



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Influencia del uso de recomendaciones de legibilidad en la facilidad de comprensión de software: Un estudio experimental

Sergio Luis Lubo Argumedo

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia

2019

Influencia del uso de recomendaciones de legibilidad en la facilidad de comprensión de software: Un estudio experimental

Sergio Luis Lubo Argumedo

Tesis presentada como requisito parcial para optar al título de:
Magister en Ingeniería de Sistemas y Computación

Director:

Ph.D. Jairo Hernán Aponte Melo

Línea de Investigación:

Evolución y Mantenimiento de Software

Universidad Nacional de Colombia

Facultad de Ingeniería, Departamento de Ingeniería de Sistemas y Computación

Bogotá, Colombia

2019

Dedicatoria

A Dios Padre por darme sabiduría y paciencia y darme a mis padres, Gaspar y Elvira, quienes, con su dedicación y amor, escucha, paciencia y comprensión, me ha permitido llegar a ser la persona y profesional que soy hoy.

A mis hermanos William, Eddie y Marie por ser mis compañeros de camino y apoyarme en todo.

A mis amigos y compañeros de trabajo por acompañarme y alentarme a seguir.

A todas y cada una de las personas que de una u otra forma me apoyaron y acompañaron en esta aventura.

Agradecimientos

A todo el equipo de trabajo, profesores, estudiantes y administrativos, de la Facultad de Ingeniería y del programa de Maestría en Ingeniería de Sistemas y Computación de la Universidad Nacional, de manera especial al profesor y director de este trabajo de investigación, Jairo Hernán Aponte Melo por profesionalismo, dedicación, apoyo y confianza en este proceso de formación.

A la profesora Bonita Sharif y al estudiante John Johnson de la Universidad de Nebraska por su participación y apoyo para alcanzar los objetivos de esta investigación, de igual forma agradezco a Nishitda Yelda de la Universidad de Youngstown por sus aportes y participación.

Al Departamento de Estadística de la Universidad Nacional de Colombia en especial a los estudiantes Hernán Torres y León Muñoz por su apoyo y aportes en el análisis estadístico de esta investigación.

A todos mis compañeros de curso en especial a Juan Manuel Cubillos, Freddy Forero, David Camilo Delgado y Sebastián Alejandro Velasco.

A mis amigos y compañeros de trabajo de NovaSec SAS por apoyarme y permitirme el tiempo para desarrollar este proceso, en especial a Fabian, Miguel y Jaime Andrés.

A todos mis compañeros, amigos y profesores de la Universidad del Magdalena, en especial a ERMR y EZZ por su apoyo y participación en este proceso.

A toda mi familia por acompañarme y alentarme día a día continuar.

Resumen

Una de las actividades más frecuentes en mantenimiento de software es la lectura y comprensión de código fuente la cual demanda una proporción considerable del tiempo de los desarrolladores de software. Si el código fuente, no está escrito siguiendo buenas prácticas de programación los tiempos de mantenimiento de este pueden incrementarse. Con el propósito de disminuir los tiempos de lectura y comprensión es necesario por tanto considerar qué tan legible es el código fuente. El consenso general es que el código fuente debe escribirse pensando en minimizar el tiempo que tardarán otros desarrolladores en leerlo y comprenderlo. En esta investigación, se realizó un experimento controlado para analizar dos reglas de legibilidad de anidamiento y bucles. Se utilizaron 32 fragmentos (métodos) Java en cuatro categorías: unos que siguen la regla, otros que no siguen la regla y que son lógicamente correctos o incorrectos. El experimento se realizó en línea con 275 participantes. Los resultados indican que minimizar el anidamiento de estructuras de control disminuye el tiempo que los desarrolladores requieren para leer y comprender código fuente, aumenta su nivel de confianza sobre el grado de comprensión del código y también mejora la capacidad para encontrar errores (*bugs*). Los resultados también muestran que la regla de evitar el uso de bucles `do/while` no tuvo impacto significativo en ninguno de los aspectos analizados. Por último, al ser un experimento totalmente en inglés, también se encontró que cuanto mayor era el nivel de conocimiento en inglés los resultados asociados a la regla de minimizar el anidamiento fueron mejores. Se analizan las implicaciones de estos hallazgos sobre legibilidad y comprensión de código fuente.

Palabras clave: program comprehension, source code readability rules, correctness, controlled experiment.

Abstract

Software developers spend a significant amount of time reading source code. If code is not written with readability in mind, it impacts the time required to maintain it. In order to alleviate the time taken to read and understand code, it is important to consider how readable the code is. The general consensus is that source code should be written to minimize the time it takes for others to read and understand it. In this paper, we conduct a controlled experiment to assess two code readability rules: nesting and looping. We test 32 Java methods in four categories: ones that follow/do not follow the readability rule and that are correct/incorrect. The study was conducted online with 275 participants. The results indicate that minimizing nesting decreases the time a developer spends reading and understanding source code, increases her level of confidence about her own understanding of the code, and also suggests that it improves her ability to find bugs. The results also show that avoiding the do-while statement had no significant impact on level of understanding, time spent reading and understanding, confidence in understanding, or ease of finding bugs. It was also found that the better knowledge of English a participant had, the more their readability and comprehension confidence ratings were affected by the minimize nesting rule. We discuss the implications of these findings for code readability and comprehension.

Keywords: program comprehension, source code readability rules, correctness, controlled experiment.

Contenido

	Pág.
1. Estado del arte.....	5
1.1 Características de nivel superficial	5
1.2 Características estructurales	7
2. Estudio experimental	9
2.1 Diseño experimental.....	9
2.2 Preguntas de investigación e Hipótesis	10
2.3 Objetos y tareas	13
2.4 Participantes	17
2.5 Procedimiento	19
3. Análisis de resultados	21
3.1 Segundo análisis.....	21
3.1.1 Nivel de comprensión	22
3.1.2 Tiempo de comprensión	23
3.1.3 Nivel de confianza	24
3.1.4 Evaluación de la legibilidad.....	24
3.1.5 Calificación de legibilidad.....	25
3.1.6 Factores secundarios: Lengua materna, Conocimiento de lenguaje Java y Competencias en inglés (Reading).....	26
3.1.7 Post cuestionario	27
3.2 Amenazas a la validez	28
3.2.1 Validez interna.....	29
3.2.2 Validez externa	29
3.2.1 Validez de constructo.....	30
3.2.2 Validez de conclusión	31
3.3 Observaciones y Discusión	31
4. Conclusiones.....	35

Lista de figuras

	Pág.
Figura 2-1 P1S1, Regla R1	15
Figura 2-2 P1S2, Regla R1	15
Figura 2-3 P1S3, Regla R1	16
Figura 2-4 P1S4, Regla R1	16
Figura 2-5 Participantes por origen y lengua materna	17
Figura 2-6 Participantes por experiencia como desarrollador profesional.....	18
Figura 2-7 Participantes por experiencia y conocimientos en Java.....	18
Figura 2-8 Participantes por competencias en inglés	19
Figura 3-1 Ranking de importancia regla R1	28
Figura 3-2 Ranking de importancia regla R2	28
Figura 3-3 Consolidado de los tiempos de lectura y respuesta.....	32

Lista de tablas

	Pág.
Tabla 2-1 Resumen del experimento	11
Tabla 2-2 Hipótesis nulas y alternativas	12
Tabla 2-3 Problemas seleccionados para R1 (Minimize nesting)	14
Tabla 2-4 Problemas seleccionados para R2 (<i>Avoid Do/While loops</i>)	14
Tabla 2-5 Muestra de una combinación de métodos y problemas para R1	20
Tabla 3-1 Pruebas <i>t-student</i> y <i>Cohen's d</i> - nivel de comprensión, regla R1	22
Tabla 3-2 Pruebas <i>t-student</i> y <i>Cohen's d</i> - nivel de comprensión, regla R2	23
Tabla 3-3 Pruebas <i>t-student</i> y <i>Cohen's d</i> - tiempo de comprensión	23
Tabla 3-4 Pruebas <i>t-student</i> y <i>Cohen's d</i> - nivel de confianza.....	24
Tabla 3-5 Pruebas <i>t-student</i> y <i>Cohen's d</i> – evaluación de legibilidad.....	25
Tabla 3-6 Pruebas <i>t-student</i> y <i>Cohen's d</i> , calificación de legibilidad y autoevaluación de inglés.....	26
Tabla 3-7 Pruebas <i>t-student</i> y <i>Cohen's d</i> , nivel de confianza y autoevaluación de inglés.....	27

Introducción

La comprensión de código fuente (*“Program Comprehension”* en inglés) es la base fundamental de muchas actividades de ingeniería del software como son: Revisión de código, corrección de defectos, refactorización o implementación de nuevas funcionalidades [1] [2] [3]. En este sentido, son muchos aspectos en que la forma como se escribe el código fuente, como adoptar un diseño, formato y convenciones de nombres en lugar de otro, puede afectar la comprensión de este. El código fuente escrito por los desarrolladores es leído no solo por ellos sino también por otros en el equipo y más allá. Por lo tanto, es crucial que el código no solo satisfaga los requerimientos funcionales, sino que debe ser escrito en una forma que facilite la legibilidad. Los efectos de un código legible van mucho más allá de lo estético, este puede tener efectos significativos en los tiempos requeridos por los desarrolladores para construir modelos mentales del software durante el desarrollo de tareas de mantenimiento del mismo [1] [3] [4]. Para el lenguaje natural, como el inglés, se han desarrollado algunas métricas como Flesch’s Reading Ease Score [5], con la cual es posible determinar el nivel de legibilidad de un texto. En código fuente estas métricas no funcionan tan bien dado que el código fuente es inherentemente bastante distinto al lenguaje natural [6]. El código fuente es rico en sintaxis y posee una capa adicional de semántica involucrada.

En trabajos recientes, Scalabrino et al [7] [8], revisaron más de cien características en fragmentos de código, pero no evidenciaron ninguna correlación de estas con algún modelo operacional de comprensión de código fuente. Los métodos anteriores derivan del trabajo de Borstle et al [9], los cuales toman en cuenta identificadores, palabras claves *“keyword”*, sentencias y longitud de las palabras en sílabas. Trockman et al [10], reanalizó el conjunto de datos de Scalabrino con el propósito de generar una métrica para la comprensión de código fuente, sin embargo, concluyeron que solo encontraron una pequeña significancia correlacional con la comprensión de código fuente al combinar múltiples características de los fragmentos de código.

Por lo tanto, la pregunta sigue siendo ¿cómo operacionalizar la legibilidad del código fuente? Para ayudar en este objetivo y estudiar el problema con más detalle desde la perspectiva del desarrollador, esta investigación presenta los resultados de un largo estudio empírico realizado con 275 participantes para determinar la legibilidad de unos fragmentos de código fuente. Estudios previos se centran fuertemente en el código fuente como métrica, sin realizar preguntas de comprensión de código a los desarrolladores. En esta investigación se presenta un primer trabajo que busca medir el nivel de legibilidad en fragmentos de código correctos e incorrectos, donde además se mide el nivel de comprensión del código a partir de preguntas.

La métrica de complejidad ciclomática de McCabe ha sido usada ampliamente para calcular el nivel de complejidad de programas. Uno de sus usos originales era realizar un seguimiento y restringir la complejidad de las rutinas de un programa mediante el uso de un umbral que nunca debería ser sobrepasado. Sin embargo, estudios empíricos han demostrado que esta métrica no tiene relación con el nivel de legibilidad percibido por los desarrolladores [11] [12] [13]. Uno de los problemas de la métrica de complejidad es que da el mismo peso para todas las estructuras de control [14]. Para superar este problema, se han propuesto métricas para estimar el esfuerzo de comprensión de las estructuras de control típicas de los lenguajes de programación imperativos [15] [16]. El objetivo de estas propuestas es asignar un peso cognitivo a cada estructura de control. Sin embargo, existe poca evidencia empírica que apoye los diferentes pesos asignados. Una excepción es un estudio reciente en que Ajami et al, evaluaron empíricamente la dificultad de la lectura de bucles (*loops*), estructuras de control (*if*), predicados y el efecto de los modismos en la comprensión de código fuente [14].

En esta investigación se provee evidencia empírica relacionada con dos prácticas de programación ampliamente recomendadas destinadas a mejorar la legibilidad del código mediante la simplificación de bucles y lógica. Los resultados empíricos proporcionan evidencia para mejorar nuestras prácticas de programación actuales y eventualmente, identificar características estructurales del código fuente que ayuden a definir mejores métricas de legibilidad. Las reglas reducción del anidamiento de estructuras de control (*minimize nesting*) y evitar el uso del bucle do-while (*avoid do-while loops*) [14] fueron seleccionadas por que son prácticas de codificación antiguas y populares sin ninguna

evidencia empírica de su efecto real en la legibilidad percibida y en la comprensión del código fuente. Por lo tanto, esta investigación busca proporcionar dicha evidencia a través de un experimento controlado. Los resultados muestran compatibilidad con la regla *minimize nesting* tanto en términos de tiempo eficientes de lectura, como en la capacidad para encontrar errores (*bugs*). Sin embargo, no se encontró compatibilidad alguna con la regla *avoid do/while loops*. Se considera y propone este estudio como un primer intento sistemático para investigar fragmentos de códigos correctos e incorrectos que aplican recomendaciones de legibilidad.

En consideración a lo anterior, se pueden resumir las principales contribuciones de esta investigación como sigue:

- Se analizan, con respecto a un conjunto de métricas de comprensión de código, dos recomendaciones (reglas) de legibilidad, reducción de anidamiento de estructuras de control (*minimize nesting*) y evitar el uso de bucles *do/while* (*avoid do/while loops*).
- Se analizan fragmentos de código lógicamente correctos e incorrectos con respecto a si estos siguen o no las reglas de legibilidad.
- Se realiza un análisis comparativo de la percepción de legibilidad a partir de los fragmentos de código correctos que siguen cada una de las reglas analizadas.

Por último, es de importancia mencionar que algunos apartes de esta tesis se encuentran publicados en el artículo “*An Empirical Study Assessing Source Code Readability in Comprehension*” [17], el cual fue aceptado en la conferencia “*The 35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*” que se llevará a cabo en Cleveland Ohio, USA, del 30 de septiembre al 4 de octubre de 2019. (Anexo C)

1.Estado del arte

Una gran variedad de características del código fuente han sido estudiadas con el propósito de identificar en qué medidas estas afectan la comprensión de software. En este capítulo se reportan experimentos previos que proporcionan evidencia empírica del impacto de las características del código fuente en la comprensión de software, contextualizando aquellos en los cuales los desarrolladores leen y comprenden fragmentos de código.

1.1 Características de nivel superficial

En este nivel existen características que cuando se toman en conjunto, pueden afectar en gran medida la comprensión de todo un sistema de software. En esta categoría, se han estudiado empíricamente, aspectos como los nombres de los identificadores, comentarios, formato y resaltado de sintaxis, entre otros, siendo los identificadores el foco principal de muchos de estos estudios, en consideración a que los identificadores representan una parte importante del texto de los programas y el contenido sintáctico de los mismos es alto. Por ejemplo, la evidencia ha mostrado que usar identificadores de palabras completas, en lugar de identificadores de una sola letra o abreviaturas, incrementa la legibilidad del código [18]; de otra parte, el uso de la notación *camel-case* resulta ser mejor que el estilo *underscore* para reducir el esfuerzo e incrementar el nivel de comprensión de código, especialmente en programadores principiantes [19]; de igual forma, identificadores largos pero más informativos mejoran la comprensión del código, especialmente cuando se requiere un nivel de comprensión profundo del mismo [20]. En este mismo sentido, los términos individuales en el código fuente se relacionan satisfactoriamente con medidas de

carga cognitiva de los desarrollares [21]. En contraposición, los patrones anti-lingüísticos¹ (*antipatterns*) en el código fuente incrementan significativamente la carga cognitiva de los desarrolladores durante las tareas de comprensión de código fuente.

El resaltado de sintaxis permite a los desarrolladores mostrar el código fuente en diferentes colores y fuentes de acuerdo con la categoría de los términos. En este sentido estudios empíricos han demostrado, por ejemplo, que, aunque los desarrolladores sienten que los fragmentos de código con colores son más fáciles de leer y estéticamente más agradables, no existe diferencia significativa entre el esfuerzo visual requerido para leer código en blanco y negro versus código con colores [22]. Del mismo modo, en el contexto más específico de directivas de preprocesador utilizadas en líneas de producto de software, una familia de experimentos controlados sugiere que los desarrolladores prefieren colores de fondo y estos tienen el potencial de mejorar la comprensión de programas [23].

Otros estudios se han orientado a analizar los posibles efectos de los grupos léxicos y sintácticos en la facilidad de comprensión de código. En cuanto al estilo de codificación, los modelos de legibilidad propuestos por Buse & Weimer [24], y Scalabrino et al. [8], han sido usados como base para investigar el impacto de prácticas de programación en la legibilidad percibida por los desarrolladores de software [25]. Los datos recopilados muestran evidencia de que la mayoría de las prácticas mejoran la legibilidad, algunas de estas obstaculizan la legibilidad y otras no parecen tener ningún efecto significativo. Un estudio sobre el mismo tema muestra que los aspectos del formato de código (por ejemplo, espacios, inicialización de variables, alineación del código, orden de las instrucciones, sobrecarga de operadores, orden de llamada de funciones) a menudo considerados triviales, pueden impactar profundamente en la comprensión de los desarrolladores [26]. Un experimento similar de *eye-tracking*² evaluó la importancia de las características estructurales y textuales del código fuente sobre su legibilidad y comprensibilidad [27]. Los resultados mostraron que la ausencia de características textuales incrementa la duración

¹ Los patrones anti lingüísticos son malas prácticas recurrentes en la nomenclatura, documentación y elección de identificadores de entidades de programas, los cuales son percibidos negativamente por los desarrolladores y que podrían afectar la comprensión del código fuente.

² Eye tracking es el proceso de evaluar, bien el punto de fijación de la mirada (donde estamos mirando), o el movimiento del ojo en relación con la cabeza.

promedio de las fijaciones, lo que sugiere que las características textuales son importantes para la comprensión del código fuente.

1.2 Características estructurales

Las características estructurales de código fuente como son las declaraciones de flujos de control y las condiciones lógicas, introducen saltos y ramas que pueden hacer que el código sea difícil de entender y mantener. Los efectos de estas características en la legibilidad de los programas de los desarrolladores ha sido una preocupación que se refleja en muchas prácticas de codificación recomendadas en innumerables textos y estándares de programación. Sin embargo, hay una pequeña evidencia empírica para confirmar o refutar la efectividad de dichas prácticas. La evidencia empírica sugiere que el bucle de lectura/proceso es más fácil de entender que el bucle de proceso/lectura, al menos para los estudiantes [28]. Además, los bucles son mucho más difíciles de entender que las sentencias IF y los bucles de cuenta regresiva son más difíciles de entender que los bucles de cuenta progresiva [14]. Hallazgos empíricos también muestran que ni la legibilidad ni la comprensión de código se ven afectadas por las variantes de los métodos en cadena [29]. Por otra parte, desde el punto de vista educativo, la evidencia ha permitido identificar varias correcciones que se pueden aplicar frecuentemente para mejorar el código ilegible entre estas se incluyen, mejoras en los nombres de las variables y métodos, la creación de nuevos métodos para reducir la duplicación de código, simplificación de estructuras y condiciones IF y simplificación de condiciones de bucles [30].

2. Estudio experimental

En este capítulo se presentan los detalles del estudio experimental realizado como parte del proyecto de investigación. Las secciones que se describen son: Diseño experimental, Preguntas de investigación e hipótesis, Objetos y tareas, Participantes, Procedimiento y Verificación.

2.1 Diseño experimental

El objetivo principal de esta investigación es analizar dos recomendaciones o reglas de legibilidad, prácticas de programación, con el propósito de proveer evidencia empírica del impacto de estas en la comprensión de código fuente. El foco principal se centra en la efectividad, asociada al nivel de comprensión alcanzado por los participantes; la eficiencia, asociada al tiempo utilizado por los participantes; y el nivel de confianza de los participantes acerca de su propio grado de comprensión del código fuente analizado. La perspectiva de la investigación es evaluar hasta qué punto las reglas evaluadas influyen en la comprensión de código fuente en el contexto en que los participantes leen y comprenden fragmentos de código fuente en lenguaje Java los cuales siguen/no siguen las reglas evaluadas y son soluciones correctas/incorrectas a los problemas de programación planteados. Las reglas analizadas se centran en la facilidad de estas para la lectura del flujo de control del código fuente. Específicamente se consideraron las siguientes dos prácticas de codificación como reglas de legibilidad de un conjunto de 25 opciones, las cuales se describen en el Anexo D:

- **R1: Minimizar el anidamiento de estructuras de control (*Minimize nesting*)**, esta recomendación consiste en que el código profundamente anidado es difícil de entender, cada nivel de anidamiento obliga al lector a agregar una condición adicional en su “pila mental” y al encontrar una llave de cierre (}) u observar el cierre de un bloque es difícil recordar cual era la condición previa [31].

- **R2: Evitar el uso de ciclos Do/While (*Avoid do/while loops*)**, esta recomendación consiste en que por lo general las condiciones lógicas se encuentran al inicio del bloque de código que conforman y que los desarrolladores normalmente leen código de arriba hacia abajo esto hace que la estructura del bucle Do/While sea un poco antinatural dado que la condición lógica se encuentra al final del bloque, por esto muchos desarrolladores terminan leyendo el código dos veces. Afortunadamente, las estructuras Do/While rara vez son necesarias y en la práctica la mayoría de los bucles Do/While pueden reescribirse como bucles While, es por estas razones que se recomienda evitar el uso de los bucles Do/While [31].

Los criterios para seleccionar estas dos recomendaciones (reglas de tipo algorítmico) fueron, en primer lugar, porque las recomendaciones de carácter superficial (por ejemplo, notación CamelCase versus Underscore) ya han sido estudiadas por diversos autores y se han publicado varios resultados interesantes, como los presentados en la sección 1.1 y las recomendaciones de diseño (reorganización de código) son complicadas de estudiar en fragmentos pequeños de código, ahora bien dentro de las reglas de tipo algorítmico fueron seleccionadas R1 y R2 por ser controversiales como el es el caso del *avoid Do/While loops* y las más recomendadas por los autores en todos los lenguajes procedimentales, como es el caso de *minimize nesting*, con el criterio adicional que estas no hubieran sido estudiadas empíricamente en otros estudios.

2.2 Preguntas de investigación e Hipótesis

La evidencia empírica buscada en la investigación podría respaldar o refutar la idea que R1 (*minimize nesting*) y R2 (*avoid do/while loops*) ayudan a los desarrolladores a comprender mejor el código fuente. En este sentido se definieron y midieron 3 variables dependientes: el tiempo en segundos gastado por cada participante durante la lectura y entendimiento de cada fragmento de código; el nivel de confianza de cada participante respecto a su grado de comprensión y el nivel de comprensión de cada participante. En este orden de ideas se definieron las siguientes preguntas de investigación:

RQ1a ¿La regla "*minimize nesting*" disminuye el tiempo invertido por los desarrolladores en la lectura y comprensión de código fuente?

RQ1b ¿La regla "*minimize nesting*" incrementa el nivel de confianza de los desarrolladores respecto a su grado de comprensión?

RQ1c ¿La regla “*minimize nesting*” mejora el nivel de comprensión de los desarrolladores al analizar código fuente?

RQ2a ¿La regla “*avoid do/while loops*” disminuye el tiempo invertido por los desarrolladores en la lectura y comprensión de código fuente?

RQ2b ¿La regla “*avoid do/while loops*” incrementa el nivel de confianza de los desarrolladores respecto a su grado de comprensión?

RQ2c ¿La regla “*avoid do/while loops*” mejora el nivel de comprensión de los desarrolladores al analizar código fuente?

De igual manera se consideraron las siguientes dos variables independientes: *Correctitud* y *Seguimiento de la regla*. La Correctitud indica si el fragmento de código Java analizado es una solución correcta al problema especificado, mientras el Seguimiento de la regla se refiere a si el fragmento de código Java sigue la recomendación de legibilidad. En la Tabla 2-1 se presenta el resumen de las variables dependientes e independientes y la Tabla 2-2 presenta las hipótesis nulas y alternativas definidas.

Tabla 2-1 Resumen del experimento

Propósito	Analizar dos recomendaciones o reglas de legibilidad, prácticas de programación, con el propósito de proveer evidencia empírica del impacto de estas en la comprensión de código fuente
Variables independientes (factores experimentales)	Correctitud, Seguimiento de la regla
Variables Dependientes	Tiempo de comprensión de código fuente, Nivel de confianza en el grado de comprensión Nivel de comprensión

Tabla 2-2 Hipótesis nulas y alternativas

Hipótesis Nula	Hipótesis alternativa
H1 ₀ . El uso de la regla “ <i>minimize nesting</i> ” no produce una reducción significativa en el tiempo utilizado por los desarrolladores para entender código fuente.	H1. El uso de la regla “ <i>minimize nesting</i> ” produce una reducción significativa en el tiempo utilizado por los desarrolladores para entender código fuente.
H2 ₀ . El uso de la regla “ <i>minimize nesting</i> ” no produce un incremento significado en el nivel de confianza de los desarrolladores acerca de su grado de comprensión del código fuente.	H2. El uso de la regla “ <i>minimize nesting</i> ” produce un incremento significado en el nivel de confianza de los desarrolladores acerca de su grado de comprensión del código fuente.
H3 ₀ . El uso de la regla “ <i>minimize nesting</i> ” no produce una mejora significativa en el nivel de comprensión de los desarrolladores al analizar código fuente.	H3. El uso de la regla “ <i>minimize nesting</i> ” produce una mejora significativa en el nivel de comprensión de los desarrolladores al analizar código fuente.
H4 ₀ . El uso de la regla “ <i>avoid do/while loops</i> ” no produce una reducción significativa en el tiempo utilizado por los desarrolladores para entender código fuente.	H4. El uso de la regla “ <i>avoid do/while loops</i> ” produce una reducción significativa en el tiempo utilizado por los desarrolladores para entender código fuente.
H5 ₀ . El uso de la regla “ <i>avoid do/while loops</i> ” no produce un incremento significado en el nivel de confianza de los desarrolladores acerca de su grado de comprensión del código fuente.	H5. El uso de la regla “ <i>avoid do/while loops</i> ” produce un incremento significado en el nivel de confianza de los desarrolladores acerca de su grado de comprensión del código fuente.
H6 ₀ . El uso de la regla “ <i>avoid do/while loops</i> ” no produce una mejora significativa en el nivel de comprensión de los desarrolladores al analizar código fuente.	H6. El uso de la regla “ <i>avoid do/while loops</i> ” produce una mejora significativa en el nivel de comprensión de los desarrolladores al analizar código fuente.

2.3 Objetos y tareas

Para cada regla se seleccionaron cuatro problemas algorítmicos, los cuales se presentan en la Tabla 2-3 y en la

Tabla 2-4. Por cada problema se diseñaron cuatro soluciones cada una de estas como un método Java. Así, una solución o método para cada problema P_k se denota como P_kS_j , donde $k \in 1..4$ y $j \in 1..4$. Con respecto a los dos factores experimentales, las cuatro soluciones de P_k representan los cuatro tratamientos diferentes, T1, T2, T3 y T4 los cuales tienen las siguientes características:

- P_kS_1 es una solución correcta para P_k que sigue la regla de legibilidad de interés (Tratamiento T1).
- P_kS_2 es una solución correcta para P_k que no sigue la regla de legibilidad de interés (Tratamiento T2).
- P_kS_3 es una solución incorrecta para P_k que sigue la regla de legibilidad de interés (Tratamiento T3).
- P_kS_4 es una solución incorrecta para P_k que no sigue la regla de legibilidad de interés (Tratamiento T4).

De acuerdo con lo anterior, para la verificación de cada regla se usaron 16 fragmentos de código fuente (métodos java). El diseño de estos métodos no incluyó dependencias de otros métodos ni objetos de otras clases, para que, de este modo, los participantes no requirieran revisar información adicional para entender estos métodos. (ver Figura 2-1, Figura 2-2, Figura 2-3, Figura 2-4). Las principales tareas creadas para el propósito del experimento se dividieron en dos categorías: *Análisis de métodos* y *Comparación de métodos*. Dentro de cada tarea de análisis de métodos, los participantes debían leer el enunciado del problema, analizar uno de los cuatro métodos de solución propuestos para el problema, calificar la legibilidad del método, responder una pregunta de opción múltiple asociada a la comprensión del método, calificar el nivel de confianza sobre su propio grado de comprensión y por último determinar si el método es correcto o no. Por otro lado, en cada tarea de comparación los participantes debían leer el enunciado del problema P_k , analizar las dos soluciones correctas del problema (P_kS_1 y P_kS_2) y después determinar cuál es la solución más legible, justificando su elección.

Tabla 2-3 Problemas seleccionados para R1 (Minimize nesting)

Problema	Enunciado
P1	<i>Given three integer numbers, the method must return the greatest.</i>
P2	<i>Given a mark which is an integer between 0 and 100, the method must return a letter. Letter A if mark ≥ 90; B if mark $\in [80, 90)$; C if mark $\in [70, 80)$; D if mark $\in [60, 70)$; and F if mark < 60.</i>
P3	<i>Given the body mass index (bmi), the method must return the category in which the index is located. The category is “very severely underweight” if $bmi \leq 15$; “severely underweight” if $bmi \in [15, 16)$; “underweight” if $bmi \in [16, 18.5)$; “healthy weight” if $bmi \in [18.5, 25)$; “overweight” if $bmi \geq 25$.</i>
P4	<i>Given three integers, the method must count how many of them are positive numbers.</i>

Tabla 2-4 Problemas seleccionados para R2 (Avoid Do/While loops)

Problema	Enunciado
P1	<i>Ask the user to answer a multiple-choice question. Show the question, get the user response, and end when the user chooses the correct answer or when she decides not to try more (typing 'q' or 'e')</i>
P2	<i>Let's assume that we want to force a user to change her password. The user must give a new password that must have 4 different characters. Besides, the given password must be different to the old one. The method receives the old password as a parameter and asks the user to give the new one.</i>
P3	<i>Add the positive numbers in an array. The method receives an integer array as a parameter and must return the sum of the positive numbers in the array.</i>
P4	<i>Count the occurrences of a character. This method receives a string and a character as parameters. It must count and return the number of occurrences of the character in the string.</i>

Figura 2-1 P1S1, Regla R1

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = numOne;
    if (numTwo > theLargest) {
        theLargest = numTwo;
    }
    if (numThree > theLargest) {
        theLargest = numThree;
    }
    return theLargest;
}
```

Figura 2-2 P1S2, Regla R1

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest;
    if (numOne < numTwo) {
        if (numTwo < numThree) {
            theLargest = numThree;
        } else {
            theLargest = numTwo;
        }
    } else {
        if (numOne < numThree) {
            theLargest = numThree;
        } else {
            theLargest = numOne;
        }
    }
    return theLargest;
}
```

Figura 2-3 P1S3, Regla R1

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = 0;
    if ((numOne > numTwo) && (numOne > numThree)) {
        theLargest = numOne;
    }
    if ((numTwo > numOne) && (numTwo > numThree)) {
        theLargest = numTwo;
    }
    if ((numThree > numTwo) && (numThree > numOne)) {
        theLargest = numThree;
    }
    return theLargest;
}
```

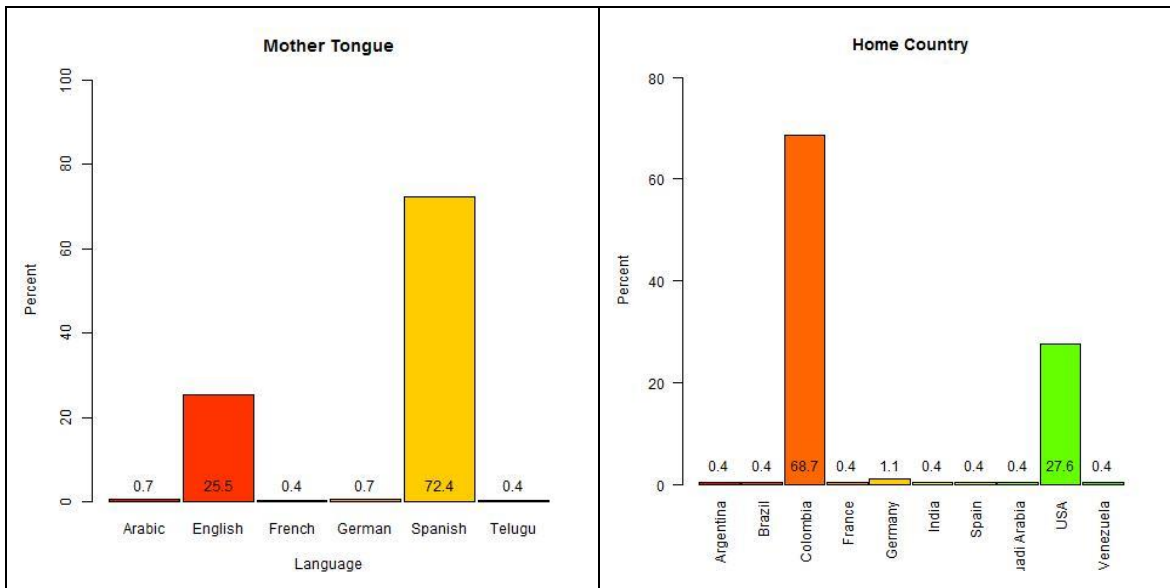
Figura 2-4 P1S4, Regla R1

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = 0;
    if (numTwo < numThree) {
        theLargest = numThree;
    } else {
        if (numTwo > numOne) {
            theLargest = numTwo;
        }
    }
    if ((numOne > numTwo) && (numOne > numThree)) {
        theLargest = numOne;
    }
    return theLargest;
}
```

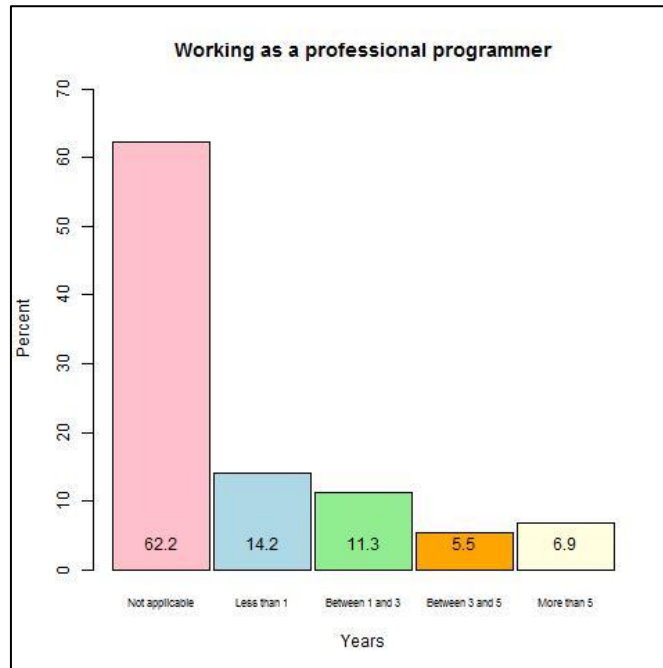
2.4 Participantes

El estudio fue realizado con 275 participantes. De estos, 208 participantes (75.6%) son estudiantes de Ingeniería de Sistemas, 49 (17.8%) son estudiantes de Maestría en Ingeniería de Sistemas y 61 (22.18%) trabajan actualmente como desarrolladores profesionales. Los participantes son de Colombia (68.7%), Estados Unidos (27.6%), Alemania (1.1%) y Argentina (0.4%). La lengua materna de los participantes es Español (72.4%), Inglés (25.5%), Árabe (0.7%), Alemán (0.7%), Telugu (0.4%) y Francés (0.4%) (ver Figura 2-5).

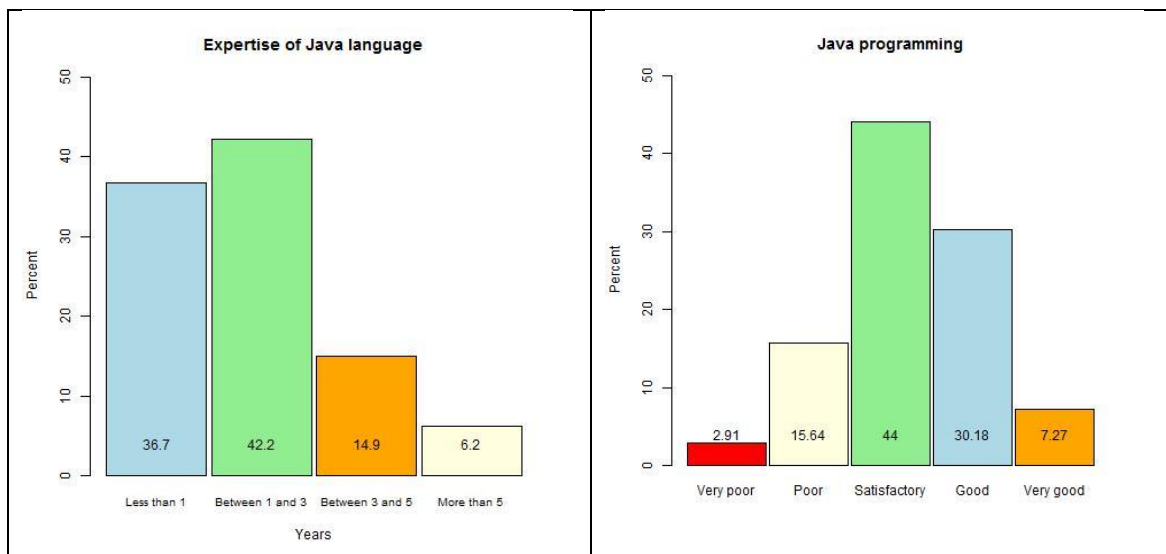
Figura 2-5 Participantes por origen y lengua materna



Con relación a la experiencia como Desarrollador profesional de los participantes, 19 (6,9%) reportaron experiencia de más de 5 años, 15 (5,5%) reportaron experiencia entre 3 y 5 años, 31 (11,3%) entre 1 y 3 años, 39 (14,2%) menor a un año y 171 (62,2%) reportaron no tener experiencia.

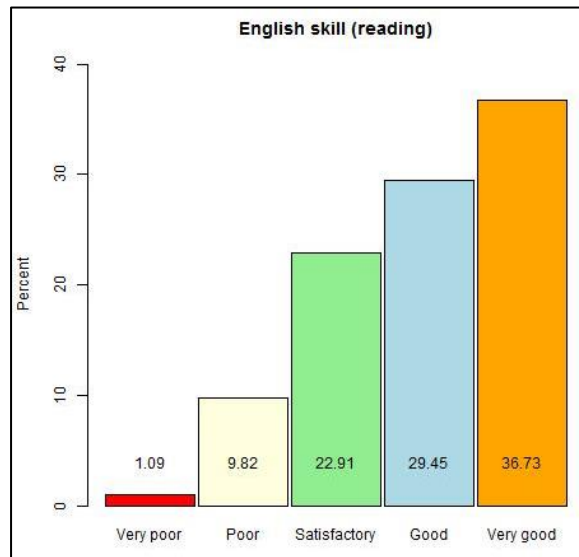
Figura 2-6 Participantes por experiencia como desarrollador profesional

Respecto a la experiencia y la autoevaluación de conocimientos del lenguaje de programación Java, más del 75% de los participantes reportaron experiencia entre 0 y 3 años y el resto de los participantes reportó experiencia entre 3 y 5 años; 8 (7.27%) de los participantes reportaron muy buenos conocimientos, 43 (30.18%) reportaron buenos conocimientos, 121 (44%) reportaron conocimientos satisfactorios, y el resto reportó bajos conocimientos en el lenguaje (ver Figura 2-7).

Figura 2-7 Participantes por experiencia y conocimientos en Java

Por último, con relación a las habilidades de lectura en inglés, el 89,09 % de los participantes reportaron nivel de satisfactorio a muy bueno (ver Figura 2-8).

Figura 2-8 Participantes por competencias en inglés



2.5 Procedimiento

La recolección de datos se realizó a través de cuestionarios en línea diseñados usando la plataforma Qualtrics. La primera sección de preguntas de los cuestionarios permitió recolectar información demográfica como: Experiencia en programación, conocimiento y experiencia en el lenguaje de programación Java y habilidades en inglés (*Reading skills*). La segunda sección de los cuestionarios se dividió en dos partes. En la parte A, se les solicitó a los participantes realizar ocho tareas de *análisis de métodos*, cuatro para la regla R1 y cuatro para la regla R2. En la parte B, los participantes debían llevar a cabo dos tareas de *Comparación de métodos*, una para la regla R1 y otra para la regla R2. Como sección de cierre (post cuestionario), los participantes debían calificar la importancia de considerar las reglas R1 y R2 para la escritura de código legible.

Para evitar el efecto *bandwagon* o efecto de arrastre³, a cada participante se le presentaron las cuatro tareas de análisis de métodos de cada regla en orden aleatorio y cada método correspondía a un problema diferente. Por ejemplo, en la Tabla 2-5 se presenta una de las combinaciones para la regla R1, en la cual el participante analizó la solución S4 para el problema P1, la solución S3 para el problema P2, la solución S2 para el problema P3 y la solución S1 para el problema P4. De igual forma llevó a cabo una tarea de comparación de métodos con las soluciones S1 y S2 para el problema P4. Las combinaciones utilizadas tanto para R1 como para R2 se detallan en el Anexo B.

Tabla 2-5 Muestra de una combinación de métodos y problemas para R1

Tarea de análisis de método				Tarea de comparación de método	
P1S4	P2S3	P3S2	P4S1	P4S1	P4S2

³ El efecto *bandwagon*, también conocido como el efecto de arrastre, "efecto de la moda", de "subirse al carro" o "efecto banda-carroza" y relacionado cercanamente al oportunismo, es la observación de que a menudo las personas hacen y creen ciertas cosas fundándose en el hecho de que muchas otras personas hacen y creen en esas mismas cosas. El efecto es peyorativamente llamado comportamiento gregario, particularmente cuando es aplicado a los adolescentes. Las personas tienden a seguir a la multitud sin examinar los méritos de una cosa en particular.

3. Análisis de resultados

El análisis de la información recolectada se realizó en dos grupos de trabajo independientes. Una vez ambos equipos terminaron, se encontró que los resultados obtenidos en cada uno son bastante similares. Por esta razón, en este documento se presentan los resultados obtenidos en el segundo grupo por considerarse más completos, concisos y claros.

3.1 Segundo análisis

Este análisis se concentró en verificar cada aspecto relacionado en las hipótesis definidas, en obtener el tamaño del efecto observado en cada uno de estos y en ampliar los resultados incluyendo consideraciones adicionales de validación. Se realizaron pruebas de t-student (*p-value*) y se calculó el tamaño de efecto mediante pruebas de Cohen, para el nivel de comprensión, los tiempos de comprensión, el nivel de confianza y de la clasificación de la legibilidad; todo esto con respecto a si el fragmento de código seguía/no seguía la regla y si el fragmento era lógicamente correcto/incorrecto. El *p-value* o valor de probabilidad, es la probabilidad de obtener un valor mayor o igual con respecto a un valor observado si la hipótesis nula es verdadera. Los valores de probabilidad muy pequeños se consideran estadísticamente significativos. Para esta investigación se consideró un nivel de significancia del 95%, es decir que en los casos donde el valor de probabilidad es menor o igual a 0.05 se rechazó la hipótesis nula. De otra parte, Cohen es el tamaño del efecto basado en la diferencia entre dos medias poblacionales, con la cual es posible interpretar el tamaño del efecto independientemente de la escala, donde el valor obtenido indica la magnitud del efecto y el signo (positivo, negativo) indica la naturaleza del efecto. Para este análisis, todas las pruebas t-student se realizaron en la misma dirección, es decir, todos los cálculos comparaban las medias de las respuestas de los fragmentos lógicamente correctos versus los fragmentos lógicamente incorrectos y los fragmentos que seguían la regla versus los fragmentos que no la seguían. Esto es importante dado que al cambiar el

orden de la comparación se afectaría el signo de la medida de Cohen, el cual es positivo para relaciones positivas y negativo para relaciones inversas.

3.1.1 Nivel de comprensión

El nivel de comprensión fue medido a través de dos preguntas diferentes: Una pregunta de opción múltiple asociada a la naturaleza del fragmento de código fuente (*pregunta de comprensión*) y una pregunta de confirmación acerca de si el fragmento es lógicamente correcto (*pregunta de confirmación lógica*). A partir de los resultados resumidos en la **Tabla 3-1**, se evidencia para la regla R1 un efecto pequeño (*Cohen's d* < 1) pero significativo (*p-value* < 0.05) en la precisión con la que los participantes respondieron correctamente la pregunta de comprensión. Adicionalmente, se evidencia que la correctitud de los fragmentos de código utilizados para verificar la regla R1, desempeñaron un papel más importante en la precisión con que los participantes respondieron correctamente la pregunta de comprensión que el seguimiento de la regla misma (*p-value*=0.04953, *Cohen's d*=0.12242 vs *p-value*<0.001, *Cohen's d*=0.39009). De igual forma, se evidencia que la correctitud de los fragmentos tuvo un efecto pequeño en la pregunta de comprensión (*p-value* < 0.0001, *Cohen's d* = 0.39009) versus un efecto medio en la pregunta de confirmación lógica (*p-value* < 0.0001, *Cohen's d*=0.69869), pero significativo en ambos casos (*p-value* < 0.05).

Tabla 3-1 Pruebas *t-student* y *Cohen's d* - nivel de comprensión, regla R1

	Pregunta de comprensión		Pregunta de confirmación lógica	
	Seguimiento regla R1	Correctitud fragmento	Seguimiento regla R1	Correctitud fragmento
<i>p-value</i>	0.04953	< 0.0001	0.25733	< 0.0001
<i>Cohen's d</i>	0.12242	0.39009	0.07056	0.69869

Para la regla R2, cuyos resultados se resumen en la Tabla 3-2, se evidencia nuevamente que la correctitud del código tuvo un papel importante en la precisión con que los participantes respondieron correctamente la pregunta de comprensión, aunque la diferencia no sea significativa.

Tabla 3-2 Pruebas *t-student* y *Cohen's d* - nivel de comprensión, regla R2

	Pregunta de comprensión		Pregunta de confirmación lógica	
	Seguimiento regla R2	Correctitud fragmento	Seguimiento regla R2	Correctitud fragmento
<i>p-value</i>	0.08911	0.08911	0.78837	<0.0001
<i>Cohen's d</i>	-0.10595	0.10595	-0.01672	0.55515

De lo anterior se puede resumir que la correctitud del código tuvo un efecto mucho mayor en el nivel de comprensión (precisión) que el seguimiento de cualquiera de las dos reglas; sin embargo, se resalta la evidencia del efecto positivo (mejora) en el nivel de comprensión de los desarrolladores al analizar código fuente que sigue la regla de legibilidad R1 (*minimize nesting*).

3.1.2 Tiempo de comprensión

El tiempo de comprensión fue medido a través del tiempo en segundos utilizado por los participantes en las actividades de análisis de métodos, para leer el enunciado del problema y analizar el fragmento de código solución propuesto para dicho problema (Tiempo de lectura). Se puede evidenciar en la **Tabla 3-3** que la regla R1 tuvo efecto pequeño pero significativo en el tiempo de lectura (*p-value* < 0.001, *Cohen's d* = -0.48068); de igual manera se evidencia que la correctitud tuvo un efecto muy pequeño, pero este no fue significativo (*p-value* = 0.11926, *Cohen's d* = -0.09707). Para la regla R2 tanto la correctitud como el seguimiento de la regla tuvieron un efecto pequeño, pero en ambos casos este no fue significativo.

De lo anterior podemos resumir que se evidencia un efecto negativo (disminución) en los tiempos de comprensión de los desarrolladores al analizar código fuente que sigue la regla de legibilidad R1 (*minimize nesting*).

Tabla 3-3 Pruebas *t-student* y *Cohen's d* - tiempo de comprensión

	R1		R2	
	Seguimiento regla R1	Correctitud fragmento	Seguimiento regla R2	Correctitud fragmento
<i>p-value</i>	< 0.0001	0.11926	0.73196	0.23294
<i>Cohen's d</i>	-0.48068	-0.09707	0.02133	0.0743

3.1.3 Nivel de confianza

El nivel de confianza fue medido a través de una pregunta, en cada una de las actividades de análisis de métodos, en la que cada participante calificó su propio nivel de confianza con respecto a la precisión de las respuestas dadas. De los resultados que se resumen en la **Tabla 3-4**, se evidencia nuevamente un efecto pequeño pero significativo para la regla R1 mientras que para la regla R2 dicho efecto también fue pequeño, pero no significativo. Con respecto a la correctitud de los fragmentos de código para la regla R1 se evidencia un efecto muy pequeño que no fue significativo, pero para la regla R2 el efecto fue pequeño y significativo. El tamaño del efecto negativo evidenciado para la correctitud para la regla R2 sugiere que los fragmentos de código correcto hicieron que los participantes tuvieran menor confianza en el nivel de comprensión.

En resumen, la evidencia indica que el seguimiento de la regla R1 hizo que los participantes tuvieran mayor confianza en su nivel de comprensión (efecto positivo). Y que la correctitud de los fragmentos de la regla R2 generaran menor confianza en el nivel de comprensión.

Tabla 3-4 Pruebas t-student y Cohen's d - nivel de confianza

	R1		R2	
	Seguimiento regla R1	Correctitud fragmento	Seguimiento regla R2	Correctitud fragmento
<i>p-value</i>	< 0.0001	0.18317	0.85384	0.02688
<i>Cohen's d</i>	0.45175	-0.08292	0.01147	-0.13799

3.1.4 Evaluación de la legibilidad

La legibilidad fue medida en las actividades de *análisis de métodos*, a través de la pregunta de calificación del nivel de legibilidad para cada fragmento de código analizado. Tal como se evidencia en la **Tabla 3-5**, el efecto de la calificación de legibilidad de los fragmentos de código que siguen la regla R1, fue mayor y significativo ($p\text{-value} < 0.0001$, $Cohen's d = 1.09733$). Esto se traduce en que los desarrolladores percibieron un alto grado de legibilidad en los fragmentos de código que seguían la regla R1. De otro lado, para la regla R2 se evidencia un efecto muy pequeño, pero no significativo; es decir que la percepción de legibilidad de los códigos utilizados para evaluar la regla R2, no generaron efecto alguno

en la percepción de los participantes. De igual forma se evidencia que para el caso de la correctitud de los fragmentos, el efecto fue inverso (negativo) pero no significativo respecto a la percepción de legibilidad.

Tabla 3-5 Pruebas *t-student* y *Cohen's d* – evaluación de legibilidad

	R1		R2	
	Seguimiento regla R1	Correctitud fragmento	Seguimiento regla R2	Correctitud fragmento
<i>p-value</i>	<0.0001	0.79348	0.23558	0.54257
<i>Cohen's d</i>	1.09733	-0.0163	0.07389	-0.03792

3.1.5 Calificación de legibilidad

La calificación cualitativa fue medida en la actividad de *comparación de métodos*, en la cual se presentó a los participantes dos fragmentos de código, uno al lado del otro. Uno de estos seguía la regla de legibilidad y el otro no; luego del análisis de los fragmentos se les solicitó establecer un orden indicando aquel que consideraran tenía mayor legibilidad. Algunos participantes no realizaron las actividades de comparación de métodos. Para el caso de la regla R1, de los 275 participantes solo 259 realizaron las actividades y para el caso de R2, 258 de 275 respondieron las preguntas. Sin embargo, para el estudio general no se excluyeron cuestionarios incompletos, dado que estos solamente omitieron la última parte. Los resultados evidencian que desde la perspectiva de los participantes ambas reglas son importantes para mejorar la legibilidad del código. Sin embargo, es superior el porcentaje de participantes que calificaron de mayor importancia la regla R1 (R1 = 86.82 %, R2 = 67.44%). Dado que los resultados del resto de los análisis no mostraron impacto significativo en las métricas para la regla R2, es posible que estos resultados estén más relacionados con lo que los participantes consideran que es código legible, o con sus preferencias personales. En lugar de indicar qué tan legible es un código fuente, posiblemente indican un sesgo en contra de los bucles *do-while* que generalmente se usan con menor frecuencia. Una alternativa posible es que las diferencias fueron más representativas cuando se analizaron los fragmentos de código que seguían la regla versus los que no uno al lado del otro.

3.1.6 Factores secundarios: Lengua materna, Conocimiento de lenguaje Java y Competencias en inglés (Reading)

Cuando se subdividió la población por el criterio de lengua materna se encontró que los participantes cuya lengua nativa es el inglés, tuvieron tiempos de lectura significativamente más cortos que los de otra lengua materna (31.06% menos que los participantes con lengua nativa español). En todos los demás análisis, el desempeño de los participantes con lengua nativa inglés y español fue muy similar, con la característica que los participantes de lengua nativa español tuvieron un rendimiento ligeramente mayor al de los participantes con lengua nativa inglés. Adicionalmente, se encontraron correlaciones pequeñas pero significativas entre los tiempos de respuesta y la correctitud de fragmentos de código para la regla R1 tanto para los participantes de lengua nativa español ($p=0.003346$, Cohen's $d= -0.2182$) como para los participantes de lengua nativa inglés ($p=0.002667$, Cohen's $d= -0.2198$). Se resalta este último hallazgo considerando que, en el conjunto completo de la población, esta correlación fue confusa por la diferencia entre los tiempos de respuesta de todos los participantes.

En el mismo sentido cuando se subdividió la población por el criterio de la autoevaluación en el nivel de competencias en inglés (Reading) se evidenciaron varios hallazgos interesantes. El grupo con las métricas de tiempo de lectura y respuesta más altos fueron los participantes que se autoevaluaron en la categoría “*Satisfactory*” (La opción media de la escala de 5 valores), mientras que los participantes que se autoevaluaron en las categorías inferiores o superiores tuvieron mejores tiempos de lectura y respuesta. El conocimiento del inglés tuvo un efecto interesante en los *p-value* y en los tamaños de efecto *Cohen's d*, para la *calificación de legibilidad*, y la *confianza de comprensión* con respecto a la regla R1. A medida que el conocimiento en inglés aumentaba, *el p-value* disminuía y el tamaño del efecto se incrementaba desde pequeñas e insignificantes correlaciones a muy grandes y significativas correlaciones para la calificación de legibilidad y a medias y significativas correlaciones para el nivel de confianza (ver **Tabla 3-6** y **Tabla 3-7**).

Tabla 3-6 Pruebas t-student y Cohen's d, calificación de legibilidad y autoevaluación de inglés

		Very por	Poor	Satisfactory	Good	Very Good
R1	<i>p-value</i>	0.837617	0.000116424	5.40957e-12	3.55568e-18	3.83824e-33
	<i>Cohen's d</i>	0.121447	0.861034	0.964534	1.055928	1.325242

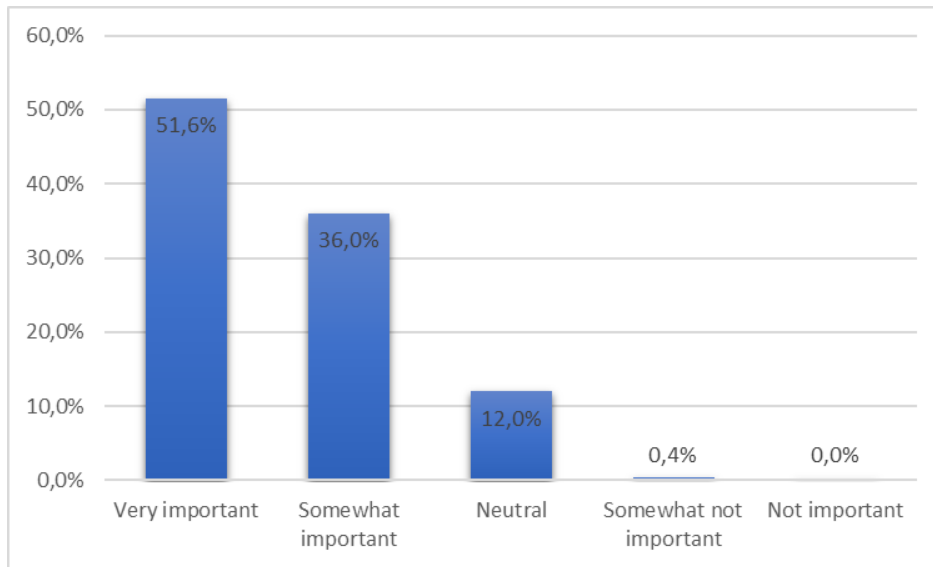
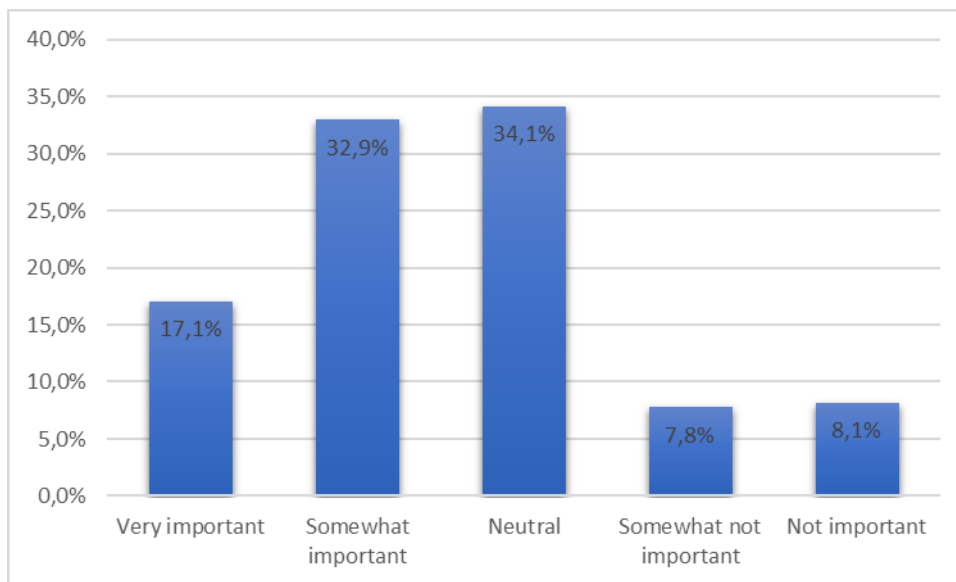
Tabla 3-7 Pruebas t-student y Cohen's d, nivel de confianza y autoevaluación de inglés

		Very por	Poor	Satisfactory	Good	Very Good
R1	<i>p-value</i>	0.447538	0.748316	0.0233969	7.33483e-05	2.68991e-11
	<i>Cohen's d</i>	-0.456435	0.068628	0.302309	0.458120	0.689388

Por último, cuando se dividió la población por el conocimiento en el lenguaje Java se evidenció que a medida que este conocimiento aumentaba, también aumentaba la significancia para la calificación de legibilidad, el nivel de comprensión y el nivel de confianza, mientras que disminuía la significancia para los tiempos de lectura y respuesta.

3.1.7 Post cuestionario

De acuerdo con el diseño del experimento, en esta parte se les solicitó a los participantes establecer un nivel de importancia para cada una de las reglas analizadas. Se utilizó una escala de Likert de 5 niveles (*Very important, Somewhat important, Neutral, Somewhat not important, and Not important*). Para la regla R1, tal como se observa en la **Figura 3-1**, el 87.6 % de los participantes considera que seguir la regla R1 para la escritura de código fuente es importante (*Very important* 133 participantes, 51.6% y *Somewhat important* 93 participantes, 36%). El resto de los participantes no considera importante el seguimiento de la regla R1 en el código fuente (*Neutral* 31 participantes, 12% y *Somewhat not important* un participante, 0.4%). Para la regla R2, tal como se observa en la **Figura 3-2**, solo el 50% de los participantes considera importante el seguimiento de la regla R2 en el código fuente (*Very important* 44 participantes, 17.1% y *Somewhat important* 85 participantes, 32.9%); para el otro 50% no es importante el seguimiento de la regla R2 (*Neutral* 88 participantes, 34.1%, *Somewhat not important* 20 participantes, 7.8% y *Not important* 21 participantes, 8.1%).

Figura 3-1 Ranking de importancia regla R1**Figura 3-2** Ranking de importancia regla R2

3.2 Amenazas a la validez

En esta sección se analizan las amenazas a la validez de los resultados y cómo estas fueron mitigadas en el diseño del estudio.

3.2.1 Validez interna

Las amenazas de validez interna son los factores exógenos que pueden influir en los resultados observados. En primer lugar, para mitigar los factores humanos se recopilaron datos demográficos de los participantes para asegurar que estos tuvieran un nivel de competencias aceptable en lenguaje de programación, habilidades medias de lectura en inglés y que no padecieran de trastornos de lectura. Adicionalmente, se indicó expresamente a los participantes las siguientes recomendaciones: (a) No usar herramientas adicionales o copiar y pegar el código para responder las preguntas asociadas, (b) No asumir la encuesta como una evaluación de sus habilidades propias de programación, (c) No preocuparse por el tiempo necesario para completar las tareas, (d) Realizar la encuesta en una sola sesión, es decir, sin pautas o interrupciones, (e) Centrarse en comprender cada método, ya que éste sólo se presentaría una única vez por cada actividad previo a la sección de preguntas. A pesar de esto, es posible que algunos de los participantes que realizaron el experimento sin supervisión (alrededor del 47% del total) hayan utilizado herramientas adicionales para entender el código fuente, o para hacer pausas durante la sesión experimental. En segundo lugar, para evitar el efecto de artefactos mal diseñados, previo al experimento se realizó un estudio piloto con dos estudiantes graduados vinculados a empresas de software. Los resultados de esta prueba piloto permitieron detectar descripciones de problemas poco claras, deficiencias en los formularios utilizados para responder las preguntas, y fragmentos de código o preguntas asociadas a estos demasiado complejas. Adicionalmente, se evaluó la viabilidad general y el tiempo requerido para completar cada una de las tareas del experimento. Finalmente, con el objetivo de controlar los efectos de orden, se organizó cada versión del experimento, 24 versiones en total, de manera que, para una regla de legibilidad en particular, el participante analizara cuatro problemas diferentes y cada una de las cuatro soluciones propuestas para estos problemas estuviera en un tratamiento diferente. Aparte de esto, las cuatro soluciones se presentaron en orden aleatorio.

3.2.2 Validez externa

Las amenazas de validez externa disminuyen el grado de generalización de los resultados. En primer lugar, considerando que el alcance de la investigación se restringió a validar el impacto de las reglas relacionadas con simplificación de bucles y simplificación lógica, los resultados podrían ser válidos sólo para lenguajes con construcciones condicionales y

bucles similares. Sin embargo, si los fragmentos de código utilizados son más complejos y/o dependen de otros métodos, los resultados podrían ser diferentes. En segundo lugar, en consideración a que solo el 22% de los participantes reportó experiencia en la industria de software y la mayoría eran estudiantes de posgrado y pregrado, es posible que las conclusiones no puedan generalizarse completamente a la población de desarrolladores profesionales. Por lo tanto, las réplicas que incluyan población de la industria u otros tipos de código fuente, son necesarios y bienvenidos.

3.2.1 Validez de constructo

La validez de constructo se ocupa de la brecha, si la hay, entre los conceptos teóricos y la representación y medición real de mismos dentro del experimento. En esta investigación no existe un tratamiento matemático de los conceptos de correctitud de programas y especificación formal de los problemas. Sin embargo, las descripciones en lenguaje natural de los problemas de programación utilizados se revisaron cuidadosamente para eliminar cualquier ambigüedad o falta de claridad que pudiera generar confusión o malentendidos. En este sentido, la prueba piloto confirmó que las entradas y salidas esperadas para cada problema de programación son claros.

La percepción de legibilidad de un código se ve afectada por muchos aspectos sintácticos y semánticos, como el tipo de estructura sintáctica utilizada, los nombres seleccionados, las dependencias, las abstracciones representadas, etc. En la presentación del código también pueden influir aspectos como el tamaño de la fuente, la longitud de las líneas, la altura de las líneas y los colores. Para mitigar la influencia de estos factores y aislar el efecto de las dos reglas de legibilidad, se seleccionaron problemas muy simples, se escribieron métodos cortos sin dependencias y comentarios y no se usaron colores. Además, en las cuatro soluciones asociadas a cada problema, se unificaron todos los aspectos sintácticos y de presentación de código, de tal forma que solo difieren en el seguimiento de la regla de legibilidad y en el atributo de corrección lógica.

Respecto a la precisión de las métricas de tiempo, las características del diseño de la encuesta en Qualtrics permitieron medir exactamente el tiempo en segundos que dedicó cada participante en leer el enunciado del problema y la solución propuesta. Adicional a esto, a los participantes no se les permitió volver a revisar el código toda vez que indicaran que habían terminado de entender la solución en cuestión.

Evaluar el nivel de comprensión de una persona es un asunto muy complicado, incluso cuando se trata de pequeños fragmentos de código fuente. En este sentido se decidió hacer una única pregunta de opción múltiple, en la que participante debía indicar cuál de las 4 afirmaciones dadas es verdadera, con respecto a la ejecución del método. Es posible que algunos de los participantes hayan respondido la pregunta correctamente sin haber entendido el código fuente. Por supuesto habría sido deseable hacer más preguntas, pero debe haber un equilibrio entre reducir el riesgo de medir incorrectamente y aumentar la complejidad del cuestionario y su calificación.

Por último, incluso con métodos cortos y simples, pueden existir varios tipos de errores lógicos (por ejemplo, un operador lógico incorrecto, un índice fuera de límites, una expresión booleana incorrecta, etc.). En el diseño de los métodos incorrectos no se utilizó una estrategia sistemática para construirlos ya que no es posible saber cómo un desarrollador escribirá una versión lógicamente incorrecta. Por lo tanto, si los errores en los métodos lógicamente incorrectos son de otros tipos, los resultados pueden ser diferentes.

3.2.2 Validez de conclusión

La validez de la conclusión aborda la selección adecuada de las pruebas estadísticas para determinar la significancia estadística de los resultados de la investigación. En todos los análisis de esta investigación se utilizaron métodos estadísticos estándar, es decir, t-test y Cohen's d, que son herramientas convencionales en estadística inferencial.

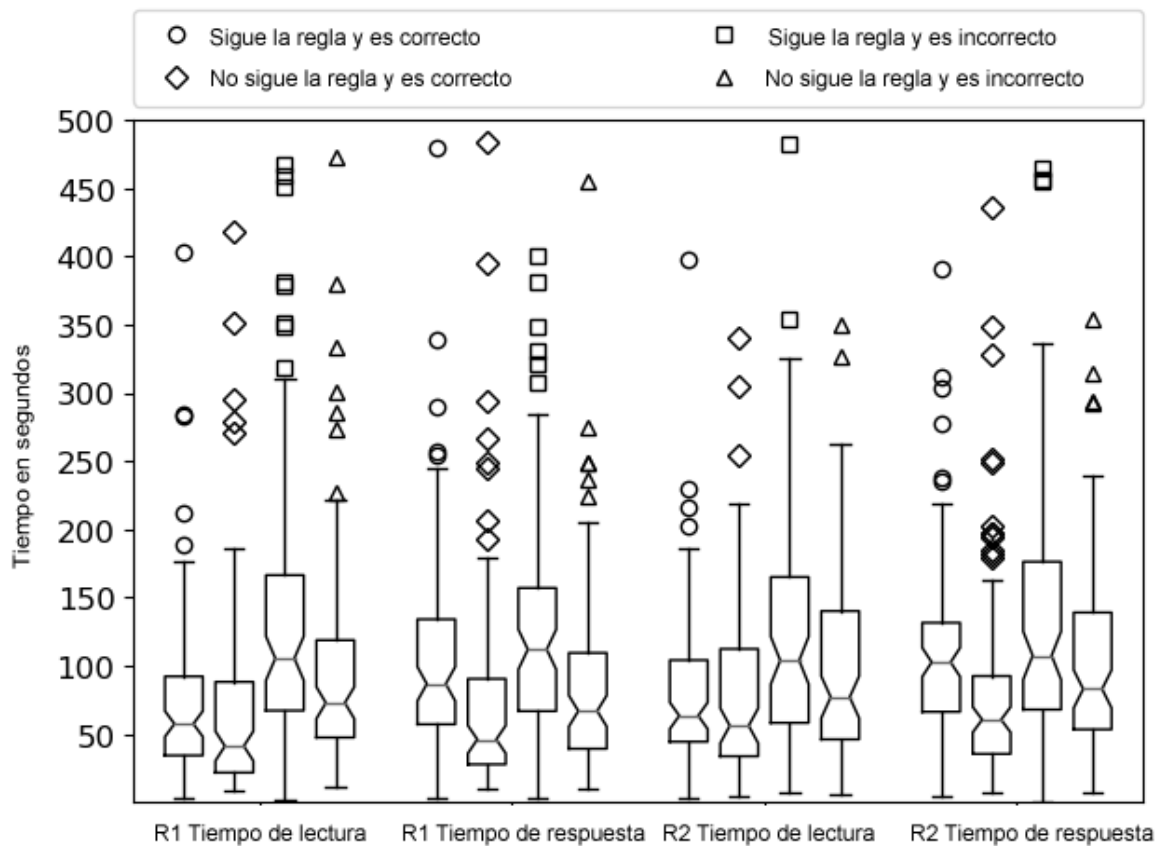
3.3 Observaciones y Discusión

Los principales hallazgos de esta investigación se refieren a que seguir la regla R1 (*minimize nesting*) parece tener un impacto significativo en la legibilidad del código fuente. Este impacto queda evidenciado en la calificación de legibilidad, en el tiempo de lectura y comprensión de código. Por el contrario, la regla R2 (*avoid do-while loops*) no evidenció tener impacto significativo en ninguna de las métricas obtenidas y analizadas.

Para el caso de las calificaciones de legibilidad, se pudo evidenciar claramente que seguir la regla R1 tuvo el mayor impacto de todas las variables analizadas (Cohen's $d > 1$),

mientras que para la regla R2 no se evidenció efecto significativo en las calificaciones de legibilidad. Respecto a los tiempos lectura y comprensión de código (ver **Figura 3-3**), se evidenció que los tiempos promedio de lectura de los fragmentos de la regla R2 fueron mayores a los de la regla R1, esto probablemente se debió a que los fragmentos de código utilizados para R2 son aproximadamente 17% más extensos que los fragmentos de la regla R1. En este mismo sentido, es probable que el efecto en los tiempos de lectura de los fragmentos que siguen la regla R1, sea por que dichos fragmentos son en promedio 6 líneas más cortos que los que no siguen la regla R2. En relación con la correctitud, esta tuvo un efecto significativo en los tiempos de respuesta de ambas reglas, esto es, que los participantes tardaron más tiempo en responder las preguntas de los fragmentos de código lógicamente incorrecto versus los fragmentos de código correctos.

Figura 3-3 Consolidado de los tiempos de lectura y respuesta



Con respecto al nivel de confianza en la comprensión, métrica que permitió a los participantes auto-evaluar su grado de comprensión, se evidenció para la regla R1 el mayor efecto observado de todas las variables medidas, lo que sugiere que el uso de la regla R1

además de hacer que el código sea legible y más rápido de leer y genera mayor confianza en los desarrolladores acerca su propio grado de comprensión.

Con respecto al nivel de comprensión, se evidenció que los fragmentos de código lógicamente correctos tuvieron mejor precisión para la *pregunta de confirmación lógica*, lo que sugiere que los participantes fueron más propensos a asumir erróneamente que el código lógicamente incorrecto era correcto, este efecto no se evidenció para el seguimiento de las reglas de legibilidad. También se encontró que el seguimiento de la regla R1 incrementó la precisión en las respuestas a la *pregunta de comprensión* lo que sugiere que el uso de la regla R1 tiene un impacto real en que tan bien los desarrolladores entienden el código fuente. Para el caso de la regla R2, no se evidenció ningún efecto ni en la *pregunta de comprensión* ni en la *pregunta de confirmación lógica*.

En resumen, a partir de estos resultados y en consideración a las preguntas de investigación formuladas en la sección 2.2, se puede evidenciar que la regla R1 (minimize nesting) disminuye el tiempo invertido por los desarrolladores en la lectura y comprensión de código fuente (RQ1a), rechazando en este caso la hipótesis nula ($H1_0$) y aceptando la hipótesis alternativa (H1), de igual forma se evidencia que la regla R1 incrementa el nivel de confianza de los desarrolladores respecto a su grado de comprensión (RQ1b), rechazando en este caso nuevamente la hipótesis nula ($H2_0$) y aceptando la hipótesis alternativa (H2) y por último que la regla R1 mejora el nivel de comprensión de los desarrolladores al analizar código fuente (RQ1c), rechazando una vez la hipótesis nula propuesta ($H3_0$) y aceptando la hipótesis alternativa (H3). De otro, la regla R2 (avoid Do/While loops) no disminuye el tiempo invertido por los desarrolladores en la lectura y comprensión de código fuente (RQ2a), aceptando la hipótesis nula ($H4_0$), la regla R2 no incrementa el nivel de confianza de los desarrolladores respecto a su grado de comprensión (RQ2b), aceptando nuevamente la hipótesis nula ($H5_0$) y que la regla R2 no mejora el nivel de comprensión de los desarrolladores al analizar código, aceptando una vez más la hipótesis nula propuesta ($H6_0$).

4. Conclusiones

En esta investigación se analizó el impacto de dos prácticas de codificación (R1: *Minimize Nesting* y R2: *Avoid Do/While loops*) en la legibilidad y comprensión (entendimiento) del código fuente. En la primera parte del experimento, cada uno de los 275 participantes realizaron ocho actividades de análisis de métodos, cuatro para R1 y cuatro para R2. En la segunda parte, cada participante llevo a cabo dos actividades de comparación de métodos, uno asociado a R1 y otro asociado a R2. La parte final del experimento solicitó a los participantes calificar la importancia de ambas prácticas de codificación. Para seleccionar los métodos asociados a una regla, se eligieron cuatro problemas sencillos de programación y para cada uno de estos se diseñaron cuatro soluciones diferentes. Dentro de las cuatro soluciones, dos son correctas y dos son incorrectas y dos siguen la regla y dos no siguen la regla. Así, se utilizaron cuatro problemas y dieciséis métodos para analizar el impacto de cada regla.

Respecto a la regla R1, se evidenció que en efecto esta práctica tiene potencial para disminuir el tiempo requerido por los desarrolladores para leer y comprender código fuente e incrementar el nivel de confianza acerca de su grado de comprensión. Respecto a la regla R2, los resultados encontrados no son significativos para ninguna de las métricas analizadas.

Para ambas prácticas, la percepción de legibilidad indica que vale la pena seguir las reglas. En el caso de R1 el 86.75% de los participantes consideraron que, para el mismo problema, el método que sigue la regla es más legible que el método que no sigue la regla. Para el caso de la regla R2 el 67.4% prefirieron el método que no usaba la estructura *do-while*. Por último, el 51.6% de los participantes consideró que la práctica de *minimize nesting* es muy importante para escribir código más legible, mientras que el 36% lo consideró como

algo, en alguna medida, importante. Y para la regla R2, la mayoría de los participantes fueron neutrales respecto a la importancia de esta para escribir código más legible.

A nivel general los resultados obtenidos muestran la importancia de la regla R1 como una práctica que debe ser promovida y enseñada a los estudiantes que están aprendiendo a programar, dado que el uso de esta reduce el tiempo de comprensión del código fuente, incrementa tanto el nivel de confianza respecto al grado de comprensión del código como el grado de comprensión mismo. De otra parte, estos resultados pueden proveer evidencia base de soporte a la construcción de guías de estilo de codificación para la industria que permita mejorar la legibilidad y comprensión y facilite la realización de las actividades de mantenimiento de software. Por último, estos resultados abren la puerta a futuras investigaciones con respecto al uso de recomendaciones adicionales a las analizadas en un contexto más amplio y profundo asociado a legibilidad y comprensión de software.

A. Anexo: Componentes de los cuestionarios

En este anexo se presentan todos los componentes (secciones) utilizados para la construcción de los cuestionarios en la plataforma Qualtrics.

A.1. Instrucciones generales

This study is concerned with software readability. **Software readability is a property that influences how easily a given piece of code can be read and understood.**

Your task is to carefully read some code snippets and then you will be asked to answer questions about them.

We will record the time you need to read the code snippets and answer the questions.

Your answers and timing will only be used to study the readability of code and not to assess your performance.

Please note that you cannot go back to questionnaire parts that you have already finished.

Please do not take notes or copy the code snippets (manually or electronically), otherwise your answers would be useless for the study.

In total, you will see eight java methods in two parts. The whole exercise will take about 20 minutes.

A.2. Cuestionario inicial

- 1) Participant ID
- 2) What is your mother tongue (native language)?

- 3) In what country do you live?
- 4) Do you have some form of reading disorder, like for example dyslexia?
- a) Yes
 - b) No
- 5) Are you currently working as a developer in a software company?
- a) Yes
 - b) No
- 6) Are you currently an undergraduate student?
- a) Yes
 - b) No
- 7) Are you currently a graduate student?
- a) Yes
 - b) No
- 8) Please rate your knowledge of the following subjects:
- a) English (in reading)
 - Very poor
 - Poor
 - Satisfactory
 - Good
 - Very good
 - b) Java programming
 - Very poor
 - Poor
 - Satisfactory
 - Good

- Very good

9) Please indicate the number of years you have been programming in:

a) Any language

- Less than 1 year
- Between 1 and 3 years
- Between 3 and 5 years
- More than 5 years

b) Java

- Less than 1 year
- Between 1 and 3 years
- Between 3 and 5 years
- More than 5 years

10) How many years have you been working as a professional programmer?

- a) Not applicable
- b) Less than 1 year
- c) Between 1 and 3 years
- d) Between 3 and 5 years
- e) More than 5 years

A.3. Tareas de análisis de métodos

En esta sección se presentan las instrucciones, los problemas para cada regla, los fragmentos de Java y las preguntas utilizadas para las tareas de análisis de métodos.

1) Instrucciones

Part A

The next eight code snippets are proposed solutions to eight different problems.

Please read the code and try to understand each solution. There is no time limit, so do not worry and be focused on understanding the code.

You will be asked very specific questions about the output of the code snippets as well as how it behaves given certain inputs. You will also be asked if the code snippet is logically correct or not.

Press “Next” when you are done understanding the solution at hand. Remember you will not be allowed to go back to the source code.

2) Problema 1, Regla R1

Given three integer numbers, the method must return the greatest.

Solución 1

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = numOne;
    if (numTwo > theLargest) {
        theLargest = numTwo;
    }
    if (numThree > theLargest) {
        theLargest = numThree;
    }
    return theLargest;
}
```

Solución 2

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest;
    if (numOne < numTwo) {
        if (numTwo < numThree) {
            theLargest = numThree;
        } else {
            theLargest = numTwo;
        }
    } else {
        if (numOne < numThree) {
            theLargest = numThree;
        } else {
            theLargest = numOne;
        }
    }
    return theLargest;
}
```

Solución 3

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = 0;
    if ((numOne > numTwo) && (numOne > numThree)) {
        theLargest = numOne;
    }
    if ((numTwo > numOne) && (numTwo > numThree)) {
        theLargest = numTwo;
    }
    if ((numThree > numTwo) && (numThree > numOne)) {
        theLargest = numThree;
    }
    return theLargest;
}
```

Solución 4

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = 0;
    if (numTwo < numThree) {
        theLargest = numThree;
    } else {
        if (numTwo > numOne) {
            theLargest = numTwo;
        }
    }
    if ((numOne > numTwo) && (numOne > numThree)) {
        theLargest = numOne;
    }
    return theLargest;
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
 - Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?

- The method does not work properly when the input numbers are 1, 1, and 2
 - The method does not work properly when the input numbers are 2, 2, and 2
 - The method does not work properly when the input numbers are 1, 2, and 3
 - The method works properly when the input numbers are all zero.
- c) How confident are you in your own level of comprehension of the previous method?
- Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
- Yes
 - No

3) Problema 2, Regla R1

Given a mark which is an integer between 0 and 100, the method must return a letter. Letter A if mark ≥ 90 ; B if mark $\in [80, 90)$; C if mark $\in [70, 80)$; D if mark $\in [60, 70)$; and F if mark < 60 .

Solución 1

```
public static char findGrade(int marks) {  
    if (marks >= 90) {  
        return 'A';  
    }  
    if (marks >= 80) {  
        return 'B';  
    }  
    if (marks >= 70) {  
        return 'C';  
    }  
    if (marks >= 60) {  
        return 'D';  
    }  
    return 'F';  
}
```

Solución 2

```
public static char findGrade(int marks) {
    char grade;
    if (marks >= 90) {
        grade = 'A';
    } else {
        if (marks >= 80) {
            grade = 'B';
        } else {
            if (marks >= 70) {
                grade = 'C';
            } else {
                if (marks >= 60) {
                    grade = 'D';
                } else {
                    grade = 'F';
                }
            }
        }
    }
    return grade;
}
```

Solución 3

```
public static char findGrade(int marks) {
    if (marks >= 90) {
        return 'A';
    }
    if (marks < 90 && marks > 80) {
        return 'B';
    }
    if (marks < 80 && marks > 70) {
        return 'C';
    }
    if (marks < 70 && marks >= 60) {
        return 'D';
    }
    return 'F';
}
```

Solución 4

```
public static char findGrade(int marks) {
    char grade;
    if (marks >= 90) {
        grade = 'A';
    } else if (marks <= 89 && marks > 80) {
        grade = 'B';
    } else if (marks <= 79 && marks > 70) {
        grade = 'C';
    } else if (marks <= 69 && marks > 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    return grade;
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
- Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
- The method does not work properly when the input mark is 80
 - The method works properly when the input mark is 70
 - The method does not work properly when the input mark is 50
 - The method does not work properly when the input mark is 0
- c) How confident are you in your own level of comprehension of the previous method?
- Very confident
 - Somewhat confident
 - Neutral

- Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
- Yes
 - No

4) Problema 3, Regla R1

Given the body mass index (bmi), the method must return the category in which the index is located. The category is “very severely underweight” if $bmi \leq 15$; “severely underweight” if $bmi \in [15; 16)$; “underweight” if $bmi \in [16; 18.5)$; “healthy weight” if $bmi \in [18.5; 25)$; “overweight” if $bmi \geq 25$.

Solución 1

```
public static String bodyMassIndexCategory(double bmi) {  
  
    if (bmi < 15.00) {  
        return "Very severely underweight";  
    }  
    if (bmi < 16.00) {  
        return "Severely underweight";  
    }  
    if (bmi < 18.5) {  
        return "Underweight";  
    }  
    if (bmi < 25.00) {  
        return "Normal (healthy weight)";  
    }  
    return "Overweight";  
}
```

Solución 2

```
public static String bodyMassIndexCategory(double bmi) {
    String result;
    if (bmi < 15.00) {
        result = "Very severely underweight";
    } else {
        if (bmi < 16.00) {
            result = "Severely underweight";
        } else {
            if (bmi < 18.5) {
                result = "Underweight";
            } else {
                if (bmi < 25.00) {
                    result = "Normal (healthy weight)";
                } else {
                    result = "Overweight";
                }
            }
        }
    }
    return result;
}
```

Solución 3

```
public static String bodyMassIndexCategory(double bmi) {

    if (bmi < 15.00) {
        return "Very severely underweight";
    }
    if (bmi >= 15.00 && bmi < 16.00) {
        return "Severely underweight";
    }
    if (bmi >= 16.00 && bmi < 18.5) {
        return "Underweight";
    }
    if (bmi > 18.50 && bmi < 25.00) {
        return "Normal (healthy weight)";
    }
    return "Overweight";
}
```

Solución 4

```
public static String bodyMassIndexCategory(double bmi) {
    String result;
    if (bmi < 15.00) {
        result = "Very severely underweight";
    } else {
        if (bmi >= 15.00 && bmi < 16.00) {
            result = "Severely underweight";
        } else {
            if (bmi >= 16.00 && bmi < 18.5) {
                result = "Underweight";
            } else {
                if (bmi > 18.50 && bmi < 25.00) {
                    result = "Normal (healthy weight)";
                } else {
                    result = "Overweight";
                }
            }
        }
    }
    return result;
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
- Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
- The method does not work properly when the bmi is 18.5
 - The method does not work properly when the bmi is 10
 - The method does not work properly when the bmi is 40
 - None of the previous statements is true
- c) How confident are you in your own level of comprehension of the previous method?

- Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
- Yes
 - No

5) Problema 4, Regla R1

Given three integers, the method must count how many of them are positive numbers.

Solución 1

```
public int countPosNumbers(int numOne, int numTwo, int numThree) {  
    int count = 0;  
    if (numOne > 0) {  
        count++;  
    }  
    if (numTwo > 0) {  
        count++;  
    }  
    if (numThree > 0) {  
        count++;  
    }  
    return count;  
}
```

Solución 2

```
public int countPosNumber(int numOne, int numTwo, int numThree) {
    if (numOne > 0) {
        if (numTwo > 0) {
            if (numThree > 0) {
                return 3;
            } else {
                return 2;
            }
        } else if (numThree > 0) {
            return 2;
        } else {
            return 1;
        }
    } else if (numTwo > 0) {
        if (numThree > 0) {
            return 2;
        } else {
            return 1;
        }
    } else if (numThree > 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Solución 3

```
public int countPosNumbers(int numOne, int numTwo, int numThree) {
    int count = 0;
    if (numOne >= 0) {
        count++;
    }
    if (numTwo >= 0) {
        count++;
    }
    if (numThree >= 0) {
        count++;
    }
    return count;
}
```

Solución 4

```
public int countPosNumbers(int numOne, int numTwo, int numThree) {
    if (numOne > 0) {
        if (numTwo > 0) {
            if (numThree > 0) {
                return 3;
            } else {
                return 2;
            }
        } else if (numThree > 0) {
            return 2;
        } else {
            return 1;
        }
    } else if (numTwo > 0) {
        if (numThree > 0) {
            return 1;
        } else {
            return 2;
        }
    } else if (numThree > 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
- Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
- The method does not work properly when the input numbers are 1, 1, and 1
 - The method does not work properly when the input numbers are 1, 0, and 2
 - The method does not work properly when the input numbers are 0, 2, and 3

- None of the previous statements is true
- c) How confident are you in your own level of comprehension of the previous method?
 - Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
 - Yes
 - No

6) Problema 1, Regla R2

Ask the user to answer a multiple-choice question. Show the question, get the user response, and end when the user chooses the correct answer or when she decides not to try more (typing 'q' or 'e')

Solución 1

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain = "y";
    while (tryAgain.compareTo("y") == 0) {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulations!");
            break;
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    sc.close();
}
```

Solución 2

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain;
    do {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulations!");
            break;
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    while (tryAgain.compareTo("y") == 0);
    sc.close();
}
```

Solución 3

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain = "y";
    while (tryAgain.compareTo("y") == 0) {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulation!");
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    sc.close();
}
```

Solución 4

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain;
    do {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulation!");
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    while (tryAgain.compareTo("y") == 0);
    sc.close();
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
 - Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
 - The method does not end when the user types "q" or "e"

- The method does not end when the user chooses the correct answer to the question
 - The method ends when the user chooses an incorrect answer to the question
 - The method ends when the user chooses the option c.
- c) How confident are you in your own level of comprehension of the previous method?
- Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
- Yes
 - No

7) Problema2, Regla R2

Let's assume that we want to force a user to change her password. The user must give a new password that must have 4 different characters. Besides, the given password must be different to the old one. The method receives the old password as a parameter and asks the user to give the new one.

Solución 1

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass = oldPass;
    boolean valid = false;
    while (newPass.equals(oldPass) || !valid) {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() != 4) {
            valid = false;
        } else {
            for (int i = 0; i < newPass.length(); i++) {
                for (int j = i + 1; j < newPass.length(); j++) {
                    if (newPass.charAt(j) == newPass.charAt(i)) {
                        valid = false;
                    }
                }
            }
        }
    }
    sc.close();
    return newPass;
}
```

Solución 2

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass;
    boolean valid;
    do {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() != 4) {
            valid = false;
        } else {
            for (int i = 0; i < newPass.length(); i++) {
                for (int j = i + 1; j < newPass.length(); j++) {
                    if (newPass.charAt(j) == newPass.charAt(i)) {
                        valid = false;
                    }
                }
            }
        }
    }
    while (newPass.equals(oldPass) || !valid);
    sc.close();
    return newPass;
}
```

Solución 3

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass = oldPass;
    boolean valid = false;
    while (newPass.equals(oldPass) || !valid) {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() < 4) {
            valid = false;
        }
        for (int i = 0; i < newPass.length(); i++) {
            for (int j = i + 1; j < newPass.length(); j++) {
                if (newPass.charAt(j) == newPass.charAt(i)) {
                    valid = false;
                }
            }
        }
    }
    sc.close();
    return newPass;
}
```

Solución 4

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass;
    boolean valid;
    do {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() < 4) {
            valid = false;
        }
        for (int i = 0; i < newPass.length(); i++) {
            for (int j = i + 1; j < newPass.length(); j++) {
                if (newPass.charAt(j) == newPass.charAt(i)) {
                    valid = false;
                }
            }
        }
    }
    while (newPass.equals(oldPass) || !valid);
    sc.close();
    return newPass;
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
 - Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
 - The method accepts only new passwords with exactly 4 characters and no duplicates

- The method accepts only new passwords with exactly 4 characters and with or without duplicates
 - The method accepts only new passwords with at least 4 characters and no duplicates
 - The method accepts only new passwords with at least 4 characters and with or without duplicates
- c) How confident are you in your own level of comprehension of the previous method?
- Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
- Yes
 - No

8) Problema 3, Regla R2

Add the positive numbers in an array. The method receives an integer array as a parameter and must return the sum of the positive numbers in the array.

Solución 1

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    while (counter < limit) {
        if (nums[counter] > 0) {
            sum += nums[counter];
        }
        counter++;
    }
    return sum;
}
```

Solución 2

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    if (limit == 0) {
        return 0;
    }
    do {
        if (nums[counter] > 0) {
            sum += nums[counter];
        }
        counter++;
    }
    while (counter < limit);
    return sum;
}
```

Solución 3

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    while (counter < limit) {
        if (nums[counter] > 0) {
            sum += nums[counter];
            counter++;
        }
    }
    return sum;
}
```

Solución 4

```
public int sumPositiveNums (int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    do {
        if (nums[counter] > 0) {
            sum += nums[counter];
            counter++;
        }
    }
    while (counter < limit);
    return sum;
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
 - Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
 - The method throws an exception when the array is empty
 - The method never ends when the array has one or more positive numbers
 - The method never ends when the array has one or more zeros
 - None of the previous statements is true
- c) How confident are you in your own level of comprehension of the previous method?
 - Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?

- Yes
- No

9) Problema 4, Regla R2

Count the occurrences of a character. This method receives a string and a character as parameters. It must count and return the number of occurrences of the character in the string.

Solución 1

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    int counter = 0;
    int index = 0;
    while (index < max) {
        if (phrase.charAt(index) != letter) {
            index++;
            continue;
        }
        index++;
        counter++;
    }
    return counter;
}
```

Solución 2

```
public int countLetter(String phrase, char lett
    int max = phrase.length();
    if (max == 0) {
        return 0;
    }
    int counter = 0;
    int index = 0;
    do {
        if (phrase.charAt(index) != letter) {
            index++;
            continue;
        }
        index++;
        counter++;
    }
    while (index < max);
    return counter;
}
```

Solución 3

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    int counter = 0;
    int index = 0;
    while (index < max) {
        if (phrase.charAt(index) != letter) {
            continue;
        }
        index++;
        counter++;
    }
    return counter;
}
```

Solución 4

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    if (max == 0) {
        return 0;
    }
    int counter = 0;
    int index = 0;
    do {
        if (phrase.charAt(index) != letter) {
            continue;
        }
        index++;
        counter++;
    }
    while (index < max);
    return counter;
}
```

Preguntas

- a) Based on your programming experience, how would you rate the readability of the previous piece of code?
- Very difficult to read
 - Difficult to read
 - Neutral
 - Easy to read
 - Very easy to read
- b) Which of the following statements is true about the code you just read?
- The method never reviews the last character of the phrase
 - The method does work properly when the phrase is empty
 - The method works properly when the letter is 'x' and the phrase is awerx
 - The method works properly only when all the characters of the phrase are equals to the letter
- c) How confident are you in your own level of comprehension of the previous method?

- Very confident
 - Somewhat confident
 - Neutral
 - Not very confident
 - Not at all confident
- d) Is the method you just saw logically correct?
- Yes
 - No

A.4. Tareas de comparación de métodos

En esta sección se presentan las instrucciones y preguntas utilizadas en las tareas de comparación de métodos. Para esta sección tal como se especifica en el diseño del experimento se utilizaron las versiones correctas y que siguen cada una de las reglas analizadas.

1) Instrucciones

Part B

In this part, you will study two for each problem. We will ask you to rank them in order of readability from best to worst.

Please rank the next two code snippets in terms of readability from best to worst. Rank 1 indicates better readability than Rank 2. Select the appropriate rank for each code snippet using the radio buttons to the right.

2) Preguntas

Please explain your ranking

A.5. Cuestionario final

- 1) Rank the importance of minimizing nesting in source code.
 - Very important
 - Somewhat important

- Neutral
- Somewhat not important
- Not important

2) Rank the importance of avoiding do-while statements in source code.

- Very important
- Somewhat important
- Neutral
- Somewhat not important
- Not important

3) Please explain any difficulties you encountered during the session.

4) What do you consider to be the most important aspects for the readability and comprehensibility of code?

5) What is your personal favorite technique, tool, or approach to help you read and comprehend code?

6) Please feel free to leave comments about the code and questions included in this exercise

B. Anexo: Distribución de problemas y soluciones

A continuación, se presenta la tabla de distribución de problemas y soluciones para cada versión de la encuesta. En cada combinación P hace referencia al problema con su respectiva numeración de 1 a 4 y S hace referencia a la solución con su respectiva numeración de 1 a 4.

Versión de la encuesta	Regla R1						Regla R2					
	Análisis de métodos				Comparación de métodos		Análisis de métodos				Comparación de métodos	
Sb1	P1S1	P2S2	P3S3	P4S4	P1S1	P1S2	P1S3	P2S4	P3S1	P4S2	P1S1	P1S2
Sb2	P1S1	P2S2	P3S4	P4S3	P2S1	P2S2	P1S3	P2S4	P3S2	P4S1	P2S1	P2S2
Sb3	P1S1	P2S3	P3S2	P4S4	P3S1	P3S2	P1S4	P2S1	P3S2	P4S3	P3S1	P3S2
Sb4	P1S1	P2S3	P3S4	P4S2	P4S1	P4S2	P1S4	P2S1	P3S3	P4S2	P4S1	P4S2
Sb5	P1S1	P2S4	P3S2	P4S3	P1S1	P1S2	P1S3	P2S4	P3S1	P4S2	P1S1	P1S2
Sb6	P1S1	P2S4	P3S3	P4S2	P2S1	P2S2	P1S3	P2S4	P3S2	P4S1	P2S1	P2S2
Sb7	P1S2	P2S1	P3S3	P4S4	P3S1	P3S2	P1S4	P2S1	P3S2	P4S3	P3S1	P3S2
Sb8	P1S2	P2S1	P3S4	P4S3	P4S1	P4S2	P1S4	P2S1	P3S3	P4S2	P4S1	P4S2
Sb9	P1S2	P2S3	P3S1	P4S4	P1S1	P1S2	P1S3	P2S1	P3S2	P4S4	P1S1	P1S2
Sb10	P1S2	P2S3	P3S4	P4S1	P2S1	P2S2	P1S3	P2S1	P3S4	P4S2	P2S1	P2S2
Sb11	P1S2	P2S4	P3S1	P4S3	P3S1	P3S2	P1S3	P2S2	P3S1	P4S4	P3S1	P3S2
Sb12	P1S2	P2S4	P3S3	P4S1	P4S1	P4S2	P1S3	P2S2	P3S4	P4S1	P4S1	P4S2
Sb13	P1S3	P2S1	P3S2	P4S4	P1S1	P1S2	P1S2	P2S3	P3S1	P4S4	P1S1	P1S2
Sb14	P1S3	P2S1	P3S4	P4S2	P2S1	P2S2	P1S2	P2S3	P3S4	P4S1	P2S1	P2S2
Sb15	P1S3	P2S2	P3S1	P4S4	P3S1	P3S2	P1S2	P2S4	P3S1	P4S3	P3S1	P3S2
Sb16	P1S3	P2S2	P3S4	P4S1	P4S1	P4S2	P1S2	P2S4	P3S3	P4S1	P4S1	P4S2
Sb17	P1S3	P2S4	P3S1	P4S2	P1S1	P1S2	P1S1	P2S4	P3S2	P4S3	P1S1	P1S2
Sb18	P1S3	P2S4	P3S2	P4S1	P2S1	P2S2	P1S1	P2S4	P3S3	P4S2	P2S1	P2S2
Sb19	P1S4	P2S1	P3S2	P4S3	P3S1	P3S2	P1S2	P2S1	P3S3	P4S4	P3S1	P3S2
Sb20	P1S4	P2S1	P3S3	P4S2	P4S1	P4S2	P1S2	P2S1	P3S4	P4S3	P4S1	P4S2
Sb21	P1S4	P2S2	P3S1	P4S3	P1S1	P1S2	P1S1	P2S2	P3S3	P4S4	P1S1	P1S2

Versión de la encuesta	Regla R1						Regla R2					
	Análisis de métodos				Comparación de métodos		Análisis de métodos				Comparación de métodos	
Sb22	P1S4	P2S2	P3S3	P4S1	P2S1	P2S2	P1S1	P2S2	P3S4	P4S3	P2S1	P2S2
Sb23	P1S4	P2S3	P3S1	P4S2	P3S1	P3S2	P1S1	P2S3	P3S2	P4S4	P3S1	P3S2
Sb24	P1S4	P2S3	P3S2	P4S1	P4S1	P4S2	P1S1	P2S3	P3S4	P4S2	P4S1	P4S2

C. Anexo: Notificación de aceptación de artículo en conferencia ICSME19

18/7/2019

Correo de Universidad Nacional de Colombia - ICSME19 notification for paper 98 (Accept)



Sergio Luis Lubo Argumedo <sluboa@unal.edu.co>

ICSME19 notification for paper 98 (Accept)

3 mensajes

ICSME19 <icsme19@easychair.org>
Para: Sergio Lubo <sluboa@unal.edu.co>

10 de junio de 2019, 15:43

Dear Sergio Lubo,

Congratulations! We are pleased to inform you that your paper

An Empirical Study Assessing Source Code Readability in Comprehension

has been accepted for inclusion in the Research Track of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), to be held in Cleveland, Ohio, USA, September 30-October 4, 2019.

The program committee selected 31 out of 135 submissions for inclusion in the research program. All submissions were reviewed by at least three members of the program committee and discussed online.

We enclose the reviews of your paper. In preparation of the final version of your paper, please make sure to incorporate the suggestions made by the reviewers.

IMPORTANT INFORMATION:

1. Please note that you are not allowed to change the author list, except to correct typos in the names. If a correction is needed (e.g. because the author name was misspelled), we need to approve the change.
2. Each accepted paper must have at least one author registered by the early registration deadline, which will be posted on the conference website soon. The registration site will be open soon. In the registration form, please indicate the paper ID: icsme-research-98
3. The camera-ready paper is due on July 31. The information about the IEEE online author kit will be sent to you in a separate email once it is available. Papers must not exceed 10 pages (including figures and appendices) plus up to 2 pages that contain ONLY references.
4. Got artifacts? The ICSME artifacts track promotes, celebrates, and catalogs excellent research artifacts in software engineering. Submit your artifact by June 24 to receive special recognition and an extra page in the proceedings to describe the artifact.
https://icsme2019.github.io/cfp_artifacts_track.html
5. Please also consider submitting to the other ICSME 2019 tracks:
 - * Late Breaking Ideas Track (deadline: June 14)
 - * Tool Demonstrations Track (deadline: June 17)
 - * Industry Track (extended abstract deadline: June 14)
 - * Journal First Track (deadline: June 14), and
 - * Doctoral Symposium (deadline: June 24),

or to any of the co-located events: SCAM or VISSOFT.

6. Please be advised that, because visa processing may take some time, you should organize your travel arrangements early. More information is available on the conference website.

7. We encourage you to like our Facebook page (<https://www.facebook.com/icsmeconf/>) and invite friends and colleagues to participate in ICSME. Or follow @IEEEICSME on Twitter.

8. We will have a program ready sometime in the middle of the summer and will send out instructions regarding presentation later.

Congratulations again on your paper acceptance. We look forward to seeing you in ICSME 2019 in Cleveland!

Best regards,

D. Anexo: Recomendaciones de legibilidad elegibles

#	Recommendation	Example	Source
1	Order of operands in relational expressions. "Put in the right-hand side the expression whose value is more constant"	if (age >= 8)... would be more readable than if (8 <= age) ...	[31]. pág.70
2	Order of if/else blocks. "Prefer dealing with the positive/simpler/more interesting case first"	if(n==0) ... would be more readable than if(n!=0) ...	[31]. pág.71
3	Ternary operator. By default, use an if/else. The ternary ?: should be used only for the simplest cases.	timeStr += (hour >= 12) ? "pm" : "am"; would be more readable than if (hour >= 12) { timeStr += "pm"; } else { timeStr += "am"; }	[31]. pág.73
4	Avoid do/while loops. ("Typically, logical conditions are above the code they guard. Because you typically read code from top to bottom, this makes do/while a bit unnatural. Many readers end up reading the code twice. "). Another reason to avoid do/while is that the <i>continue</i> statement can be confusing inside it.	Check the discussion at http://www.java-forums.org/new-java/34604-when-while-vs-do-while.html Stack Overflow discussion: http://stackoverflow.com/questions/3347001/do-while-vs-while	[31]. pág.74

5	Return early from a function	<pre>if (r == null) return false; if (r.element.equals(x)) return true; if (r.element.compareTo(x) < 0) return contains(x,r.rightBranch); return contains(x,r.leftBranch);</pre> <p>would be more readable than</p> <pre>if (r == null) result = false; else if (r.element.equals(x)) result = true; else if (r.element.compareTo(x) < 0) result = contains(x,r.rightBranch); else result = contains(x,r.leftBranch); return result;</pre>	[31]. pág.75
6	Minimize nesting.	<pre>int theLargest = a; if (b > theLargest) theLargest = b; if (c > theLargest) theLargest = c; return theLargest;</pre> <p>would be more readable than</p> <pre>if (a<b) if (b<c) theLargest = c else theLargest = b; else if (a<c) theLargest = c else theLargest = a return theLargest;</pre>	[31]. pág.77

<p>7</p>	<p>Use continue inside loops to reduce nesting. In this case, <i>continue</i> is the analogous technique to returning early</p>	<pre>for (int i = 0; i < results.size(); i++) { if (results[i] != NULL) { non_null_count++; if (results[i].name != "") { System.out.println("Considering candidate..."); ... } } }</pre> <p>would be less readable than</p> <pre>for (int i = 0; i < results.size(); i++) { if (results[i] == NULL) continue; non_null_count++; if (results[i].name == "") continue; System.out.println("Considering candidate..."); ... }</pre>	<p>[31]. pág.78</p>
<p>8</p>	<p>Break down giant expressions into more digestible pieces (Using explaining variables)</p>	<pre>String name = sc.nextLine().trim(); if (name.length() != 0) ...</pre> <p>would be more readable than</p> <pre>if (sc.nextLine().trim().length() != 0) ...</pre>	<p>[31]. pág.84</p>

9	<p>Define a new variable to capture the meaning or purpose of an expression (using a summary variables)</p>	<pre>if (request.user.id == document.owner_id) { // user can edit this document... } ... if (request.user.id != document.owner_id) { // document is read-only... }</pre> <p>would be less readable than</p> <pre>final boolean user_owns_document = (request.user.id == document.owner_id); if (user_owns_document) { // user can edit this document... } ... if (!user_owns_document) { // document is read-only... }</pre>	
10	<p>Manipulate conditional expressions using De Morgan's laws:</p> <p>1) not (a or b or c) \Leftrightarrow (not a) and (not b) and (not c) 2) not (a and b and c) \Leftrightarrow (not a) or (not b) or (not c)</p>	<pre>if (!(a && !b)) would be less readable than if (!a b)</pre>	[31]. pág.85
11	<p>Do not abuse short-circuit evaluation of the && and operators</p> <p>1) false && ... evaluates to false 2) true ... evaluates to true</p>	<pre>assert(!(bucket = FindBucket(key)) !bucket->IsOccupied());</pre> <p>would be less readable than</p> <pre>bucket = FindBucket(key); if (bucket != NULL) assert(!bucket->IsOccupied());</pre>	[31]. pág.86
12	<p>Break down giant expressions into more digestible pieces (solving the problem the "opposite" way)</p>	<p>Check Problem 4. The problem is to determine whether two intervals overlap</p>	[31]. pág.86

<p>13</p>	<p>Eliminate variables that do not improve readability</p> <ul style="list-style-type: none"> ● Useless temporary variable <ul style="list-style-type: none"> ○ It isn't breaking down a complex expression. ○ It doesn't add clarification. ○ It's used only once, so it doesn't compress any redundant code. ● Eliminating Intermediate Results <ul style="list-style-type: none"> ○ In general, it's a good strategy to complete the task as quickly as possible. ● Eliminating control flow variables <i>"Their sole purpose is to steer the program's execution—they don't contain any real program data"</i> (the variable done in the last example) 	<pre> now = datetime.datetime.now() root_message.last_view_time = now would be less readable than root_message.last_view_time = datetime.datetime.now() ----- var remove_one = function (array, value_to_remove) { var index_to_remove = null; for (var i = 0; i < array.length; i += 1) { if (array[i] === value_to_remove) { index_to_remove = i; break; } } if (index_to_remove !== null) { array.splice(index_to_remove, 1); } }; would be less readable than var remove_one = function (array, value_to_remove) { for (var i = 0; i < array.length; i += 1) { if (array[i] === value_to_remove) { array.splice(i, 1); return; } } }; ----- boolean done = false; while (<i>/* condition */</i> && !done) { ... if (...) { done = true; continue; } } is less readable than while (<i>/* condition */</i>) { ... if (...) { break; } } </pre>	<p>[31]. pág.94</p>
<p>14</p>	<p>Reduce the scope of each variable</p>		<p>[31]. pág. 97</p>
<p>15</p>	<p>Prefer write-once variables</p>		<p>[31]. pág.103</p>

16	Moving the variable definition down to the place where they are first used	<pre>def ViewFilteredReplies(original_id): filtered_replies= [] root_message= Messages.objects.get(original_id) all_replies= Messages.objects.select(root_id=original_id) root_message.view_count += 1 root_message.last_view_time = datetime.datetime.now() root_message.save() for reply in all_replies: if reply.spam_votes <= MAX_SPAM_VOTES: filtered_replies.append(reply) return filtered_replies Would be less readable than def ViewFilteredReplies(original_id): root_message = Messages.objects.get(original_id) root_message.view_count += 1 root_message.last_view_time = datetime.datetime.now() root_message.save() all_replies = Messages.objects.select(root_id=original_id) filtered_replies = [] for reply in all_replies: if reply.spam_votes <= MAX_SPAM_VOTES: filtered_replies.append(reply) return filtered_replies</pre>	[31]. pág.101
----	--	---	---------------

17	<p>Extracting from unrelated subproblems</p> <ul style="list-style-type: none"> Extracting a function that can be reused in the future which is self-contained and unaware of how applications are using it. 	<pre> var findClosestLocation = function (lat, lng, array) { var closest; var closest_dist = Number.MAX_VALUE; for (var i = 0; i < array.length; i += 1) { // Convert both points to radians. var lat_rad = radians(lat); var lng_rad = radians(lng); var lat2_rad = radians(array[i].latitude); var lng2_rad = radians(array[i].longitude); // Use the "Spherical Law of Cosines" formula. var dist = Math.acos(Math.sin(lat_rad) * Math.sin(lat2_rad) + Math.cos(lat_rad) * Math.cos(lat2_rad) * Math.cos(lng2_rad - lng_rad)); if (dist < closest_dist) { closest = array[i]; closest_dist = dist; } } return closest; }; Would be less readable than var spherical_distance = function (lat1, lng1, lat2, lng2) { var lat1_rad = radians(lat1); var lng1_rad = radians(lng1); var lat2_rad = radians(lat2); var lng2_rad = radians(lng2); // Use the "Spherical Law of Cosines" formula. return Math.acos(Math.sin(lat1_rad) * Math.sin(lat2_rad) + Math.cos(lat1_rad) * Math.cos(lat2_rad) * Math.cos(lng2_rad - lng1_rad)); }; var findClosestLocation = function (lat, lng, array) { var closest; var closest_dist = Number.MAX_VALUE; for (var i = 0; i < array.length; i += 1) { var dist = spherical_distance(lat, lng, array[i].latitude, array[i].longitude); if (dist < closest_dist) { closest = array[i]; closest_dist = dist; } } return closest; }; </pre>	[31]. pág.110
----	---	---	---------------

18	Code should be organized so that it is doing one task at a time	<pre>var vote_changed = function (old_vote, new_vote) { var score = get_score(); if (new_vote !== old_vote) { if (new_vote === 'Up') { score += (old_vote === 'Down' ? 2 : 1); } else if (new_vote === 'Down') { score -= (old_vote === 'Up' ? 2 : 1); } else if (new_vote === "") { score += (old_vote === 'Up' ? -1 : 1); } } set_score(score); };</pre> <p>Would be less readable than</p> <pre>var vote_value = function (vote) { if (vote === 'Up') return +1; if (vote === 'Down') { return -1; } return 0; } var vote_changed = function (old_vote, new_vote) { var score = get_score(); score -= vote_value(old_vote); // remove the old vote score += vote_value(new_vote); // add the new vote set_score(score); };</pre>	[31]. pág.123
----	---	---	---------------

19	Use methods to cleanup irregularity	<pre>// Turn a partial_name like "Doug Adams" into "Mr. Douglas Adams". // If not possible, 'error' is filled with an explanation. string ExpandFullName(DatabaseConnection dc, string partial_name, string* error); DatabaseConnection database_connection; string error; assert(ExpandFullName(database_connection, "Doug Adams", &error) == "Mr. Douglas Adams"); assert(error == ""); assert(ExpandFullName(database_connection, " Jake Brown ", &error) == "Mr. Jacob Brown III"); assert(error == ""); assert(ExpandFullName(database_connection, "No Such Guy", &error) == ""); assert(error == "no match found"); assert(ExpandFullName(database_connection, "John", &error) == ""); assert(error == "more than one result"); Would be less readable than CheckFullName("Doug Adams", "Mr. Douglas Adams", ""); CheckFullName(" Jake Brown ", "Mr. Jake Brown III", ""); CheckFullName("No Such Guy", "", "no match found"); CheckFullName("John", "", "more than one result"); void CheckFullName(string partial_name, string expected_full_name, string expected_error) { // database_connection is now a class member string error; string full_name = ExpandFullName(database_connection,partial_name, &error); assert(error == expected_error); assert(full_name == expected_full_name); }</pre>	[31]. pág.38
----	-------------------------------------	--	--------------

20	Pick a meaningful order and use it consistently	<pre>details = request.POST.get('details') location = request.POST.get('location') phone = request.POST.get('phone') email = request.POST.get('email') url = request.POST.get('url')</pre> <p>Would be more readable than</p> <pre>if details: rec.details = details if phone: rec.phone = phone # Hey, where did 'location' go? if email: rec.email = email if url: rec.url = url if location: rec.location = location # Why is 'location' down here now?</pre>	[31]. pág.39
----	---	--	--------------

21	Organize declaration into blocks	<pre>class FrontendServer { public: FrontendServer(); ~FrontendServer(); // Handlers void ViewProfile(HttpRequest* request); void SaveProfile(HttpRequest* request); void FindFriends(HttpRequest* request); // Request/Reply Utilities string ExtractQueryParam(HttpRequest* request, string param); void ReplyOK(HttpRequest* request, string html); void ReplyNotFound(HttpRequest* request, String error); // Database Helpers void OpenDatabase(string location, string user); void CloseDatabase(string location); };</pre> <p>Would be more readable than</p> <pre>class FrontendServer { public: FrontendServer(); void ViewProfile(HttpRequest* request); void OpenDatabase(string location, string user); void SaveProfile(HttpRequest* request); string ExtractQueryParam(HttpRequest* request, string param); void ReplyOK(HttpRequest* request, string html); void FindFriends(HttpRequest* request); void ReplyNotFound(HttpRequest* request, string error); void CloseDatabase(string location); ~FrontendServer(); };</pre>	[31]. pág.40
----	----------------------------------	---	--------------

22	Avoid ambiguous pronouns while commenting	<pre>//Insert the data into the cache, but check if it's too big first</pre> <p>Would be less readable than</p> <pre>//Insert the data into the cache, but check if the data is too big first</pre> <p>As it might refer to the data or cache and one should figure that out reading the rest of the code</p>	[31]. pág.60
23	Don't use comments when you can use a function or a variable	<pre>// does the module from the global list <mod> depend on the // subsystem we are part of? if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem())) This could be rephrased without the comment as ArrayList moduleDependees = smodule.getDependSubsystems(); String ourSubSystem = subSysMod.getSubSystem(); if (moduleDependees.contains(ourSubSystem))</pre>	[32]. pág.67

24	Use exceptions rather than return codes	<pre>public class DeviceController { ... public void sendShutDown() { DeviceHandle handle = getHandle(DEV1); // Check the state of the device if (handle != DeviceHandle.INVALID) { // Save the device status to the record field retrieveDeviceRecord(handle); // If not suspended, shut down if (record.getStatus() != DEVICE_SUSPENDED) { pauseDevice(handle); clearDeviceWorkQueue(handle); closeDevice(handle); } else { logger.log("Device suspended. Unable to shut down"); } } else { logger.log("Invalid handle for: " + DEV1.toString()); } }...} Would be less readable than public class DeviceController { ... public void sendShutDown() { try { tryToShutDown(); } catch (DeviceShutDownError e) { logger.log(e); } } private void tryToShutDown() throws DeviceShutDownError { DeviceHandle handle = getHandle(DEV1); DeviceRecord record = retrieveDeviceRecord(handle); pauseDevice(handle); clearDeviceWorkQueue(handle); closeDevice(handle); } private DeviceHandle getHandle(DeviceID id) { ... throw new DeviceShutDownError("Invalid handle for: " + id.toString()); ... }...}</pre>	[32]. pág.104
----	---	--	---------------

25	Pick the simplest set of inputs that completely exercise the code	<pre>CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3", "4, 3, 1");</pre> <p>Would be difficult than</p> <pre>CheckScoresBeforeAfter("1, 2, 3", "3, 2, 1");</pre>	[31]. pág.156
----	---	--	---------------

Bibliografía

- [1] R. Brooks, «Towards a theory of the comprehension of computer programs,» *International journal of man-machine studies*, vol. 18, pp. 543-554, 1983.
- [2] E. Soloway y K. Ehrlich, «Empirical studies of programming knowledge,» *IEEE Transactions on software engineering*, Vols. %1 de %2SE-10, pp. 595-609, 1984.
- [3] M.-A. Storey, «Theories, methods and tools in program comprehension: past, present and future,» de *13th International Workshop on Program Comprehension (IWPC'05)*, 2005.
- [4] A. Mayrhauser y A. M. Vans, «Program understanding behavior during debugging of large scale software,» de *Papers presented at the seventh workshop on Empirical studies of programmers*, 1997.
- [5] R. Flesch, «A new readability yardstick,» *Journal of Applied Psychology*, vol. 32, pp. 221-233, 1948.
- [6] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif y S. Tamm, «Eye Movements in Code Reading: Relaxing the Linear Order,» de *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, Florence, 2015.
- [7] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk y R. Oliveto, «Automatically Assessing Code Understandability: How Far Are We?,» de *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, Urbana-Champaign, IL, USA, 2017.
- [8] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk y R. Oliveto, «Improving code readability models with textual features,» de *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016.

- [9] J. Börstler, M. E. Caspersen y M. Nordström, «Beauty and the Beast: On the Readability of Object-oriented Example Programs,» *Software Quality Journal*, vol. 24, pp. 231-246, 2016.
- [10] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner y B. Vasilescu, «"Automatically assessing code understandability" reanalyzed: combined metrics matter,» de *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018.
- [11] K. B. Lakshmanan, S. Jayaprakash y P. K. Sinha, «Properties of Control-Flow Complexity Measures,» *IEEE Transactions on Software Engineering*, vol. 17, pp. 1289-1295, 1991.
- [12] G. K. Gill y C. F. Kemerer, «Cyclomatic complexity density and software maintenance productivity,» *IEEE Transactions on Software Engineering*, vol. 17, pp. 1284-1288, 1991.
- [13] A. Jbara, A. Matan y D. G. Feitelson, «High-MCC functions in the Linux kernel,» de *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012.
- [14] S. Ajami, Y. Woodbridge y D. G. Feitelson, «Syntax, Predicates, Idioms - What Really Affects Code Complexity?,» de *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017.
- [15] J. Shao y Y. Wang, «A new measure of software complexity based on cognitive weights,» *Canadian Journal of Electrical and Computer Engineering*, vol. 28, pp. 69-74, 2003.
- [16] Y. Wang, «Cognitive Complexity of Software and its Measurement,» de *2006 5th IEEE International Conference on Cognitive Informatics*, 2006.
- [17] J. C. Johnson, S. Lubo, N. Yedla, J. Aponte y B. Sharif, «An Empirical Study Assessing Source Code,» de *35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH USA, 2019.
- [18] D. Lawrie, C. Morrell, H. Feild y D. Binkley, «What's in a Name? A Study of Identifiers,» de *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006.
- [19] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell y B. Sharif, «The Impact of Identifier Style on Effort and Comprehension,» *Empirical Software Engineering*, vol. 18, pp. 219-276, 2013.

- [20] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel y M. Beigl, «Descriptive Compound Identifier Names Improve Source Code Comprehension,» de *Proceedings of the 26th Conference on Program Comprehension*, Gothenburg, 2018.
- [21] S. Fakhoury, Y. Ma, V. Arnaudova y O. Adesope, «The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load,» de *Proceedings of the 26th Conference on Program Comprehension*, Gothenburg, 2018.
- [22] T. R. Beelders y J.-P. L. Plessis, «Syntax highlighting as an influencing factor when reading and comprehending source code,» *Journal of Eye Movement Research*, vol. 9, 2015.
- [23] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachsel, M. Papendieck, T. Leich y G. Saake, «Do Background Colors Improve Program Comprehension in the #Ifdef Hell?,» *Empirical Software Engineering*, vol. 18, pp. 699-745, 2013.
- [24] R. P. L. Buse y W. R. Weimer, «Learning a Metric for Code Readability,» *IEEE Transactions on software engineering*, vol. 36, pp. 546-558, 2010.
- [25] R. M. Santos y M. A. Gerosa, «Impacts of Coding Practices on Readability,» de *Proceedings of the 26th Conference on Program Comprehension*, Gothenburg, 2018.
- [26] M. E. Hansen, R. L. Goldstone y A. Lumsdaine, «What Makes Code Hard to Understand?,» *CoRR*, vol. abs/1304.5257, 2013.
- [27] A. Wulff-Jensen, K. Ruder, E. Triantafyllou y L. E. Bruni, «Gaze Strategies Can Reveal the Impact of Source Code Features on the Cognitive Load of Novice Programmers,» de *Advances in Neuroergonomics and Cognitive Engineering*, 2019.
- [28] E. R. Iselin, «Conditional statements, looping constructs, and program comprehension: an experimental study,» *International Journal of Man-Machine Studies*, vol. 28, pp. 45-66, 1988.
- [29] J. Borstler y B. Paech, «The Role of Method Chains and Comments in Software Readability and Comprehension-An Experiment,» *IEEE Transactions on Software Engineering*, vol. 42, pp. 886-898, 2016.

- [30] T. Sedano, «Code Readability Testing, an Empirical Study,» de *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, 2016.
- [31] D. Boswell y T. Foucher, *The Art of Readable Code*, O'Reilly Media, Inc., 2011.
- [32] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1 ed., Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.