de la versión final del código.

Al momento de automatizar la medición se presentaron algunos desafíos los cuales pueden ser vistos como limitaciones. Por ejemplo, en la medición de los atributos relacionados a la subcaracterística "1.1.3. Naming compliance" donde se debe analizar el nombre de una clase, método o parámetro. En estos se debe determinar si el nombre está escrito como lo indica la guía, es decir, en formato Camel case. La dificultad encontrada fue que, si bien se puede determinar fácilmente si la primera palabra comienza o no con una letra en mayúscula o minúscula, es difícil determinar si las palabras siguientes comienzan con mayúscula o no. Esto se debe a que no se sabe en qué punto termina la palabra anterior, la misma puede estar abreviada, e incluso las palabras que forman el nombre pueden estar en inglés, o en español, o en una combinación de varios idiomas.

Otro desafío fue la medición del atributo "*1.1.2.3. Compliance with the number of statements per line".* La dificultad radica en que Java contiene varios tipos de sentencias (declaraciones, de flujo de control, asignaciones, imports, etc.) y cada una de ellas requiere una estrategia diferente para su medición. Esto se puede observar en la especificación de la métrica *#LCUS* del Apéndice III.

En estos desafíos mencionados, para su correcta medición es necesaria la intervención del usuario. Más allá de estas cuestiones, la herramienta prueba ser de utilidad para agilizar el proceso de medición y como referencia para aprender a aplicar la Google Java Style Guide.

Finalmente, cabe mencionar que existen varias líneas de investigación futura. Una de ellas, en la cual ya se comenzó a trabajar, es la automatización de la medición del resto de los atributos que fueron mapeados en este trabajo. Por otro lado, se pretende agregar a los resultados enlaces a la documentación referida a cada atributo y la especificación de la métrica que originó su medición. Por último, se evalúa la posibilidad de enumerar los incumplimientos a la Google Java Style Guide y proporcionar más información sobre ellos, por ejemplo, en qué línea de código se está produciendo dicho incumplimiento.

Agradecimientos. Esta línea de investigación está soportada parcialmente por la Facultad de Ingeniería de la UNLPam, Argentina, mediante los proyectos 09/F079 y POIRe2023-09.

Referencias

1. Somerville, I.: Ingeniería de software. 6ta Edición. Addison Wesley Publishers, (2002).
2. Pressman, R. S.: Ingeniería del software. Un enfoque práctico. 7ma Edición. Mc Graw Hill Educación, (2010).
3. ISO/IEC 9126-1: Software Engineering - Software Product Quality - Part 1- Quality Model, Int'l Org. for Standardization, Geneva, (2001).
4. ISO/IEC 25010: Systems and Software Engineering– Systems and software product Quality Requirements and Evaluation (SQuaRE)– System and software quality models, (2011).
5. Avila D., Báez E., Zapata M., López D., Zurita D.: Unreadable Code in Novice Developers. In: Rocha, Á., Adeli, H., Dzemyda, G., Moreira, F., Ramalho Correia, A.M. (eds) Trends and Applications in Information Systems and Technologies. WorldCIST 2021. Advances in Intelligent Systems and Computing, vol 1368. Springer, Cham. https://doi.org/10.1007/978-3-030-72654-6_5, (2021).
6. Google Java Style Guide, https://google.github.io/styleguide/javaguide.html, último acceso 2024/03/21.
7. Broekhuis S.: The Importance of Coding Styles within Industries. 35th Twente Student Conference on IT (TScIT 35), pp. 1-8, (2021).
8. Kernighan, B.W., Plauger, P.J.: The Elements of Programming Style, 1st Edición. McGraw-Hill, New York, (1974).
9. Sun Microsystems: Java code conventions. https://www.oracle.com/technetwork/java /codeconventions-150003.pdf, último acceso 2024/03/21, (1997).
10. Reddy, A.: Java™ coding style guide. Sun MicroSystems, (2000).
11. Guia de desarrollo, https://uniandesdsit.github.io/coding-guidelines/style/ STYLE_GUIDE.html, Universidad de los Andes, Colombia, último acceso 2024/03/21.
12. Becker, P., Olsina, L., Papa, M.F.: Approach to Improving Java Source Code Considering Non-compliance with a Java Style Guide. In: Pesado, P. (eds) Computer Science – CACIC 2022. CACIC 2022. Communications in Computer and Information Science, vol 1778. Springer, Cham. https://doi.org/10.1007/978-3-031-34147-2_9, (2023).
13. Documentación de google-java-format, https://github.com/google/google-java-format, último acceso 2024/03/21.
14. Olsina, L.: The Foundational Ontology ThingFO: Architectural Aspects, Concepts, and Applicability. In: Fred, A., Aveiro, D., Dietz, J., Bernardino, J., Masciari, E., Filipe, J. (eds.) Knowledge Discovery, Knowledge Engineering and Knowledge Management. IC3K 2021. Communications in Computer and Information Science, Vol 1718, Springer. Chapter 5, pp. 73–99, (2023).
15. Olsina, L., Papa, M.F., Becker, P.: NFRsTDO v1.2's Terms, Properties, and Relationships -- A Top-Domain Non-Functional Requirements Ontology, pp. 1–9, https://doi.org/10.48550/arXiv.2302.01096, (2023).
16. Becker, P., Olsina, L., Papa, M.F.: Especificación de Métricas de Mantenibilidad para Mejorar Código Java: Caso Aplicado en un Curso Avanzado de Ingeniería en Sistemas. Actas del 11vo Congreso Nacional de Ingeniería Informática / Sistemas de Información (CoNaIISI), 2-3 de Nov., Tucumán, Argentina, pp. 1-15. (2023).
17. Molina, H., Olsina L. Diseñando la Evaluación de Calidad en Uso de una Herramienta Didáctica para Crear Interfaces Gráficas en JavaTM, In: XV Argentine Symposium on Software Engineering, (ASSE2014), 43 JAIIO, CABA, Argentina, pp. 51-65. (2014).