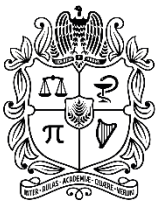


UNIVERSIDAD
NACIONAL
DE COLOMBIA

Consistencia lingüística: Métrica para evaluar la calidad semántica entre los requisitos funcionales y el código fuente

Carlos Alberto Botero Restrepo

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín Antioquia, Colombia

2020

Consistencia lingüística: Métrica para evaluar la calidad semántica entre los requisitos funcionales y el código fuente

Carlos Alberto Botero Restrepo

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

Director:
Ph.D. Carlos Mario Zapata Jaramillo

Línea de Investigación:
Ingeniería de software
Grupo de investigación:
Lenguajes computacionales

Universidad Nacional de Colombia
Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión
Medellín, Colombia
2020

A mi esposa por su apoyo, a mi director por sus enseñanzas, a todas las personas que creyeron en mi, a la academia por obligarme a crecer y a Dios por el tiempo de vida para lograrlo.

Agradecimientos

A la academia; volver a ella siempre será recordar lo poco que sabemos, lo valioso que es aprender y el esfuerzo que lleva hacerlo. La universidad es un lugar donde se respira responsabilidad, aprender por la sociedad y no por uno mismo, de manera directa la academia nos hace responsables de ser mejores y de hacer una sociedad con más criterio. Gracias a la Universidad Nacional de Colombia y sus profesores porque allí me obligaron a crecer.

A mi familia y mi director Carlos Mario Zapata Jaramillo, porque me apoyaron y acompañaron durante el proceso de estudio, valorando los resultados obtenidos; comprendieron mis silencios y ausencias cuando fue necesario y fueron más allá, apoyándome en todo lo que parecía sin sentido, pero que me hace feliz.

A Dios, porque unió todo lo necesario para que yo pueda tener esta oportunidad.

Resumen

El desarrollo de software se debe ver como proceso más que como una simple codificación. En este proceso, a los requisitos funcionales se les transforma su sintaxis en lenguaje controlado sin afectar la semántica del sistema de software. Sin embargo, debido a algunas prácticas de los desarrolladores, al comparar el código fuente con los requisitos funcionales se encuentran diferencias en los términos, lo que afecta el tiempo del entendimiento, aumenta los costos y vuelve inútiles los recursos que sirven para comprender el sistema de software en la fase de mantenimiento.

Existen estudios en los que se resalta la importancia de la comprensión y cohesión con base en el nombramiento del código fuente para que este tenga sentido cuando se lee. Por esto, algunos autores exponen métricas de calidad de software y herramientas que complementan los entornos de desarrollo, para dar sugerencias de nombramiento y velar por su uso. Sin embargo, para el entendimiento del código fuente no se toma en cuenta la documentación ni los modelos. A pesar de que se reconoce la importancia de la documentación, no hay métricas que permitan evaluar cuánto de los modelos está en el código fuente ni de qué manera y, aún con las métricas existentes, la comprensión del código fuente sigue siendo un proceso que tarda más tiempo del esperado en la fase de mantenimiento.

Por lo anterior, en esta Tesis de Maestría se propone una métrica de consistencia lingüística, la cual, usando una mezcla de LSA (*Análisis semántico latente* por sus siglas

en inglés) y una adaptación a la *distancia de Levenshtein,* permite cuantificar la distancia entre el código fuente y los modelos con base en el cambio de terminología. La métrica propuesta se prueba de manera didáctica mediante un juego y un caso de laboratorio, en el que se expone cómo se afecta el entendimiento de un sistema de software cuando los términos cambian y cómo se mide usando la métrica de consistencia lingüística propuesta.

Con la métrica propuesta se confirma de manera cuantitativa que el cambio de las palabras afecta el entendimiento del sistema de software, se identifican los cambios que crean mayor deterioro en el entendimiento del código fuente y se muestra que la terminología utilizada en la educción de requisitos y plasmada en los modelos se debe tomar como recurso fundamental.

Palabras clave: Consistencia lingüística, métrica, distancia semántica, calidad de software, código fuente.

Linguistic consistency: Metric to evaluate semantic quality between functional requirements and source code

Abstract

Software development should be seen as a process rather than simple coding. In this process, syntax of functional requirements is transformed into controlled language leaving intact the semantics of the software system. Nonetheless, in software systems—due to some developer practices— differences are found in the terms when comparing the source code with the functional requirements, affecting comprehension times, increasing costs, and rendering useless the resources used for understanding the software system in the maintenance stage.

Two factors are highlighted in some studies: Importance of code understanding and cohesion based on naming the source code in a way that makes sense when reading it. For this reason, some authors present software quality metrics and tools for complementing development environments, offering naming suggestions, and ensure their usage. However, documentation and models are left behind for understanding the source code. Although the importance of documentation is recognized, no metrics are available for

evaluating how much of the model is in the source code and how it is there. Even with existing metrics, source code understanding remains as a process taking longer than expected in the maintenance stage.

For the aforementioned reasons, in this MEng. Thesis, we propose a linguistic consistency metric by using a mixture of LSA (*Latent Semantic Analysis*) and an adaptation to the *Levenshtein distance* in order to quantify the distance between the source code and the model based on the change of terminology. The proposed metric is tested in a didactic way by using a game and a lab case, which shows how the understanding of software is affected when the terms change and how it is measured by using the proposed linguistic consistency metric.

We confirm in a quantitative way with the proposed metric that changes in wording affect software system understanding, we identify which changes diminish source code understanding, and we show that terminology used in the requirements elicitation and reflected in the models should be taken as a fundamental resource.

Key words: Linguistic consistency, metric, semantic distance, quality software, source code.

Contenido

Lista de figuras

Pág.

Lista de tablas

Pág.

Lista de Símbolos y abreviaturas

Abreviaturas

Abreviatura	Término
1.LSA	Análisis semántico latente
2.RL	Relación de Levenshtein

1. Introducción

1.1 Justificación

En el desarrollo de software se espera que los recursos usados para implementar el código fuente sean herramientas útiles en la fase de mantenimiento, pero es común encontrar que los términos usados en las diferentes fases de la construcción de los sistemas de software no coinciden, generando, como explica Siegmund (2016), problemas de tiempo, sobrecostos y deterioro de los recursos para el mantenimiento.

Según Fahmi y Choi (2007) los sistemas de software se pueden definir más como un proceso que como una simple codificación. Asimismo, Zapata y Manrique (2014) lo ratifican como la transformación del lenguaje natural a uno controlado, que no presenta los problemas del exceso de información, uso de sinónimos y las ambigüedades que se tienen en el discurso del interesado. Por esto, el sistema de software puede sufrir cambios sintácticos, más no semánticos durante dicha transformación.

En esta misma línea, Etzkorn y Delugach (2000) y Allamanis *et al.* (2014) exponen lo relevante de la coherencia, la cohesión y la comprensión que debe tener el código fuente, ligadas con un buen nombramiento de los términos, de manera que tengan sentido cuando se lean. Pastor y Molina (2007) exponen la importancia de los requisitos y su consistencia, al igual que la coherencia con el código fuente como un proceso automático donde el código fuente se genera a partir de los modelos. Sin embargo, Siegmund (2016), a pesar de reconocer el impacto que tiene en costos y tiempo el entendimiento del código fuente,

expone que hay prácticas que se transmiten entre generaciones que afectan el nombramiento del código fuente, porque algunos sistemas de software cuentan con limitantes que son más importantes para su correcto funcionamiento que el entendimiento de su código fuente.

Para afrontar los problemas asociados con el entendimiento del código fuente, se analizan diferentes métricas de calidad de software de autores como Avidan y Feitelson (2017), Binkley *et al.* (2011) y Allamanis *et al*. (2014), entre otros, los cuales resaltan la importancia de la coherencia y consistencia en el código fuente, las prácticas de programación y el impacto que tiene el cambio de la terminología para su entendimiento. Allamanis *et al.* (2014) y Marcilio *et al.* (2019) también estudian el tema de manera práctica, al exponer herramientas que permiten automatizar los procesos de validación de las métricas y otras que sugieren la forma de nombrar en el código fuente de manera que sea más comprensible cuando se lee. Sin embargo, en los estudios, métricas y herramientas encontradas no se tienen en cuenta los modelos para realizar las sugerencias o alertas a los desarrolladores, lo que deja una brecha entre los requisitos funcionales y el código fuente en términos de consistencia lingüística y, por tanto, no se resuelve el problema del entendimiento del software en la fase de mantenimiento en su totalidad.

El entendimiento y consistencia en los sistemas de software se orientan con la arquitectura que sigan, sea entendiendo el sistema línea a línea o de manera global. Sin embargo, independientemente de la arquitectura, se tienen conceptos como el que se promulga en el principio cuatro del manifiesto ágil, que enuncia: “Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto”. Sin embargo, no hay un método con el que se mida esta consistencia, y la intención de mantener los mismos términos en el código fuente y los modelos, por lo que se puede perder esa consistencia si las personas no tienen la voluntad de seguirlo.

Por todo lo anterior, y al identificar las diferentes prácticas de nombramiento del código fuente y cómo afectan el entendimiento del sistema, en esta Tesis de Maestría se propone

una métrica de calidad semántica llamada *métrica de consistencia lingüística*, con la cual se establece la distancia entre las palabras usadas en los modelos y el código fuente mediante el uso de LSA (*Análisis semántico latente* por sus siglas en inglés) y una adaptación a la *distancia de Levenstein*, para cubrir las palabras relacionables con su significado y otras más que se usan y no tienen sentido por si solas como siglas, abreviaturas y traducciones a otros idiomas, entre otras. Así, se cubre todo el dominio del sistema de software y se identifica la consistencia al detalle que se requiera.

Para validar la necesidad de tener la métrica propuesta, se construye un juego didáctico basado en Gómez (2010), que expone las percepciones de diferentes personas respecto del adecuado uso del nombramiento de los elementos en el código fuente y lo relevante de la consistencia con los demás productos de trabajo. También, se expone un caso de laboratorio, en el cual se muestra el proceso desglosado del uso de la métrica propuesta y cómo se obtienen mejoras con su uso para el entendimiento del código fuente a la luz de los modelos.

Al medir la consistencia lingüística se conoce cuánto se afecta el entendimiento del código fuente respecto de los modelos por el cambio en la terminología y, con ello, cuáles cambios de términos son los que causan mayor deterioro. También, se identifica cómo los cambios sintácticos afectan la semántica del software y deterioran el entendimiento de las sentencias funcionales vistas desde el código fuente. Por esto, el uso de la métrica propuesta ayuda a mejorar el entendimiento del sistema de software e involucra a los modelos en el proceso, logrando que un sistema de software sea más consistente y acoplado a lo largo de todas sus fases, lo que permite enfocarse en los puntos que causan mayor deterioro en el entendimiento. Adicionalmente, la terminología usada en la educción de requisitos y plasmada en los modelos se convierte en un insumo importante que no se debe alterar en el proceso de construcción de un sistema de software.

1.2 Planteamiento del problema

Estudios encontrados para solucionar el problema de la comprensión del código fuente con base en el nombramiento se realizan mediante análisis y experimentación, como los de Avidan y Feitelson (2017), los cuales estudian el impacto que tiene el cambio en los nombres de los diferentes identificadores en un lenguaje especifico, de modo que se pueda saber cuáles identificadores afectan más la comprensión del código fuente. Binkley *et al.* (2011) investigan el uso de palabras adecuadas para los elementos de código fuente y qué estructura de palabras puede conducir a una mejor comprensión.

En esta misma línea, Blinman y Cockburn (2005), para medir el problema del entendimiento del código fuente en la fase de mantenimiento, realizan estudios experimentales para conocer cómo los cambios en los nombres del código fuente en diferentes estilos de arquitectura de software afectan su comprensión. Para identificar cuáles son los nombres que afectan de manera más significativa la comprensión del código fuente, cuestionan a las personas sobre cuánto recuerdan luego de leerlo y concluyen que los nombres descriptivos ayudan a reducir la complejidad del programa.

Otras métricas que estudian Etzkorn y Delugach (2000), Wang *et al.* (2014) y Allamanis *et al.* (2014) sobre experiencias semánticas, se enfocan en las convenciones de nombramiento, cohesión de la clase y calidad de la documentación de la clase, entre otros. Estos autores exponen la importancia del nombramiento en el código fuente, pero no involucran los modelos para sus evaluaciones o análisis.

Los sistemas de software cuentan para el control de calidad del código fuente con diferentes estrategias, herramientas y métricas encontradas en los diferentes antecedentes. Sin embargo, en ninguna de las métricas encontradas se toman en cuenta la documentación o los modelos, a pesar de ser un recurso fundamental para la

construcción del código fuente. También, en la comunidad científica se estudia la importancia de la documentación de los sistemas de software y su entendimiento, pero no hay métricas que permitan evaluar cuánto de los modelos está realmente en el código fuente y de qué manera. Adicionalmente, según Siegmund (2016), la comprensión del código fuente sigue siendo un proceso que tarda más tiempo del esperado en la fase de mantenimiento y no es comprensible rápidamente a la luz de los requisitos funcionales, a pesar de las diferentes métricas de comprensión del código fuente, herramientas de apoyo para los ambientes de desarrollo y marcos metodológicos para desarrollo de software que se implementan.

1.3 Objetivos

1.3.1 Objetivo General

Construir una métrica de calidad semántica que permita medir la consistencia lingüística entre el código fuente y los requisitos funcionales.

1.3.2 Objetivos específicos

- Identificar las prácticas de nombramiento que se usan actualmente en el desarrollo de software y determinar cómo afectan el entendimiento del sistema.

- Establecer la distancia semántica que existe entre una palabra usada en el modelo y los diferentes estilos de nombramiento identificados para un mismo concepto en el código fuente.

- Construir una fórmula que permita evaluar cuantitativamente la calidad semántica de un sistema y sirva como instrumento de medición sobre el cual se apalanque la métrica de calidad de consistencia lingüística de un sistema de software, basada en la terminología usada en los requisitos funcionales y el código fuente.

- Evaluar con la métrica de consistencia lingüística creada diferentes sistemas, para establecer la distancia entre el modelo y el código fuente.

1.4 Metodología

Para este trabajo investigativo, la metodología usada es una adaptación del método científico con un refuerzo para la exploración y el análisis basados en Kitchenham y Charters (2007). Durante las primeras etapas se usa una base de recolección estructurada y no estructurada, en la cual se hace un diagnóstico sobre la lingüística en el código fuente de un sistema de software y su comparación con los modelos y se usa el método científico para el desarrollo y la validación.

1.4.1 Exploración

Se realiza una revisión sistemática de literatura con base en el protocolo que establecen Kitchenham y Charters (2007), encontrando la información disponible y relevante para la investigación. Esta información se recolecta de manera estructurada, como diagnóstico que enmarca la relación que tiene un sistema de software como referencia lingüística de su código fuente en comparación con los modelos. Además, se buscan otras métricas de calidad de software referentes a la sintaxis y a la semántica en el código fuente, estrategias de nombramiento y análisis sobre comparaciones de las diferentes métricas. Con esto, se tiene la información necesaria y relevante para analizar y fortalecer la idea de que aún se

necesitan herramientas de evaluación de calidad del software que sean garantes de su legibilidad y entendimiento a la luz de los modelos, para mejorar estas características en la fase de mantenimiento.

En esta etapa, como una adaptación a lo que enuncian Kitchenham y Charters (2007) como “planificación de la revisión”, se definen protocolos de búsqueda mediante el uso de palabras clave (*software quality, linguistic consistency, software quality metrics, semantic distance, functional requirements, source code, codebase etc.*) con el fin de encontrar las diferentes fuentes que aporten al tema que se aborda en esta Tesis de Maestría y se tienen conversaciones frecuentes a manera de entrevistas con ingenieros experimentados en el desarrollo de software sobre el nombramiento de los elementos del código fuente. ¿En qué se basan? ¿Con qué argumentos nombran una variable? No se obtienen respuestas de búsqueda directa en los modelos, sino argumentos orientados con opiniones, prácticas heredadas y capacitaciones recibidas con expresiones como “lo mas comprensible posible, no importa que sea largo” (frase escuchada en capacitación laboral).

La revisión realizada arroja que no existe documentación que aborde puntualmente la forma de medir la relación semántica entre los modelos y el código fuente de un sistema de software. Por este motivo, los documentos en los cuales se basa esta Tesis de Maestría son estudios relacionados con el nombramiento del código fuente, enunciados sobre las necesidades que aún tienen los sistemas de software referentes al entendimiento y la semántica y el análisis de diferentes estilos de nombramiento presentes en la literatura y en la práctica. Además, se emplean conversaciones a manera de entrevistas a personas del medio del desarrollo de software sobre el nombramiento del código fuente, análisis del nombramiento que se hace en algunas fuentes de código abierto, ejemplos encontrados en los antecedentes estudiados para la presente Tesis de Maestría y acciones empíricas sobre las posibles formas de medición de la consistencia lingüística que tiene un sistema de software.

1.4.2 Análisis

Con la información recolectada, se encuentra que existe una amplia variedad en el nombramiento de los elementos en el código fuente. De esta variedad se toma lo relevante del contexto semántico para analizar el sentido de la escritura, con lo cual se enfoca la presente Tesis de Maestría en el entendimiento de las sentencias en el código fuente y se dejan por fuera del alcance las restricciones particulares de los lenguajes de programación y las formas de escritura de una palabra. Si una palabra se escribe en forma Camell, Pascal u otra que no altere su significado, se abstrae en el análisis de esta Tesis de Maestría para dar foco únicamente al entendimiento del sistema de software por el cambio de una palabra por otra y no por el cambio en la forma de escritura de la misma palabra. Algunas alteraciones a la hora de escribir una palabra como "*Person*" Vs "*PersonController*" no necesariamente implican que se disminuya el entendimiento para una persona que desarrolla sistemas de software o, por el contrario, que lo facilite. Sin embargo, si en los diagramas que se opta por tomar como referentes no se nombra igual, sí existe un cambio en las palabras y, por tanto, en el entendimiento de ambos. Los diferentes lenguajes de programación y la división de responsabilidades traen consigo restricciones de nombramiento que implican adaptaciones a los que originalmente el interesado piensa (por ejemplo, en Java no se permiten dos clases con el mismo nombre cuando pertenecen al mismo paquete y, se usan varias capas para dividir responsabilidades), pero es algo que la persona que hace los diseños más específicos debe pensar cuando la arquitectura se propone. De allí se parte que, si no se tienen nombres finales en los modelos y los elementos tienen nombres como los que se listan en la Tabla 1-1, se consideran alteraciones que se miden y presentan desviación con la métrica de consistencia lingüística.

A continuación, se listan algunas formas de nombramiento en el código fuente de repositorios abiertos, de entrevistas a personas y de contexto laboral alterando sus nombres (véase la Tabla 1-1).

Tabla 1-1: Tipos de nombramiento de elementos en el código fuente encontrados en experiencias laborales y fuentes de código abierto.

Fuente: Elaboración propia.

Tipos de nombramiento en el código fuente	**Características de los tipos de nombramiento de elementos en el código fuente**	
	Descripción	**Ejemplo**
Con prefijos o sufijos	Adición de características al elemento.	BillController, LongAmount, T012_PERSONA_SERVICE
Mezcla de idiomas	Mezcla de diferentes idiomas en una misma palabra o entre el modelo y el código fuente.	PersonaController, PagosBill, (Bill - SalesCheck)
Sinonimia	Uso de una palabra en el modelo y un sinónimo en el código fuente.	(Student - Pupil) (Libro - texto)
Abreviaciones	Reducción a las palabras originales del modelo.	(Person - per) (PagarFacturas - PFacturas)
Siglas	Reducción del número de letras a usar en las palabras o grupos de palabras.	(Person - p) (PagarFacturas - PF) (Cedula de ciudadanía - cc)
Palabras o letras sin sentido	Palabras que se encuentran y no tienen relación alguna con el elemento que representan en el modelo.	(Person - a)
Iguales	Palabras que están iguales en el modelo y el código fuente.	(Bill - Bill)

La Tabla 1-1, es el resultado de las respuestas que dieron personas con más de cinco años de experiencia en el desarrollo de software a las preguntas: ¿Cómo nombra los elementos del código fuente? Y ¿Dónde aprendió a nombrar los elementos del código fuente? Además de búsquedas en repositorios de desarrollo móvil (iOS y Android), web (Html y javascript) y cliente servidor (Outsystems y Powerbuilder) públicos y privados.

La complejidad que implica el rango de palabras de la Tabla 1-1, hace necesario delimitar el alcance de la presente Tesis de Maestría a premisas concretas y volver en múltiples ocasiones a la etapa de exploración, con el fin de buscar diferentes formas de medir la distancia entre las palabras, para que sea posible sugerir métodos más acertados para las diferentes situaciones encontradas.

1.4.3 Desarrollo

Con las herramientas recolectadas en los pasos anteriores surgen diferentes formas de medir la distancia entre las palabras. Se inicia con versiones más simples y se evoluciona con las validaciones obtenidas (véase la Tabla 1-1) hasta llegar a una más completa, la cual cuenta con una composición en la que se considera si la palabra es diferente, pero es coherente y real. Puede ser diferente por un error de digitación, ser una representación de una palabra más compleja o una sigla que no cuenta con significado por si sola. Si la palabra existe, tiene relación semántica medible con otra palabra (la original que está en el diseño) o si, por el contrario, es un error sintáctico que afecta la coherencia del texto.

Para la cuantificación de la distancia entre las palabras, se opta por medirlas todas bajo dos técnicas diferentes; la primera cubre las palabras con sentido y la segunda las palabras que no cuentan con un significado por si solas. Se mide con las dos técnicas a todas las palabras, porque las que no tienen sentido pueden tener una relación con las originales, incluso mayor a un símil. Por ejemplo, si la palabra original es “*estudiante*”, puede ser de mayor entendimiento “*estudiant*” que “*pupilo*”; en este caso “*estudiant*” es una palabra que

no tiene valor por si sola, pues puede ser un error de digitación que sí afecta el entendimiento, pero no lo hace nulo.

Por lo anterior, con las herramientas recolectadas en los pasos anteriores, se investigan métodos que cumplen con las necesidades encontradas para analizar dos palabras: El Análisis Semántico Latente (LSA) que explican Venegas (2003) y Gutiérrez (2005), y una adaptación de la *distancia de Levenshtein* (1965) y que analizan Ene y Ene (2017). La primera es una técnica que permite evaluar la distancia semántica entre dos palabras con base en un corpus de conocimiento y, por tanto, tiene en cuenta el contexto y la segunda ayuda a evaluar de manera más cercana los errores sintácticos, las abreviaturas y las palabras que por sí solas no cuentan con sentido, evaluando los cambios que tienen una palabra y otra.

La evaluación de las palabras con un solo método bajo el criterio de un evaluador, además de ampliar el trabajo, sesga los resultados. Por este motivo, todas las palabras se barren con ambos métodos, de manera que se pueda evaluar bajo un criterio pragmático y uno mecánico y tomar el mejor resultado, para que la métrica sea objetiva y no basada en el criterio subjetivo de su evaluador.

Ambos métodos, el LSA (*Análisis semántico latente* por sus siglas en inglés) y la *distancia de Levenshtein*, tienen unidades de medida diferentes, por lo que se adecúan de manera que ambos métodos generen el mismo tipo de unidad, se puedan sumar cuando se necesite y se pueda ponderar una porción de software que mezcle palabras con diferentes características.

Luego de analizar todas las palabras de una porción de código, se suma el total y se da un resultado. A este proceso se le nombra *métrica de consistencia lingüística* y muestra la consistencia que tiene el sistema de software a lo largo de su construcción y, a su vez, qué tan comprensible es el código fuente a la luz de los requisitos funcionales o qué tanto entendimiento perdió con el cambio en la terminología.

1.4.4 Validación

Para esta etapa se usan dos herramientas diferentes; la primera es un juego construido como prueba de laboratorio y usado en diferentes etapas del proceso de la Tesis de Maestría como una validación empírica de la necesidad de la métrica de consistencia lingüística; la segunda es un caso de laboratorio, el cual se usa a lo largo del trabajo investigativo como explicación didáctica y sustento práctico de la *métrica de consistencia lingüística* y que se modifica de manera que el código fuente sea consistente con el modelo. De esta manera se expone la mejora del entendimiento en la lectura del código fuente con base en el modelo.

En esta etapa también se exponen las características de la *métrica de consistencia lingüística* propuesta en esta Tesis de Maestría, basadas en las opiniones de las personas que interactúan con el juego y qué aportes encuentran con el uso de la métrica, al igual que las conclusiones de la Tesis de Maestría y el trabajo futuro que queda luego de el análisis de los resultados obtenidos.

1.5 Estructura de la Tesis.

Esta Tesis de Maestría se estructura de la siguiente manera: en el Capítulo 2 se presenta el marco teórico, en el cual se abordan las consideraciones de los autores sobre la calidad de software, la relevancia del problema y la necesidad de cuantificar el problema del entendimiento del código fuente en la fase de mantenimiento de un sistema de software;

en el Capítulo 3 se describen los antecedentes sobre métricas que buscan consistencia con el código fuente, herramientas que buscan buen nombramiento y comparativos entre estas métricas y herramientas con la temática del entendimiento del código fuente basado en los modelos; en el Capítulo 4 se propone una métrica de calidad de software llamada *métrica de consistencia lingüística,* que permite cuantificar la distancia entre el código fuente y los modelos con base en el cambio de las palabras, se indican las formas de medir las distancias de las palabras y se presenta la fórmula definitiva para los bloques de código fuente; en el Capítulo 5 se expone un juego creado para validar de manera empírica de la necesidad de la métrica de consistencia lingüística y un caso de laboratorio donde se muestra y explica el uso de la métrica de consistencia lingüística; finalmente, en el Capítulo 6 se presentan las conclusiones, recomendaciones y el trabajo futuro que se puede derivar de esta Tesis de Maestría.

2. Marco teórico

2.1 Contexto del problema

Según Zapata y Manrique (2014), para la construcción de los sistemas de software se pasa por diferentes fases como el análisis y diseño antes de obtener el código fuente del sistema de software. Sin embargo, cuando es necesario intervenir el código fuente construido en la fase de mantenimiento, es común encontrar que lo que se diseña en los modelos no está exactamente igual en el código fuente y, según Siegmund (2016), entender lo que se construye es algo que tarda más tiempo del esperado, ocasionando problemas de tiempo, costos, utilidad de los recursos y mantenibilidad en los sistemas de software.

Las métricas y estrategias para mitigar el problema del entendimiento y la calidad de los sistemas de software se centran en gran parte en el código fuente para resolver el problema del entendimiento. Sin embargo, según Fahmi y Choi (2007), el desarrollo de los sistemas de software se puede definir como un proceso más que como una simple codificación y los modelos como abstracción a más alto nivel del código fuente permiten una más rápida comprensión de las funcionalidades; estos autores sólo incluyen paradigmas particulares de programación, como la programación orientada a aspectos y la programación orientada a objetos. Fahmi y Choi (2007) consideran que la implementación técnica puede pasar a un segundo plano, pues se puede mejorar mediante

la transpilación del lenguaje, por lo que los retos en los sistemas de software se transforman para adecuarse a las necesidades y requisitos que presenta un interesado y a las variables de riesgo que pueden afectar la comprensión y sostenibilidad del código fuente.

Según Zapata y Manrique (2014), durante la educción de los requisitos, el analista de software debe comprender lo que expresa el interesado en lenguaje natural, de manera que pueda plasmar sus necesidades y expectativas en los diseños. Sin embargo, esto implica que el analista interprete las palabras del interesado, las cuales pueden tener problemas de sinonimia, información que irrelevante para el sistema de software y redundancias, entre otros. Estas expresiones se deben manejar de manera que no lleguen al sistema de software. Sin embargo, Zapata y Manrique (2014) también afirman que no se debe cambiar la semántica de lo que expresa el usuario en el proceso de construcción del código fuente. Zapata y Manrique (2014) exponen que la transformación del lenguaje natural en un lenguaje controlado que se pueda ejecutar en un procesador es una interpretación simple de la programación, pero también explican por qué el código fuente (*lenguaje controlado*) debe ser semánticamente igual o más reducido en su universo de palabras que el requisito funcional (*lenguaje natural*), lo que hace suponer el lenguaje controlado como fácil de comprender. Sin embargo, Autores como Tiarks (2011) y Mayrhauser *et al.* (1997), cuando los cita Siegmund (2016) identifican que, en la fase de mantenimiento, los desarrolladores tardan la mayor parte del tiempo en entender el código fuente (*lenguaje controlado*) y no en ejecutar la nueva implementación o la solución.

La dificultad del entendimiento del código fuente se sigue presentando a pesar de las métricas actuales, las facilidades que presentan los IDE (*entorno de desarrollo integrado* por sus siglas en inglés) y diferentes prácticas que se crean para tratar de evitarlo. Adicionalmente, Siegmund (2016) enuncia que, a pesar de que los lenguajes cada vez se asemejan más a expresiones naturales, el impacto sobre el tiempo de comprensión de los sistemas de software sigue siendo un factor importante en la fase de mantenimiento.

Este comportamiento muestra que, cuando los términos usados en el requisito funcional no son consistentes con el código fuente, no significa de manera estricta que el requisito funcional no se codifique técnicamente bien o que tenga errores de lógica. El análisis de sistemas de código abierto y las respuestas de desarrolladores que trabajan en sistemas de software que presentan estos problemas, muestran que normalmente los términos de los requisitos funcionales se encuentran como sinónimos, abreviaturas o incluso como incoherencias semánticas que, a pesar de ser difíciles de asociar con el requisito funcional, en la codificación hacen lo correcto. Sin embargo, si el código fuente es difícil de entender con la lectura directa y las herramientas de interpretación (como los modelos y demás documentación) son difíciles de usar como apoyo, por lo que puede tomar más tiempo realizar cualquier cambio.

Un ejemplo de las inconsistencias que se pueden presentar entre los términos de los modelos y el código fuente es el siguiente: si para el interesado, en el requisito funcional y en documentación existe una operación que se llama “Imprimir”, en el código fuente se espera exista algo igual. Sin embargo, esta operación se podría encontrar como “GenerarReporte”, “Imp”, “MyMethod”, “GR”, etc., lo que no hace que el sistema funcione mal, pero, para un desarrollador que retome este código fuente o haga mantenimiento basado en la documentación funcional, es más complejo y, por tanto, tarda más tiempo entenderlo. En la Figura 2-1 se expone lo que espera encontrar una persona que realiza mantenimiento al código fuente que tiene consistencia lingüística con los modelos y en la Figura 2-2 se muestra una situación común donde el código fuente tiene baja consistencia lingüística con los modelos.

Figura 2-1: Código fuente consistente con los modelos y el interesado.

Fuente: Elaboración propia.

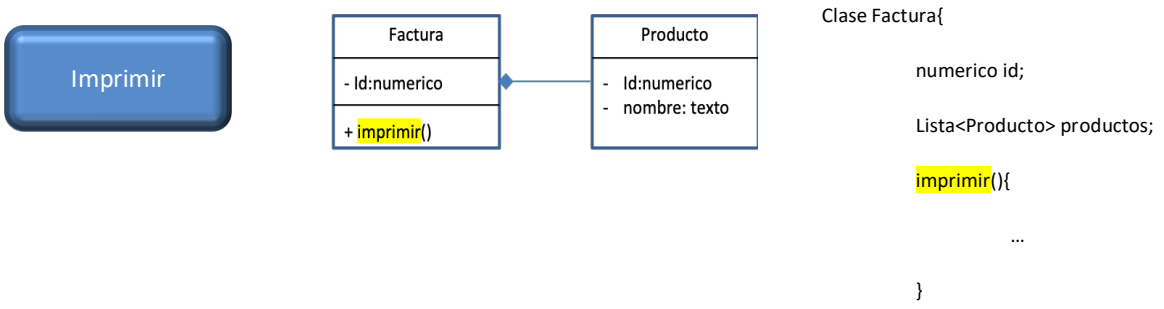
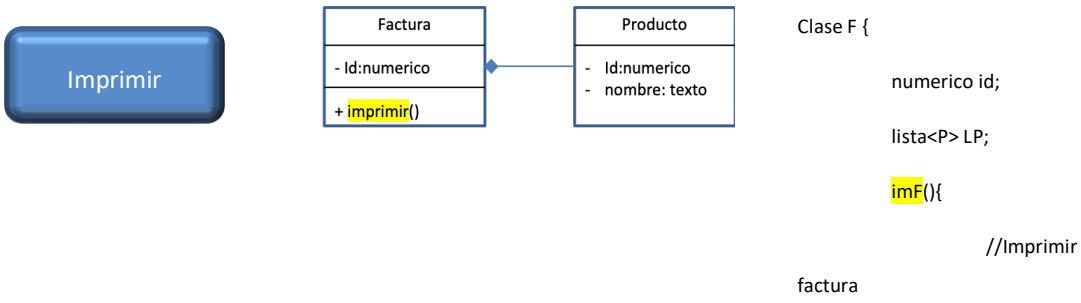


Figura 2-2: Código fuente poco consistente con los modelos y el interesado.

Fuente: Elaboración propia.



Según Zapata y Manrique (2014), el código fuente como lenguaje controlado y, por ende, subconjunto del lenguaje natural que se usa durante la educción de requisitos, se limita a las expresiones que existen en el lenguaje natural. Este lenguaje se usa con el objetivo de delimitar sus expresiones, evitar malas interpretaciones y ambigüedades para que un compilador lo ejecute sin fallas. No obstante, el uso de algunas prácticas permite desviar el lenguaje controlado de manera que el código fuente sí compile, se ejecute correctamente

y cumpla con las métricas de calidad existentes. Sin embargo, comprenderlo a la luz del universo de palabras que hay en los modelos es difícil porque no concuerdan.

Siegmund (2016) muestra cómo en sus inicios en los sistemas de software se presta más atención a la consistencia y coherencia desde su definición inicial, de manera que analizarlos y sostenerlos es más fácil. Sin embargo, cuando cita a Boysen (1977) y Sackman *et al*. (1968), enuncia que hay factores adicionales al entendimiento que afectan el funcionamiento y el nombramiento de los elementos a lo largo del programa, pero esas prácticas se justifican por el momento tecnológico de estos estudios.

Para la mantenibilidad del código fuente existen factores diferentes al nombramiento de los elementos del sistema de software y características adicionales que se deben tener en cuenta según su relevancia, las cuales varían para cada sistema. Siegmund (2016) expone que hay sistemas de software con limitaciones que son de alta relevancia porque afectan su correcto funcionamiento y estas limitaciones pueden primar sobre características de mantenibilidad del software. Limitaciones sistémicas como la memoria física disponible, el procesamiento y la memoria volátil, entre otras, son características no funcionales que impactan el orden de prioridades para evaluar la calidad del sistema de software, ante lo cual el nombramiento del código fuente puede perder relevancia. Las limitantes que se mencionan y que Siegmund (2016) expone, hoy se tienen en menor escala, como el espacio en disco que ocupa el nombre de una variable, un atributo o una operación, la capacidad del compilador y los métodos de desarrollo, que permiten hacer el código más comprensible posible sin consecuencias no funcionales importantes. Sin embargo, en los inicios del desarrollo de software o en sistemas con las limitaciones mencionadas, un nombre de variable corto aporta para la optimización del sistema, ahorrando almacenamiento y procesamiento.

Los problemas que trae no poder comprender el código fuente en la fase de mantenimiento son de alto impacto para el sistema de software y, por esto, se estudian diferentes perspectivas del problema para entregar pautas y reglas que permitan mitigar el problema.

Etzkorn y Delugach (2000) exponen la relevancia que tienen la coherencia, comprensión y cohesión del código fuente para poderlo leer de manera mas sencilla y facilitar el entendimiento. Sin embargo, la comprensión y coherencia la abordan con base en el código fuente como producto de trabajo aislado; se busca que sea consistente en las líneas de código pero no con el resto de los productos de trabajo del sistema de software.

En esta misma línea y partiendo de que las limitantes mencionadas ya no aplican en muchos casos, Allamanis *et al.* (2014) exponen que el código fuente debe tener sentido cuando se lee para facilitar su entendimiento, el cual se relaciona directamente con el buen nombramiento de los elementos y, a pesar de que no retoman productos de trabajo anteriores como los modelos para sus análisis sobre el nombramiento de los elementos del código fuente, sí son importantes para la presente Tesis de Maestría los argumentos de Etzkorn y Delugach (2000) y Allamanis *et al.* (2014) sobre la comprensión y coherencia del código fuente y, especialmente, el nombramiento como parte fundamental del entendimiento del sistema de software.

Wang *et al*. (2014) abordan la manera en que las prácticas tradicionales de desarrollo de software se transmiten entre generaciones, afectando las concepciones culturales y estableciendo preconceptos de nombramiento que hoy se hacen sin una justificación razonable, generando más problemas que beneficios al sistema de software. En este tipo de prácticas se puede convertir el nombre de una variable en algo inconsciente, como se refleja en las respuestas de algunas personas, al indagar sobre los motivos para nombrar los elementos del código fuente. Sin embargo lo que exponen Siegmund (2016), Etzkorn y Delugach (2000) y Allamanis *et al.* (2014) refleja que este desinterés, puede causar problemas posteriores de entendimiento con consecuencias importantes.

En esta misma línea, Siegmund (2016) y Wang *et al*. (2014) concluyen que las enseñanzas que recibe un desarrollador influencian directamente sus prácticas de nombramiento sobre el código fuente. Adicionalmente, si se tienen factores no funcionales que afecten el nombramiento, el desarrollador no puede enfocarse en el código fuente como elemento de comprensión del sistema, es decir, hay prácticas de nombramiento que son justificables

por deficiencias en factores como las cualidades sistémicas no funcionales; sin embargo, cuando estas prácticas de nombramiento las replican los desarrolladores que trabajan sobre entornos que no tienen estas limitantes, no existen las justificaciones y afectan la comprensión del código fuente.

Partiendo de lo anteriormente expuesto, se puede identificar que en el desarrollo de software se mezclan dos situaciones. Por un lado, los preconceptos de nombramiento que se tienen para el desarrollo de software según Wang *et al*. (2014) y Siegmund (2016), que en este momento son injustificados y, para efectos de legibilidad, se les llama en el presente marco "*malas prácticas*"[1] y, en específico, "*malas prácticas lingüísticas*". Por otro lado, están los problemas que surgen con el paso del tiempo y que se mencionan anteriormente, a las que se les da el nombre para efectos prácticos de "*problemas actuales*".

Un sistema de software no es algo estático y la evaluación de "*malas prácticas*" para que el código fuente se haga cumpliendo las recomendaciones del momento, no garantiza que no se deteriore o cambie su diagnóstico frente el cambio de criticidad impuesto a la evaluación de prácticas de desarrollo. Esto lo exponen Marcilio *et al.* (2019) mediante el uso de herramientas automatizadas, dónde cambiar la criticidad de las variables puede cambiar el resultado de calidad del sistema de software. Esta situación ocurre de manera cíclica y, por ende, si la construcción del sistema de software tiene "*malas prácticas*", posiblemente el sistema de software se deteriore si no se solucionan; esta situación es común según los resultados de Marcilio *et al.* (2019), cuando exponen que los

[1] Termino comúnmente usado en el campo del desarrollo de software para prácticas consideradas inadecuadas.

desarrolladores, a pesar de ser conscientes de que hay malas prácticas de desarrollo, no las solucionan por estar en el día a día o porque son *"menores"*.

En relación con la coherencia y la consistencia en los sistemas de software, existen pautas para que los productos de trabajo de una misma fase de un sistema de software sean consistentes y coherentes (diferentes partes del código fuente o consistencia entre diferentes modelos de un mismo sistema de software) como los expuestos anteriormente. Según Pastor y Molina (2007), la madurez de algunos modelos permite generar el código fuente de manera automática, lo que ayuda a la consistencia entre los modelos y el código fuente. Sin embargo, esta forma de trabajo aún no se adopta completamente y la gran mayoría de desarrollos de sistemas de software se ejecutan de manera manual, o con automatizaciones que cuentan con inconsistencias en los nombramientos, dejando así posibles problemas para la comprensión del código fuente en la fase de mantenimiento.

Por todo lo anterior, se hace una búsqueda de las diferentes métricas y herramientas asociadas con la comprensión del código fuente y la consistencia en los sistemas de software, de manera que el problema del entendimiento del sistema de software se aborda desde un punto de vista pragmático, donde el desarrollo del sistema de software se toma como un proceso y se mide la distancia que existe entre los diferentes productos de trabajo, a fin de identificar donde está la perdida del entendimiento por el cambio en la terminología y qué otros estudios aportan en la construcción de una métrica que ayude al entendimiento del sistema de software como proceso en la fase de mantenimiento.

3. Antecedentes

Para el desarrollo de software, a medida que crece la complejidad y tamaño de los sistemas, también crecen las variables que pueden causar problemas o implicar riesgo, como es el caso del entendimiento del código fuente. Por este motivo, en la presente Tesis de Maestría se realiza la búsqueda y análisis de diferentes métricas de calidad para que el código fuente sea más legible y comprensible, las mejoras que se presentan en los IDE (*entorno de desarrollo integrado* por sus siglas en inglés), las características de los procesos de desarrollo de sistemas de software y se analizan los cambios en los métodos de desarrollo, a fin de crear una base de conocimiento suficiente para la construcción de una nueva métrica que permita medir la distancia entre los requisitos funcionales y el código fuente.

Para identificar las acciones que hacen al sistema de software fácil de mantener, se parte de definiciones que circulan en la comunidad científica y otras veces se adaptan de manera empírica. Rosenberg y Sheppard (1994), cuando relacionan métricas de diferente índole durante el desarrollo de software, como la complejidad ciclomática de McCabe, la complejidad ciclomática extendida y el porcentaje de comentarios, entre otras, buscan métricas asociadas con el código fuente y con la codificación compleja que se puede presentar en porciones referentes a la estructura, es decir, miden la cantidad de ciclos o condiciones anidadas en exceso, para, en lo posible, evitarlas y que no se dificulte la lectura del código fuente desde un punto de vista técnico y sintáctico.

Rosenberg y Sheppard (1994) analizan métricas comúnmente usadas para mejorar el entendimiento del código fuente. Sin embargo, la adopción de estas métricas no garantiza

que el código fuente haga lo que expresa el interesado o que esté acorde con los modelos, pues en las métricas que analizan Rosenberg y Sheppard (1994) el único insumo necesario es el código fuente. Esto deja de lado la premisa de esta Tesis de Maestría de mejorar el entendimiento del código fuente basado en la documentación. Por otro lado, estas métricas se basan en la coherencia en el código fuente, pero no en la consistencia a lo largo del proceso de construcción de un sistema de software.

Avidan y Feitelson (2017) realizan un experimento controlado con profesionales pertenecientes al área del desarrollo de software. En tal experimento, se presenta el código fuente original de un sistema de software a un grupo de personas y luego el mismo código pero con variaciones en los nombres de los identificadores, de manera que éste no sea comprensible solo con lectura. Con esto, de manera experimental, muestran cómo los nombres de las variables son significativos para la comprensión de una aplicación y muestran que hay puntos de entendimiento con más importancia que otros, por ejemplo; los nombres de los parámetros son más relevantes que los de las variables a la hora de la lectura del código fuente. Avidan y Feitelson (2017) se basan en muestras pequeñas para acotar el estudio semántico de código fuente, a fin de mejorar su entendimiento, y concluyen que tienen mayor relevancia los nombres de los objetos, atributos y métodos que las variables locales o su manipulación, lo que orienta los casos a usar en la construcción de una nueva métrica, de forma que su uso tenga mayor impacto.

Por otro lado, Etzkorn y Delugach (2000) buscan mantener coherencia en el código, medir su grado de comprensión, cohesión, complejidad y la calidad de documentación de una clase, entre otros. Avidan y Feitelson (2017) debaten sobre el nombramiento de las clases, las variables, los atributos y los métodos del código fuente, pero sin tener en cuenta el papel de la consistencia lingüística entre los modelos y el código fuente, dejando prácticas pendientes por medir en los procesos de evaluación de calidad del software, que puedan garantizar que lo que expresa el interesado se comprenda de igual forma en el código fuente del sistema de software.

Según Binkley *et al.* (2011) se deben usar las palabras adecuadas para el nombramiento del código fuente, donde los identificadores deben ser palabras completas o incluso abreviaturas, porque conducen a una mejor comprensión que los identificadores compuestos de letras simples. Bajo esta misma línea, Blinman y Cockburn (2005), mediante un experimento para identificar cuánto recuerdan las personas al leer un código fuente alterando la estructura de sus nombres, concluyen que los nombres de las variables deben ser descriptivos, porque ayudan a reducir la complejidad del programa y recordar el código fuente.

Avidan y Feitelson (2017), Binkley *et al.* (2011), Blinman y Cockburn (2005) y Rosenberg y Sheppard (1994) realizan sus análisis sobre sistemas de software particulares y bajo condiciones especificas, con lo cual, a pesar de encontrar resultados concretos, brindan un espectro de resultados amplio y dejan detalles por fuera. Estos autores no dan relevancia a la relación entre el modelado y su implementación para medir sus diferencias lingüísticas y, específicamente, cómo afecta esto la comprensión del código fuente.

Asimismo, estos estudios y los de Etzkorn y Delugach (2000) se apoyan en la experimentación, el uso de herramientas del mercado para evaluar la calidad de nombramiento del software, el análisis de código fuente y la comparación de sintaxis y la semántica en aras de mejorar la comprensión del código fuente. Esto muestra lo diverso de los estudios y la problemática que sigue siendo la calidad del código fuente y el tiempo que lleva su comprensión en las fases avanzadas y de mantenimiento.

Además de las prácticas mencionadas, existen otras orientadas a la optimización de los recursos y el buen uso del lenguaje de programación, las cuales aportan a la calidad del sistema de software. Sin embargo, la métrica propuesta en esta Tesis de Maestría se enfoca en las características semánticas. Allamanis *et al.* (2014) enuncian algunas que se orientan al entendimiento de un sistema de software, sugiriendo que la semántica del código fuente se liga con la comprensión, debe ser coherente y tener sentido cuando se lee. Así, el entendimiento se liga con el buen nombramiento de los elementos del código

fuente en los sistemas de software. Esto permite acotar el problema que se analiza en la presente Tesis de Maestría de las "*malas prácticas lingüísticas*" a un ámbito más específico como "*malas prácticas de nombramiento*".

En la Tabla 3-1 se hace un comparativo entre las características que se presentan en los estudios de Avidan y Feitelson (2017); Binkley *et al*. (2011); Blinman y Cockburn (2005); Rosenberg y Sheppard (1994) con la necesidad expuesta en esta Tesis te Maestría de evaluar la consistencia durante la construcción de un sistema de software.

Tabla 3-1: Resumen comparativo entre la temática de la Tesis de Maestría y algunas métricas de calidad de software.

Fuente: Elaboración propia.

Cualidad	**Características Avidan y Feitelson (2017); Binkley *et al.* (2011); Blinman y Cockburn (2005); Rosenberg y Sheppard (1994)**		
	Semejanzas	**Diferencias**	**Aportes**
Comprensión	Se estudia el cambio en los nombres de los identificadores. Se evalúa el uso de palabras adecuadas. Se mide la comprensión del código fuente.	Se toma el código fuente como un elemento aislado. No se busca consistencia con los modelos y demás elementos del sistema de software.	Se encuentran los identificadores que contribuyen más a la comprension. Se sugieren las palabras adecuadas para el código fuente. Se exponen qué partes del código fuente son más importantes para comprender el sistema de software.
Complejidad	Se concluye que los nombres descriptivos facilitan la lectura y la recordación del código fuente. Se sugieren prácticas de nombramiento para codificar.	Las sugerencias de los nombres se basan en la estructura del código fuente y no en los modelos. Las prácticas sintácticas y de nombramiento sólo se basan en la lectura aislada del código fuente.	Se explica una forma de medir: cuánto se recuerda del código fuente al leerlo. Se exponen qué palabras del código fuente aportan más al entendimiento del sistema de software. Se aportan conclusiones basadas en experimentación.

Para el control de estas situaciones, en el mercado existen herramientas ASAT (*Automatic Static Analysis Tools*) que permiten automatizar la evaluación de estas "*malas prácticas*". Marcilio *et al.* (2019) exponen cómo el uso de estas herramientas ayuda al desarrollador a estar informado de manera ágil y continua de sus prácticas sobre el código fuente, pero puede desecharlas o darles poca relevancia según la criticidad que se les asigna. Además, estas herramientas no permiten generar análisis del sistema de software como proceso. Por el contrario, como su nombre lo indica, son herramientas de análisis de código estático para evaluar el código fuente como componente independiente del proceso. A pesar de que se logran mejores ejecuciones y prácticas de programación, no se garantiza que se desarrolle lo que espera el usuario y tampoco que se desarrolle en los términos de la documentación para facilitar el entendimiento en procesos posteriores.

En las Figuras 3-1 y 3-2 se muestra cómo en la herramienta ASAT (*Automatic Static Analysis Tools*) SONAR se exponen los resultados del análisis de código fuente con su criticidad, exponiendo cómo en un sistema de software, específicamente su código fuente, es de alta relevancia que sea comprensible para ser posteriormente de fácil mantenimiento.

Figura 3-1: Mediciones herramienta SONAR

Fuente: https://www.javiergarzas.com

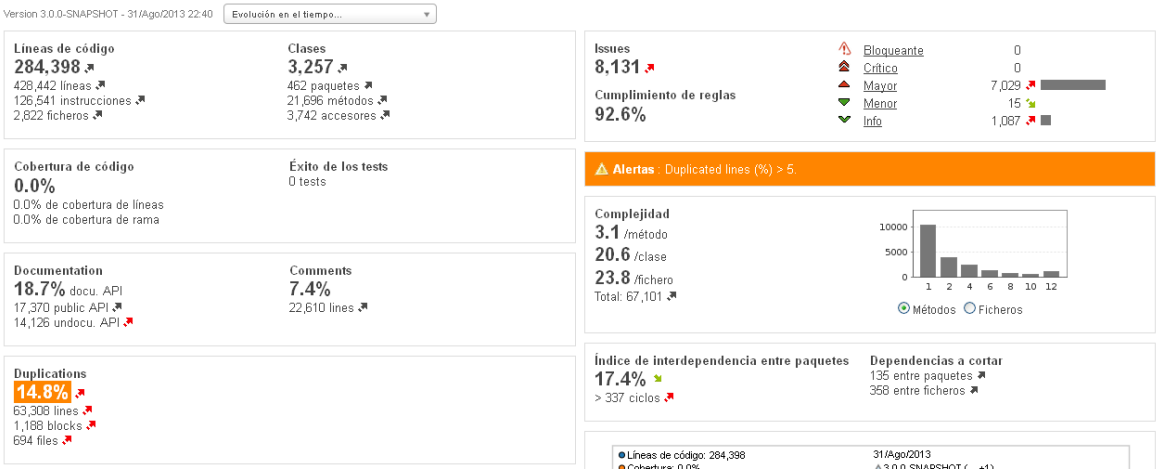
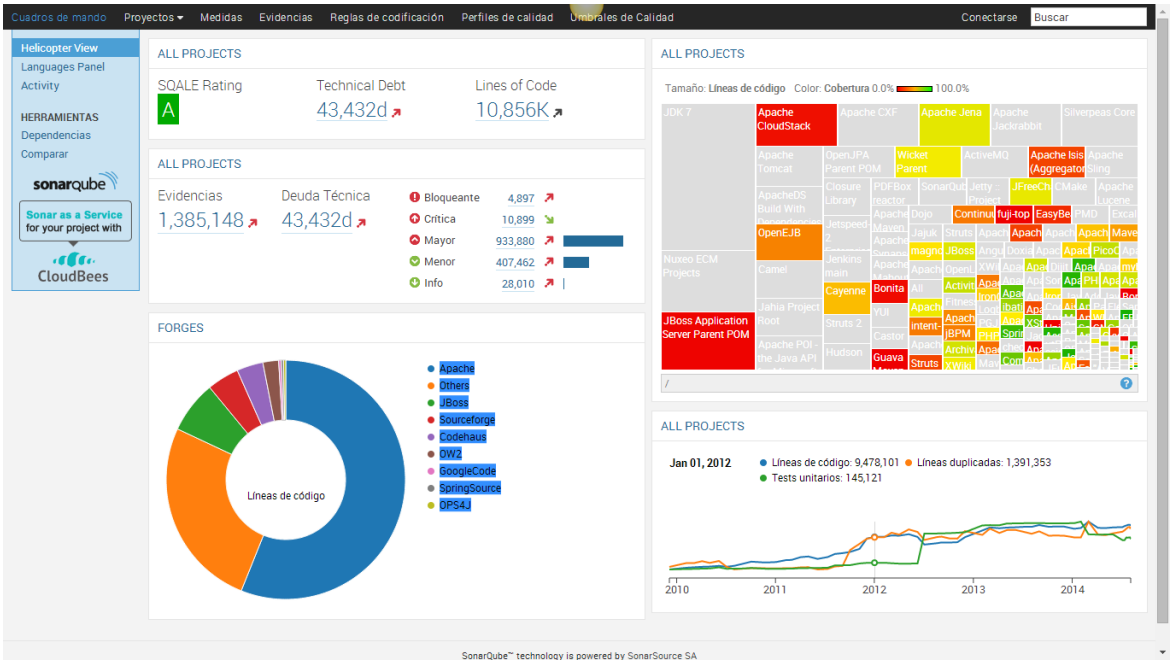


Figura 3-2: Estadística de resultados de evaluación con SONAR

Fuente: https://testeandosoftware.com/



Del mismo modo, Allamanis *et al*. (2014) hacen referencia al framework "*Naturalize*", el cual incluye sugerencias automatizadas de nombramiento basadas en patrones y permite generar estándares para el nombramiento de los elementos del código fuente de un sistema de software, pero no se basa en los requisitos funcionales. De esta manera, no se profundiza en la medición de la consistencia lingüística entre el código fuente y los modelos como un asunto relevante. Con el uso de esta herramienta se pretende que el código fuente sea más legible para evitar confusiones, pero no se garantiza que el sistema de software sea consistente desde la expresión del interesado hasta su compilación. En la herramienta "*Naturalize*", las sugerencias se basan en la consistencia del código fuente, datos que tiene por defecto y modelos de lenguaje que se le pueden inyectar como código base. Estas porciones de código se usan como insumo de la herramienta a fin de hacer natural y consistente el código en su estructura. Sin embargo, en la herramienta no se usa un modelo del diseño para garantizar la consistencia con el resto del sistema de software. En las Figuras 3-3 y 3-4 se ilustra el funcionamiento de "*Naturalize*", la forma en que se

sugiere el nombramiento para diferentes porciones de código fuente con base en las características del lenguaje de programación *Java* y los porcentajes de las palabras para mejorar más el entendimiento del código fuente al leerlo. Todo esto se realiza sin tomar en cuenta los modelos.

Figura 3-3: Uso de la herramienta "*Naturalize*" para nombrar el parámetro m

Fuente: Página oficial http://groups.inf.ed.ac.uk/naturalize

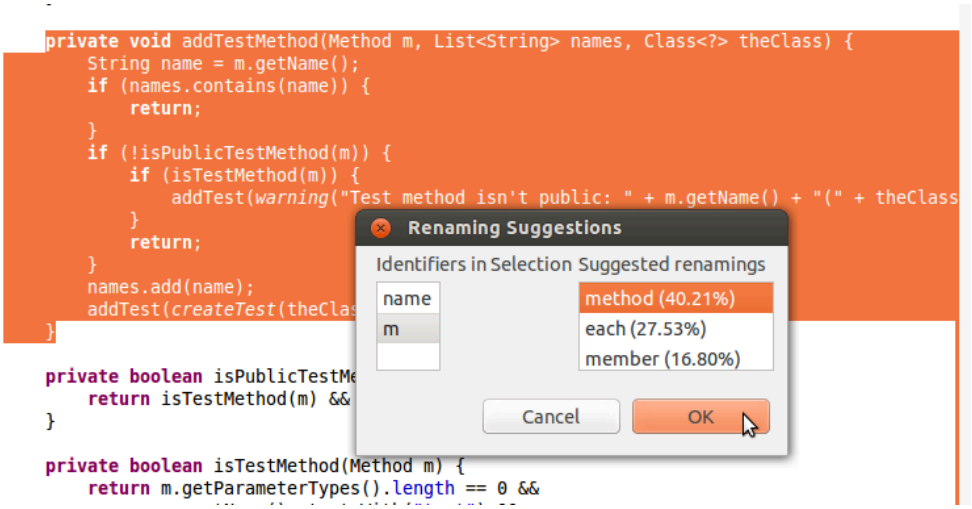
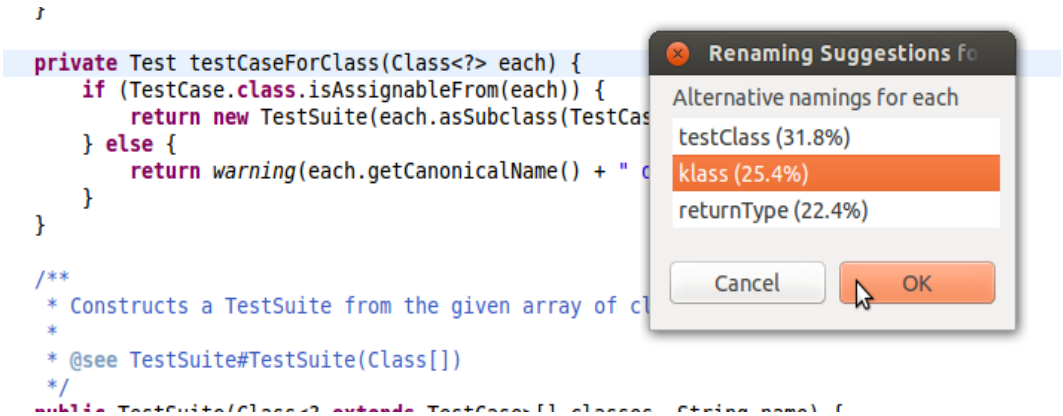


Figura 3-4: Uso de la herramienta "*Naturalize*" para nombrar el parámetro each

Fuente: Página oficial http://groups.inf.ed.ac.uk/naturalize



A manera de resumen, se exponen en la Tabla 3-2 el comparativo de las herramientas expuestas "*Naturalize*" y las herramientas ASAT (*Automatic Static Analytics tools*) con la temática de la presente Tesis de Maestría.

Tabla 3-2: Resumen comparativo entre la temática de la Tesis de Maestría y algunas herramientas para evaluar la calidad de software.

Fuente: Elaboración propia.

Herramienta	**Características herramientas para la calidad del software**		
	Semejanzas	**Diferencias**	**Aportes**
Naturalize	Se alerta sobre mejoras al código fuente en nombramiento. Se busca la consistencia en el código fuente. Se evalúa de manera cuantitativa cuál es mejor nombre en el código fuente.	No se basa en los modelos para dar las sugerencias de nombramiento.	Sugerencias de nombramiento basadas en patrones.
ASAT (*Automatic Static Analytics tools*)	Se alerta sobre la comprensión del código basado en su complejidad. Se mide la calidad del código teniendo en cuenta su mantenibilidad. Se permite evaluar por criticidad para enfocarse en lo más relevante.	No se busca la consistencia del código fuente ni se analiza de manera semántica. Se toma el código fuente como un trozo aislado del sistema de software. No se usan complementos al código fuente como los modelos.	Se expone cómo, a pesar del alertamiento, muchos problemas no se resuelven. Se resalta la importancia del alertamiento temprano.

Las métricas sintácticas de calidad de software orientadas hacia las buenas prácticas del un lenguaje de programación o al método de desarrollo buscan la medición de parámetros como la forma de escritura del código fuente. Allamanis *et al*. (2014) mencionan que cada desarrollador tiene su estilo característico de programación, preferencias sobre el nombramiento, maneras de relacionar los objetos y patrones de diseño para la codificación.

Tiarks. (2011), Mayrhauser *et al*. (1997) y Siegmund (2016) mencionan este tipo de características sintácticas como puntos a estandarizar, a fin de tener un estilo consistente que facilite la legibilidad y el mantenimiento del código fuente. Sin embargo, a pesar de tener estas prácticas, en algunos casos se requiere la adición de comentarios en su código fuente porque se considera poco legible. La lectura del código fuente con comentarios lleva a pensar que se construye bien, pero no necesariamente es así; en las conversaciones sostenidas con desarrolladores que trabajan en sistemas de software que tienen problemas de entendimiento y aportes de las comunidades de desarrollo de software, se expone que el código fuente se puede comentar excesivamente (métrica que también se controla en las herramientas ASAT) pero mal construido de fondo y, adicionalmente, los comentarios no aparecen en los modelos como documentación necesaria para el entendimiento del sistema. Por este motivo, para esta Tesis de Maestría los comentarios en el código fuente no se toman en cuenta a la hora de evaluar su consistencia con los modelos.

Etzkorn y Delugach (2000) analizan una serie de métricas para la calidad de los sistemas de software orientadas a evaluar el tamaño, la complejidad y la legibilidad, usando para cada cualidad métricas que permitan identificar los aportes al entendimiento del código fuente. Para el tamaño usan métricas como *el total de líneas de código* y *las declaraciones ejecutables* (*medidas con delimitadores*), para la complejidad usan *la complejidad ciclomática de McCabe* (número de rutas lógicas), *complejidad ciclomática extendida* y *porcentaje de GOTO* y para la legibilidad, la *longitud promedio de los nombres de las variables* de los sistemas de software.

Por lo amplio del estudio y la variedad de métricas que analizan Etzkorn y Delugach (2000), en la Tabla 3-3 se comparan las métricas mencionadas con las características que tiene la temática de esta Tesis de Maestría, exhibiendo semejanzas y diferencias según la cualidad que representan las diferentes métricas de su estudio. Algunas de estas métricas se acoplan con las herramientas ASAT (*Automatic Static Analysis Tools*) expuestas previamente y, a pesar de los beneficios que presentan, Marcilio *et al*. (2019) Anotan que

las sugerencias que arrojan estas herramientas a veces las ignoran los desarrolladores o las llevan a un segundo plano, por que su impacto se asocia con la mantenibilidad y no con el correcto funcionamiento del sistema de software.

Tabla 3-3: Comparativo de cualidades de las métricas de calidad de software que analizan Etzkorn y Delugach (2000).

Fuente: Elaboración propia basado en Etzkorn y Delugach (2000).

Cualidad	**Características Etzkorn y Delugach (2000)**		
	Semejanzas	**Diferencias**	**Aportes**
Tamaño	Se estudian mediciones por volumen sintáctico como: el recuento de líneas o el tamaño de archivos y de comentarios.	Las métricas estudiadas son de orden sintáctico y no semántico.	Se muestra cómo las medidas de la sintaxis se pueden asociar con la semántica.
Complejidad	Se evalúa la estructura lógica de los módulos de código individuales. Se evalúa la complejidad del código a nivel de ejecución, por ejemplo con su complejidad ciclomática extendida. Algunas métricas expuestas se usan para determinar la comprensión de la lógica del código fuente.	Se busca caracterizar el código fuente como un bloque lógico independiente de los requisitos.	Se identifica que la difícil lectura del código fuente ya tiene algunas formas de medición.
Legibilidad	Se incluye el porcentaje de comentarios (las líneas que contienen comentarios / el total de líneas de código). Se identifican cambios en las métricas durante el desarrollo y en el momento del lanzamiento.	Las métricas analizadas son semánticas, pero no permiten relacionar la legibilidad del código fuente con los requisitos funcionales.	Se expone cómo la legibilidad del código fuente no depende de lo extenso que sea o de los comentarios que tenga.

Durante el análisis de las Tablas 3-1, 3-2 y 3-3 se destaca que la comprensión del código fuente como consecuencia de los requisitos funcionales y las métricas de calidad es un problema de los sistemas de software y una preocupación de la comunidad científica, pero los diferentes estudios no garantizan aún que lo que está en los modelos se encuentre de manera consistente en el código fuente.

También, se encuentra asociado el entendimiento de un sistema de software con su estilo de construcción, Según Blinman y Cockburn (2005), dependiendo de la arquitectura, si la construcción del sistema de software es de tipo *bottom-up* como lo describe Shneidermann (1977), su entendimiento se impulsa con los datos que tiene el programador; es decir, el desarrollador construye con base en su conocimiento sintáctico, leyendo y comprendiendo el sistema de software línea por línea. Por otro lado, si el sistema de software se construye *Top-Down*, Brooks (1983) espera que el programador no lea un sistema de software línea por línea para comprenderlo sino que lo entienda por bloques de funcionalidad; en otras palabras, el sistema de software se entiende de manera global.

Como análisis de estos tipos de arquitectura, se expone en la Tabla 3-4 la relación que presenta la temática de la presente Tesis de Maestría con las arquitecturas que comparan Blinman y Cockburn (2005).

Tabla 3-4: Comparativo de cualidades de las arquitecturas que exponen Blinman y Cockburn (2005).

Fuente: Elaboración propia.

Herramienta	Características arquitecturas expuestas por Blinman y Cockburn (2005)		
	Semejanzas	**Diferencias**	**Aportes**
***bottom-up,* (Shneidermann, 1977)**	El programador interpreta el código fuente línea por línea.	El programador usa para la interpretación del código fuente principalmente la sintaxis.	El código fuente debe ser comprensible linea por linea.
***Top-down,* (Brooks, 1983)**	El código fuente es un conjunto que se interpreta con funcionalidades.	El programador no debe interpretar el sistema de software leyendo linea por linea.	El sistema de software es un grupo de funcionalidades que se entienden descentralizadas.

En algunos métodos de trabajo también se menciona la importancia del trabajo conjunto entre el desarrollador y las personas encargadas del área funcional. En textos como el manifiesto ágil se enuncia que *los responsables de negocio y los desarrolladores deben trabajar juntos de forma cotidiana durante todo el proyecto*. Sin embargo, esto no garantiza la consistencia en el nombramiento durante el desarrollo o que el código fuente sea entendible en la fase de mantenimiento. A medida que los proyectos avanzan, es posible que el equipo cambie; por ello, de no tener consistencia durante su construcción, el problema del entendimiento del código fuente basado en los modelos para los sistemas de software en la fase de mantenimiento no se mitiga.

Luego de tomar como antecedentes los análisis prácticos, herramientas y métricas investigadas y a pesar de existir diferentes estudios orientados a la semántica e incluso al nombramiento del código fuente, en los diferentes contextos que se aborda el problema no se encuentra un caso puntual donde se trate de medir la consistencia en la lingüística del código fuente de un sistema de software contra sus modelos. A pesar de que se

reconoce la importancia de la documentación, no hay métricas que permitan evaluar cuánto de los modelos está en el código fuente ni de qué manera y aún con la existencia de diferentes métricas sobre la comprensión de código fuente, entenderlo sigue siendo un proceso que tarda más tiempo del esperado en la fase de mantenimiento.

Por todo lo anterior, se exhibe la necesidad de crear una métrica de calidad de software comparando los diferentes productos de trabajo de un sistema de software, cuantificando su consistencia lingüística y evaluando la distancia semántica que hay entre los términos. Así, se ligan más al desarrollo los insumos adicionales al código fuente, se incentiva la actualización de la documentación y se logra que el sistema de software sea más consistente y acoplado a lo largo de todas sus fases, facilitando el entendimiento del código fuente en la fase de mantenimiento con el uso de los modelos.

4. Propuesta de solución: Métrica de consistencia lingüística

Para abordar la problemática del difícil entendimiento del código fuente en la fase de mantenimiento, luego de lo encontrado en los antecedentes, se descubre la necesidad de identificar las prácticas de nombramiento del código fuente y cómo afectan el entendimiento del sistema. Luego, si hay diferencias entre las palabras usadas en el nombramiento de los elementos en el código fuente y los modelos, se establece la distancia semántica mediante una fórmula que permita evaluar cuantitativamente la calidad semántica de un sistema de software basada en la terminología, se prueba en sistemas controlados y, finalmente, se construye una *métrica de consistencia lingüística* que permita medir la consistencia lingüística entre el código fuente y los requisitos funcionales.

En esta Tesis de Maestría, para abordar los problemas encontrados y aportar a los vacíos que quedan en el entendimiento de los sistemas de software luego de aplicar las métricas expuestas en los antecedentes, se propone la *métrica de consistencia lingüística*, en la cual se relacionan dos productos de trabajo del sistema de software a fin de mejorar la consistencia entre los requisitos funcionales y el código fuente.

La métrica se crea a partir de autores como Briand *et al.* (1997), que validan métricas de cohesión o superposición y se basan en propiedades para evaluar de manera teórica una métrica. Según este método, a pesar de que se hace una evaluación intuitiva y rigurosa, el cumplimiento de las propiedades que se encuentran en la teoría no es suficiente para decir que una métrica es válida o no, sino que el fallo de una propiedad sirve para concluir

que una métrica se define mal. Las recomendaciones de Briand *et al.* (1997), llevan a que la *métrica de consistencia lingüística* propuesta en esta Tesis de Maestría se construya de manera que sea cuantificable y predictiva.

Según Weyuker (1988) se debe tener una relación sintáctica expresada mediante reglas que permitan descartar un mal uso de la métrica. Las reglas de Weyuker (1988), a pesar de ser orientadas a la sintaxis, permiten determinar que la creación de la *métrica de consistencia lingüística* debe cubrir todo el universo de palabras que se puedan usar en un sistema de software sean coherentes o no, de manera que, al efectuar la validación completa esta métrica, no deje vacíos en su medición.

Briand *et al.* (1997) y Weyuker (1988) exponen cualidades que debe cumplir una métrica de calidad de software, de las cuales se concluye que la métrica propuesta en la presente Tesis de Maestría, debe ser cuantificable, predictiva y completa. Por ello, se busca acotar el alcance, los productos de trabajo y la forma de medir la consistencia lingüística en un sistema de software. Para que la métrica propuesta en esta Tesis de Maestría sea cuantitativa, se crea una fórmula que permita con mayor exactitud conocer la consistencia lingüística entre los requisitos funcionales y el código fuente, para que sea predictiva, la métrica se crea de forma que tenga un comportamiento predictivo y lógico, es decir, si el código se entiende o no debe ser consecuente con el resultado de la métrica y, para que sea completa, la métrica debe poder cubrir cualquier parte del sistema de software, siempre que cuente con los insumos necesarios. Sin embargo, es necesario delimitar el alcance de la métrica propuesta.

4.1 Alcance

Esta Tesis de Maestría se orienta bajo un carácter exploratorio descriptivo pues, para la construcción de la métrica propuesta, se parte de diferentes estudios teóricos sobre el

entendimiento y la relación de las palabras de manera semántica. Se tienen en cuenta comparaciones y métodos de desarrollo de software que regulan tipos de nombramiento al código fuente, así como análisis de algunos sistemas de código abierto para encontrar formas de nombramiento que aceptan las comunidades de desarrollo.

La métrica propuesta en esta Tesis de Maestría es de orden cuantitativo, pues la métrica de consistencia lingüística propuesta genera resultados bajo una unidad de medida porcentual, para evaluar la consistencia lingüística que tiene un sistema de software durante su ciclo de vida. Este tipo de unidad de medida exacta permite al evaluador encontrar qué palabras impactan el entendimiento, es decir, le permite medir cuánto cambia el entendimiento del sistema de software con el cambio en las palabras.

Además de crear una métrica que sea cuantificable, predictiva y completa, en la Tabla 1-1 se muestra la existencia de un universo de posibilidades muy amplio para el nombramiento de las variables. Por ello, se hace necesario acotar la métrica a premisas que hacen más precisos y controlables los resultados[2]:

- El idioma en que deben estar los modelos y el código fuente es inglés.

- La distancia entre las palabras es cero si y solo si, ambas palabras están iguales en el modelo y el código fuente. No importa el uso de mayúsculas y minúsculas, pero si afectan la adición, cambio o sustracción de caracteres.

[2] Que no se tengan estas premisas no implica que la métrica no se pueda hacer bajo la técnica posteriormente descrita, pero, los resultados pueden presentar alguna desviación.

- El modelo debe ser consecuente con la porción de código a probar, es decir, si se evalúan los nombres de una porción de código debe ser con su par en la documentación.

- Los modelos no deben tener inconsistencias en los términos que usa el usuario cuando expresa los requisitos funcionales.

4.2 Productos de trabajo para medir la consistencia lingüística

La métrica propuesta en esta Tesis de Maestría se puede usar para medir la consistencia lingüística entre dos productos de trabajo cualquiera que pertenezcan al ciclo de vida del sistema de software, incluso entre dos transformaciones del sistema de software dentro de una misma fase. Los términos que se usan en ambos casos para nombrar el mismo elemento deben ser iguales y, si no lo son, se puede medir su distancia.

Sin embargo, Para el alcance de esta Tesis de Maestría, el primer producto de trabajo seleccionado para la comparación es el código fuente, pues según Siegmund (2016), es el que más dificultad y gasto de tiempo representa cuando hay que entender un sistema de software en la fase de mantenimiento, además de ser el resultado final del esfuerzo de implementación del sistema de software. El segundo producto de trabajo para la comparación es el modelo, en el cual, según Zapata y Manrique (2014) se refina el problema y las ambigüedades y los múltiples nombramientos para un mismo elemento deben desaparecer.

Según Pastor y Molina (2007), estos dos productos de trabajo del sistema de software deben tener un nivel de consistencia tal que para ellos se tiene planteado que existan estrategias como MDD (*desarrollo dirigido por modelos* por sus siglas en inglés) y MDA (*arquitectura dirigida por modelos* por sus siglas en inglés), en los cuales la madurez de un modelo permite generar el código fuente de manera automática. El problema con la consistencia lingüística entre el modelo y el código fuente se presenta cuando no se hace de manera automática sino manual o el proceso automático no se hace de manera correcta. A pesar de que en este punto el código fuente se debe leer exactamente igual al modelo, esto no sucede en la práctica, como lo dice Siegmund (2016).

En complemento con lo anterior, de los antecedentes analizados en esta Tesis de Maestría, incluyendo las conversaciones sostenidas con los desarrolladores, se concluye que un factor relevante para que el código fuente no sea entendible a la luz de los modelos es el uso de palabras que no están presentes en los modelos y que desvían el entendimiento del sistema de software. Está práctica es la que se cuantifica con la métrica que se propone en esta Tesis de Maestría y se le da el nombre de *métrica de consistencia lingüística*.

Por todo lo anterior, a pesar de que la *métrica de consistencia lingüística* que se propone en esta Tesis de Maestría se puede usar para evaluar dos productos de trabajo cualquiera de un sistema de software, el modelo y el código fuente se usan como los dos productos de trabajo establecidos para comparar la consistencia lingüística en todos los ejemplos y durante toda esta Tesis de Maestría. Por ello, durante la solución propuesta, la *métrica de consistencia lingüística* es la cuantificación de la desviación del entendimiento entre un modelo y el código fuente debido al cambio en los términos usados.

La *métrica de consistencia lingüística* se puede utilizar para evaluar porciones de código fuente de un sistema de software que tiene documentación, o en el momento de la implementación. Con el uso de la métrica propuesta se pretende que el sistema de software se desarrolle con base en la documentación y que se pueda comprender el código fuente analizando los modelos. Así, se mejoran los tiempos de entendimiento de un

sistema de software ya construido y se aporta a su calidad semántica, pues es una forma estricta de evaluar que se refleje en el código fuente lo que expresa el interesado.

4.3 Proceso de selección de las porciones del sistema de software a evaluar

Sin importar el marco de trabajo usado, la estrategia para la educción de requisitos o la documentación, es indispensable tener el modelo y el código fuente del requisito funcional a evaluar. El modelo debe ser una abstracción fiel del requisito funcional y de referenciación directa a los nombres, por ejemplo, un esquema preconceptual como el que expone Zapata (2007) o un modelo UML como el de clases, con el fin de que la *métrica de consistencia lingüística* tenga los insumos necesarios. Como premisa de precisión para la métrica de consistencia lingüística, se parte de que el diagrama es 100% consistente con los requisitos y durante la educción de los mismos no se tuvo ninguna alteración a la expresión del interesado.

Los sistemas de software pueden ser muy complejos; por este motivo, se pueden evaluar porciones de ellos, siempre que se tenga el código fuente y los diagramas respectivos. Por ejemplo, si se toma una clase en particular del código fuente, se debe tener la contraparte de la documentación, como puede ser el diagrama de clases. Es importante que se tenga la documentación exacta de la porción de código a evaluar porque, si se toma un nivel muy abstracto, se pueden presentar cambios respecto del código fuente. Es decir, si el diagrama dice “*Factura*” ya que era una charla inicial con el interesado, pero el código fuente dice “Cuenta” se tendrá una inconsistencia. Se debe usar el último diagrama que usó el programador para implementar el código fuente para obtener los mejores resultados.

En el análisis de los antecedentes se encuentra que se pueden tener entre los modelos y el código fuente diferentes tipos de palabras que intentan representar lo mismo. En el mejor de los casos se tienen textos iguales, pero también se usan sinónimos, inconsistencias lingüísticas y abreviaturas, entre otros, que obligan a un emparejamiento manual de las palabras que se tienen en los diagramas (en lo posible de bajo nivel) y el código fuente del sistema de software. En esta propuesta de la *métrica de consistencia lingüística* es necesario un análisis de cada producto de trabajo o conocimiento del desarrollador para emparejar tales palabras.

Para medir la consistencia lingüística de un sistema de software se requieren los siguientes pasos:

- Seleccionar un requisito funcional representado mediante un modelo sin alterar ningún término (se asume que en este punto la consistencia lingüística entre las palabras del interesado y el modelo es 100%).

- Buscar la representación del requisito funcional seleccionado en el código fuente.

- Ubicar cada término en ambas partes (código fuente y modelo del requisito funcional), nombres de clases, variables, operaciones, etc.

- Tomar los términos pares del modelo y del código del sistema de software y evaluar la distancia semántica entre ambas palabras mediante la técnica sugerida, una mezcla del *análisis semántico latente* y una adaptación a la *distancia de Levenshtein* descritas en el numeral 4.4 de esta Tesis de Maestría.

- Convertir esta distancia semántica a una medida porcentual, donde el 100% es el uso de la misma palabra tanto en el requisito funcional como en el código fuente,

es decir, si la distancia semántica entre todos los pares de palabras es cero significa una consistencia lingüística del 100%.

- Sacar el promedio ponderado y presentarlo como resultado final de la métrica de consistencia lingüística para la porción seleccionada del sistema de software.

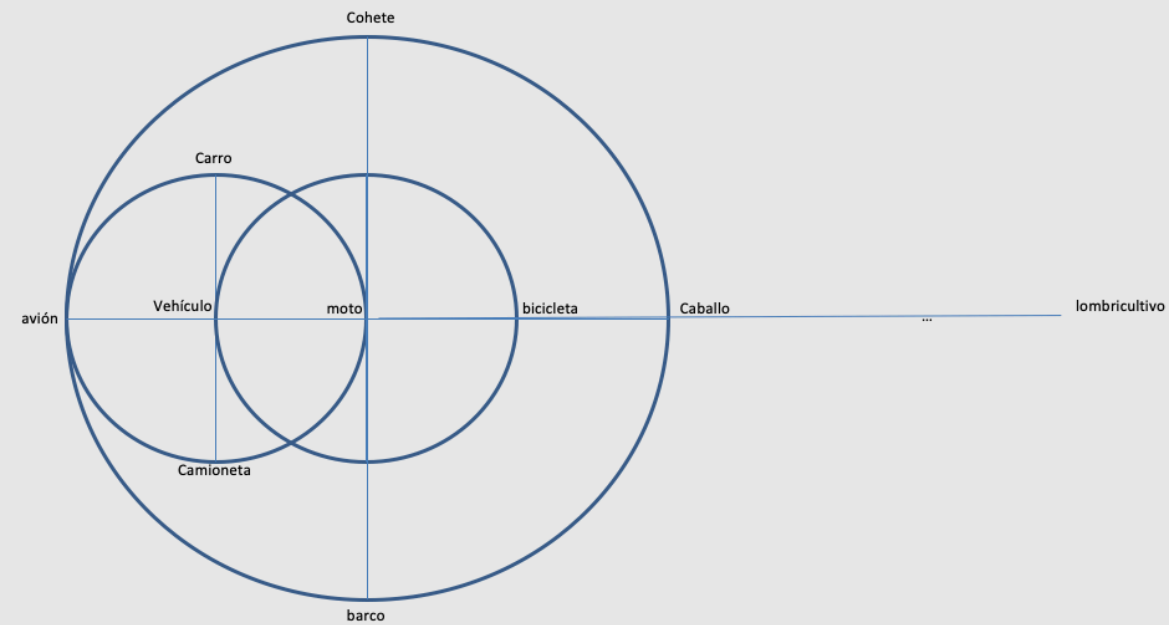
Con el proceso descrito, se puede evaluar la consistencia lingüística de cualquier fragmento de código fuente de un sistema de software, para el cual se tenga su respectivo modelo. Sin embargo, en este primer acercamiento el emparejamiento de las palabras es manual, porque es necesario el conocimiento de la persona para que indique qué porción del código fuente pertenece a su respectivo modelo y qué palabra del código fuente empareja con la palabra del modelo.

4.4 Cálculo de la distancia semántica

La distancia semántica se define en la literatura de diferentes maneras. Según Acosta (1989), la distancia semántica es un símil a un círculo con centro en el significado de la palabra y con algunas otras palabras alrededor a determinadas distancias (distancias semánticas). A su vez, cada palabra es un centro de otro círculo, formando una agrupación de campos que se distancian según tengan afinidades unas palabras con otras. Una representación de esto se plasma en la Figura 4-1 donde, a pesar de no ser una medida exacta, se expone cada palabra como centro y otras a distancias representativas según su relación. Acosta (1989) haciendo alusión a Lotmar (1919), establece que las parafasias pueden generar una "*distorsión en el hallazgo de la palabra*" o un "*descarrilamiento del significado"*. Por esto, cabe acatar que para el cálculo de la distancia de las palabras, las parafasias fonológicas y semánticas, a pesar de conservar relación con la palabra inicial, tienen impacto en el entendimiento.

Figura 4-1: Representación geométrica de la distancia entre las palabras.

Fuente: Elaboración propia basada en Acosta (1989)



Por el nivel de complejidad y de abstracción necesarios para medir la distancia semántica, para esta Tesis de Maestría se estudian varios métodos. En un principio, desde una base empírica se le otorga un porcentaje de distancia a cada palabra según si ésta es un sinónimo, una sigla o una abreviatura. Sin embargo, validando con Acosta (1989) se explica que los estudios se basan en encuestas poblacionales cerradas desde las cuales se saca una medida para la relación semántica. En estas encuestas, Acosta (1989) muestra que establecer de esta manera las distancias sesga la medida al contexto, de manera que, para las distancias semánticas, es relevante su contexto (pragmática); ante esta problemática, se buscan métodos que puedan resolver esta situación que no sean empíricos. Según Venegas (2003), es posible medir distancias entre palabras, frases, oraciones, párrafos grandes y textos identificando inclusive el contexto, siempre que se tenga una base de conocimiento amplia.

Por lo anterior, y por acotaciones como las de Gutiérrez (2005), que enuncia que en la psicolingüística el significado de cualquier pieza semántica es contextualmente dependiente, al hacer la búsqueda en los diferentes espacios de investigación, se tiene como primera opción en la métrica de consistencia lingüística el uso de LSA (*Análisis semántico latente* por sus siglas en inglés). Venegas (2003) expone el LSA para medir la distancia semántica entre estas palabras con sentido. Para las demás palabras, cuyo cambio no se pueda medir por su significado, se usa un estilo de medición basado en cambios básicos en las letras que explican Ene y Ene (2017) y es la *distancia de Levenshtein*, a la cual, por temas de cálculo se modifica como se expone en el numeral 4.4.2.

4.4.1 Análisis semántico latente (LSA)

El *Análisis semántico latente* (LSA), lo expone Venegas (2003) como una forma de medir la relación semántica entre piezas de texto de diferentes tamaños como palabras, párrafos o grandes textos. Para el caso de la *métrica de consistencia lingüística*, se utiliza sólo para distancia entre dos palabras y tiene como ventaja adicional que, para calcular la distancia semántica, considera el contexto. Esto se debe a que el algoritmo que incluye necesita un corpus de conocimiento que lo nutra.

Venegas (2003) expone que el LSA en sus inicios se consideró como una teoría y método de representación del conocimiento humano, como un modelo estadístico que permite comparar similitudes semánticas entre piezas de texto. Esta comparación se realiza en un espacio semántico multidimensional, generado a partir de un valor singular de descomposición (SVD), con el cual es posible determinar distancias entre palabras y párrafos o párrafos y párrafos. El LSA hace parte de la psicolingüística computacional apoyando en la generación del aprendizaje al ayudar a relacionar grandes textos.

Venegas (2003) y Gutiérrez (2005) también abordan el LSA como parte de la lingüística dentro del campo de la psicolingüística. En este campo, con el LSA se aborda el significado de las palabras como una representación humana para hacer predicción de juicios con base en la relación que tienen las palabras entre sí. Sin embargo, estas predicciones se basan en piezas de texto que pertenecen a un mismo dominio de conocimiento y, para su mejor comportamiento, estas piezas textuales deben ser tan robustas cómo sea posible. De esta manera se cuenta con más relaciones entre palabras dentro de un mismo contexto. Para ajustar el resultado a valores aceptables, se espera que el corpus de conocimiento a usar para establecer las relaciones del LSA sea amplio en el número de textos que lo conforman, pero restringido en cuanto al dominio de conocimiento que tienen estos textos. Se espera, también, que los textos sean en lenguaje natural y amplio en concurrencias de las palabras.

El LSA tiene algunas variaciones como el LSI (*indexación semántica latente* por sus siglas en inglés) que se usa para relacionar las palabras y se usa comúnmente en los buscadores web. Sin embargo para la métrica propuesta, el LSA usado en algunas *lexical database* es lo suficientemente robusto para cubrir lo esperado en cuanto a la relación que tienen las palabras con sentido.

Cumplir para cada caso a medir con las características anteriormente expuestas es complejo. Por este motivo, para los ejemplos prácticos que se quiere realizar, existen diferentes *lexical database* que ya incorporans corpora de conocimiento nutridos, prácticos y fáciles de usar como WordNet, BDOS (*Bi-Direction One Step*) o HS (*Derivación de la jerarquía* por sus siglas en inglés), lo que los hace una herramienta comparativa adecuada para validar lo argumentado en la Tesis de Maestría sin profundizar más de lo necesario en LSA o en una específica *lexical database*. Esta medida da un resultado porcentual de relación, por ejemplo, entre las palabras “*Car*” y “*Vehicle*” existe una relación del 92%.

Este tipo de comparación es adecuado cuando las palabras que se quiere comparar son completas y con sentido. Sin embargo, si se usa sólo el LSA, no se cumple con la premisa de cubrir todas las palabras del sistema de software, pues el LSA depende de un corpus de conocimiento nutrido y con procesamiento previo, el cual puede dejar por fuera algunas palabras. Esto se debe al carácter subjetivo de los corpora con que se nutre o a la falta de relación de una palabra con su abreviatura u otro tipo de representación que tenga el código fuente. Para estos casos, donde no es necesario ningún contexto o las palabras no son de utilidad sin explicación no se cubren con el LSA y, por tanto, es necesario usar un método que se base en cambios básicos de las palabras y permita medir la distancia entre cualquier tipo de cadenas de caracteres.

4.4.2 Distancia de Levenshtein

Para cubrir los casos de las palabras que no estén en la *Lexical database* o en los que el cálculo de la distancia de las palabras no depende de su entendimiento porque pueden no tener sentido, se usa una adaptación realizada a la *distancia de Levenshtein* como herramienta que no necesita contexto.

La *distancia de Levenshtein* es una medida que se basa en operaciones básicas de sustitución, eliminación y adición de letras para llegar de una cadena de caracteres a otra. Según Ene y Ene (2017), la *distancia de Levenshtein* o distancia de edición permite comparar las cadenas de caracteres sin importar si son varias palabras o si tienen caracteres especiales entre ellas, lo que la hace apropiada para los casos que deja por fuera el *análisis semántico latente*.

Como ejemplo del uso de la *distancia de Levenshtein* se tiene la comparación de dos palabras como “cama” y “cana”; en este caso se tiene la substitución de la “m” por la “n”,

lo que da una distancia de uno. Otro ejemplo se da entre las palabras “cama” y “calma”, con una adición de la letra “l”, lo que también entrega una *distancia de Levenshtein* de uno.

Como se puede observar, los resultados de los dos tipos de medición dan en unidades de medida diferentes, el uno porcentual y el otro en unidades básicas; por este motivo, para dar un resultado único y de más fácil lectura, a la *distancia de Levenshtein* se le hace una adaptación de manera que sus cálculos arrojen resultados porcentuales y concuerden con los del LSA para poderlos ponderar en una única fórmula.

4.4.2.1 Distancia de Levenshtein adaptada

Para la *distancia de Levenshtein*, cada cambio es una diferencia entre las cadenas de texto; por este motivo, si no hay cambios, quiere decir que la relación entre las palabras es del 100% (las palabras son iguales) y el máximo de cambios que puede sufrir una de las cadenas de caracteres para ser igual a la otra usando la *distancia de Levenshtein* depende del número de letras que tenga la cadena más larga. Según esto, la mayor diferencia que se puede presentar es que una cadena de caracteres no tenga ni una letra igual a la otra y así el cambio serán todas las letras y la *distancia de Levenshtein* será de las N letras de la cadena de caracteres más larga.

Para que este cambio se convierta en un porcentaje, se debe dividir el resultado de la *distancia de Levenshtein* entre el número de letras de la cadena de caracteres más larga y luego multiplicarlo por cien para que de un resultado porcentual de la distancia. Sin embargo, el LSA da un porcentaje de relación y no de distancia. Por esto, se toma el 100% que es la máxima relación menos el cálculo realizado anteriormente. De esta forma, el resultado de ambas medidas arroja el mismo tipo de medida porcentual, lo que hace que el cálculo de ambas se base en la relación de las palabras y no de la distancia entre ellas.

De este modo, la fórmula resultante de la adaptación a la *distancia de Levenshtein* se propone en la Ecuación (4.1):

$$100\% - \frac{\boldsymbol{Distancia\ de\ Levenshtein}}{\textbf{Número letras palabra mas laga}} * 100 = relación\ de\ Levenshtein \quad (4.1)$$

4.4.3 Fórmula para calcular la consistencia lingüística

Con el fin de sistematizar aún más la métrica, todos los pares de palabras se evalúan con los dos métodos expuestos, una *Lexical database* y la *distancia de Levenshtein* modificada como se propone; el mayor porcentaje de consistencia arrojado de los dos métodos se toma como porcentaje definitivo. Luego de tener el porcentaje de cada palabra, se calcula el promedio ponderado para obtener la métrica de consistencia lingüística. El valor obtenido representa el porcentaje de entendimiento que se tiene del código fuente a la luz de los requisitos funcionales con base en los cambios de la terminología. Por ejemplo, una consistencia del 90% indica que el código fuente se puede entender visualizando los modelos a nivel de términos, pero que un 10% del entendimiento se perdió y por tanto, será 10% más difícil entender el código fuente a la luz del modelo por el cambio en la terminología.

Para el cálculo de la métrica de consistencia lingüística se requiere la siguiente función:

Sean:

- “x”, “y” las dos palabras a relacionar.

- LSA(x,y): *Análisis Semántico Latente* entre las dos palabras “x”, “y”

- RL(x,y): *Relación de Levenshtein* (véase la Ecuación 4.1) entre las palabras “x”, “y”

Por cada palabra se calcula su relación semántica así:

La relación semántica entre dos palabras para un sistema de software según la métrica de consistencia lingüística se propone en la Ecuación 4.2:

$$RS(x,y) = \begin{cases} LSA(x,y), & LSA(x,y) \geq RL(x,y) \\ RL(x,y), & RL(x,y) > LSA(x,y) \end{cases} \quad (4.2)$$

El resultado final de la métrica de consistencia lingüística es el promedio de la evaluación de cada una de las relaciones semánticas de las palabras que están en el modelo y en el código fuente seleccionados como porción del sistema de software al que se le realizó la medición:

La fórmula para el cálculo de la métrica de consistencia lingüística para un sistema de software se propone en la Ecuación 4.3:

$$Consistencia\ ling\text{üística} = \frac{\sum RS(x,y)}{Total\ de\ pares\ de\ palabras\ evaluadas} \quad (4.3)$$

5. Validación

La validación de esta Tesis de Maestría se hace con dos productos de trabajo. El primero es un juego construido durante el proceso de la Tesis de Maestría, con el cual se evalúa de manera empírica la necesidad de crear una métrica que evalúe la consistencia lingüística entre los requisitos funcionales y el código fuente y el segundo un caso de laboratorio donde se explica de manera didáctica el uso de la métrica y se evalúan los resultados ante una prueba controlada.

5.1 Juego "*Nómbralos cerebro*"

Para evaluar de manera empírica la necesidad de crear una métrica de la consistencia lingüística entre los requisitos funcionales y el código fuente, se crea un juego, el cual se usa en diferentes etapas de la presente Tesis de Maestría. Las personas que participan en el juego compiten por ser quien aporta más entendimiento al código fuente respecto del modelo, cambiando únicamente las palabras del código fuente. El juego se crea con base en el juego Twister de Hasbro® según las reglas oficiales de la página https://products.hasbro.com/ en Abril del 2020, pero se presenta bajo una adaptación didáctica que permita comprender el uso de la métrica de consistencia lingüística haciendo uso de las recomendaciones y la plantilla de Gómez (2010).

5.1.1 Plantilla de presentación del juego basada en Gómez (2010)

Temática del juego:

Por medio de herramientas didácticas, presentar una métrica de consistencia lingüística para comprenderla y evaluar su medición.

Propósito del juego:

Reforzar la importancia del uso de la terminología en diferentes fases de la construcción de un sistema de software.

Enseñar cómo se usa la métrica de consistencia lingüística por medio de enfoques didácticos para medir la calidad semántica de un sistema de software.

Conceptos básicos:

Consistencia lingüística: Métrica de calidad a evaluar y con la que se mide la consistencia entre los términos usados en los requisitos funcionales y el código fuente.

Para el cálculo de la métrica de consistencia lingüística se mide la distancia semántica entre dos palabras, la usada en los requisitos funcionales y la que está en el código fuente. Los jugadores no tienen la necesidad de hacerlo de manera manual, pero para conocimiento de la métrica, ésta se calcula usando una mezcla de dos tipos de medición de distancia entre dos palabras; la relación que muestra la relación ontológica de una

Lexical database nutrida con su propio corpus (para el ejercicio es wordNet) y la *distancia de Levenshtein* modificada expuesta anteriormente en esta Tesis de Maestría.

Se toma la relación de mayor valor como la medida respectiva entre las dos palabras seleccionadas.

Como ejemplo, si en el requisito funcional dice "*value*" y en el código fuente dice "*price*" la *distancia de Levenshtein* es de 20% y la relación con WordNet para la cual se usa la página http://www.olesk.com es de 91%. EL valor usado para la métrica es el mayor, es decir, 91% de consistencia entre estas dos palabras

Técnica candidata:

Nombre de la temática: nombramiento, azar, semántica, pensamiento, consistencia lingüística

Preguntas de caracterización técnica:

¿Cómo nombrar un elemento en el código fuente?

¿Cómo hacer consciente al desarrollador sobre los nombres de las partes del sistema?

¿Cómo medir la calidad respecto de la forma en que se nombra el software?

¿Cómo medir la consistencia lingüística?

¿Cómo mejora el proceso de desarrollo de software con una buena consistencia lingüística?

¿Cuánto afecta el entendimiento del sistema de software el diferente nombramiento en los modelos y el código fuente?

Plantilla técnica juego original:

Para exponer de manera precisa y acorde con la plantilla de Gómez (2010), en la Tabla 5-1 se muestran las características del juego original Twister de Hasbro® como materiales, generalidades y reglas que permiten conocer la dinámica del juego e identificar al ganador. El juego expuesto se evalúa según los criterios de Gómez (2010) y se usa como base del juego propuesto para evaluar la métrica de consistencia lingüística.

Tabla 5-1: Plantilla técnica juego base Twister ruleta. (Parte 1/2)

Fuente: Elaboración propia basado en https://www.hasbro.com y Gómez (2010)

PLANTILLA TÉCNICA		
I. GENERALIDADES		
Nombre Técnica	Twister ruleta	
Objetivo del Juego	Girar la ruleta y decir en voz alta los movimientos. Los jugadores deben mover esa parte del cuerpo al círculo de ese color tan rápido como puedan. Si se apoya una rodilla o un codo en el tapete, o si se cae, ¡Está ELIMINADO! ¡El último jugador que quede en pie es el ganador!	
Jugadores	3 o más Jugadores.	
II. MATERALES		
Nombre	**Cantidad**	**Descripción**
Flecha de Ruleta	1	Flecha para girar y de manera aleatoria indicar el color a pisar según donde apunte en el tablero de la ruleta.
Tablero	1	Tablero para marcar cuál de los colores tocan los jugadores.
Base	1	Base para que gire la flecha libremente y le de aleatoriedad al giro.

Tabla 5-1: Plantilla técnica juego base Twister ruleta. (Parte 2/2)

Fuente: Elaboración propia basado en https://www.hasbro.com y Gómez (2010)

III. REGLAS DEL JUEGO	
Nro	**Descripción**
1	Sólo una mano o un pie por círculo.
2	El primer jugador que llegue a un círculo, lo ocupa.
3	Quien gira la ruleta es el árbitro y decide en caso de desacuerdos.
4	Cuando se realice un movimiento, no se puede mover otra vez a menos que el árbitro esté de acuerdo, incluso si otro jugador intenta pasar por encima.
5	Si los seis círculos de un color están ocupados, se gira la ruleta otra vez
6	Si el árbitro dice una combinación realizada, se mueve la mano o el pie a otro círculo de ese mismo color. (Si los seis círculos están ocupados, se gira otra vez.)

Técnicas más apropiadas según la caracterización de la temática:

Tabla 5-2: Preguntas para definir la técnica del juego según la temática (Parte 1/2)

Fuente: Elaboración propia basado Gómez (2010).

Pregunta	**Respuesta**	**Puntaje**
¿Uno de los propósitos del juego es simular escenarios para que los desarrolladores tomen decisiones según los objetivos propuestos?	Si	3
¿Se describe una o varias situaciones donde los desarrolladores asumen roles específicos?	Si	3
¿Los desarrolladores no tiene el mismo nivel de conocimiento de las condiciones del escenario, es decir hay información de la situación solo en poder de algunos desarrolladores?	No	0

Tabla 5-2: Preguntas para definir la técnica del juego según la temática (Parte 2/2)

Fuente: Elaboración propia basado Gómez (2010).

Pregunta	Respuesta	Puntaje
¿En el juego existen conflictos o circunstancias que deben resolver los desarrolladores mediante comunicación?	No	0
¿Inicialmente todos los desarrolladores del juego reciben la misma descripción del escenario?	Si	3
¿Debe existir un moderador del juego encargado de describir la situación inicial y dirigir el desarrollo del juego?	No (pero se recomienda)	0
¿La situación o escenario simulado tiene inicio, momento para la toma de decisiones y desenlace del escenario?	Si	3
¿Al finalizar el juego es importante hacer una socialización para analizar las decisiones tomadas, los factores que influyen en ellas y sus consecuencias?	Si	3
¿Este tipo de juegos se presta para plantear a los desarrolladores el interrogante acerca de la estrategia optima u acciones a seguir para maximizar los beneficios como parte del proceso de análisis del comportamiento?	Si	1
¿Los desarrolladores según el rol asumido toman decisiones que reflejan su sistema de creencias personales?	Si	3
¿No existen reglas estrictas, sino que el desarrollo del juego depende de las decisiones de los desarrolladores?	Si	1

Puntaje final: 20

Puntaje final = 20/23 = 0.86

En este apartado, las preguntas diferenciadoras tienen tres unidades y las preguntas estándar una unidad. Al dividir el puntaje obtenido sobre el máximo posible es superior 0.85 y, por tanto según Gómez (2010) es viable adaptar el juego.

Dado que el juego cumple con los criterios de Gómez (2010), se adapta en la Tabla 5-3 a un juego que permite evaluar de manera empírica la necesidad de la creación de la métrica de consistencia lingüística propuesta.

Incorporar el conocimiento específico al juego:

Plantilla técnica juego adaptado:

Tabla 5-3: Plantilla técnica juego adaptado *Nómbralos Cerebro.* (Parte 1/5)

Fuente: Elaboración propia basado en Gómez (2010)

PLANTILLA TÉCNICA	
I. GENERALIDADES	
Nombre Técnica	Nómbralos cerebro
Objetivo	Evaluar de manera empírica la necesidad de crear una métrica que evalúe la consistencia lingüística entre los requisitos funcionales y el código fuente Ser el desarrollador que mejore más la consistencia lingüística del sistema de software. Comprender la importancia que tiene el buen nombramiento del código fuente, de manera que éste sea consistente con los requisitos funcionales, los cuales en este caso se representan con un diagrama de clases. Crear en cada desarrollador prácticas de buen nombramiento del código fuente. Despertar habilidades de codificación que reflejen prácticas de calidad semánticas orientadas a la consistencia lingüística.

Tabla 5-3: Plantilla técnica juego adaptado *Nómbralos Cerebro* (Parte 2/5)

Fuente: Elaboración propia basado en Gómez (2010)

PLANTILLA TÉCNICA		
Objetivo	Mostrar cómo se utiliza la métrica de consistencia lingüística para medir la distancia semántica que tienen los requisitos funcionales del código fuente.	
Número de Desarrolladores	Hasta 6 desarrolladores.	
II. MATERIALES		
Nombre	**Cantidad**	**Descripción**
Pensamiento aleatorio para nombrar en forma de flecha	1	Macro en Excel virtual
Lista de opciones circular	1	"Cerebro del desarrollador" construido con Macros en Excel y que se encarga de ayudar a tomar las decisiones de nombramiento a cada desarrollador. Es una lista de opciones circulares que contiene las posibles decisiones que puede tomar un desarrollador para nombrar un elemento del código fuente.
Base	1	La base es un Excel con macros. Es necesario al abrir el documento de Excel y activar las macros para que se ejecuten el "cerebro" y demás funcionalidades que son programadas para dar más didáctica al juego, funcionen como se espera y el juego sea más dinámico.

Tabla 5-3: Plantilla técnica juego adaptado *Nómbralos Cerebro* (Parte 3/5)

Fuente: Elaboración propia basado en Gómez (2010)

PLANTILLA TÉCNICA		
Nombre	**Cantidad**	**Descripción**
Tablero	1	Lugar con las palabras que el cerebro del desarrollador le indica tomar (Sinónimo, abreviatura, “No se me ocurre nada”, “Sé consistente”). Diagrama de clases que representa la necesidad funcional. Lugar donde el desarrollador afecta el código fuente según la orden que obtuvo de su cerebro, cambiando una palabra por lo que este le indique (sinónimo, abreviatura o una palabra cualquiera que no tenga que ver con el software cuando salga “no se me ocurre nada”, o la misma palabra del diagrama cuando salga “Sé consistente”). Porcentaje de afectación del código fuente según se modifiquen los términos utilizados. Comparativo entre las afectaciones del código fuente que hace cada jugador.
III. REGLAS DEL JUEGO		
Nro	**Descripción**	
1	Los desarrolladores parten de un código fuente con 0% de consistencia lingüística a la luz de un diagrama de clases dado. Cada desarrollador, cuando tiene que editar el código fuente, pone su cerebro a dar vueltas para nombrar el código fuente como mejor le parezca.	
2	El cerebro del desarrollador sólo indica cómo nombrarlo, pero no qué porción de código fuente editar. Es decisión del desarrollador cómo lo hace y particularmente de su decisión impactará en diferente porcentaje la consistencia lingüística del código fuente.	

Tabla 5-3: Plantilla técnica juego adaptado *Nómbralos Cerebro* (Parte 4/5)

Fuente: Elaboración propia basado en Gómez (2010)

III. REGLAS DEL JUEGO	
Nro	**Descripción**
3	El desarrollador no puede cambiar una palabra por otra que tenga la misma característica de sinónimo o abreviatura, a menos que no tenga mas opciones, es decir, si al desarrollador su cerebro le indica que debe usar un sinónimo, él no debe reemplazar un término que ya es sinónimo del original a menos que el sistema se nombre completamente con sinónimos.
4	Cada desarrollador tendrá su medida de cuánto mejoró o empeoró la consistencia lingüística.
5	Si el cerebro le indica al desarrollador poner una abreviatura, él toma una palabra de la lista de “Abreviaturas”.
6	Si el cerebro le indica al desarrollador poner un sinónimo, él toma una palabra de la lista de “sinónimo”.
7	Si el cerebro le indica al desarrollador ser consistente, él desarrollador debe ser consistente, es decir, si en el diagrama la clase se llama “*Cow*” lo consistente será que en el código fuente la clase se llame “*Cow*” (las palabras consistentes están ocultas en la columna R por si hay dudas).
8	Si al cerebro le sale “No se me ocurre nada”, el desarrollador no cambiará el nombre de ningún elemento del código fuente.
9	El campo de la celda “*Afectación desarrollador*” indica cuando mejoró o desmejoró el desarrollador de turno la consistencia del sistema de software. Se debe actualizar su aporte según lo que esté en este campo suma o resta a lo que lleve y de esta manera se sabrá que desarrollador al final mejoró más el código fuente.

Tabla 5-3: Plantilla técnica juego adaptado *Nómbralos Cerebro* (Parte 5/5)

Fuente: Elaboración propia basado en Gómez (2010)

IV. Criterios de selección del ganador
Se representa un sistema en la fase de mantenimiento, cada desarrollador debe refactorizar el sistema de software mejorando el nombramiento en el código fuente, de tal manera que el desarrollador tome conciencia de la importancia del nombramiento del código fuente y que éste también es una herramienta del entendimiento del sistema. No existe un momento que determine que el juego termina, cuando todos los desarrolladores toman conciencia de la importancia de la consistencia lingüística entre la definición, el modelado, la implementación y el cómo se mide esto mediante la consistencia lingüística se detiene y determina el ganador. El ganador será aquel que mejore más la consistencia lingüística del código fuente.

5.1.2 Materiales del Juego

En las Figuras 5-1 y 5-2 se propone la hoja de cálculo en Microsoft 365® Excel® que contiene las macros necesarias para que el juego funcione, el diagrama de clases que se usó durante los diferentes momentos en que se puso a prueba el juego, el código fuente que representa los modelos y los resultados que se obtienen de parte de cada participante al igual que el aporte final al sistema de software.

Figura 5-1: "Cerebro" y afectación del desarrollador al código fuente

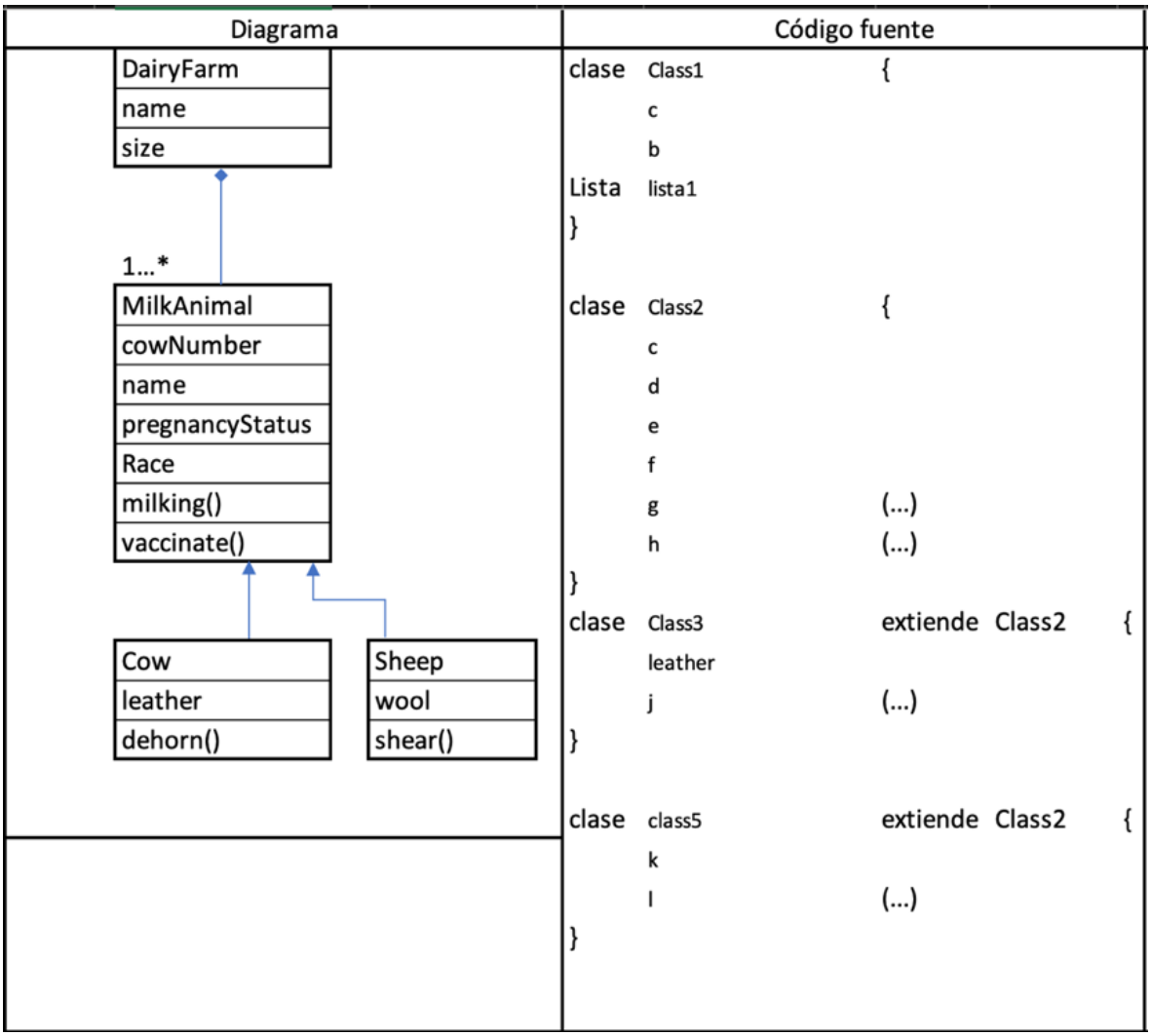
Fuente: Elaboración propia

Afectación desarrollador	0.00
Consistencia	
Desarrollador 1	0.00
Desarrollador 2	0.00
Desarrollador 3	0.00
Desarrollador 4	0.00
Desarrollador 5	0.00
Desarrollador 6	0.00
Aportes Total	
0.00	

En la Figura 5-2 se muestra el diagrama inicial y el código fuente entregado a los participantes del juego. Estas palabras se usan en inglés para cumplír con las especificaciones descritas en el numeral 4.1 (Alcance) de la presente Tesis de Maestría.

Figura 5-2: Diagrama inicial y código fuente entregado a los participantes

Fuente: Elaboración propia



Adicional a esto, se entregan las posibles palabras a relacionar entre las que se encuentran las que hacen que el código fuente sea 100% consistente con el modelo. Estas palabras se presentan en la Tabla 5-4 como insumo a los jugadores dado que los cálculos se hacen previos al inicio del juego para que sea más dinámico.

Tabla 5-4: Conjunto de palabras entregadas a los jugadores como opciones.

Fuente: Elaboración propia

Se consistente	Sinónimos	Abreviaturas/errores
DairyFarm	Farm	FAR
	CowFarm	CF
	MilkFarm	DF
name	cowName	NAM
size	measure	SI
	extension	SIZ
MilkAnimal	Animal	MA
	Mammal	MANIMAL
cowNumber	Number	CN
name	nickname	NANIMAL
pregnancyStatus	pregnancy	PS
Race	line	R
milking	produce	MILK
vaccinate	inject	VACUNE
Sheep	goat	SH
	lamb	OVEJA
wool	hair	W
shear	shave	SH
Cow	beef	C
leather	skin	LT
dehorn	removeHorns	TOPIZAR

5.1.3 Comentarios sobre el juego

El juego se probó en la clase de lingüística computacional de la Universidad Nacional de Colombia sede Medellín del semestre 2019-01 y se concluyó por parte de las personas que la *métrica de consistencia lingüística* mediante el juego permite reforzar:

- La importancia de los modelos para la construcción del código fuente.
- Cómo la consistencia entre los modelos y el código fuente es relevante para el entendimiento completo del sistema de software.
- La afectación al tiempo de entendimiento del código fuente que se ocasiona por el cambio en la terminología de los modelos.

5.2 Caso de laboratorio del uso de la consistencia lingüística

El caso de laboratorio es un paso a paso a manera didáctica, que facilita el entendimiento de la forma de cálculo de la métrica de consistencia lingüística, expuesto mediante una prueba controlada, basada en un código fuente real existente en varios sistemas.

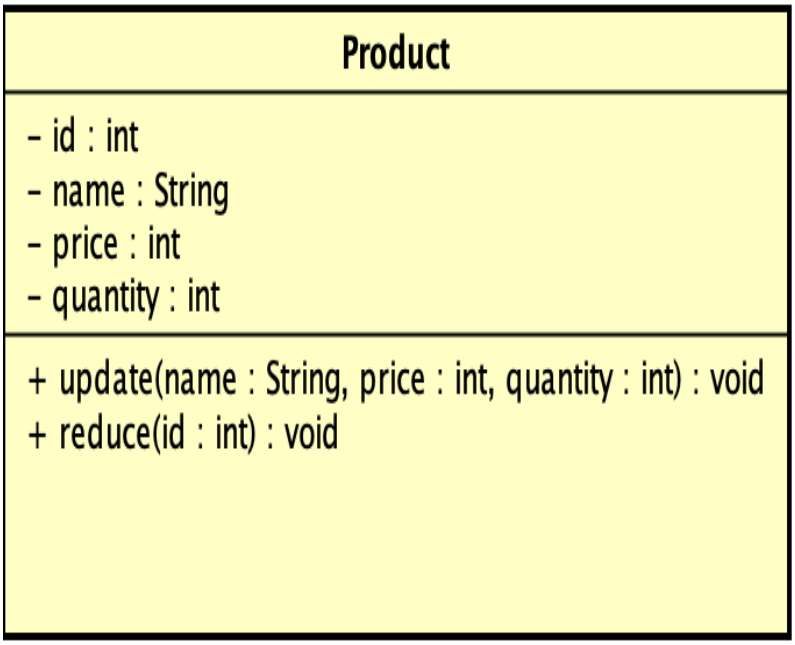
En el ejemplo, se toma la porción de un sistema de software para uso ilustrativo, se identifican los puntos que afectan el entendimiento del sistema de software, se analizan y posteriormente se modifican de forma que los resultados para la métrica de consistencia

lingüística queden al 100% y con ello, se facilite la lectura del código fuente a la luz de los requisitos funcionales.

- Se selecciona el modelo o porción del modelo que representa el requisito funcional. Este puede ser un esquema preconceptual como el propuesto por Zapata (2007) o un diagrama UML, pero debe ser validable con el código fuente. Para este caso se selecciona el diagrama de clases UML de la clase “product” (Véase la Figura 5-3).

Figura 5-3: Representación UML de la clase “product”.

Fuente: Elaboración propia.



- Se identifica la porción de código fuente correspondiente con la representación anterior. Para el ejemplo el lenguaje de programación es Java (Véase la Figura 5-4), pero siempre que se pueda hacer un emparejamiento con el código fuente, se puede usar cualquier lenguaje de programación.

Figura 5-4: Porción del código fuente correspondiente a la clase “product”.

Fuente: Elaboración propia.

```
public class Product {

    private int id;
    private String productName;
    private int value;
    private int quantity;

    public void update (String name, int value, int amount) {
        //....
    }

    public void delete(int product) {
        // ...
    }

    //Getters and Setters

}
```

- Se identifican los términos pares a los que se les va a evaluar su distancia semántica y calcular su distancia:

En este proceso se buscan los mismos términos en ambos productos de trabajo, es decir, se busca el nombre de la clase en el modelo y el nombre de la clase en el requisito funcional, se busca el nombre de cada uno de los atributos y métodos de la clase incluidos sus parámetros en el código fuente y su par en el diagrama. En este caso, sólo se llega a este nivel de detalle porque así lo tiene el modelo. Si el modelo da más detalle de la implementación se sigue el procedimiento con cada una de las palabras que esté en el código fuente. Además, como se encuentra en el estudio de Avidan y Feitelson (2017), son los puntos que más aportan al entendimiento del sistema de software.

Una vez identificados cada uno de los pares de palabras, se someten a evaluación por medio de la distancia que arroja el *lexical database* y también a la modificación que se hizo a la *distancia de Levenshtein*, definida en esta Tesis de Maestría como *relación de Levenshtein* (véase la Ecuación (4.1)).

Para el *análisis semántico latente*, en este ejemplo se usa una lexical database ya poblada y habilitada en internet en abril del 2020. Esta es http://www.olesk.com, la cual, a pesar de que solo incluye el idioma inglés es práctica para el ejemplo con WordNet. En esta se evalúan todas las palabras, pero las contracciones de dos o más palabras se evalúan por separado para tener una mejor distancia semántica, en el ejemplo "*productName*" se evalúa como "*product name*", así el sistema es más preciso en su valoración. sin embargo, en esta los cambios de idioma no tienen relación semántica.

Para la medición de la *distancia de Levenshtein* de los pares de palabras seleccionados, también hay herramientas en la web que ya la calculan. Para abril del 2020 para los ejemplos de esta Tesis de Maestría se usa https://es.planetcalc.com. Sin embargo, esta medida arroja un número de operaciones básicas para transformar una palabra en otra, por lo que se usa la fórmula previamente expresada (véase la Ecuación (4.1)) para transformar la *distancia de Levenshtein* a una medida porcentual que represente la relación entre las palabras.

- Se toma el mejor de los dos análisis realizados a cada par de palabras, es decir, si se analizan dos palabras y la *relación de Levenshtein* da como resultado un 13.3% y el cálculo de la distancia semántica usando una lexical database da un 25% se toma la mayor, el 25% sería la relación que tienen las dos palabras. luego de esto, se debe usar la totalidad de los cálculos realizados a las palabras y promediar los resultados, de manera que se pueda tener un ponderado total de la porción del sistema de software analizado.

En la Tabla 5-5 se muestran los pares de palabras extraídos de los modelos y el código fuente y los cálculos para obtener sus distancias semánticas.

Tabla 5-5: Palabras extraídas de los modelos y el código fuente y los cálculos de sus distancias.

Fuente: Elaboración propia.

Código fuente	Modelo	Relación semántica %	Relación de Levenshtein %	Distancia semántica %
Product	Product	100	100	100
id	id	100	100	100
productName	name	99.2	27.3	99.2
value	price	91	20	91
quantity	quantity	100	100	100
update	update	100	100	100
name	name	100	100	100
value	price	91	20	91
amount	quantity	99	12.5	99
Reduce	delete	62	33.3	62
product	id	41	14.3	41
Métrica de consistencia lingüística:		89.4%		

$$Consistencia\ ling\ddot{u}\acute{i}stica = \frac{\sum RS(x,y)}{Total\ de\ pares\ de\ palabras\ evaluadas}$$

$$89.4\% = \frac{(100 + 100 + 99.2 + 91 + 100 + 100 + 100 + 91 + 99 + 62 + 41)}{11}$$

- Se analizan los resultados obtenidos validando que cambios impactan más el sistema de software y con él su entendimiento.

Del cuadro anterior se observa que, en este caso particular todos los pares de elementos analizados tienen una mejor medida basados en la *lexical database* utilizada y adicional se pueden concluir que:

- El nombre de la clase está bien y consistente entre el código fuente y los modelos.

- Los atributos "*id*" y "*quantity*" están bien nombrados y su consistencia con el modelo es del 100%. Sin embargo, los atributos "*price*" y "*name*" no son consistentes con el modelo. Estos presentan diferencias que afectan el rápido entendimiento del código fuente del sistema de software utilizando los modelos como guía.

- El método "*update*" está bien nombrado en el código fuente respecto de los modelos, pero sus atributos no; solo el atributo "*name*" es 100% consistente para la métrica de consistencia lingüística, "*price*" y "*quantity*" no están iguales y por tanto están afectando el entendimiento del código fuente.

  - El caso del método "*reduce*" es el que más impacta el entendimiento del sistema de software porque no coinciden ni el nombre del método, ni el nombre de sus parámetros. Por tanto, si un desarrollador quisiera buscarlo desde los modelos, se le dificulta más el entendimiento e impacta en tiempo el mantenimiento del sistema de software.

- Se refactoriza el código de manera que, la consistencia lingüística sea del 100% entre los modelos y el código fuente del sistema de software.

Refactorizando el código fuente para que sea 100% consistente con el modelo en la terminología queda como se muestra en la Figura 5-5.

Figura 5-5: Porción de código fuente refactorizado con cumplimiento del 100% en la consistencia lingüística respecto del modelo presentado en la Figura 5-3.

Fuente: Elaboración propia.

```
public class Product {

    private int id;
    private String name;
    private int price;
    private int quantity;

    public void update (String name, int price, int quantity) {
        //....
    }

    public void reduce(int id) {
        // ...
    }

    //Getters and Setters

}
```

Luego de realizar los cambios al código fuente de forma que los términos concuerden con los del modelo se puede determinar que, con el cambio en los términos usados, el código fuente queda 100% consistente con el modelo y por tanto, una persona que vea el modelo y necesite buscar algo en el código fuente lo podrá hacer de manera más intuitiva y directa.

6. Conclusiones y trabajo futuro

6.1 Conclusiones

- Con la *métrica de consistencia lingüística* se confirmó de manera cuantificable que el cambio de las palabras a lo largo del desarrollo de un sistema de software deteriora su entendimiento y, por tanto, a medida que los cambios sean mayores, el mantenimiento será más lento y complicado.

- Con el uso de la *métrica de consistencia lingüística*, se pudo evaluar qué puntos son los que causan mayor deterioro en la comprensión del sistema de software y, por tanto, en dónde enfocarse para mejorar el entendimiento semántico relacionado con los modelos de un sistema de software.

- Los cambios sintácticos impactan de manera directa la semántica del código fuente de un sistema de software, pues al modificar la manera en que se copia una misma palabra se altera el contexto y por tanto el entendimiento de una sentencia funcional.

- La terminología utilizada en la educción de requisitos y plasmada en los modelos se debe tomar como parte fundamental de los recursos y no se debe modificar, pues hacerlo impacta de manera directa el entendimiento del sistema y por tanto su calidad.

- La cuantificación del entendimiento de un sistema de software permitió conocer los puntos que son factores de riesgo para el retraso en el futuro mantenimiento y facilitó su corrección.

- La relación de los términos de los modelos y del código fuente mediante la *métrica de consistencia lingüística* permitió ligar más al desarrollo los insumos adicionales al código fuente, volviendo todo un instrumento acoplado que se debe mantener y actualizar por igual, evitando documentación inútil en etapas avanzadas de los proyectos.

- En un sistema de software se debe poder cuantificar la calidad en la mayor cantidad de aspectos posibles, porque cambios tan pequeños como el cambio de una palabra por un sinónimo pueden impactar el entendimiento del código fuente de manera negativa sin que esto lo haga conscientemente un desarrollador.

6.2 Trabajo futuro

- Mejorar la forma con que se hace el emparejamiento de los diferentes productos de trabajo; el proceso es para el alcance de esta Tesis de Maestría de forma manual y, por tanto, consume gran parte del tiempo de evaluación de la métrica.

- Ampliar la métrica a diferentes etapas del diseño de software, de manera que se puedan ejecutar ejercicios prácticos con diferentes modelos para garantizar su consistencia lingüística.

- Incorporar la *métrica de consistencia lingüística* a herramientas ASAT (*Automatic Static Analysis Tools*) y de nombramiento, de manera que se pueda evaluar durante el proceso de construcción del sistema de software automáticamente

- Evaluar la consistencia de nombramiento en el código fuente para arquitecturas que tienen capas, sin necesidad de tener modelos con tan alto grado de especificidad.

Referencias

J. Acosta. “Información semántica, distancias semánticas y conceptos”, Revista interuniversitaria de formación del profesorado Buenos Aires, Vol. 4, pp. 71-78. 1989.

M. Allamanis, E. T. Barr, C. Bird, C. Sutton. “Learning natural coding conventions”, in International Symposium on Foundations of Software Engineering (FSE). ACM, New York, pp. 281-293, 2014.

E. Avidan, D. G. Feitelson. “Effects of Variable Names on Comprehension: An Empirical Study”, in IEEE International Conference on Program Comprehension, Jerusalem, pp. 55-65. 2017.

D. Binkley, M. Hearn, D. Lawrie. “Improving identifier informativeness using part of speech information”, in 8th Working Conference on Mining Software Repositories, Honolulu, pp. 203–206, 2011.

S. Blinman, A. Cockburn. “Program comprehension: Investigating the effects of naming style and documentation”, in 6th Australasian User Interface Conf, Newcastle. pp. 73–78, 2005.

B. Boehm, V. Basili. “Software Defect Reduction Top 10 List”. IEEE Computer, Vol. 34, pp. 135-137. 2001.

L. C. Briand, J. W. Daly y J. Wüst. “A Unified Framework for Cohesion Measurement in Object-Oriented Systems”, in Fourth International Software Metrics Symposium, Albuquerque, pp. 43-53, 1997

S. Blinman, A. Cockburn. “Program comprehension: Investigating the effects of naming style and documentation”. In 6th Australasian User Interface Conf, Newcastle. pp. 73–78, 2005.

L. Castro, F. Baiao, G. Guizzardi. “A survey on Conceptual Modeling from a Linguistic Point of View”. Relatórios Técnicos do Departamento de Informática Aplicada da UNIRIO, Vol. 3, pp. 3-12. 2009.

A. Ene, A. Ene. “An application of the Levenshtein algorithm in vocabulary learning”, in International conference on electronics, Computers and artificial intelligence, Targoviste , pp. 1-4, Junio, 2017.

L. Etzkorn, H. Delugach. “Towards a semantic metrics suite for object-oriented design Proceedings”, in 34th International Conference on Technology of Object-Oriented Languages and Systems- TOOLS 34, Santa Barbara, pp. 71-80, 2000.

S. Fahmi, H. J. Choi. "Software Reverse Engineering to Requirements", in International Conference on Convergence Information Technology, Gyeongju, pp. 2199-2204, 2007.

M. C. Gómez. "Definición de un método para el diseño de juegos orientados al desarrollo de habilidades gerenciales como estrategia de entrenamiento empresarial", Tesis de Maestría, Maestría en Ingeniería Administrativa, Universidad Nacional, Colombia. 2010.

R. Gutiérrez. "Análisis Semántico Latente: ¿Teoría psicológica del significado?", Revista signos, Vol. 38, N. 59, pp. 303-323, 2005.

G. Jorge y Botana. "El Análisis de la Semántica Latente y su aportación a los estudios de Usabilidad" Revista electrónica: No solo usabilidad: revista sobre personas, diseño y tecnología, Vol. 5, N. 5, 2006.

B. Kitchenham, S. Charters. "Guidelines for performing Systematic Literature Reviews in Software Engineering", Keele University and Durham University Joint Report , 2007-01.

B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, S. Linkman. "Systematic literature reviews in software engineering--a tertiary study," Information and Software Technology, Vol. 52, N. 8, pp. 792-805. 2010.

D. Marcilio, R. Bonifacio, E. Monteiro, E. D. Canedo, G. Pinto. "Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube" IEEE/ACM 27th in International Conference on Program Comprehension, Montreal, pp. 209-219, 2019.

G. Miller. “WordNet: a lexical database for English” Communications of the ACM, Vol. 38, pp. 39-41, 1995.

O. Pastor, J. C. Molina. “Model-Driven Architecture in practice: A Software Production Environment Based on Conceptual Modeling” Springer, New York, 2007.

L. H. Rosenberg, S. B. Sheppard. “Metrics in software process assessment, quality assurance and risk assessment”, in IEEE 2nd International Software Metrics Symposium, London, pp. 10-16, 1994.

J. J. Siegmund. “Program Comprehension: Past, Present, and Future”, in IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Suita, Vol. 5, pp. 13-20, 2016.

K. Sun, Y. Ji, L. Rui, X. Qiu. “An improved method for measuring concept semantic similarity combining multiple metrics”, in 5th IEEE International Conference on Broadband Network & Multimedia Technology, Beijing, pp. 268-272, 2013.

R. Tiarks. “What Programmers Really Do: An Observational Study”. Softwaretechnik-Trends, Vol. 31, pp. 36-37. 2011.

R. Venegas. “Análisis Semántico Latente: una panorámica de su desarrollo”, Revista signos, Vol. 36, N. 53, pp. 121-138, 2003.

A. Von Mayrhauser, M. Vans, A. Howe. “Program Understanding Behaviour during Enhancement of Large-scale Software”. Journal of Software Maintenance: Research and Practice, Vol. 9, pp. 299-327. 1997.

Y. Wang, C. Wang, X. Li, S. Yun, M. Song. “How are identifiers named in open source software? About popularity and consistency”. International Journal of Computer and Information Technology, Vol. 3, pp. 616-625. 2014.

E. J. Weyuker. “Evaluating Software Complexity Measures,” IEEE Transactions on Software Engineering, Vol. 14, pp. 1357-1365. 1988.

C. M. Zapata, A. Jaramillo, F. Arango. “Una propuesta para mejorar la completitud de requisitos utilizando un enfoque lingüístico”, Ingeniería & Desarrollo, Vol. 19, N. 19, pp. 1-16. 2006.

C. M. Zapata, B. Manrique. “Transformación de lenguaje natural a controlado en la educción de requisitos: una síntesis conceptual basada en esquemas preconceptuales”. Revista Facultad de Ingeniería Universidad de Antioquia, Vol. 70, N. 70, pp. 132-145. 2014.

C. M. Zapata. “Definición de un esquema preconceptual para la obtención automática de esquemas conceptuales de UML”, Tesis doctoral, Doctorado en ingeniería, Universidad Nacional, Colombia. 2007.