Derivación de Atributos a partir de la Guía de Estilo Java de Google y su Cuantificación mediante Métricas
Dianela Sosa, María Fernanda Papa, Pablo Becker, Luis Olsina
SADIO Electronic Journal of Informatics and Operations Research (EJS) Vol. 24 No. 1 (2025) e-ISSN 1514-6774
https://doi.org/10.24215/15146774e069 | https://revistas.unlp.edu.ar/ejs
Sociedad Argentina de Informática e Investigación Operativa | Universidad Nacional de La Plata | Buenos Aires | Argentina

Derivación de Atributos a partir de la Guía de Estilo Java de Google y su Cuantificación mediante Métricas

Deriving Attributes from Google's Java Style Guide and Quantifying Them Using Metrics

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. Habitualmente, en el desarrollo de software intervienen equipos numerosos y descentralizados, lo que representa un desafío para 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. No obstante, su utilización puede ser compleja, especialmente para desarrolladores juniors o en contextos con plazos ajustados. 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 mapea guías de estilo de programación a atributos y a sus métricas que los cuantifican. Además, muestra el empleo de la herramienta JavaStyleInspector para analizar código Java y generar reportes que permitan 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.

Abstract. Software development typically involves large, decentralized teams, making it difficult to create software that is easy to understand and maintain. This situation highlights the need to code software programs following clear and widely recognized style guides for the language used. However, applying these guides can be complex, especially for junior developers or in time-sensitive scenarios. In this sense, it is important to have not only an approach that allows mapping the guides to attributes and these to their metrics, but also a tool that checks and recommends improvements when the code does not adhere to these guides. This paper illustrates the use of a systematic approach that maps programming style guides to attributes and the metrics that quantify them. Additionally, it demonstrates the use of the JavaStyleInspector tool to analyze Java code and generate reports that facilitate rapid code improvement in alignment with the Google Java Style Guide. The tool's application can have a positive impact on the teaching of style guides in computer science-related careers and on the daily work of a software industry professional.

Palabras claves: Código fuente – Medición – Métrica – Google Java Style Guide – Herramienta

Keywords: Source code – Measurement – Metric – Google Java Style Guide – Tool

Received September 2024; Accepted December 2024; Published March 2025

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*". En el estándar ISO/IEC 25010 [4], el cual reemplaza a [3], se define calidad como "*el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor*". Cabe destacar que, a fines del año 2023, se publicó una nueva edición del estándar ISO/IEC 25010, con una definición de calidad semejante a su versión anterior.

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 posee las subcaracterísticas Analizabilidad, Modificabilidad, Estabilidad, Capacidad de ser probado y Adherencia a la mantenibilidad (maintainability compliance), el estándar ISO/IEC 25010 agrupa las subcaracterísticas de Modularidad, Reusabilidad, Analizabilidad, Modificabilidad y Capacidad de ser probado.

Este trabajo se centra en la característica Mantenibilidad y su subcaracterística Adherencia (Compliance). Para su desarrollo se toma como fundamento el estándar ISO/IEC 9126, que define Compliance como "*el grado en que el producto de software adhiere a estándares o convenciones relacionados con la mantenibilidad*". No se utilizó el estándar ISO/IEC 25010 dado que considera a la subcaracterística Compliance de un modo implícito. El foco de la investigación recae 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 establecen prácticas uniformes entre los desarrolladores dentro de una organización, lo que resulta en un código más comprensible y fácil de mantener. Esta uniformidad no solo facilita la integración de nuevos desarrolladores al equipo de trabajo, sino que también optimiza el tiempo y esfuerzo necesarios para comprender y modificar el producto de software.

En la actualidad, se observa una preocupante falta de motivación entre los desarrolladores a la hora de adherirse a los estándares de codificación [5]. Esta actitud suele estar asociada a la percepción de que seguir estas pautas es una tarea aburrida o innecesaria, especialmente cuando los plazos de entrega son ajustados y la prioridad recae en garantizar la funcionalidad del producto. Sin embargo, aunque el código fuente pueda ejecutarse sin cumplir estrictamente con las guías de codificación, su calidad se ve comprometida, particularmente en lo que respecta a la Mantenibilidad.

Como se mencionó, la adopción de una guía de estilo ofrece múltiples beneficios, aunque su implementación requiere tiempo y práctica. Un desafío significativo radica en que estas guías suelen ser complejas y ambiguas, sumado a que suelen estar redactadas en un idioma que no es el nativo del programador, lo que puede representar una barrera adicional, especialmente para desarrolladores con menos experiencia. Esto genera inconsistencias en la aplicación de las directrices a lo largo del código. Como consecuencia, durante el proceso de consolidación del equipo en el uso de las guías de estilo, resulta necesario disponer de enfoques que faciliten la evaluación de la adherencia del código generado a las guías elegidas.

Contar con un enfoque que permita derivar de cada ítem de las guías de codificación a uno o más atributos medibles de forma objetiva y/o automática puede ser beneficioso para validar la adherencia del código fuente. Este enfoque, además, puede facilitar un aprendizaje basado en la práctica y minimizar el esfuerzo requerido tras la definición inicial de los atributos a medir, promoviendo una aplicación más efectiva y consistente de las guías. Otra manera de facilitar la adopción de las guías de codificación es el empleo de herramientas que verifiquen la adherencia a estas y corrijan cualquier incumplimiento identificado.

En este contexto, el presente trabajo tiene como objetivo aplicar un enfoque que mapea las guías de estilo de codificación a atributos y sus métricas correspondientes, complementado con el uso de la herramienta *JavaStyleInspector*. Esta herramienta, específicamente diseñada y desarrollada para este fin, facilita a los programadores mantener su código fuente Java alineado a la *Google Java Style Guide* [6], mediante la evaluación del nivel de cumplimiento de los ítems de dicha guía.

Es importante destacar que este artículo es una versión extendida del trabajo presentado en [7]. A diferencia de [7], aquí se presenta con mayor detalle el enfoque utilizado que permite la derivación a partir de ítems de la guía en atributos y sus métricas que las cuantifican. También, se profundiza en los beneficios y limitaciones del uso del enfoque, junto a los desafíos encontrados durante el desarrollo y uso de *JavaStyleInspector*.

A continuación, en la Sección 2 se presentan los Trabajos Relacionados 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 del ejemplo dado en la Sección 3. En la Sección 5 se discuten los beneficios y limitaciones del enfoque junto a los desafíos que se sortearon en el desarrollo y uso de la herramienta. Finalmente, en la Sección 6 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 [8]. 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 [9] 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 [10], las cuales fueron extendidas y refinadas por Reddy en 2000 [11] 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*" [12].

El estándar propuesto por Google (ver Fig. 1) 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.

Como se mencionó en la sección introductoria, este trabajo es una versión extendida de [7], el cual profundiza y continúa la investigación presentada en [13]. En este último trabajo, se mostró de forma resumida cómo mapear las guías de estilo de programación a atributos y sus métricas, enmarcándola dentro de la actividad de definición de requisitos no funcionales. 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. A diferencia de [13], en [7] 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 [13].

En el presente artículo se presenta con mayor detalle el enfoque que permite la derivación de un ítem o elemento de la guía en uno o más atributos, profundizando en aspectos que no fueron tratados en [7] ni en [13]. Además, se discuten los beneficios y

limitaciones del enfoque, junto a los desafíos encontrados durante el desarrollo y uso de la herramienta *JavaStyleInspector*.

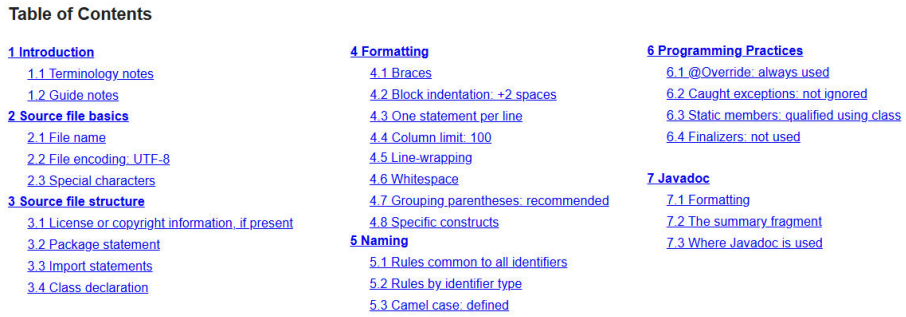


Fig. 1. Tabla de contenido de la Google Java Style Guide [6].

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 con más detalle en la Sección 5.

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 [14]. 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 Derivar Ítems de Guías de Estilo en Atributos

El enfoque aquí detallado se basa en la arquitectura ontológica FCD-OntoArch (*Foundational, Core, Domain, and instance Ontological Architecture for sciences*) [15]. Esta arquitectura, como se puede apreciar en la Fig. 2, posee cinco niveles de ontologías, a saber: nivel fundacional, nivel central (core), nivel alto de dominio, nivel bajo de dominio y nivel de instancia. Mientras que los conceptos en los niveles fundacional y central son independientes del dominio, los tres niveles restantes son dependientes del dominio.

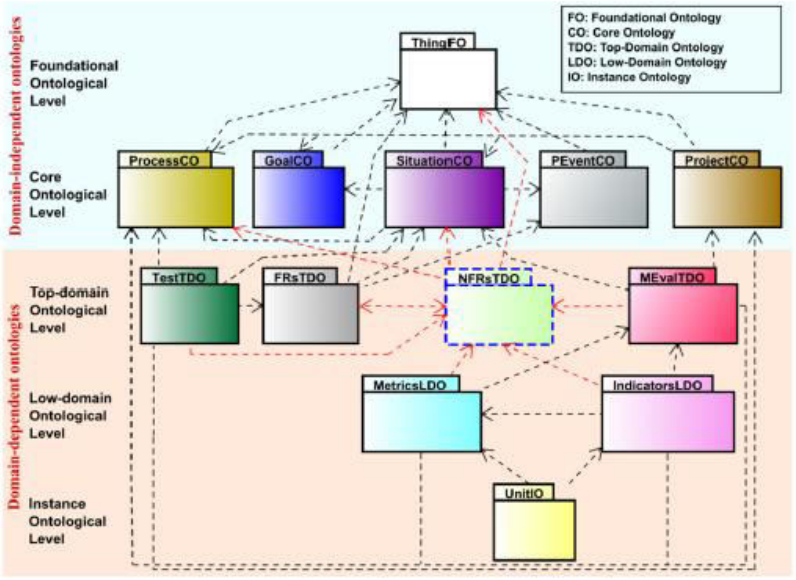


Fig. 2. Ubicación de los componentes NFRsTDO y MetricsLDO dentro de la arquitectura ontológica FCD-OntoArch.

Las ontologías que se requieren para explicar el enfoque se encuentran a nivel de dominio en FCD-OntoArch y son NFRsTDO [16] y MetricsLDO [17]. Si bien ambas ontologías se encuentran a nivel de dominio, difieren en su nivel. Esto queda definido observando el final de su nombre, donde TDO significa nivel de dominio alto y LDO nivel de dominio bajo.

La ontología NFRsTDO especifica y define todos los términos, propiedades y

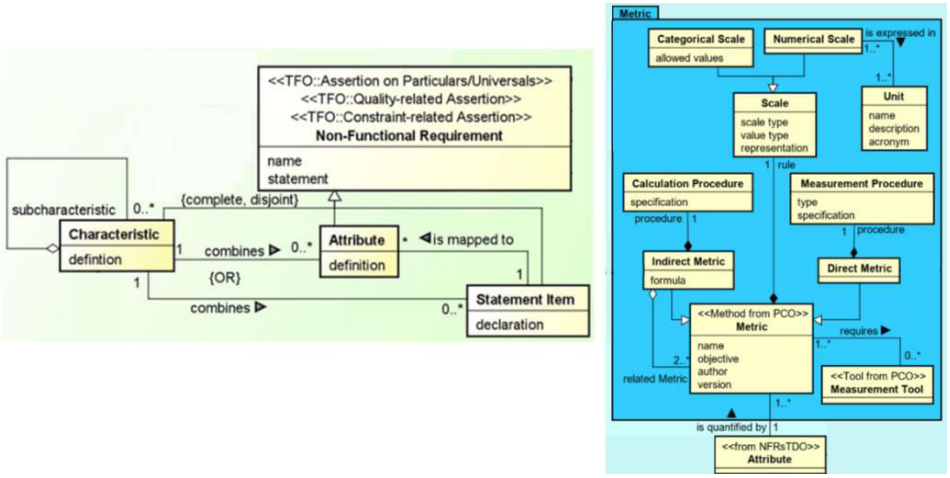
relaciones referidos a Requisitos no funcionales desde el punto de vista de calidad y calidad/costo. Algunos de los conceptos involucrados son Requisito no funcional, Característica, Atributo e Ítem. La ontología MetricsLDO especifica y define los términos, propiedades y relaciones referidos a Métricas. Algunos de los conceptos útiles para entender el enfoque son Métrica, Métrica directa e indirecta, Procedimiento de medición y de cálculo, Escala (Numérica y Categórica) y Unidad. Estos conceptos, junto con sus propiedades y relaciones, constituyen un marco de referencia conceptual que orienta el mapeo o derivación desarrollado en este trabajo. Este marco permite identificar con precisión el significado de cada concepto y los aspectos esenciales a definir para alcanzar el objetivo planteado. La Tabla 1 presenta la definición de cada uno de los conceptos mencionados, y la Fig. 3 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 [16, 17].

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 ejemplificar cómo trabajar con el enfoque se expone el caso de dos ítems de la Google Java Style Guide, a saber: “3.3.3. Ordering and spacing” y “5.2.2. Class names”. Notar que, con el fin de resaltar el uso de los conceptos mostrados en la Fig. 3, a medida que se describa el ejemplo se subrayarán los términos y las relaciones presentes en las ontologías. Por otro lado, cabe mencionar

que algunas partes del ejemplo están redactadas en inglés ya que el mismo está tomado del caso presentado en [13], el cual fue escrito en inglés.



(a) (b)

Fig. 3. Extracto de las ontologías de NFRsTDO (a) y MetricsLDO (b) donde se muestran las relaciones entre conceptos.

Como se aprecia en la Fig. 3.a y en la definición de Statement Item en la Tabla 1, los ítems presentes en la Google Java Style Guide son un tipo de requerimiento no funcional que pueden ser mapeados a atributos.

El primer ítem de la Google Java Style Guide a ejemplificar es el "3.3.3. Ordering and spacing". En la Fig. 4 se muestra el ítem tal cual está definido en la guía, el cual indica que las sentencias import deben ordenarse de la siguiente forma: primero, un bloque con todas las sentencias import estáticas y luego otro bloque con todas las sentencias import no estáticas. Luego hace una aclaración que indica que, si están presentes los dos tipos de bloque, estos deben estar separados por una sola línea en blanco. Y no debe haber ninguna otra línea en blanco entre las sentencias import. Esta declaración puede ser reformulada teniendo en cuenta tres recomendaciones, a saber: 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. A continuación, las tres recomendaciones reescritas a partir del ítem 3.3.3, se mapearon a tres atributos, como se muestra en la Tabla 2.

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. 4. Definición del ítem "3.3.3 Ordering and spacing" de Google Java Style Guide [6].

Tabla 2. Mapeo del ítem “3.3.3. Ordering and spacing” de la Google Java Style Guide a tres atributos.

Recomendación	*Atributo*
Las sentencias import estáticas y no estáticas deben estar ordenadas en dos bloques.	*Compliance with the ordering of types of imports*
Entre esos dos bloques se debe colocar una línea en blanco a modo de separación.	*Spacing compliance between static and non-static import blocks*
Cada sentencia import debe estar en distintas líneas sin que haya líneas en blanco adicionales.	*Spacing compliance between import sentences*

La derivación a partir de un ítem en tres atributos contribuye a aumentar la precisión en su evaluación. Cuando se aplica la Google Java Style Guide como una lista de verificación, no es posible determinar de manera directa cuáles recomendaciones dentro de un ítem se están incumpliendo, dado que la información que brinda es «cumple» o «no cumple». Al tener mapeado el ítem a tres atributos, uno por cada recomendación, se puede saber exactamente qué es lo que se incumple. Esto permite dirigir los esfuerzos correctivos hacia los aspectos específicos del código que requieren ajustes.

El otro ítem de la Google Java Style Guide a ejemplificar es “5.2.2. Class names”, que se encuentra declarado en la guía tal cual como se muestra en la Fig. 5. De todo lo expresado en la guía se tomó como recomendación que las clases deben estar nombradas con UpperCamelCase, lo cual implica que las primeras letras de cada palabra deben estar en mayúsculas. Este es el caso donde un ítem se mapea a un único atributo, el cual se denomina *Class naming compliance*.

Fig. 5. Definición del ítem “5.2.2. Class names” de Google Java Style Guide [6].

Como se aprecia en la Fig. 3.a los atributos son combinados en características y subcaracterísticas. Por lo tanto, se especificó un modelo jerárquico que 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. En relación con la subcaracterística “1.1. Adherencia”, se definieron tres subcaracterísticas derivadas del análisis de los títulos incluidos en la Google Java Style Guide (Fig. 1). Por ejemplo, el título “3. Source file structure” se

renombró como “1.1.1. Source file structure compliance”, el título “4. Formatting” se renombró como “1.1.2. Formatting compliance” y el título “5. Naming” se renombró como “1.1.3. Naming compliance”.

La subcaracterística “1.1.1. Source file structure compliance” agrupa los tres *atributos* que surgieron del mapeo del ítem “3.3.3. Ordering and spacing” más un atributo adicional. Dentro de la subcaracterística 1.1.2 se agrupan cuatro atributos referidos al uso de llaves opcionales, el número de variables por declaración, sentencias por línea y tamaño máximo de una línea. Finalmente, la subcaracterística “1.1.3. Naming compliance” reúne los atributos relacionados con los nombres de clases, métodos y parámetros, entre los que se encuentra el atributo *Class naming compliance*. El modelo jerárquico completo, con los atributos indicados en cursiva, se presenta en la segunda columna de la Tabla 3.

Tabla 3. Correspondencia entre ítems de la guía de estilo Google y atributos presentes en [13].

Ítems de Google Java Style Guide	Características y *atributos* (*en itálica*)
	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*
	1.1.1.2. Spacing compliance between static and non-static import blocks
	1.1.1.3. Spacing compliance between import sentences
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*
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*
4.8.2.1. One variable per declaration	*1.1.2.2. Compliance with the number of variables per declaration*
4.3. One statement per line	*1.1.2.3. Compliance with the number of statements per line*
4.4. Column limit: 100	*1.1.2.4. Compliance with the maximum line size*
5. Naming	**1.1.3. Naming compliance**
5.2.2. Class names	*1.1.3.1. Class naming compliance*
5.2.3. Method names	*1.1.3.2. Method naming compliance*
5.2.6. Parameter names	*1.1.3.3 Parameter naming compliance*

Cada elemento del modelo jerárquico se definió conforme a lo establecido en la ontología. En la Fig. 3.a se observa que las características, subcaracterísticas y atributos poseen como propiedad el campo definición. A modo de ejemplo, en la Tabla 4 se transcriben las definiciones correspondientes a la característica 1. Maintainability, sus subcaracterísticas y los atributos presentes en el ejemplo que se está desarrollando. Para una definición de todos los elementos del modelo jerárquico ver el Apéndice I del material suplementario.

Tabla 4. Definición de algunas de las características, subcaracterísticas y atributos presentes en el modelo jerárquico. El contenido de la tabla está en ingles dado que es un extracto de lo presentado en [13].

Característica / Subcaracterísticas / *Atributos*	Definición
1. Maintainability	... the software product (e.g. source code) can be modified. Modifications may include corrections, improvements, or adaptation of the software to changes in the environment, and in requirements and functional specifications. Note: Adapted from ISO 9126-1 [3].
1.1. Compliance	... the software product (e.g. the source code) adheres to standards or conventions relating to maintainability. Note: Adapted from ISO 9126-1 [3].
1.1.1. Source file structure compliance	... the source code file adheres to the structure specified in the coding style guide for a given programming language.
1.1.1.1. Compliance with the ordering of types of imports	... the source code file adheres to the ordering of types of imports that is specified in the coding style guide.
1.1.1.2. Spacing compliance between static and non-static import blocks	... the source code file adheres to the spacing between static and non-static import blocks that are specified in the coding style guide.
1.1.1.3. Spacing compliance between import sentences	... the source code file adheres to the spacing between import sentences that is specified in the coding style guide.
...	
1.1.2 Formatting compliance	… the source code file adheres to the formatting specified in the coding style guide for a given programming language.
...	
1.1.3 Naming compliance	… the source code file adheres to the naming convention specified in the coding style guide for a given programming language.
1.1.3.1 Class naming compliance	… the source code file adheres to the class naming convention specified in the coding style guide.
...	

Una vez mapeados los ítems de la guía a atributos y agrupados en el modelo jerárquico –que se puede ver completo en la segunda columna de la Tabla 3– se deben especificar las métricas que cuantifican a cada atributo (ver Fig. 3.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 5. 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. 3.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 dos métricas directas, las cuales se definieron en las Tablas 6 y 7. A diferencia de la métrica directa que posee un procedimiento de medición, las métricas indirectas poseen un procedimiento de cálculo.

Tabla 5. Especificación de la métrica indirecta "Percentage of compliance with the ordering of types of imports (%COTI)" que cuantifica el atributo *1.1.1.1*.

Indirect Metric	
Quantified Attribute name: Compliance with the ordering of types of imports **Metric Name:** Percentage of compliance with the ordering of types of imports (%COTI)	
Objective: Determine the percentage of Java files that have the imports ordered by type with respect to the total Java files to be measured. **Author:** Pablo Becker and Luis Olsina	**Version:** 1.0
Calculation Procedure: **Formula:** $$\%COTI = \left(\frac{\sum_{i=1}^{\#JF} AOTI_i}{\#JF}\right) * 100$$	
Scale: Numeric **Value Type:** Real	**Scale Type name:** Ratio **Representation:** Continuous
Unit: **Name:** Percentage	**Acronym:** %
Related Direct Metrics: AOTI: Availability of ordering of types of imports #JF: Number of Java files	

Tabla 6. Especificación de la métrica directa "Availability of ordering of types of imports (AOTI)" necesaria para calcular la métrica indirecta "Percentage of compliance with the ordering of types of imports (%COTI)".

Direct Metric	
Quantified Attribute name: Location of imports in Java file **Metric Name:** Availability of ordering of types of imports (AOTI)	
Objective: Determine the correct location of imports in a Java file. **Author:** Pablo Becker and Luis Olsina	**Version:** 1.0

Measurement Procedure:
Type: Objective
Specification:
AOTI = SI = NSI = 0
if (there are static imports sentences) SI = 1
if (there are non-static imports sentences) NSI = 1
if (SI = 1 and NSI = 1){
if (all static imports are declared previously to all non-static imports) AOTI = 1
}else AOTI = 1	

Scale: Numeric **Value Type:** Integer	**Scale Type name:** Absolute **Representation:** Discrete
Unit: **Name:** ordered import types	**Acronym:** OIT

Tabla 7. Especificación de la métrica directa "Number of Java files (#JF)" necesaria para calcular otras métricas indirectas.

Direct Metric	
Quantified Attribute name: Size of Java files **Metric Name:** Number of Java files (#JF)	
Objective: Determine the total number of Java files in a Java project. **Author:** Pablo Becker and Luis Olsina	**Version:** 1.0
Measurement Procedure: **Type:** Objective **Specification:** #JF=0 for each file .java in the project directory #JF++	
Scale: Numeric **Value Type:** Integer	**Scale Type name:** Absolute **Representation:** Discrete
Unit: Name: Java file	**Acronym:** JF

La métrica indirecta que cuantifica el atributo 1.1.1.3. se denomina “Percentage of compliance with the spacing between import sentences” y se especifica en la Tabla 8. Esta métrica indirecta se relaciona con la métrica directa “Availability of spacing between import sentences” definida en la Tabla 9 y la métrica directa “Number of Java files” especificada en el ejemplo anterior (Tabla 7).

Tabla 8. Especificación de la métrica indirecta “Percentage of compliance with the spacing between import sentences (%CSBIS)” que cuantifica el atributo *1.1.1.3*.

Indirect Metric	
Quantified Attribute name: Spacing compliance between import sentences **Metric Name:** Percentage of compliance with the spacing between import sentences (%CSBIS)	
Objective: Determine the percentage of Java files that do not have a space between import sentences with respect to the total Java files to be measured. **Author:** Pablo Becker and Luis Olsina	**Version:** 1.0
Calculation Procedure: **Formula:** $$\%CSBIS = \left(\frac{\sum_{i=1}^{\#JF} ASBIS_i}{\#JF}\right) * 100$$	
Scale: Numeric **Value Type:** Real	**Scale Type name:** Ratio **Representation:** Continuous
Unit: **Name:** Percentage	**Acronym:** %
Related Direct Metrics: ASBIS: Availability of spacing between import sentences #JF: Number of Java files	

Tabla 9. Especificación de la métrica directa “Availability of spacing between import sentences (ASBIS)” necesaria para calcular la métrica indirecta “Percentage of compliance with the spacing between import sentences (%CSBIS)”.

Direct Metric	
Quantified Attribute name: Space between import sentences **Metric Name:** Availability of spacing between import sentences (ASBIS)	
Objective: Determine if each import sentence is separated by a blank line. **Author:** Pablo Becker and Luis Olsina	**Version:** 1.0

Measurement Procedure:
Type: Objective
Specification:
ASBIS = 1
if (there is more than one import sentence)
for each import sentence
if (the current import sentence is not the last import sentence)
if (there is a blank line between the current import sentence and the next import sentence) ASBIS = 0	

Scale: Numeric **Value Type:** Integer	**Scale Type name:** Absolute **Representation:** Discrete
Unit: **Name:** spaced import sentences	**Acronym:** SIB

Como último ejemplo, el atributo 1.1.3.1 se cuantificó con la métrica indirecta denominada “Percentage of class naming compliance”, que tiene tres métricas directas relacionadas, a saber: “Availability of valid class name”, “Number of classes” y “Number of Java files” (Tabla 7). La Fig. 6 muestra una representación gráfica de las especificaciones de las métricas necesarias para cuantificar el atributo 1.1.3.1.

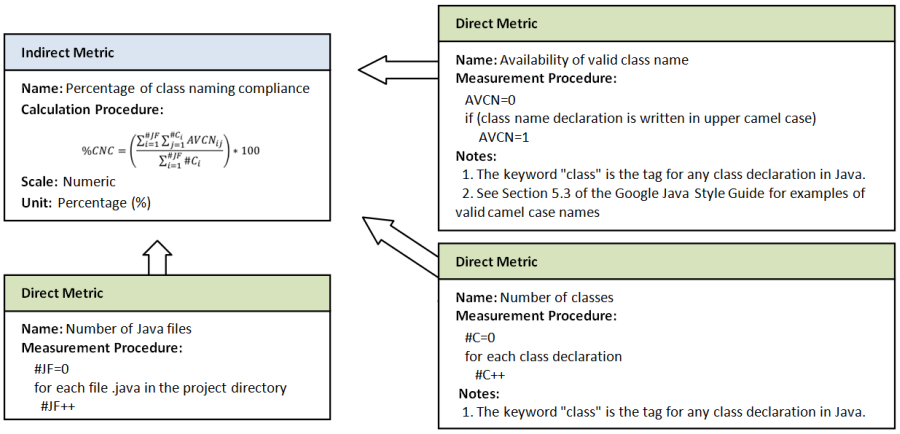


Fig. 6. Especificación gráfica de la Métrica indirecta que cuantifica el atributo "*1.1.3.1 Class naming compliance"* y sus Métricas directas.

De este modo se fueron definiendo todas las métricas que cuantifican los atributos del modelo en base al enfoque aquí presentado. Para una definición completa de las métricas ver el Apéndice II del material suplementario. A modo de resumen, en la Fig. 7 se esquematiza el enfoque utilizado con la finalidad de visualizar de manera clara las etapas del proceso, desde la identificación de los ítems de la guía de estilo hasta su mapeo con los atributos y métricas asociados.

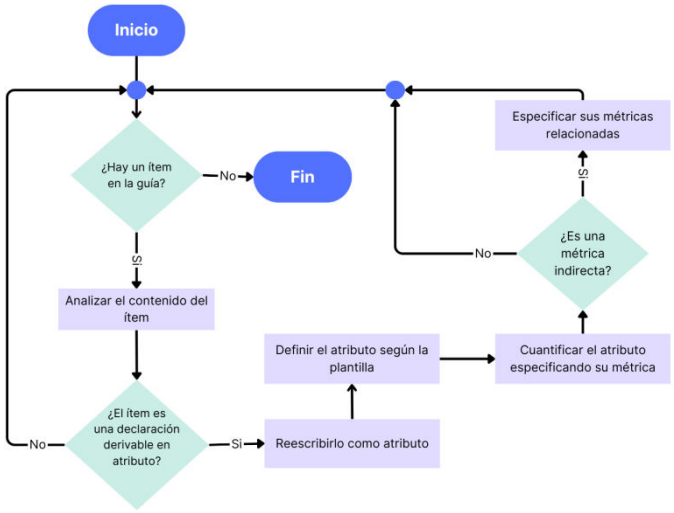


Fig. 7. Esquema del enfoque sistemático que mapea guías de estilo de programación a atributos y a sus métricas.

Google Java Style Guide consta de 87 ítems distribuidos en 7 secciones (ver Fig. 1), y el enfoque se aplicó a 37 ítems. Los ítems seleccionados fueron aquellos que establecían una regla (declaración válida a ser derivada en atributo) y no se limitaban

sólo a definiciones, comentarios o recomendaciones. Por ejemplo, mientras que el ítem "5.3 Camel case: defined" no fue considerado porque enuncia una definición, el ítem "4.8.3.1 Array initializers: can be 'block-like'" tampoco se consideró porque brinda una recomendación. Por otro lado, existen agrupaciones de otros ítems que fueron transformados en subcaracterísticas. Por ejemplo, los ítems de la guía agrupados en las secciones 1. Introduction, 6. Programming Practices y 7. Javadoc no se consideraron debido a que la primera sección es informativa, y las otras dos requieren para su medición interpretar código fuente, lo cual escapa del alcance de este trabajo.

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. Compliance with the number of top-level class declarations per source file"***.**

En resumen, los 37 ítems seleccionados se mapearon a 42 atributos, los cuales fueron combinados en características y subcaracterísticas para formar un árbol de requisitos no funcionales. Una vez diseñado el árbol, se definieron todos sus componentes y se especificaron las métricas indirectas, y sus directas, para cuantificar cada atributo. Luego, de los 42 atributos diseñados se seleccionaron 22 y se automatizaron las métricas correspondientes en JavaStyleInspector implementando más de 60 procedimientos de cálculo y de medición. Todo este trabajo fue documentado en español en [7]. En el presente artículo, como el foco está en la descripción del enfoque y en la discusión de sus beneficios y limitaciones, se decidió seguir el ejemplo publicado en [13], que es un conjunto pequeño del trabajo realizado previamente en [7].

4 Automatización de la Medición en JavaStyleInspector

4.1 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. 8 se puede observar el stack tecnológico empleado en el desarrollo.

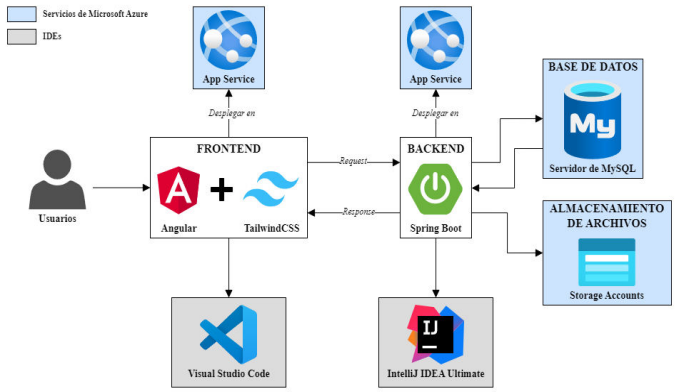


Fig. 8. Stack tecnológico utilizado en el desarrollo de JavaStyleInspector.

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.).

4.2 Aplicación de la herramienta JavaStyleInspector

En esta sección se realizará un paralelismo entre el proceso de medición seguido manualmente y la aplicación de la herramienta JavaStyleInspector para el mismo código fuente. Con este paralelismo se pretende demostrar la necesidad de contar con una herramienta automática que evalúa la adherencia de un código java a la Google Java Style Guide. Para ejemplificar el proceso de medición se utiliza el archivo "GUICalculator.java" (ver Apéndice III del material suplementario), el cual fue utilizado en [13]. Usar el mismo código fuente permite, no solo validar los resultados obtenidos por JavaStyleInspector con los realizados manualmente en dicho estudio, sino también, documentar con mayor detalle el proceso manual realizado.

Como primera acción de uso de JavaStyleInspector, 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. 9 punto 1). A continuación, se selecciona del árbol de requisitos no funcionales los atributos a medir. Para el ejemplo, se eligieron los once atributos que participaron del trabajo documentado en [13] y expuestos en la segunda columna de la Tabla 3. La Fig. 10 muestra algunos de los atributos escogidos que pertenecen a la subcaracterística 1.1. Compliance y su subcaracterística "1.1.1. Source file structure compliance". Solo queda presionar el botón medir para que los resultados se presenten en pantalla (Fig. 9 punto 3). Los resultados pueden ser descargados en formato pdf.

Para la medición manual es necesario inspeccionar el código. La Fig. 11 muestra un extracto desde donde el encargado de la medición obtiene los valores a medir para los atributos ejemplificados en este documento.

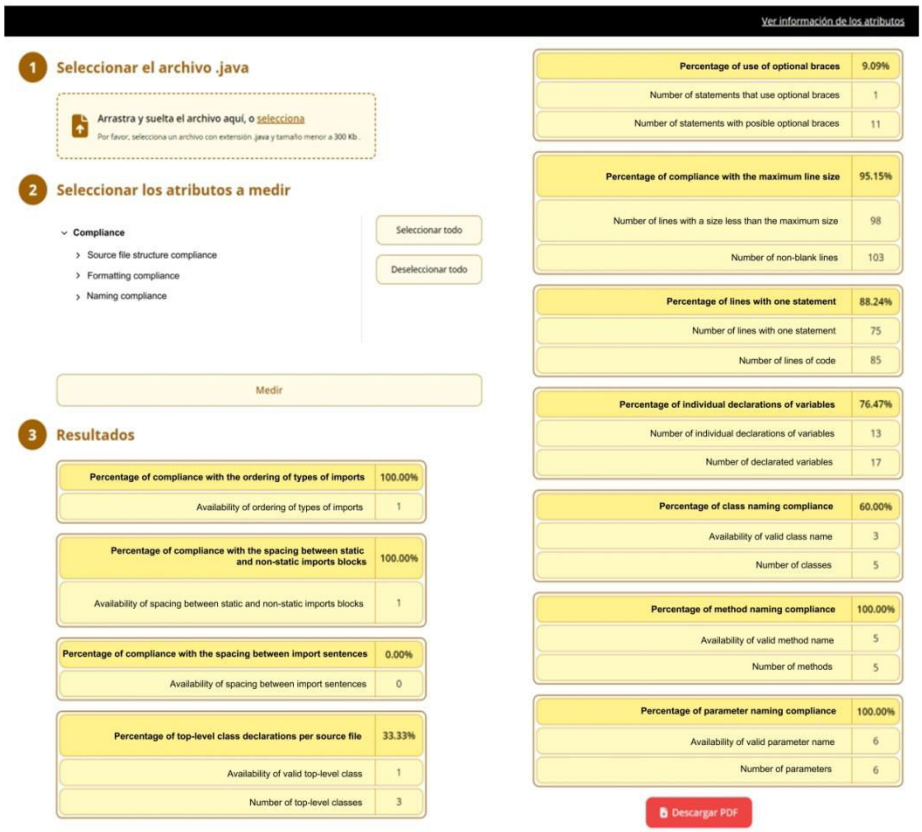


Fig. 9. Captura de pantalla de JavaStyleInspector donde se muestran los pasos para medir y sus resultados.

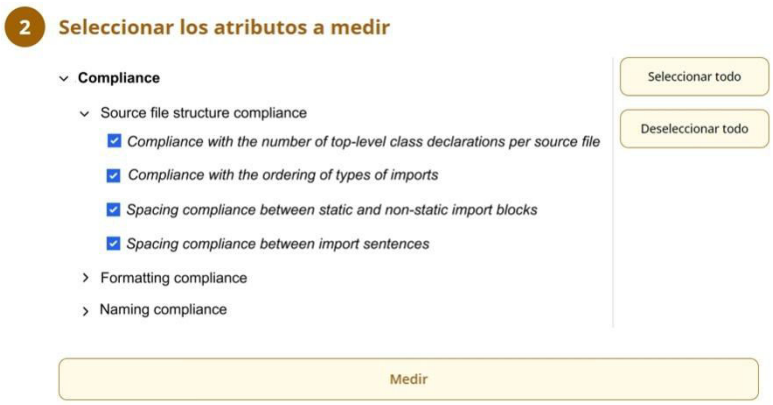


Fig. 10. Captura de pantalla de JavaStyleInspector donde se seleccionan los atributos a medir.

```
1  /*
2  This code was adapted from https://javadesdecero.es/codigos/calculadora-codigo-fuente/
3  */
4
5  package MyCalculator;
6
7  import javax.swing.JFrame;
8  import javax.swing.JPanel;
9  import javax.swing.JButton;
10
11 import java.awt.BorderLayout;
12 import java.awt.GraphicsDevice;
13 import java.awt.GraphicsEnvironment;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionEvent;
16 import java.awt.event.ActionListener;
17
18 import java.math.BigDecimal;
19
20 class GUICalculator {
21     public static void main(String[] args) {
```

Fig. 11. Extracto del archivo que contiene el código fuente Java de "GUICalculator.java" donde se puede medir los atributos *1.1.1.1.*, *1.1.1.2.* y *1.1.1.3.* Las líneas de código resaltadas en naranja indican un incumplimiento del ítem de la Google Java Style Guide.

El primer atributo a medir es el *1.1.1.1. Compliance with the ordering of types of imports*. Su cuantificación se basa en la fórmula establecida en el procedimiento de cálculo de la métrica indirecta denominada "Percentage of compliance with the ordering of types of imports (%COTI)" especificada en la Tabla 5. Para determinar este porcentaje, es necesario obtener previamente los valores de las métricas directas "Availability of ordering of types of imports (AOTI)" y "Number of Java files (#JF)", las cuales están detalladas en las Tablas 6 y 7, respectivamente. El archivo .java es único, por lo tanto #JF tiene un valor 1 y AOTI posee un valor 1 dado que hay solo sentencias de import no estáticas; por lo tanto, la condición SI=1 and NSI=1 evalúa como falsa entonces se pasa por la sentencia else donde a AOTI se le asigna 1. Una vez obtenidos estos dos valores se sustituye en la Fórmula 1 (especificada en Tabla 5) para obtener el valor de %COTI=100.

$$\%COTI = \left(\frac{\sum_{i=1}^{1} 1}{1}\right) * 100 = \left(\frac{1}{1}\right) * 100 = 100 \qquad (1)$$

La cuantificación del atributo *1.1.1.3. Spacing compliance between import sentences* está dada por la métrica indirecta denominada "Percentage of compliance with the spacing between import sentences (%CSBIS)". Al aplicar la Fórmula 2, especificada en su procedimiento de cálculo, dio un valor de %CSBIS=0. Para llegar a este resultado fue necesario aplicar el procedimiento de medición de la métrica directa denominada "Availability of spacing between import sentences (ASBIS)" y retomar el valor de #JF=1. El valor medido de ASBIS fue de 0 dado que, como se puede apreciar en la Fig. 11, hay 10 sentencias import y entre ellas existen líneas en blanco resaltadas en naranja. El procedimiento chequea si hay más de una sentencia import, si así fuera, recorre cada sentencia import verificando que no sea la última y comprobando que no existan líneas en blanco. Si ocurre esto último coloca el valor de ASBIS en 0.

$$\%CSBIS = \left(\frac{\sum_{i=1}^{1} 0}{1}\right) * 100 = \left(\frac{0}{1}\right) * 100 = 0 \quad (2)$$

Como último ejemplo del procedimiento de medición, se cuantifica al atributo *1.1.3.1. Class naming compliance* a partir de la definición de la métrica indirecta denominada "Percentage of class naming compliance (%CNC)". Para aplicar su procedimiento de cálculo es necesario obtener el valor de las métricas directas relacionadas "Availability of valid class name (AVCN)", "Number of classes (#C)" y "Number of Java file (#JF)". Todo esto está especificado en la Fig. 6. Las clases que se encuentran dentro del archivo .java (ver Apéndice III del material suplementario) son GUICalculator, calculatorFrame, calculatorpanel, InsertAction y CommandAction por lo tanto #C es igual a 5. Mientras que las clases GUICalculator, InsertAction y CommandAction fueron bien nombradas según la regla upperCamelCase, las clases calculatorFrame y calculatorpanel están mal nombradas. Esto lleva a que AVCN tenga un valor de 1 para las tres primeras clases y de 0 para las restantes. La Tabla 10 muestra los valores medidos utilizados para obtener el valor de %CNC en 60.

Tabla 10. Valores medidos y calculados para cuantificar el atributo *1.1.3.1. Class naming compliance* a partir de la definición de la métrica indirecta denominada "Percentage of class naming compliance (%CNC)".

Percentage of class naming compliance (%CNC)			60
Availability of valid class name (AVCN)			3
AVCN_1_1	(GUICalculator)	1	
AVCN_1_2	(calculatorFrame)	0	
AVCN_1_3	(calculatorpanel)	0	
AVCN_1_4	(InsertAction)	1	
AVCN_1_5	(CommandAction)	1	
Number of classes (#C)			5
#C_1		5	
Number of Java file (#JF)			1

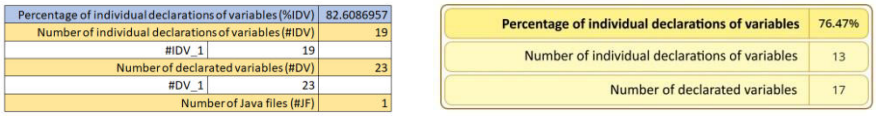
Una vez obtenidos los resultados que se presentan en la Tabla 11 y en la Fig. 9 punto 3 se efectuó el análisis de los mismos. Para ello, estos valores se compararon con los obtenidos de forma manual en [13]. Como se puede apreciar en la Tabla 11 (columnas 2 y 3), hubo tres valores diferentes. Dos de los cuales son cuestiones de redondeo (valores resaltados en naranja en la Tabla 11) y un tercero que tiene una diferencia de 6,4 puntos (resaltado en rojo en la Tabla 11). Para ver todos los valores de las métricas directas e indirectas ir al Apéndice IV del material suplementario.

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 caso de [13] se contabilizan los parámetros como un tipo de variable. La diferencia mencionada se puede observar en la Fig. 12 donde se muestran los valores medidos para los atributos "Number of individual declarations of variables (#IDV)" y "Number of declared variables (#DV)". 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.

Tabla 11. Comparativa de las mediciones realizadas en [13] y los valores obtenidos por JavaStyleInspector. Además de la correspondencia entre ítems de la guía de estilo Google y atributos presentes en [13].

Características y *atributos* (*en itálica*)	Valores de medición		Diferencia
	Manual	Herramienta	
1. Maintainability			
1.1. Compliance			
1.1.1. Source file structure compliance			
1.1.1.1. Compliance with the ordering of types of imports	100	100	0
1.1.1.2. Spacing compliance between static and non-static import blocks	100	100	0
1.1.1.3. Spacing compliance between import sentences	0	0	0
1.1.1.4. Compliance with the number of top-level class declarations per source file	33,33	33,33	0
1.1.2. Formatting compliance			
1.1.2.1. Compliance with the use of optional braces	9,09	9,09	0
1.1.2.2. Compliance with the number of variables per declaration	82,61	76,47	6,14
1.1.2.3. Compliance with the number of statements per line	88,23	88,24	0,01
1.1.2.4. Compliance with the maximum line size	95,14	95,15	0,01
1.1.3. Naming compliance			
1.1.3.1. Class naming compliance	60	60	0
1.1.3.2. Method naming compliance	100	100	0
1.1.3.3 Parameter naming compliance	100	100	0



(a) (b)

Fig. 12. (a) Valores medidos manualmente en [13] – (b) Valores medidos con JavaStyleInspector. La diferencia se debe a distintas interpretaciones del item "4.8.2.1. One variable per declaration" de la Google Java Style Guide.

Durante la etapa de pruebas de usabilidad, JavaStyleInspector fue utilizada por tres de los autores de este artículo, los cuales poseen 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 se están recolectando datos de calidad en uso para eficacia, eficiencia y satisfacción, tal como se realizó en [18].

5 Beneficios, desafíos de uso y limitaciones

5.1 Enfoque de mapeo o derivación

El enfoque detallado en este documento, que permite mapear ítems de una guía de estilo de programación a atributos y sus métricas específicas, ofrece algunos beneficios significativos, aunque también presenta desafíos y limitaciones que deben ser consideradas. Si bien el enfoque fue ejemplificado utilizando la Google Java Style Guide, puede adaptarse para cualquier otra guía de estilo de programación otorgando flexibilidad y aplicabilidad en diferentes contextos de desarrollo. Por lo tanto, los integrantes de los equipos de desarrollo pueden capitalizar el conocimiento adquirido en la aplicación del enfoque en diferentes contextos laborales siempre en pos de mejorar la mantenibilidad de su código.

Por otro lado, el enfoque propuesto ofrece un mapeo flexible o adaptable de ítems de la guía a atributos, de modo que un ítem puede mapearse a uno o varios atributos, y varios ítems pueden agruparse bajo un mismo atributo particular. Esta flexibilidad, ejemplificada al finalizar la Sección 3, facilita una medición más precisa, modular y adaptable del cumplimiento de las guías de estilo. La asociación de un ítem a múltiples atributos favorece una mayor granularidad de análisis identificando qué aspectos específicos no se cumplen y la agrupación de ítems relacionados en un único atributo simplifica la evaluación de requisitos que persiguen un objetivo común, evitando redundancias y optimizando el proceso de medición.

La definición de métricas que especifican exactamente cómo cuantificar cada atributo ofrece ventajas, en especial si se las compara con el sistema de checklist que habitualmente utilizan las guías de estilo de programación. Las métricas permiten una medición precisa, objetiva y reproducible a lo largo del tiempo. Mientras que el uso de un checklist se limita a indicar si un ítem de la guía se cumple o no, sin dar detalles de qué es lo que se incumple; el uso de métricas da información mucho más detallada y puntual. Lo que facilita la corrección del código fuente de una manera más informada, dirigida y eficiente. Por otro lado, al indicar exactamente cómo debe ser la medición y las propiedades del valor medido, proporciona información relevante que permite análisis más profundos y que los resultados puedan ser comparados a lo largo del tiempo. Permitiendo así que el equipo de desarrollo entre en un ciclo de mejora continua a partir de análisis comparativos y de evaluar cómo evoluciona el cumplimiento de las guías en las distintas etapas del ciclo de desarrollo de software.

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.

En contrapartida, 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 los beneficios ya mencionados. Y, por otro lado, a medida que los integrantes del equipo van adquiriendo experiencia en el

uso del enfoque van logrando mayor facilidad al momento de definir atributos y métricas.

5.2 JavaStyleInspector

Respecto a la herramienta JavaStyleInspector, se puede indicar que 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. Esto se pudo apreciar en la Sección 4 donde se mostraron los dos procesos para alcanzar la medición del código "GUICalculator.java". 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.

Al momento de automatizar la medición, contar con la especificación de las métricas provistas por el enfoque fue muy beneficioso dado que facilitó la programación de la herramienta, al brindar definiciones claras y criterios precisos sobre qué y cómo medir en el código fuente. Esto simplificó la implementación de los algoritmos, redujo al mínimo la ambigüedad en la interpretación de los ítems de la guía y permitió que los resultados generados por la herramienta fueran consistentes. Sin embargo, 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 Anexo III de la referencia [7].

En los 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.

6 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 detalladamente al 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 para un lenguaje específico) 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 [13].

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 [13]. 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.

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 [7]. 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 el proyecto 09/F079, y el proyecto POIRe 2023-9 de la UNLPam.

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 2025/01/03.
7. Sosa, D., Papa, M.F., Becker P., Olsina L. *Mapeo de Atributos a partir de Guías de Estilo de Programación y Automatización de sus Métricas.* In proceedings of ASSE'24, Argentine Symposium on Software Engineering, Vol: 53 JAIIO, pp. 1-14, Bahia Blanca, Argentina, August 12-16, ISSN: 2451-7593, (2024).
8. Broekhuis S.: The Importance of Coding Styles within Industries. 35th Twente Student Conference on IT (TScIT 35), pp. 1-8, (2021).
9. Kernighan, B.W., Plauger, P.J.: The Elements of Programming Style, 1st Edición. McGraw-Hill, New York, (1974).
10. Sun Microsystems: Java code conventions. https://www.oracle.com/technetwork/java /codeconventions-150003.pdf, último acceso 2024/03/21, (1997).
11. Reddy, A.: Java™ coding style guide. Sun MicroSystems, (2000).
12. Guia de desarrollo, https://uniandesdsit.github.io/coding-guidelines/style/ STYLE_GUIDE.html, Universidad de los Andes, Colombia, último acceso 2025/01/03.
13. 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).
14. Documentación de google-java-format, https://github.com/google/google-java-format, último acceso 2025/01/03.
15. 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).
16. 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).
17. 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 noviembre, Tucumán, Argentina, pp. 1-15. (2023).
18. 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).