EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Orizaba

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCIÓN I.- TESIS

TRABAJO PROFESIONAL

"INGENIERÍA INVERSA DEL LENGUAJE DE PROGRAMACIÓN NATURALÍSTICO SN"

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:

I.S.C. Pedro de Jesús González Palafox

DIRECTOR DE TESIS:

Dr. Ulises Juárez Martínez

CODIRECTOR DE TESIS:

Dr. Oscar Pulido Prieto

ORIZABA, VERACRUZ, MÉXICO. **MARZO 2023**

Orizaba, Veracruz, 30/marzo/2023
Dependencia: **División de Estudios de Posgrado e Investigación**
Asunto: **Autorización de Impresión**
OPCION: I

C. PEDRO DE JESÚS GONZÁLEZ PALAFOX
CANDIDATO A GRADO DE MAESTRO EN:
SISTEMAS COMPUTACIONALES
P R E S E N T E.-

De acuerdo con el Reglamento de Titulación vigente de los Centros de Enseñanza Técnica Superior, dependiente de la Dirección General de Institutos Tecnológicos de la Secretaría de Educación Pública y habiendo cumplido con todas las indicaciones que la Comisión Revisora le hizo respecto a su Trabajo Profesional titulado:

" Ingeniería inversa del lenguaje de programación naturalístico SN"

comunico a Usted que este Departamento concede su autorización para que proceda a la impresión del mismo.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

Cuauhtémoc Sánchez R.

DR. CUAUHTÉMOC SÁNCHEZ RAMÍREZ
JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

EDUCACIÓN SECRETARÍA DE EDUCACIÓN PÚBLICA | TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLOGICO DE ORIZABA
DIV. DE EST. DE POSGRADO E INVEST.

OG-13-F06

Orizaba, Veracruz, 23/marzo/2023
Asunto: **Revisión de trabajo escrito**

C. CUAUHTÉMOC SÁNCHEZ RAMÍREZ
JEFE DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN
P R E S E N T E.-

Los que suscriben, miembros del jurado, han realizado la revisión de la Tesis del (la) C.

PEDRO DE JESÚS GONZÁLEZ PALAFOX

La cual lleva el título de:

Ingeniería inversa del lenguaje de programación naturalístico SN

Y concluyen que se acepta.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

PRESIDENTE:	DR. ULISES JUÁREZ MARTÍNEZ	FIRMA
SECRETARIO:	M.C. MA. ANTONIETA ABUD FIGUEROA	FIRMA
VOCAL:	DRA. LISBETH RODRÍGUEZ MAZAHUA	FIRMA
VOCAL SUP.:	DR. OSCAR PULIDO PRIETO	FIRMA

TA-09-18

TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE ORIZABA
DIV. DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

CARTA DE ORIGINALIDAD

En la ciudad de Orizaba, Veracruz, el día 21 del mes de abril del año 2023, el que suscribe Pedro de Jesús Gonzalez Palafox, alumno del programa de Maestría en Sistemas Computacionales con número de control M15011160, manifiesta que es autor del trabajo de tesis titulado "Ingeniería inversa del lenguaje de programación naturalístico SN" y declaro que el trabajo es original, ya que sus contenidos son producto de mi directa contribución intelectual. Todos los datos y las referencias a materiales ya publicados están debidamente identificados con su respectivo crédito e incluidos en las notas bibliográficas y en las citas que se destacan como tal y, en los casos que así lo requieran, cuento con las debidas autorizaciones de quienes poseen los derechos patrimoniales. Por lo tanto, me hago responsable de cualquier litigio o reclamación relacionada con derechos de propiedad intelectual, exonerando de toda responsabilidad al Tecnológico Nacional de México / Instituto Tecnológico de Orizaba.

Pedro de Jesús Gonzalez Palafox
Nombre y firma

TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE ORIZABA
DIV. DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

CARTA DE CESIÓN DE DERECHOS

En la ciudad de Orizaba, Veracruz, el día 21 del mes de abril del año 2023, el que suscribe Pedro de Jesús Gonzalez Palafox, alumno del programa de Maestría en Sistemas Computacionales con número de control M15011160 manifiesta que es autor del trabajo de tesis bajo la dirección de Dr. Ulises Juárez Martínez y cede los derechos del trabajo de tesis titulado "Ingeniería inversa del lenguaje de programacíon naturalístico SN" al TecNM / Instituto Tecnológico de Orizaba para su difusión y divulgación, con fines académicos y de investigación.

Queda estrictamente prohibido reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del Tecnológico Nacional de México/Instituto Tecnológico de Orizaba. Este puede obtenerse escribiendo a la siguiente dirección: msc@orizaba.tecnm.mx . Si el permiso se otorga, cualquier usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Pedro de Jesús González Palafox

Nombre y firma

Agradecimientos

En primer lugar, quisiera expresar mi gratitud al Dr. Ulises Juárez Martínez, quien ha sido un mentor excepcional. Gracias por su guía, paciencia, motivación y compromiso en cada etapa de este proceso.

También quiero agradecer al Dr. Oscar Pulido Prieto, mi co-director de tesis, por su valioso apoyo y por compartir sus conocimientos y experiencia. Gracias por su paciencia, sus consejos y su compromiso en este proyecto.

Quiero agradecer a mis padres, quienes han sido mi mayor fuente de inspiración y motivación. Gracias por su amor incondicional, su apoyo y su aliento constante en cada paso de este camino. Su presencia en mi vida ha sido fundamental para mi desarrollo como persona y profesional, y estoy profundamente agradecido por todo lo que han hecho por mí.

También quiero agradecer a mis amigos, quienes han estado a mi lado durante todo este proceso. Gracias por su apoyo incondicional, su paciencia y su compañía en los momentos más difíciles. Sus palabras de aliento y sus ánimos han sido fundamentales para mantenerme motivado y enfocado en este proyecto.

Finalmente, quiero expresar mi agradecimiento al CONACYT por el apoyo financiero otorgado para la realización de este proyecto. Gracias por creer en mi capacidad y por brindarme los recursos necesarios para llevar a cabo este trabajo.

Índice general

Índice de tablas

Índice de figuras

Índice de códigos

Resumen

La programación naturalística es un tema de gran interés porque permite reducir la brecha entre el dominio del problema y el dominio de la solución en el proceso de creación del software. El objetivo del presente trabajo es aplicar un proceso de ingeniería inversa al lenguaje de programación SN para identificar las áreas de oportunidad del lenguaje y posteriormente complementar su documentación y tras llevar a cabo lo anterior, mejorar el sistema de instalación de SN para obtener un proceso sencillo que permita la difusión del lenguaje de forma estable. Finalmente, identificar los mejores mecanismos para la difusión del lenguaje, poniendo a disposición de la comunidad de desarrolladores una versión de SN que permita familiarizarse con el paradigma naturalístico.

Abstract

Naturalistic programming is a topic of great interest because it allows reducing the gap between the problem domain and the solution domain in the software creation process. The goal of this work is to apply a reverse engineering process to the SN programming language to identify the language's areas of opportunity and subsequently complement the language documentation and improve the SN installation system to obtain an installation process that allows the stable diffusion of the language.

Finally, identify the best mechanisms for language diffusion, making available to the developer community a version of SN that allows them to become familiar with the naturalistic paradigm.

Introducción

La programación naturalística se define como una técnica de programación que utiliza abstracciones cuya expresividad se acerca a los lenguajes naturales. El objetivo es preservar las necesidades del cliente en su idioma, mientras que el texto de estas necesidades es simultáneamente la especificación de requisitos y el código fuente del programa.

En consecuencia, el objetivo del paradigma naturalístico es reducir la brecha entre el dominio del problema y el de la solución. En la literatura, se reportan dos enfoques principales, uno se enfoca en transformar lenguajes naturales controlados en código de alto nivel, como Java y Python; en el otro, la descripción de los requisitos es al mismo tiempo el código fuente del programa. Si bien los traductores empleados en el primer enfoque no ofrecen un nuevo paradigma, son pocos los lenguajes de programación naturalísticos de propósito general.

El presente trabajo se enfoca en el lenguaje de programación naturalístico SN, el cual necesita una versión estable que permita su difusión con el objetivo de extender el paradigma naturalístico. Para lograr la difusión del lenguaje, es necesario complementar la documentación existente del mismo e identificar los medios de difusión más eficientes.

Este trabajo se organiza en cinco capítulos principales:

En el capítulo 1 se cubre el marco teórico, planteamiento del problema, objetivo general y objetivos específicos, y la justificación.

En el capítulo 2 se describen los trabajos relacionados con el paradigma naturalístico. Se analizan los lenguajes de programación naturalísticos de propósito general reportados. Además, se presenta la propuesta de solución y las herramientas a utilizar para cumplir con los objetivos

del presente proyecto.

En el capítulo 3 se muestra el desarrollo de la metodología del proyecto, se exploran las características de SN, la arquitectura del lenguaje y la arquitectura del compilador.

En el capítulo 4 se presentan los resultados de la ejecución del proceso de ingeniería inversa.

Finalmente, en el capítulo 5 se presentan las conclusiones y recomendaciones.

Capítulo 1

Antecedentes

En este capítulo se presenta el marco teórico en el que describen los conceptos fundamentales dentro del contexto del presente proyecto de tesis, además se presenta el objetivo general y los objetivos específicos. Finalmente se presenta la justificación del proyecto.

1.1. Marco teórico

Este apartado expone los conceptos más importantes relacionados con el trabajo presentado.

1.1.1. Programación naturalística

El diccionario de inglés de Cambridge define "naturalístico" como "Similar a lo que existe en la naturaleza"[1]. De esta definición, una cosa naturalística es artificial y se contrapone a lo natural, pero se diseña para ser similar a su contraparte natural.

En [1] se define programación naturalística como el paradigma que toma elementos de los lenguajes naturales para diseñar lenguajes formales de programación más expresivos desde la perspectiva humana, con la intención de reducir la brecha existente entre el planteamiento del

[1]Diccionario de inglés Cambridge - https://dictionary.cambridge.org/es/diccionario/inglesespanol/naturalistic (accedido Mayo 27, 2022)

problema y la solución del mismo.

La programación naturalística incorpora elementos lingüísticos tomados del lenguaje natural como: anáforas, temporalidad y abstracciones, por mencionar algunos, los cuales son un medio para definir e implementar un lenguaje de programación naturalístico de propósito general.

1.1.2. Lenguaje natural

El término lenguaje es amplio y ambiguo. Se utiliza para describir y comunicar ideas y sentimientos por medio de la palabra. El lenguaje consiste en la materialización de signos que simbolizan objetos e ideas.

En el libro [2], el autor afirma que el lenguaje despliega la capacidad simbólica de los seres humanos, la capacidad de comunicar mediante una articulación de sonidos y signos «significantes», provistos de significado. El autor señala que existen diferentes lenguajes cuyo significante no es la palabla; pero el lenguaje esencial que caracteriza al ser humano es el lenguaje «lenguaje-palabra»; afirma que el lenguaje no sólo es un instrumento para comunicar sino también un instrumento del pensar[2].

En [3] el autor determina la figura de la esencia del lenguaje humano. Concretamente ésta: Las palabras del lenguaje nombran objetos - las oraciones son combinaciones de esas denominaciones.

Deíxis

El Oxford English Dictionary define el término deixis como:

"The function or use of deictic words or expressions whose meaning depends on where, when, or by whom they are used."

Por su traducción al español:

[2]Tesis expuesta en *La política* (1979), donde afirma que pensar es «onomatología», *logos* construido en palabras y mediante palabras.

"La función o uso de palabras o expresiones deícticas cuyo significado depende de dónde, cuándo o por quién se utilizan."

El término deíxis comprende el sentido de una oración, el cual depende del contexto de trabajo. La deixis existe en todos los lenguajes naturales, ya que permite expresar ideas, reducir el número de palabras, por ejemplo, la oración: "tú tomáras clases por la tarde y yo tomaré la misma clase", reduce a "tú tomáras clases por la tarde y yo también" sin que pierda su significado. En algunas ocasiones, las oraciones resultantes de la omisión de palabras son ambiguas, sin embargo, el proceso cognitivo humano resuelve la ambigüedad.

Endófora

Endófora es una referencia indirecta realizada dentro del mismo texto, clasificadas según el momento de definición del objeto referenciado.

1. Anáfora: La referencia indirecta realizada después de declarar el referente.

   Ejemplo: Toma la memoria de la mesa, después conéctala a la computadora.

2. Catáfora: La referencia indirecta realizada antes de la declaración del objeto referente.

   Ejemplo: Después de tomarla de la mesa, conecta la memoria a la computadora.

3. Exófora: Reemplaza una palabra definida en un texto anterior por otra.

   Ejemplo: ¿Te interesa el reloj que está en el aparador? ¿Verdad que es lindo? La homófora es un tipo particular de exófora, en la cual, la referencia depende del contexto específico.

Sintagmas

Según el diccionario de la Real Academia Española:

"Palabra o conjunto de palabras que se articula en torno a un núcleo y que puede ejercer alguna función sintáctica."

Se conoce como sintagma, al conjunto de palabras que tienen una función específica dentro de una oración. Por lo tanto, los sintagmas son de alta importancia en los lenguajes naturalísticos, ya que, proporcionan la capacidad de crear instrucciones complejas con un alto nivel de expresividad por medio de sustantivos y verbos, que dotan a las instrucciones de un contexto específico. Los sintagmas se clasifican en:

- **Sintagma nominal (*noun phrase*)**: Según el diccionario de la Real Academia Española: "m. Gram. sintagma que tiene por núcleo un nombre.". Sintagma cuyo núcleo es un sustantivo o pronombre. Usualmente es en el sintagma nominal donde se designa al sujeto que participará en el predicado verbal.

- **Sintagma verbal (*verb phrase*)**: Según el diccionario de la Real Academia Española: "m. Gram. sintagma que tiene por núcleo un verbo.". En este sintagma, se tiene un conjunto de palabras cuyo núcleo es un verbo, el cual concuerda con el número y género que se establece en el sintagma nominal.

- **Sintagma preposicional (*prepositional phrase*)**: El diccionario de la Real Academia Española lo define como: "m. Gram. sintagma que tiene por núcleo una preposición o que está encabezado por ella.". Este tipo de sintagma tiene como núcleo una preposición, la cual funciona como enlace mientras que el resto de sintagmas que le acompañan forman un término.

- **Sintagma adjetival (*Adjective phrase*)**:La definición del diccionario de la Real Academia Española es: "m. Gram. sintagma que tiene por núcleo un adjetivo.". Esta clase de sintagma se conforma por un conjunto de palabras, pero su núcleo es el adjetivo; por lo que se considera al adjetivo como la palabra con mayor jerarquía.

Contexto

Según el diccionario de la real academia española:

"Entorno lingüístico del que depende el sentido de una palabra, frase o fragmento determinados."

Dada la definición de la real academia española, el contexto es el elemento lingüístico que provee a las oraciones de sentido. Los diseñadores de lenguajes proporcionan una clara definición de la abstracción con el fin de evitar la pérdida de los elementos que requiere el contexto.

El contexto se define en [4] como las circunstancias que ocurren alrededor de un evento determinado. A partir del contexto se puede interpretar o entender un evento. El contexto está compuesto por una serie de circunstancias (como el espacio y el tiempo) que facilitan el entendimiento de un mensaje.

Expresividad

Según la definición del Oxford English Dictionary, expresividad es:

"Showing or able to show your thoughts and feelings"

Su traducción al español es:

"Capacidad de demostrar pensamientos y sentimientos."

En los lenguajes, la expresividad es una característica centrada en la capacidad de describir ideas de forma precisa, ya sea de manera oral o escrita, lo cual facilita la interpretación de una idea. En programación, la expresividad comprende el nivel de descripción que tiene el código, esto con la intención de que los usuarios comprendan el objetivo del programa dentro del dominio aplicado.

Ambigüedad

Según el diccionario de la real academia española:

"Dicho especialmente del lenguaje: Que puede entenderse de varios modos o admitir distintas interpretaciones y dar, por consiguiente, motivo a dudas, incertidumbre o confusión."

La ambigüedad es un atributo presente en palabras, ideas o sintagmas, cuyo significado permite tener más de una interpretación, cada una sujeta al pensamiento del receptor del mensaje.

Gramática

Se define la gramática[3] como el estudio de los componentes de la lengua y sus combinaciones. La gramática estudia la estructura de las palabras y sus accidentes, así como la manera en que se combinan para formar oraciones; incluye la morfología y la sintaxis.

Sintaxis

En lingüística, la sintaxis estudia los elementos de una lengua y sus combinaciones, establece la estructura que debe tener una oración; proporciona instrucciones sobre cómo combinar y vincular palabras, formar oraciones y definir conceptos de manera coherente. En los lenguajes de programación no es diferente, en [5] se define la sintaxis como las reglas que deben cumplir los diferentes elementos de un lenguaje de programación para considerarse como sentencias válidas.

Semántica

En [5] se define a la semántica como la encargada de comprobar el sentido correcto de las sentencias válidas. La semántica establece cómo se comportan los diferentes elementos del lenguaje y determina la validez de las construcciones sintácticas.

1.1.3. Lenguaje formalista

Un lenguaje formalista es, según [6], una representación determinista con una especificación más formal del lenguaje natural. Los lenguajes controlados de tipo formalista se asemejan más

[3]Definición de gramática - Diccionario de la Real Academia Española - https://dle.rae.es/gramática(accedido Mayo. 27, 2022)

a un lenguaje natural formal definido con claridad, es predecible y más fácil de utilizar que un lenguaje formal. Un lenguaje formalista se enfoca a que la lógica sea más entendible por los programadores, de modo que se evita el indeterminismo ya que se requiere que dichos lenguajes se traduzcan a un lenguaje formal de forma clara y predecible.

1.1.4. Programación orientada a aspectos

La programación orientada a aspectos ofrece un mecanismo que permite la encapsulación de los requisitos no funcionales, de manera correcta en entidades bien definidas, lo cual evita la dispersión por todo el código, lo que facilita la separación de responsabilidades. El principal exponente del paradigma orientado a aspectos es AspectJ, dicho lenguaje permite agregar funcionalidad transversal a clases de Java, por tanto, el primero es ortogonal y complementario del segundo.

Pero los aspectos poseen una limitante que se deriva de la forma en que se expresan los cortes, que en el caso de AspectJ es por medio de un patrón de la firma que es expresivo de acuerdo a la sintaxis de la firma de los métodos de Java, lo que da como resultado cortes poco tolerantes a cambios cuando se requiere agregar o modificar la funcionalidad de métodos particulares; ante dicho problema, se menciona que los aspectos son "frágiles".

Aspecto

Un aspecto (del inglés *Aspect*) es una funcionalidad transversal repetida a lo largo del sistema, implementada de forma modular y separada del resto del sistema [7].

Puntos de unión

Un punto de unión (del inglés *join point*) es un evento identificable durante la ejecución del programa, a partir del cual el aspecto puede conectarse con algún otro componente para realizar una acción [7].

Corte en puntos

Los cortes en puntos (*pointcut*), también conocidos como cortes, son la combinación de varios puntos de unión en la que el aspecto lleva a cabo su comportamiento, uniendo las composiciones de cortes por medio de operadores lógicos [8].

Aviso

Un aviso (*advice*) implementa la funcionalidad transversal, es decir, una acción realizada como parte del aspecto. Un aviso define la modificación del código en el contexto del corte en puntos [8].

Asunto

Un asunto es algo de interés para las personas involucradas en el proyecto de software, considera cualquier tipo de requerimiento, concepto o código. Un asunto es el mismo con independencia del paradigma utilizado [9].

1.1.5. SN

SN es un prototipo de lenguaje naturalístico de proposito general, que se basa en el idioma inglés y que deriva de los elementos que se plantean en el modelo presentado en [6]. El prototipo devuelve como resultado bytecode de Java, utiliza referencias indirectas, permite un mayor nivel de descripción al momento de definir un procedimiento al asemejarlo a un sintagma verbal, define sustantivos que opcionalmente poseen una forma plural, adjetivos que se combinan con sustantivos ya sea durante la definición o instanciación y por último, el prototipo presenta una capacidad limitada para describir circunstancias para un sustantivo o un adjetivo.

Circunstancias

Una circunstancia es el contexto que complementa un evento, en un lenguaje natural una circunstancia complementa a un enunciado de forma no secuencial. Las circunstancias permiten definir situaciones que ocurren como consecuencia de un suceso, o que condicionan la ocurrencia del mismo.

En SN una circunstancia es un mecanismo que permite establecer restricciones con base en adjetivos permitidos para la composición, adjetivos requeridos o adjetivos no permitidos.

1.2. Situación tecnológica, económica y operativa de la empresa

El Instituto Tecnológico de Orizaba (ITO) es una institución que fue fundada en el año de 1957 ante las necesidades propias del desarrollo industrial, que en ese entonces iniciaba su despegue en la zona centro del Estado de Veracruz. Ubicada en Oriente 9, Colonia Emiliano Zapata, CP. 94320 en la ciudad de Orizaba, Veracruz.

A partir del año 2016, el ITO pasó a formar parte del Tecnológico Nacional de México. Actualmente este plantel ofrece ocho especialidades a nivel de licenciatura: Ingeniería Eléctrica, Electrónica, Mecánica, Industrial, Química, Sistemas Computacionales, Informática y Gestión Empresarial. En la División de Estudios de Posgrado e Investigación se ofrecen cinco maestrías: Maestría en Ingeniería Administrativa, Ingeniería Industrial, en Ciencias de la Ingeniería Química, Ingeniería Electrónica y Maestría en Sistemas Computacionales y un doctorado: Doctorado en Ciencias de la Ingeniería. Así como diplomados en diferentes áreas como apoyo a la actualización al personal de las empresas de la región, al haber consolidado la enseñanza a nivel superior; el Instituto Tecnológico de Orizaba implanta un sistema de calidad en busca de la excelencia educativa.

1.3. Planteamiento del problema

Las técnicas actuales de programación y de resolución de problemas, consideran que factores como la parte mental, la comunicación humana, los lenguajes de programación y en general los aspectos técnicos, son los más influyentes en la tradicional brecha que existe entre el dominio del problema y el dominio de la solución. Para atender esta problemática, diversos autores consideran que el estado actual de la programación ha alcanzado un nivel suficiente donde un cambio de paradigma permite desarrollar nuevos lenguajes de programación que reduzcan la brecha entre el dominio del problema y el dominio de la solución.

Un lenguaje naturalístico, al reducir la brecha entre los dominios del problema y de la solución, permite reducir los tiempos de desarrollo y disminuir la documentación requerida.

1.4. Objetivos

Esta sección describe el objetivo general junto con los objetivos específicos de este proyecto de tesis.

1.4.1. Objetivos

Aplicar ingeniería inversa al lenguaje de programación naturalístico SN para la reorganización de componentes, mejora de arquitectura y difusión del lenguaje, así como mejora la expresividad y la comunicación entre clientes y analistas.

1.4.2. Objetivos específicos

1. Revisar la arquitectura del lenguaje SN para conocer el estado actual de su diseño y organización.

2. Actualizar la documentación del lenguaje SN por medio de los artefactos necesarios a partir de la aplicación del proceso de ingeniería inversa.

3. Probar ejemplos previamente compilados para asegurar el correcto funcionamiento del lenguaje.

4. Depurar el sistema de instalación de SN para facilitar la difusión del lenguaje.

1.5. Justificación

Los sistemas de software en campos como la ciencia, la ingeniería y los negocios son naturalmente complejos por las estructuras y los procesos que capturan la complejidad específica del dominio, sin embargo, los mecanismos existentes de descripción de requerimientos resultan inadecuados para transmitir a las personas la información relevante para los sistemas de software, como resultado, el software se vuelve un frágil monumento de código que solo algunos se atreven a modificar, impidiendo así a los verdaderos expertos controlar el software de una manera directa.

El paradigma naturalístico ofrece la posibilidad de analizar, modelar e implementar software, preservando las ideas de los clientes con un menor grado de deformación, mejorando la documentación, con código fuente autodocumentado. Para tener estos beneficios a nivel industrial existe la necesidad de difundir el paradigma para que la comunidad de desarrolladores lo conozca y adquiera los conocimientos necesarios para su uso. El lenguaje SN debe realimentarse por lo cual es necesario ponerlo a disposición de forma pública.

Capítulo 2

Estado de la práctica

En este capítulo se dan a conocer los principales trabajos relacionados con el proyecto de tesis.

2.1. Trabajos relacionados

Los sistemas de software en campos como la ciencia, la ingeniería y los negocios son naturalmente complejos por las estructuras y los procesos que la capturan complejidad especifica del dominio, sin embargo, los mecanismos existentes de descripción resultan inadecuados para transmitir a las personas información relevante para los sistemas de software, como resultado, el software se vuelve un frágil monumento de código que solo algunos se atreven a modificar, impidiendo así a los verdaderos expertos controlar el software de una manera directa y sistemática. Los investigadores están constantemente buscando formas alternativas para expresar en forma más cercana a la manera en que se piensa, es por ello que a lo largo de los años varios lenguajes han sido diseñados específicamente enfocados en los problemas de usabilidad y expresividad, sin embargo, ninguno de esos lenguajes ha tenido éxito en el desarrollo de software, pues su principal problema es que los modelos de computación simplificados no son capaces de soportar las demandas de la programación de sistemas.

En [10] se abordaron algunos de los problemas relacionados con los lenguajes de programación existentes, sin embargo, aún se mantienen lejos de soportar las expresiones del lenguaje natural que se busca. No obstante, la programación orientada a aspectos ha dejado importantes lecciones con las cuales es posible alimentar las siguientes generaciones de lenguajes que soporten sistemas complejos.

En ese sentido, una de las características principales de los lenguajes naturales es el uso de una diversidad de relaciones anafóricas. Una anáfora es, esencialmente, referencialidad entre enunciados. Los pronombres son ejemplos de anáforas dependientes del contexto: esto, aquello, eso, ella, cual, etc. Pero las expresiones de referencia pueden ser más que pronombres. En un uso lingüístico general, la anáfora se refiere a la dependencia referencial independiente de la forma morfológica e indiferente de si depende o no del contexto.

Hay dos aspectos relacionados con la referencia: a qué referirse y cómo referirse a él. El lenguaje natural no tiene un conjunto extenso de cosas bien definidas a las cuales puede hacer referencia, de hecho, la mejor palabra para describir es cosa, la cual puede ser cualquier cosa, en el cómputo estos grupos podrían ser direcciones de memoria y registros, o funciones y variables, pero puede ir más allá, se pueden representar cosas tales como: un punto en el tiempo, “el párrafo previo” y “todas las secciones de este resumen”.

Sin embargo, los lenguajes naturales no son tan caóticos como parece. Las cosas tienden a caer en un pequeño número de clases. El desafío de tomar como base los lenguajes naturales para producir un lenguaje de programación es decidir cuáles cosas deben ser elementos referenciables en el contexto de la programación informática, dada la amplia gama de dominios de aplicación. Se debe mantener en mente que un lenguaje naturalístico debe tener una propiedad importante: permitir construir una abstracción compleja sobre una relativamente pequeña cantidad de abstracciones primitivas.

Las dificultades para capturar los requerimientos de sistemas generan, según se indica en [11], un brecha entre las técnicas de programación y cómo se quiere que realmente se comporten las computadoras. Esta brecha es causada por varios problemas:

- **El problema mental:** Desarrollar software consta de tres fases: pensar el qué, es decir, la idea, transformar la idea a una forma en la que el lenguaje de programación pueda entenderlo y finalmente escribirlo en el lenguaje de programación. Esta doble transformación presenta el grave problema de tener que adaptar las ideas a muy específicos lenguajes de programación.

- **El problema del lenguaje de programación:** Las ideas no cambian con el pasar del tiempo, un determinado algoritmo es siempre el mismo, sin embargo, tal algoritmo tiene que ser escrito una y otra vez para cada lenguaje. Podría no parecer relevante para algo como un algoritmo, sin embargo, para aplicaciones grandes definitivamente importa.

- **El problema del lenguaje natural:** Personas de distintos países participan hoy día en proyectos con personas que hablan distintos idiomas. Esto conduce a escribir, comentar y documentar el código usualmente en inglés lo que resulta ineficiente para todos los hablantes no nativos. Lo ideal es que cada quien pudiese escribir, comentar y documentar programas en su idioma nativo y que otros pudieran leer esos programas en sus propios idiomas nativos.

- **El problema técnico:** La capacidad de escribir programas informáticos en su idioma nativo también libera a los desarrolladores de tener que aprender una y otra vez nuevos lenguajes de programación. Una vez que una idea se expresa en lenguaje natural, es válida durante mucho tiempo. Ser capaces de escribir programas de forma natural es la clave para poder solucionar tanto el problema mental como el problema de los lenguajes de programación.

Considerando lo anterior, se desarrolla el prototipo Pegasus. La arquitectura de Pegasus tiene tres características básicas: leer el lenguaje natural, generar una salida de código de programación (Java) y expresar los resultados en otros idiomas del lenguaje natural. Pegasus tiene una instancia entre la lectura del lenguaje natural y la generación de código de programación: se llama a esta instancia el cerebro, el cual está conformado de tres partes:

- **La mente**: Tiene la tarea de entender el significado de la oración. Pegasus analiza la estructura gramatical de la oración. Después analizar la estructura global, cada cláusula se analiza por separado. En este momento se utiliza la gramática básica de Pegasus; usando esta gramática Pegasus puede identificar los predicados de las cláusulas en la oración, así como los sujetos, los objetos y los adverbios.

- **La memoria a largo plazo:** Es el diccionario que almacena la información léxica para cada idea, contiene sus representaciones como entidad, acción y propiedad en todos los lenguajes naturales respaldados por Pegasus. La memoria a largo plazo también almacena conocimiento semántico.

- **La memoria a corto plazo:** Al igual que la memoria a corto plazo humana, ésta se encarga de guardar el contexto actual. Para resolver una idea en el contexto actual, Pegasus busca en la memoria a corto plazo, comenzando por las entidades mencionadas recientemente.

Una vez que Pegasus ha resuelto toda la idea y ejecuta los resultados a través de una biblioteca de significados, genera la salida en un lenguaje de programación como Java. Al finalizar la ejecución Pegasus puede expresar las ideas leídas en todos los lenguajes naturales compatibles (inglés y alemán al momento de difundido el artículo). Los autores están seguros que la programación en lenguaje natural será una tendencia futura en el desarrollo de técnicas de programación, enfocadas hacia el pensamiento humano.

El trabajo con los lenguajes naturales continúa en [12], los autores conceptualizan la idea de tipos de lenguaje natural y posibles aplicaciones para su uso en programación. Los autores afirman que el lenguaje natural tiene el mismo objetivo que los lenguajes de programación; comunicar información, incluido el conocimiento y comportamiento. El uso de los lenguajes naturales en la programación prevé sacar provecho de las características esenciales de los primeros aplicando sus propiedades básicas, listadas a continuación:

- **Evitar redundancia**: Los lenguajes naturales reducen la cantidad de datos comunicados por unidad al evitar palabras redundantes; se tiende a expresar información con la menor

cantidad de palabras posibles. De esta manera, se omiten detalles innecesarios y se evita tener en la memoria conceptos temporales.

- **Localidad:** El lenguaje natural se estructura en declaraciones de tamaño limitado, oraciones y párrafos. Por lo general, existen referencias solo a elementos de la estructura actual o la anterior sin abordar elementos más distantes, a diferencia de los lenguajes de programación que pueden hacer referencia a variables definidas al comienzo de un fragmento largo de código.

- **Inmediación:** Evitar la cansada tarea de transformar las ideas de manera adecuada para un compilador o intérprete elegido, logrando que la máquina se adapte a los pensamientos humanos y no al revés.

Para programar es necesario elaborar una idea, reconvertirla y darle forma para que se adapte a las condiciones de un lenguaje de programación determinado, los desarrolladores se adaptan a las máquinas en lugar de que las máquinas se adapten a los desarrolladores. Para eliminar este paso intermedio entre la idea y el comportamiento, se tienen que revisar las técnicas de programación actuales e incorporar elementos naturalísticos como:

- **Referencias naturales:** Se basan en caracterizar a objetos por sus propiedades. “Tomar la manzana roja” donde la referencia natural de la manzana es su propiedad roja.

- **Generalización:** Expresar enunciados generales; hechos generales sobre propiedades o relaciones de objetos. “El gato es un animal”, “Una casa tiene techo”.

- **Descripciones de instancias:** Los tipos naturalísticos podrían servir como “constructores” ya que son capaces de iniciar una instancia al momento de crearla. “Una casa roja”, en este caso la asignación de la propiedad color es implícita omitiendo las redundantes asignaciones explícitas como “color=rojo”.

Los autores señalan que los tipos naturalísticos funcionan como constructores, donde una inicialización describe las cualidades que debe cumplir una instancia para considerarse de un

tipo, teniendo como ventaja la descripción implícita de propiedades según el tipo. Al mismo tiempo, los tipos naturalísticos permiten la existencia de objetos sin un tipo definido como una abstracción no concreta, un supertipo universal. El tipo naturalístico es un conjunto de cualidades, representado por predicados lógicos que todas las instancias deben completar para pertenecer a ese tipo. Su estructura es:

[cantidad] [propiedad] [concepto] [condición]

1. **Una cantidad:** Es un número escrito o representado. “1,2,3”; “uno, dos, tres”

2. **Una propiedad:** Es una cualidad atribuida, un adjetivo que representa una propiedad para los lenguajes naturales.

3. **Un concepto:** Es un “sustantivo” en los lenguajes naturales, cosas como “una casa”, “un tigre”.

4. **Una condición:** La condición se considera como una cualidad de tipo que permite codificar predicados arbitrarios, condiciones.

Los tipos naturalísticos, señalan los autores, también soportan la definición de características ausentes mediante la negación semántica de la propiedad; permitiendo mantener una relación binaria que sirve para proteger a las entidades de poseer una determinada propiedad y al mismo tiempo con una propiedad opuesta. En adición, los tipos naturalísticos tienen la flexibilidad con respecto a expresarse de forma exacta, relativa o abstracta. Las cantidades exactas se presentan mediante símbolos de número entero, las cantidades relativas soportan la referencia dada por el contexto donde se utilizan, las entidades abstractas permiten distinguir entre instancias plurales o singulares sin entrar en más detalles.

Por último, con respecto a los tipos naturalísticos, éstos poseen el soporte de condiciones que permiten codificar predicados arbitrarios. En un sistema de tipos naturalísticos es posible definir: A como subtipo de B, si y solo si, A es compatible con todos los tipos que constituyen a B.

Sin embargo, los tipos naturalísticos son nominativos en el sentido de que la relación jerárquica entre conceptos debe establecerse explícitamente. Independientemente de que un tipo posea las características necesarias para pertenecer al subtipo, no lo sería a menos que el contexto lo establezca.

Posteriormente en [13] se presentó un nuevo enfoque hacia la programación naturalística utilizando requisitos independientes del lenguaje. El enfoque adoptado toma requisitos en forma de casos de uso multilingües a partir de los cuales un generador de código produce el código de destino. La riqueza de este nuevo enfoque consiste en la implementación de código XML que representa la información de los escenarios de casos de uso.

Para resolver el código XML en código de lenguaje de programación, Java en este caso, el enfoque opera en tres etapas. Primero transforma los escenarios de casos de uso en patrones del modelo semántico creando las reglas de mapeo correspondientes. El modelo semántico permite que se utilice información semántica al explorar e interpretar la sintaxis de los requisitos, independientemente del lenguaje. En la segunda etapa el enfoque extrae la implementación de las reglas de mapeo generadas como un código XML siguiendo una estructura específica. En la tercera etapa el código XML resultante se utiliza para construir las entradas que solicita el generador de código ReDSeeDS. El generador ReDSeeDS transforma las entradas en un código Java siguiendo la arquitectura MVP (Modelo-Vista-Presentador).

Con las bondades del lenguaje natural ya conocidas y con herramientas desarrolladas hacia el mismo esfuerzo surge la pregunta: ¿Por qué la programación de todos los días no se realiza utilizando lenguajes naturales? En [14] se desarrolla una serie de planteamientos que exploran esta cuestión y encaminan a definir una propuesta conceptual. En [10] y [11] los investigadores han intentado utilizar lenguajes naturales para construir herramientas de desarrollo de software como las encontradas en [12] y [13]. Sin embargo, los investigadores se han enfrentado al hecho de que las computadoras no pueden lidiar con la ambigüedad de los lenguajes naturales que los lenguajes de programación han resuelto con especificaciones formales. En el razonamiento humano es el proceso cognitivo que ayuda a los programadores a resolver esta ambigüedad, ayuda a las personas a comprender y distinguir no solo los verbos y sustantivos, sino también

el significado en un contexto en particular. Como solución, se propone obtener un lenguaje de programación naturalístico, restringido y formalizado, para ser procesado por las computadoras.

En [14] se analizaron varias herramientas que utilizan una gramática controlada del idioma inglés con un medio para traducir, esta versión controlada, a un lenguaje de programación o a generadores de código de programación que controlan la ambigüedad. Sin embargo, los autores observaron que muchas de las tecnologías dependen de diccionarios de datos o bibliotecas precompiladas, lo cual reduce su uso a dominios particulares y aleja el objetivo de definir un modelo naturalístico de propósito general. También se encontraron lenguajes con alto nivel de expresividad, con un proceso cognitivo similar al humano para lidiar con la ambigüedad, no obstante, estos están limitados a dominios particulares y utilizan un lenguaje natural limitado.

En 2019 el equipo del Instituto Tecnológico Nacional de México, Campus Orizaba diseñó un lenguaje de programación naturalístico de propósito general, basado en el idioma inglés, llamado SN [6]. El lenguaje soporta el uso de referencias indirectas, trabaja con abstracciones complejas a partir de la composición de un sustantivo y uno o varios adjetivos que especifican el comportamiento de la abstracción. SN soporta bloques de instrucciones de repetición y bloques de instrucciones condicionales. La sintaxis de SN permite trabajar con instrucciones similares al lenguaje natural. SN es un prototipo de lenguaje naturalístico de proposito general que permite escribir código con un conjunto formalizado del idioma inglés incluyendo elementos del lenguaje natural. SN necesita refinarse, algunas de las limitaciones principales son: una ejecución lenta - debido al uso intensivo de *reflection*; la dependencia de los lenguajes intermedios Scala y AspectJ y que su API es básica y limitada.

En [15] se presenta el modelo conceptual, en el cual se basa SN, que describe los elementos necesarios para el diseño de lenguajes naturalísticos de propósito general que integren en su gramática: abstracciones, elementos temporales y referencias indirectas.

Los elementos mínimos que considera el modelo para la definición de un modelo naturalístico para el diseño de lenguajes de programación de propósito general son los siguientes:

1. Sustantivo como abstracción base con la que se identifica una “cosa” concreta, o un con-

junto de “cosas”.

2. Adjetivo como una abstracción que se emplea para complementar al sustantivo.

3. Verbo como acción que realiza un sustantivo.

4. Circunstancia como el conjunto de cosas que ocurren alrededor de un hecho, se le denomina circunstancia a los elementos que afectan de algún modo al significado de la oración.

5. Sintagma para definir mecanismos de composición que se basen en el uso de sintagmas nominales para definir entidades.

6. Definición de instrucciones en términos de elementos que se describieron con anterioridad (anáforas).

7. Definición de los tipos de forma explícita y estática, de modo que se utilicen en la construcción de instrucciones que se conformen con sintagmas.

Cabe destacar que dichos elementos sólo se consideraron para un lenguaje naturalístico que se enfoque en el idioma inglés con un nivel de expresividad que se encuentre más cercano a la forma de cómo los programadores expresan sus ideas, pero con la formalidad suficiente como para que una computadora lo procese sin ambigüedad.

Los elementos seleccionados permiten el diseño de otros lenguajes de programación naturalísticos de propósito general sin requerir diccionarios de datos u otras técnicas. SN, a partir del modelo implementado, contribuye a reducir la brecha entre los dominios del problema y la solución. SN genera como resultado un código auto-documentado. Como se mencionó anteriormente, SN necesita refinarse porque carece de la optimización necesaria para su uso en entornos reales de desarrollo.

En [1] se continúa el trabajo con el lenguaje prototipo SN. Los autores sabían que el principal obstáculo del lenguaje natural es la ambigüedad y la dependencia del contexto. Hasta ahora las computadoras procesan sin problema lenguajes de programación porque estos se basan en

formalismos que sin ambigüedad. Para controlar la ambigüedad del lenguaje natural se ha propuesto usar un modelo formalizado y restringido del lenguaje natural. En esta nueva revisión del lenguaje SN los autores señalan la necesidad del uso de plurares para colecciones de datos y los siguientes elementos del modelo que son deseables pero complementarios para un lenguaje naturalístico:

1. Soporte completo para la deixis, lo que implica que las instrucciones se definan en función de elementos que se describan no sólo antes, sino también después en el mismo texto.

2. Soporte para identificadores que permitan trabajar con las abstracciones por medio de referencias directas.

3. Tipificación basada en propiedades, lo que implica clasificar un sustantivo con base en sus propiedades y por tanto, que se tenga la capacidad para agregar nuevas propiedades a una abstracción cuando se cree.

Como trabajo a futuro los autores proponen realizar casos de pruebas más complejos que permitan dar mayor validez al modelo. Además, los autores proponen trabajar con la integración de mecanismos que permitan el uso de gramáticas embebidas dada la complejidad para la descripción de elementos de dominios específicos.

2.2. Análisis comparativo

La Tabla 2.1 presenta un anális comparativo de los trabajos relacionados descritos en la primer parte de este capítulo. En este análisis se observan las diferencias y similitudes entre los trabajos. La tabla consta de 5 columnas: artículo, problema, tecnologías utilizadas (simplemente Tecnologías, para abreviar), resultados y estado.

Tabla 2.1: Análisis comparativo del estado del arte

Artículo	Problema	Tecnologías	Resultados	Estado
[10]	En el desarrollo de software para determinados dominios son difíciles de capturar sus requisitos con las técnicas actuales. Principalmente por el problema mental de transformar la idea.	AspectJ	Expone la necesidad de un lenguaje naturalístico y plantea una propuesta de solución con las características deseables de dicho lenguaje.	Finalizado.
[11]	Tras establecer la necesidad de un lenguaje naturalístico, es necesario establecer los alcances de dicho lenguaje y establer los principios teóricos de un lenguaje naturalístico.	Pegasus NaturalJava AspectJ	Un modelo prototipo de lenguaje naturalístico que consta de tres elementos para su correcto funcionamiento y un informe para ejemplificar los posibles usos y funcionalidad	Finalizado.
Continua en la siguiente página				

Tabla 2.1: Análisis comparativo del estado del arte - continuación

	Problema	Tecnologías	Resultados	Estado
[12]	Las herramientas de definición de objetos a pesar de haber supuesto un avance histórico para la computación resultan insuficientes para las necesidades de la programación naturalística.	No menciona ninguna tecnología ocupada.	Realiza la definición de los tipos naturalísticos a fin de describir con precisión las características necesarias y características suficientes que conforman los tipos naturalísticos.	Finalizado.
[14]	Desde hace décadas diversos autores iniciaron el estudio de los posibles beneficios de utilizar elementos del lenguaje natural en los lenguajes de programación. Este estudio es un repaso por los principales conceptos y las principales tecnologías que utilizan en menor o mayor medidas elementos del lenguaje natural.	No menciona ninguna tecnología ocupada.	El estudio demostró una paulatina integración de distintos elementos naturalísticos en los lenguajes de programación, sin embargo, no existe una implementación suficiente de dichos elementos para afirmar la existencia de un lenguaje naturalístico.	Finalizado.
Continua en la siguiente página				

Tabla 2.1: Análisis comparativo del estado del arte - continuación

	Problema	Tecnologías	Resultados	Estado
[15]	Siendo conocidas las ventajas ofrecidas por los elementos del lenguaje natural, no hay lenguaje alguno que provea o incorpore tales elementos y provocan el estancamiento de las tecnologías.	AspectJ Scala ANTLR	Como resultado obtuvo un modelo funcional para abordar las necesidades y las áreas de oportunidad no cubiertas por las tecnologías existentes. Implementa el modelo de forma satisfactoria, pero con alcance limitado.	Mejoras a futuro.
[1]	Tras el éxito en la primera iteración de la implementación del modelo del lenguaje SN, es necesario enforcarse en los aspectos a fortalecer del mismo.	No menciona ninguna tecnología ocupada.	Este artículo presenta un modelo de programación naturalística de propósito general obtenido a partir de un análisis tanto de diversos trabajos, como de la propia estructura de la lengua inglesa.	Mejoras a futuro.

2.3. Solución propuesta

La propuesta de solución es un profundo análisis de la arquitectura de SN, así como de su documentación y proceso de instalación y configuración para generar una versión estable de SN que permita su difusión.

El proceso consta de cuatro fases principales para la resolución del problema planteado en el

capítulo 1 del presente trabajo.

2.3.1. Justificación de la solución seleccionada

Se seleccionó la siguiente propuesta, tomando en cuenta las necesidades del proceso de ingeniería inversa.

Revisión de la arquitectura del lenguaje SN

Esta fase consta de realizar una profunda revisión de la arquitectura del lenguaje SN y de su documentación a fin de comprender el estado actual del lenguaje, lo que permite obtener una visión concreta de las acciones a realizar a partir de las necesidades de SN.

Actualización de la documentación

A partir de los puntos de mejora encontrados durante la revisión de la arquitectura de SN, determinar los mecanismos de ingeniería inversa que mejor satisfagan las necesidades del lenguaje tanto para su arquitectura como para su documentación.

Evaluación de ejemplos previamente compilados

Con el fin de asegurar el correcto funcionamiento del lenguaje SN tras los cambios realizados a partir del proceso de ingeniería inversa, los ejemplos preexistentes para SN, requieren evaluarse y corregirse en caso de ser necesario a raíz de los cambios realizados.

Depuración de la instalación y configuración

Para facilitar la difusión del lenguaje SN, es necesario llevar a cabo una simplificación del proceso de instalación y configuración del lenguaje naturalístico SN.

Capítulo 3

Aplicación del proceso de ingeniería inversa

En este capítulo se presenta de forma detallada el desarrollo del proceso de ingeniería.

Como parte de este proceso y para cumplir con los objetivos establecidos se presenta como primer punto el análisis del proceso de instalación de SN para identificar los puntos de mejora del proceso.

Posterior al análisis de la instalación, se presenta una revisión de los elementos de documentación de SN, se analiza el dominio y el alcance de la documentación existente para identificar los elementos necesarios para complementar la documentación.

Por último se presenta la ejecución el proceso de ingeniería inversa abarcando sus diferentes fases (iteraciones).

3.1. Revisión del proceso de instalacíon de SN

Previo a comenzar la aplicación del proceso de ingeniería inversa se revisó el proceso de instalación SN, esta revisión tenía como objetivo identificar las dependencias necesarias para el correcto funcionamiento de SN. Adicionalmente se revisó la configuración necesaria de las

variables de entorno para el correcto funcionamiento de la fase de compilación y la fase de ejecución de SN.

3.1.1. Dependencias de SN

La información correspondiente a las dependencias de SN se determinó a partir de lo contenido en [16] (a partir de ahora "SN MANUAL"). En dicho manual se especifica que como requisitos de SN son necesarios:

1. Compilador Scala en su versión 2.12.3. Se especifica que el compilador no debe incluir sbt (sbt es una herramienta de compilación de código abierto para proyectos Scala y Java, similar a Maven y Ant de Apache).

2. AspectJ en su versión 1.8.10

   Con respecto a la versión de Java necesaria para la compilación y ejecución de código SN, no se especificó ninguna versión en particular; las pruebas realizadas se llevaron a cabo ceon Java en su versión 11 y en su versión 8.

3.1.2. Proceso de instalación de SN

El siguiente proceso de instalación se encuentra especificado en el "SN MANUAL".

1. Descomprimir el archivo **SN.zip** en la carpeta donde se desea instalar SN.

2. Crear la variable de entorno *SN_PATH* y agregar las direcciones donde se ubican los archivos **aspectjrt.jar** y **scala-library.jar** (carpeta *lib* de sus respectivos compiladores).

3. Agregar los compiladores *scalac* y *ajc* a la variable de entorno *PATH* o, agregar las direcciones donde se encuentran dichos compiladores (carpeta *bin* de cada uno).

4. Agregar la dirección *sn/bin* a la variable de entorno *PATH*.

El “SN MANUAL” no especifica alguna ruta particular en la que deba instalarse el compilador de Scala o AspectJ, por lo tanto la instalación de ambos programas se realizó de acuerdo al proceso de instalación por defecto.

En la Tabla 3.1 se especifican las rutas contenidas en la variable de entorno *PATH*, por otro lado la tabla 3.2 muestra las rutas contenidas en la variable de entorno *SN_PATH*.

Tabla 3.1: Rutas de la variable de entorno *PATH* según el manual de SN.

C:\aspectj1.8\bin
C:\Program Files\scala\bin
C:\sn\bin

Tabla 3.2: Rutas de la variable de entorno *SN_PATH* según el manual de SN.

C:\Program Files\scala\lib
C:\aspectj1.8\lib

Una vez configuradas las rutas tal y como se especifica en el manual, se procedió a compilar el programa de prueba “HelloWorld”:

```
main HelloWorld:
System prints "Hola Mundo" and newline.
```

Sin embargo, con la configuración contenida en el “SN MANUAL” al utilizar el comando *snc*, el cual tiene la función de compilar el código SN, no se recibía ninguna respuesta en la consola. Para solucionar temporalmente este inconveniente se optó por aumentar el grado de especificidad de las rutas en las variables de entorno.

Esta modificación se determinó a partir de consultar los archivos *sn.bat* y *snc.bat* ubicados en la carpeta *bin* de la instalación de SN. En estos archivos se identificaron las rutas consultadas mediante línea de comando al comento de compilar y ejecutar un programa en SN.

En el código 3.1 se aprecia en la línea *10* y en la línea *28* las rutas que se buscan en la variable de entorno *CLASSPATH* para la ejecución de un programa SN. Por otro lado, en 3.2 se especifica en la línea *9*, *25* y *30* la ruta, también contenida en *CLASSPATH*, de la dependencia necesaria para la compilación del código fuente SN.

Código 3.1: Código de **sn.bat**

```
1  @echo off
2  IF NOT %1 == -d (
3    GOTO VALIDATE_1
4  )
5  GOTO VALIDATE_D
6
7  :VALIDATE_1
8  IF EXIST %~dp1 (
9    REM scala -classpath "%~dp0..\lib\naturalistic-api.jar;c:\aspectj1.8\
       lib\aspectjrt.jar\;." %1
10   aj5 -classpath "C:\aspectj1.8\lib\aspectjweaver.jar;C:\Program Files\
       scala-2.12.3\lib\scala-library.jar;C:\Program Files\sn\lib\
       naturalistic-api.jar;." -javaagent:"C:\aspectj1.8\lib\aspectjweaver.
       jar" %1
11   GOTO END
12 )
13 GOTO END
14
15 :VALIDATE_D
16 IF %1 == -d (
17   GOTO VALIDATE_2
18 )
19 GOTO END
20
21 :VALIDATE_2
22 IF NOT EXIST "%3" (
23   GOTO END
```

```
)

IF EXIST "%2" (
  REM scala -classpath "C:\Program Files\sn\lib\naturalistic-api.jar;c:\aspectj1.8\lib\aspectjrt.jar" -sourcepath %2 %3
  aj5 -classpath "C:\aspectj1.8\lib\aspectjweaver.jar;C:\Program Files\scala-2.12.3\lib\scala-library.jar;C:\Program Files\sn\lib\naturalistic-api.jar;%2" -javaagent:"C:\aspectj1.8\lib\aspectjweaver.jar" %3
)
:END
```

Código 3.2: Código de **snc.bat**

```
@echo off
IF NOT %1 == -d (
  GOTO VALIDATE_1
)
GOTO VALIDATE_D

:VALIDATE_1
IF EXIST %~dp1 (
  java -classpath "%~dp0..\lib\naturalistic-api.jar" -jar "%~dp0..\lib\sn-compiler.jar" "%~dp1%1"
  GOTO END
)
GOTO END

:VALIDATE_D
IF %1 == -d (
  GOTO VALIDATE_3
)
GOTO END

```

```
:VALIDATE_3
IF "%3" == "" (
  GOTO VALIDATE_2
)

java -classpath "%~dp0..\lib\naturalistic-api.jar" -jar "%~dp0..\lib\sn-
    compiler.jar" "%~dp1%2\%3"
GOTO END

:VALIDATE_2
IF EXIST "%2" (
  java -classpath "%~dp0..\lib\naturalistic-api.jar" -jar "%~dp0..\lib\
    sn-compiler.jar" "%~dp1%2"
)
:END
```

En la Tabla 3.3 se muestran las modificaciones realizadas tras el análisis de los archivos *BASH*, se sustituyó la variable de entorno *SN_PATH* por la variable de entorno *CLASSPATH* y se especificaron los archivos de las dependencias requeridas para la compilación y ejecución de programas SN.

Tabla 3.3: Rutas de la variable de entorno *CLASSPATH* a partir de la información contenida en el archivo BASH de SN.

C:\Program Files\scala\lib\scala-library.jar
C:\aspectj1.8\lib\aspectjweaver.jar
C:\Program Files\sn\lib\naturalistic-api.jar

3.1.3. Compilación y ejecución de programas en SN

SN trabaja con archivos con extensión **.sn** por medio de los comandos *snc* y *sn*, el primer comando compila y genera los archivos ejecutables y el segundo permite ejecutar los programas. La sintaxis para la compilación es la siguiente:

snc <nombre del archivo>.sn

Para ejecutar un programa se requiere de una abstracción *main*, la cual genera un archivo *.class* que ejecute de la siguiente forma:

sn <Nombre de la abstracción main>

El nombre de la abstracción *main* en la forma de:

main <Nombre de la abstracción>:

Resalta el hecho que los programas sólo se pueden ejecutar mediante un punto de entrada, dicho punto de entrada es la abstracción *main*. En este sentido la única función de la abstracción *main* es servir de punto de acceso al programa.

Un detalle a tomar en consideración es que el estado actual del compilador sólo permite compilar un archivo a la vez, además solo permite trabajar con los archivos que se encuentran en la misma carpeta.

Para comprobar el correcto funcionamiento del compilador SN se ejecutaron los programas prueba proporcionados por [17], de los 12 ejemplos se seleccionaron 3 para comprobar el funcionamiento del compilador y el resultado de la ejecución de los programas SN.

Código 3.3: Código **1HelloWorld.sn**

```
1 main HelloWorld:
2   System prints "Hola mundo" and newline.
```

```
D:\Documentos\Maestría\Tesis\SN\Ejemplos>snc 1HelloWorld.sn
**************************
scalac.bat -classpath ".;c:\program files (x86)\scala\lib\scala-library.jar;c:\program files\sn
\lib\naturalistic-api.jar" -encoding Cp1252 -d "D:\Documentos\Maestría\Tesis\SN\Ejemplos" -nowa
rn "D:\Documentos\Maestría\Tesis\SN\Ejemplos\1HelloWorld.scala"
**************************
ajc.bat -classpath ".;c:\program files (x86)\scala\lib\scala-library.jar;c:\program files\sn\li
b\naturalistic-api.jar" -1.5
**************************

D:\Documentos\Maestría\Tesis\SN\Ejemplos>sn HelloWorld
Hola mundo
```

Figura 3.1: Compilación, ejecución y resultado del programa 3.3

En la Figura 3.1 se muestra la compilación y ejecución del programa 3.3 se define la abstracción “HelloWorld”. Posteriormente se le indica al sistema imprimir una cadena de texto y una nueva línea.

Código 3.4: Código **4Condicionales.sn**

```
noun Age:
  verb itself Verify years as Number:
    execute the next instruction when years is greater or equal than 18.
    System prints "Adult" and newline.
    execute the next instruction when years is lesser than 18.
    System prints "Younger" and newline.
main Condicionales:
  an Age.
  the Age Verify 15.
```

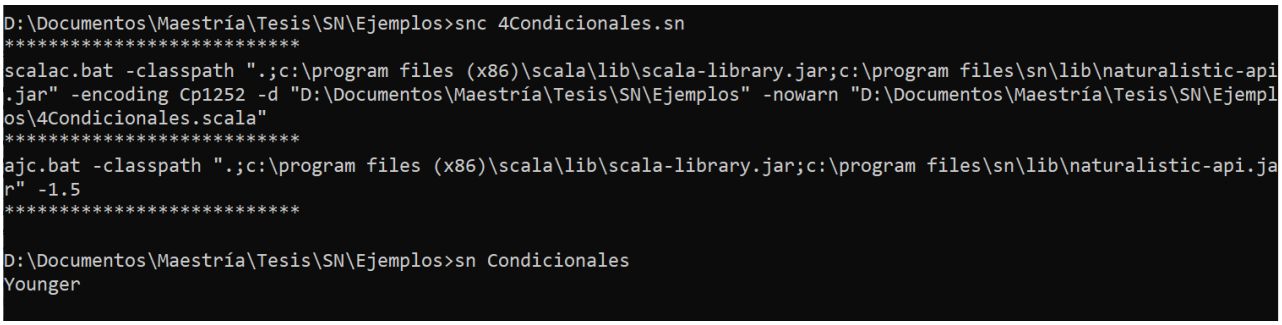

Figura 3.2: Compilación, ejecución y resultado del programa 3.4

En la Figura 3.2 se muestra la compilación y ejecución del programa 3.4 primero se define el *noun* (sustantivo) *Age* en el cual se define el verbo *Verify* como número el cual escribirá en consola *"Adult"* si el número es mayor o igual a 18 o *"Younger"* si el número es menor a 18.

Se crea la abstracción "Condicionales", se crea una instancia de edad y se verifica la edad 15, obteniendo como resultado en consola *"Younger"*.

Código 3.5: Código **8Formula.sn,**

```
noun Formula.
adjective Percentage:
  attribute quantity as a Real Number.
  attribute percent as an Integer Number.
  derived attribute result as a Real Number:
    aux is the real of percent; aux / 100; quantity * it; and return it.
adjective Speed:
  attribute distance as a Real Number.
  attribute time as an Real Number.
  derived attribute result as a Real Number:
    distance / time; and return it.
main MainFormula:
  a Speed Formula with 10 as distance and 5 as time; and System prints
   the result of it and newline.
  spf is an Speed Formula with 10 as distance and 5 as time; and System
   prints the result of spf and newline.
```

```
15  System prints the result of an Speed Formula with 10 as distance and 5
      as time and newline.
16  System prints the result of a Percentage Formula with 10 as percent
     and 500 as quantity and newline.
```

```
D:\Documentos\Maestría\Tesis\SN\Ejemplos>snc 8Formula.sn
**************************
scalac.bat -classpath ".;c:\program files (x86)\scala\lib\scala-library.jar;c:\program files\sn\lib\naturalistic-api
.jar" -encoding Cp1252 -d "D:\Documentos\Maestría\Tesis\SN\Ejemplos" -nowarn "D:\Documentos\Maestría\Tesis\SN\Ejempl
os\8Formula.scala"
**************************
ajc.bat -classpath ".;c:\program files (x86)\scala\lib\scala-library.jar;c:\program files\sn\lib\naturalistic-api.ja
r" -1.5 -sourceroots "D:\Documentos\Maestría\Tesis\SN\Ejemplos\." -nowarn -outxml
**************************

D:\Documentos\Maestría\Tesis\SN\Ejemplos>sn MainFormula
2
2
2
50.0
```

Figura 3.3: Compilación, ejecución y resultado del programa 3.5

En el programa 3.5 se define el *noun* (sustantivo) *Formula* y los *adjectives* (adjetivos) *Percentage* y *Speed*.

El adjetivo *Percentage* cuenta con los atributos numéricos *quantity*, *percent* y con el atributo numérico derivado *result*.

El adjetivo *Speed* cuenta con los atributos numéricos *distance*, *time* y con el atributo numérico derivado *result*.

En la abstracción *MainFormula*, en la línea 13, se define el adjetivo compuesto *Speed Formula*, se asignan valores a los atributos y se muestra en consola el atributo derivado *result*. Esta operación se repite en las líneas 14 y 15 con una sintaxis diferente, este ejemplo muestra las características de los lenguajes naturales que están presentes en SN gracias al paradigma naturalístico.

Finalmente en la línea 16 se define el adjetivo compuesto *Percentage Formula*, se asignan valores a sus atributos y se imprime el atributo derivado *result*.

3.2. Revisión del estado de la documentación de SN

La segunda etapa del proceso de ingeniería inversa consistió en una revisión de la documentación existente de SN y de su compilador. Por lo anterior se consideraron dos tipos de documentación:

1. Documentación para el usuario; la cual tiene como objetivo explicar al usuario final el funcionamiento del software, en el caso de SN, la documentación para el usuario corresponde a "SN MANUAL". Dicho documento contiene información de la instalación de SN, compilación y ejecución de programas SN.

2. Documentación de desarrollo; este tipo de documentación refiere a los documentos de ingeniería de software generados durante el diseño y desarrollo de un sistema. Por otro lado, también se considera los comentarios internos en el código que tienen como objetivo explicar el funcionamiento de segmentos concretos de código.

Como se mencionó anteriormente, con respecto a la documentación para el usuario se cuenta con "SN MANUAL" el cual proporciona información sobre el proceso de instalación de SN, así como compilación y ejecución de programas en SN. Adicionalmente, proporciona la sintaxis básica de SN junto con los elementos naturalísticos más destacados.

Sin embargo, este documento no muestra todas las capacidades del lenguaje. Por ejemplo, en [6] el autor muestra ejemplos de SN para la interacción con base de datos; en [16] no se mencionan esas capacidades, por lo cual se considera que [16] es insuficiente para dar a conocer SN, debido a que la información contenida en dicho manual no abarca todas las capacidades que el lenguaje ofrece, lo que genera una visión limitada sobre el potencial de SN.

Con respecto a la documentación de desarrollo, SN no cuenta con ningún producto de ingeniería de software correspondiente a su proceso de desarrollo. El código del compilador de SN tampoco se encuentra documentado en ninguna sección. De forma puntual se encuentra el nombre del desarrollador de SN, sin embargo, ningún segmento de código de encuentra documentado

por lo que llevar a cabo el análisis de las diversas secciones del compilador resulta complicado dado su estado actual.

3.3. Aplicación del proceso de ingenería inversa

En esta sección se detalla el proceso de ingeniería inversa llevado a cabo sobre SN, principalmente sobre el compilador de SN.

3.3.1. Ingenería inversa

"El término ingeniería inversa tiene su origen en el mundo del hardware. Una compañía desensambla un producto de hardware de otra empresa con la intención de entender los secretos de diseño y fabricación de su competidor. Dichos secretos podrían entenderse fácilmente si se obtuvieran las especificaciones de diseño y fabricación. Pero esos documentos son propiedad de la empresa competidora y no están disponibles para la compañía que hace la ingeniería inversa. En esencia, la ingeniería inversa exitosa deriva en una o más especificaciones de diseño y fabricación para un producto al examinar especímenes reales del mismo."[18]

Según [18] la ingeniería inversa para el software es muy similar. Los "secretos" por entender son oscuros porque jamás se desarrollaron especificaciones. Por tanto, la ingeniería inversa para software es el proceso de analizar un programa con la intención de crear una representación del mismo en un nivel superior de abstracción que el código fuente.

En [19] menciona a la ingeniería inversa como un proceso de reingeniería que *"tiene la misión de desestrañar los misterios y secretos de los sistemas en uso. Consiste principalmente en recuperar el diseño de una aplicación a partir del código"*.

En el trabajo [19], el autor señala que antes de comenzar propiamente las actividades de ingeniería inversa, el código fuente no estructurado ("sucio") debe reesctructurarse. Según [18] la reestructuración de código es el tipo más común de reingeniería, señala que aunque los sistemas heredados puedan tener una arquitectura sólida, si los módulos individuales se codificaron en una

forma que los hace difíciles de entender, probar y mantener, la reestruccturación es necesaria. El código fuente reestructura y la documentación interna se actualiza.

En el libro [20], el autor cita a Bjarne Stroustrup con respecto a qué significa un código limpio:

"I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well."

El autor también cita a Grady Booch en [20] con respecto al mismo asunto:

"Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control."

3.3.2. Reestructuración de código

Como primer paso para reestructuración de código se contabilizó la cantidad de líneas totales del código fuente de SN. En la tabla 3.4 se muestra la cantidad total de líneas, líneas de código, líneas comentadas y líneas vacias del compilador de SN. Esta primera medición se llevó a cabo para contrastar el impacto de la reestructuración del código previo a la documentación interna.

Tabla 3.4: Cantidad total de líneas, líneas de código, líneas comentadas y líneas vacias antes de la reestructuración.

líneas totales	líneas de código fuente	líneas comentadas	líneas en blanco
27387	20882	4462	2043

Para ejemplificar el estado inicial del código fuente en el código 3.6 se muestra las primeras líneas de código de la clase **snc.java**, en esta clase se encuentra el método *main* por lo que es el punto de inicio del proceso de compilación de un programa SN. En esta clase se encuentra el método *main*

Código 3.6: **snc.java** original de la línea 21 a la 90

```
1
2  public class snc {
3
4    private static String author = "Author: Oscar Pulido-Prieto";
5
6    /*private static ArrayList<String> className = new ArrayList<String>()
       ;
7    private static ArrayList<String> aspectName = new ArrayList<String>();
       */
8    private static boolean aspect = false;
9    //private static boolean main = false;
10   private static boolean single = false;
11
12   private static int quantity = 0;
13
14   private static ArrayList<String> fileNames = new ArrayList<String>();
15
16   public static void main(String[] args) {
17
18     //snc.redirectStream();
19
20     File f = new File(System.getProperty("java.class.path"));
21     File d = f.getAbsoluteFile().getParentFile();
22     classpath = d.toString();
23     //System.out.println("::: " + d);
24     //System.out.println("*****************************************");
25
26     //args = new String[] {"D:\\RESPALDOS DE ESCRITORIO CORTADOS\\
       lenguaje\\Pruebas LMAF\\Overloading0.sn"};
27     /**
28     NaturalisticLoader.seekClasses(NaturalisticLoader.getPaths());
29     System.err.println("-------------------------------------");
```

```
30      NaturalisticLoader.seekClasses(new String[] {classpath + "\\
    naturalistic-api.jar"});
31      System.exit(0);
32      */
33
34      // AQUÍ SE ABRE EL IDE
35      if(args.length == 0) {
36        /*System.out.println("Error: no file or dir");
37        System.exit(0);*/
38
39        main.ide.Window.initIDE();
40        /**throw new RuntimeException("Error: no file or dir");*/
41      } else {
42
43        compileFiles(args);
44      }
45
46  }
47
48  private static String classpath;
49
50  public static void compileFiles(String[] args) {
51
52    if(args[0].contains("\"")) {
53      //System.out.println(args[0]);
54      args[0] = args[0].replaceAll("\"", "\\\\");
55      //System.out.println(args[0]);
56    }
57
58    //NaturalisticLoader.seekClasses(new String[] {".\\naturalistic-api.
    jar"});
59    //NaturalisticLoader.seekClasses(new String[] {classpath + "\\
    naturalistic-api.jar"});
```

```
60
61      String[] classpathElements = NaturalisticLoader.getPaths();
62
63      NaturalisticLoader.seekClasses(classpathElements);
64
65      String localClasspath = ".";
66      for(String element : classpathElements) {
67        //System.out.println(element);
68        localClasspath += ";" + element;
69      }
70      //System.exit(0);
```

Durante el proceso de reestructuración de código se eliminaron las variables que no eran utilizadas en ningún punto del software, también se eliminaron las líneas de código comentadas, las cuales se asume corresponden a código descartado durante el desarrollo el compilador. De esta manera el código resultante posee un menor nivel de ambigüedad por lo cual resultó más sencillo el llevar a cabo el análisis de la funcionalidad del mismo.

En la tabla 3.5 se muestra la cantidad total de líneas, líneas de código, líneas comentadas y líneas vacias del compilador de SN al finalizar el primer paso del proceso de reestructuración de código. El código se reestructuró siguiendo las consideraciones de lo que un código limpio debe ser según [20].

Tabla 3.5: Cantidad total de líneas, líneas de código, líneas comentadas y líneas vacias al finalizar la reestructuración.

líneas totales	líneas de código fuente	líneas comentadas	líneas en blanco
22347	20882	1093	372

Como segundo paso del proceso de reestructuración de código se realizó la documentación del código del compilador de SN. Para este paso se tomaron en cuenta las consideraciones de Oracle en [21] para documentar código java con el objetivo de utilizar la herramienta de *JavaDoc*.

Código 3.7: **snc.java** de la línea 20 a la 90

```
1
2   /**
3   * Start point of SN compiler
4   *
5   * @author Oscar Pulido Prieto.
6   * @version 0.1
7   * @since 2019
8   */
9 public class snc {
10  /**
11  * This static variable indicates the presence of aspects in the
     program to be compiled.
12  */
13  private static boolean aspect = false;
14
15  /**
16   * Variable in charge of storing the address where the libraries
     needed to run the SN compiler (Scala and AspectJ) are located.
17   */
18  private static String classpath;
19
20  /**
21   * Store de CompilationUnit.
22   */
23  private static CompilationUnit cu;
24
25  /**
26   * Array containing the different files that compose the program to be
      compiled.
27   */
28  private static ArrayList<String> fileNames = new ArrayList<String>();
29
```

```
30  private static Stategy strategy;
31  /**
32   * Beginning of the SN compiler.
33   * In first instance it accesses the libraries necessary for the
     compilation of the programs in SN.
34   * The method can receive as parameter the .sn files to compile. In
     case it does not receive input files, it opens the SN IDE.
35   * @param args Files containing SN source code
36   */
37  public static void main(String[] args) {
38    File f = new File(System.getProperty("java.class.path"));
39    File d = f.getAbsoluteFile().getParentFile();
40    classpath = d.toString();
41    if(args.length == 0) {
42      main.ide.Window.initIDE();
43    } else {
44      compileFiles(args);
45    }
46
47  }
48
49  /**
50   * The SN compiler machinery
51   * Changes the input path format of the source files to regex.
52   * It gets the paths to the Scala and AspectJ libraries contained in
     the classpath. After loading the library paths
53   * it checks that the Scala and AspectJ versions are compatible with
     the SN compiler.
54   * It prepares the necessary Strings to execute the Scala (scalac) and
      AspectJ (ajc) compile command.
55   * From the array with all the files to be compiled, they are
     separated individually.
56   * After having the source files separated, it carries out the
```

```
    compilation through readfile/readfiles obtaining Scala and AspectJ
    code.
57  * By means of executeCommand the Scala code and the AspectJ code are
    compiled.
58  * Finally deleteClassFiles is invoked to delete the files with Scala
    code and AspectJ code.
59  * This process can be avoided by placing a file without extension
    called aux-file.
60  * @param args Source file to compile
61  */
62 public static void compileFiles(String[] args) {
63
64   if(args[0].contains("\"")) {
65     args[0] = args[0].replaceAll("\"", "\\\\");
66   }
67
68   String[] classpathElements = NaturalisticLoader.getPaths();
69   NaturalisticLoader.seekClasses(classpathElements);
70   String localClasspath = ".";
```

En el código 3.7 se muestra el mismo segmento que en 3.6. Como puede apreciarse, el código se encuentra limpio, sin comentarios de código residual y cada una de sus partes está según las recomendaciones de Oracle. Este cambio permite que el código fuente del compilador sea revisado y mantenido por otros desarrolladores en el futuro pues les brinda información relevante sobre el funcionamiento del compilador y cada una de sus partes.

3.3.3. Análisis de la arquitectura física del compilador SN

Por medio del proceso de ingeniría inversa se obtuvieron elementos de la arquitectura física del compilador, con el objetivo de conocer el estado actual del código. En la Figura 3.4 se muestra la representación de la arquitectura física del compilador, la cual consiste en dos componentes principales:

1. El paquete *main* que contiene las clases principales del compilador.

2. El paquete *naturalistic* que contiene las clases donde se definen las reglas de generación de código.

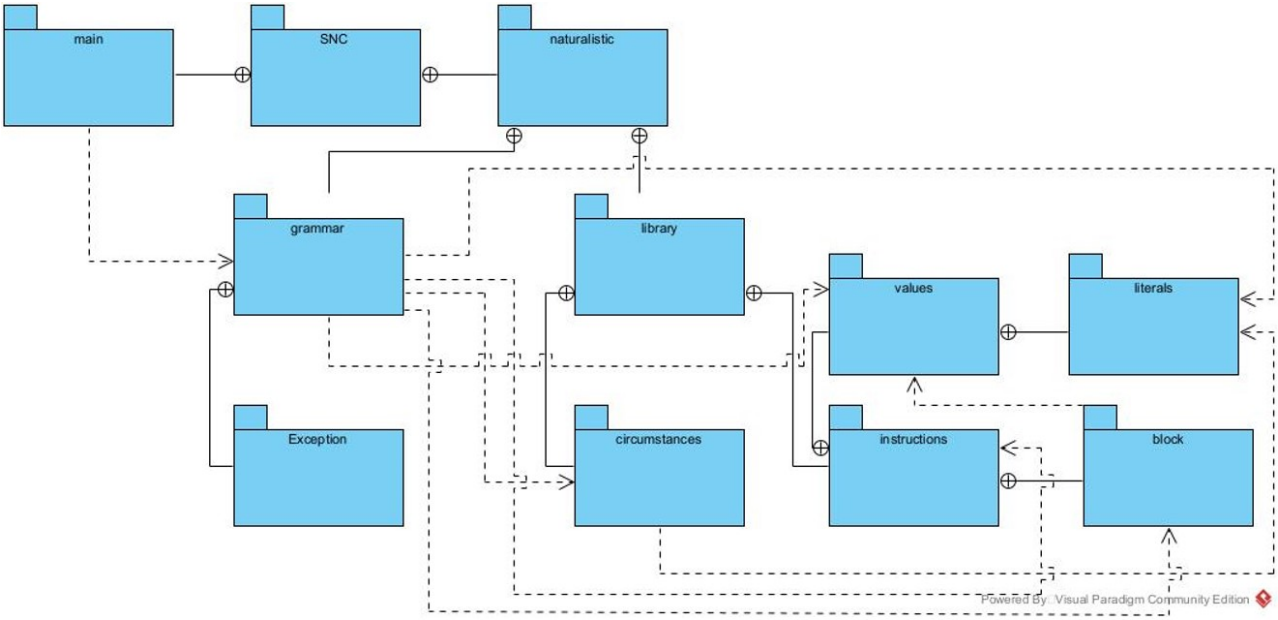

Figura 3.4: Diagrama de paquetes del compilador de SN

En la Figura 3.5 se muestra el paquete *library*, además de contener a los paquetes *circumtances* e *intructions*, este paquete contiene las clases para la generación de código de las abstracciones naturalísticas, sustantivos y adjetivos; la abstracción *main*; atributos singulares y plurales; verbos y la unidad de compilación.

En la Figura 3.6 se muestra el paquete *circumstances* que contiene las clases necesarias para la generación de código de las circunstancias. En SN una circunstancia es un mecanismo que permite establecer restricciones con base en qué adjetivos se permiten para la composición, qué adjetivos se requieren o qué adjetivos no se permiten [4].

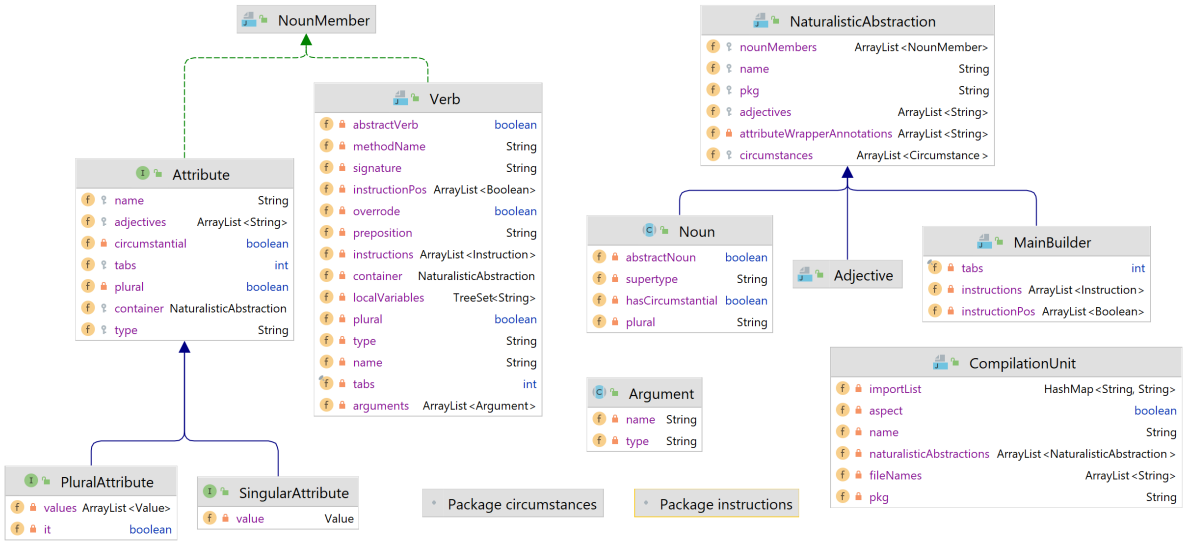

Figura 3.5: Diagrama de clases del paquete *library*

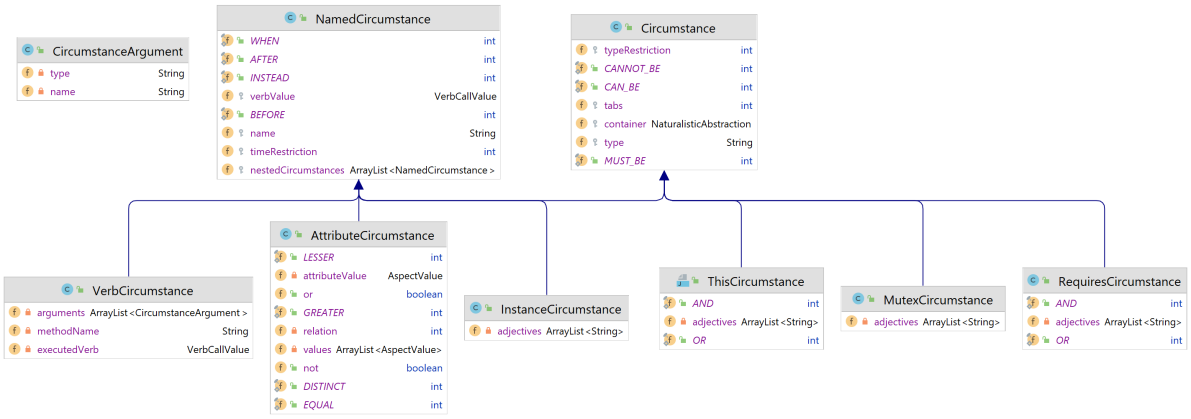

Figura 3.6: Diagrama de clases del paquete *circumstances*

Por otro lado, en la Fig 3.7 se muestra el paquete *instructions* que contiene a los paquetes *blocks* y *values* y contiene las clases de generación de código de las instrucciones. Internamente el compilador de SN etiqueta las instrucciones como *BasicInstrucion* o como *BlockInstruction*. Los bloques de instrucciones están formados por instrucciones básicas. Adicionalmente SN tiene

dos instrucciones que trata de forma independiente, dichas instrucciones son:

- EmbeddedGrammar[4]: Se utiliza para trabajar con gramáticas embebidas.
- VerbTypeInstruction: Se utiliza para las instrucciones que devuelven algún valor cuando son llamados.

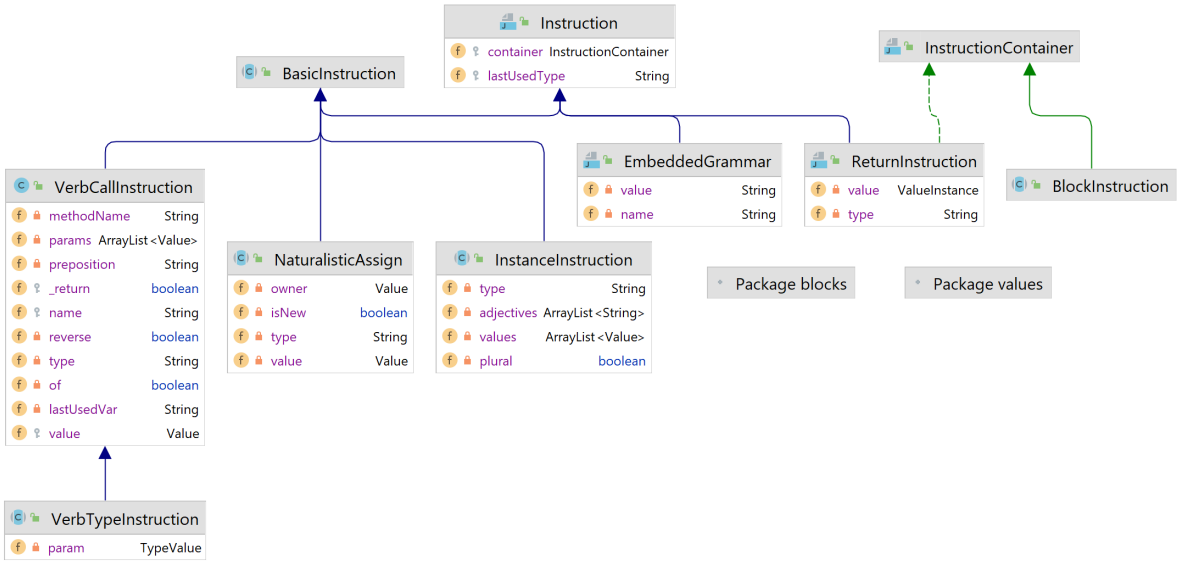

Figura 3.7: Diagrama de clases del paquete *instructions*

En la Figura 3.8 se muestra el paquete *blocks* que contiene las clases para gestionar los bloques de instrucciones, tanto de repetición como de decisión. Para ambos casos, los bloques de instrucciones se catalogan como *LineInstruction* o *MultipleInstruction*, los bloques de *MultipleInstruction* son bloques de instrucciones en los que cada instrucción del bloque está escrito en una línea diferente, en cambio los *LineInstruction*, aunque también trabajan con múltiples instrucciones, se caracterizan por estar escritas en una sola línea.

[4]El lenguaje SN se enfoca a describir expresiones naturalísticas, de modo que se requiere de un mecanismo que permita definir formalismos de un dominio particular para funcionar de forma adecuada, un mecanismo para describir instrucciones de un dominio particular por medio de gramáticas embebidas.

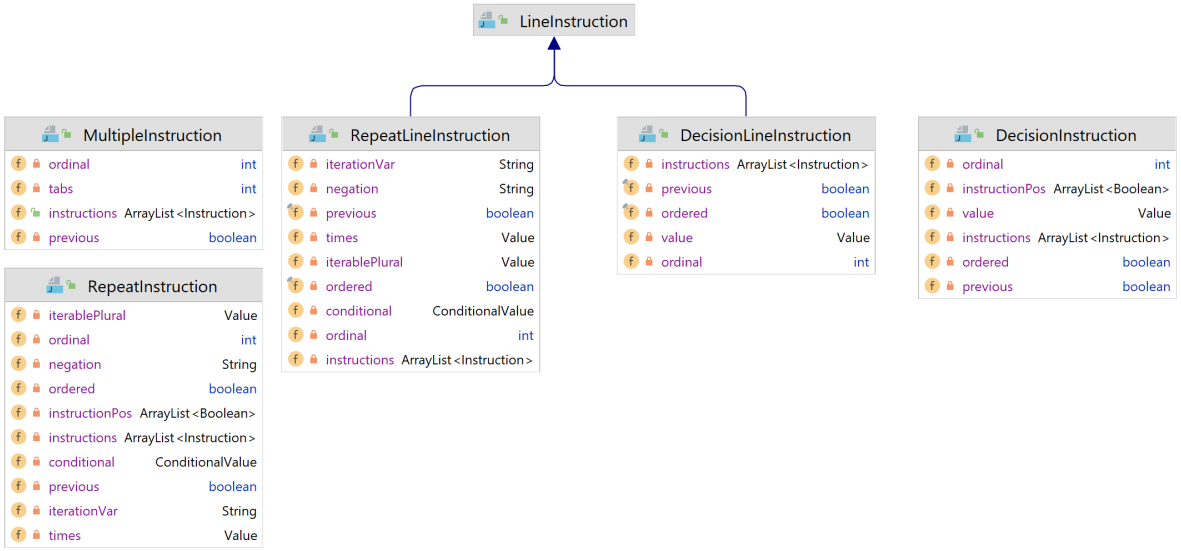

Figura 3.8: Diagrama de clases del paquete *blocks*

En la Figura 3.9 se muestra el paquete *values* que contiene al paquete *literals* y contiene las clases de generación de código necesarias para que SN trabaje con valores. Entre los elementos más destacables se encuentra la clase para trabajar con identificadores, la clase para definir el tipo del valor y las clases necesarias para trabajar con valores plurales[5].

[5]Se espera que los plurales posean atributos y verbos, pero dado que su definición depende del sustantivo al que representan, sus elementos se definen en el mismo sustantivo anteponiendo la palabra reservada plural.

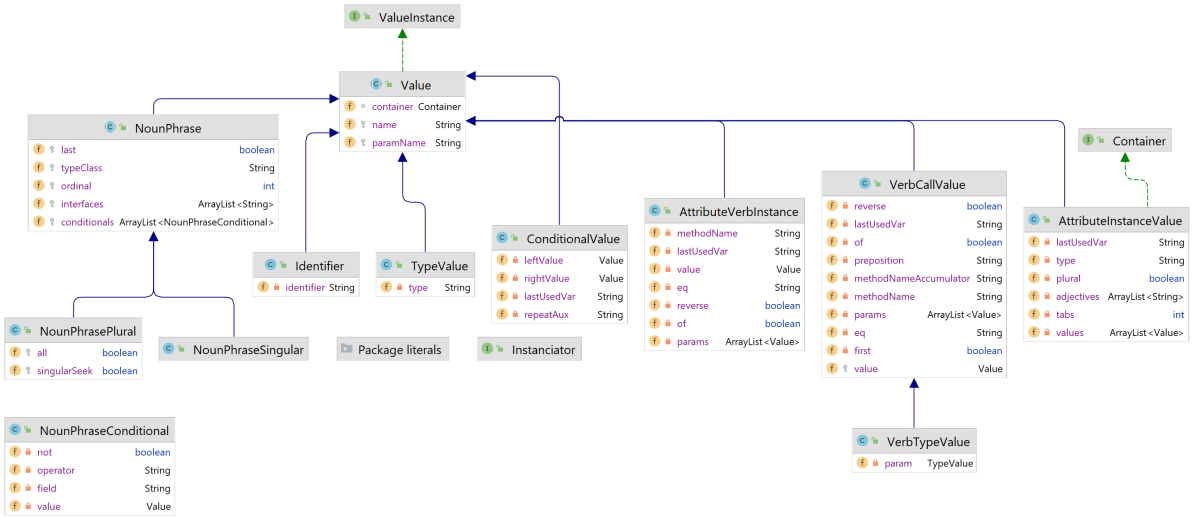

Figura 3.9: Diagrama de clases del paquete *values*

Finalmente, en la Figura 3.10 se muestra el paquete *literals* que contiene las clases para la generación del código de los tipos de datos implementados por defecto en SN.

Los literales[6] implementados en SN son:

- Boolean
- Character
- Integer
- Null
- Real
- String

[6]Una literal es un valor constante formado por una secuencia de caracteres. Cualquier declaración en SN que defina un valor constante -un valor que no cambie durante la ejecución del programa- es una literal.

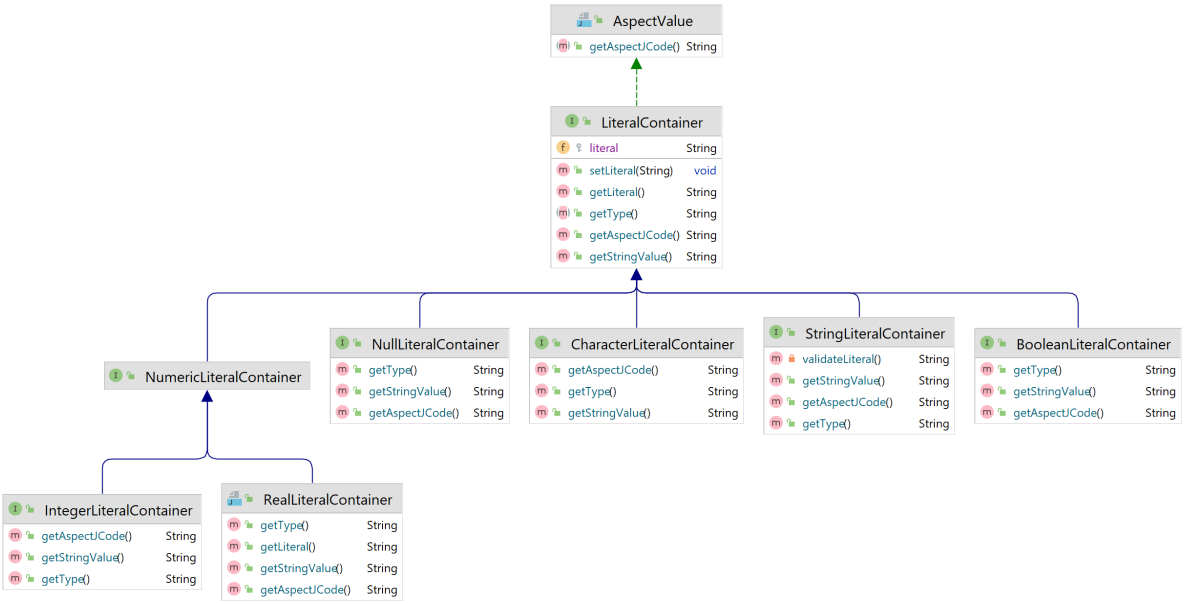

Figura 3.10: Diagrama de clases del paquete *literals*

3.3.4. Análisis de la arquitectura lógica del compilador SN

Se realizó un análisis de la arquitectura lógica del compilador mediante ingeniería inversa con el objetivo conocer el funcionamiento interno del compilador y comprender su funcionamiento. Como resultado de dicho análisis se obtuvo el esquema de arquitectura lógica del funcionamiento del compilador de SN. El compilador de SN se divide en tres etapas las cuales se describen a continuación.

Fase de análisis

La fase de análisis es el punto de inicio de la compilación, en esta fase el código fuente SN pasa por el analizador léxico y como resultado genera la tabla de tokens y la tabla de símbolos. Una vez finalizada esta fase, se lleva a cabo el análisis sintáctico del texto de entrada, de esta fase se obtiene el árbol de sintaxis abstracto (AST por sus siglas en inglés) del programa introducido.

Posteriormente, se lleva a cabo el análisis semántico del código fuente, este analizador realiza

la revisión de restricciones sintácticas de SN, adicionalmente, el análizador semántico crea una pila de datos la cual se utiliza para el soporte de referencias indirectas.

Una vez concluida la fase de análisis se obtiene como resultado un AST válido de SN y la tabla de símbolos. En la Figura 3.11 se presenta el esquema de la fase de análisis del compilador.

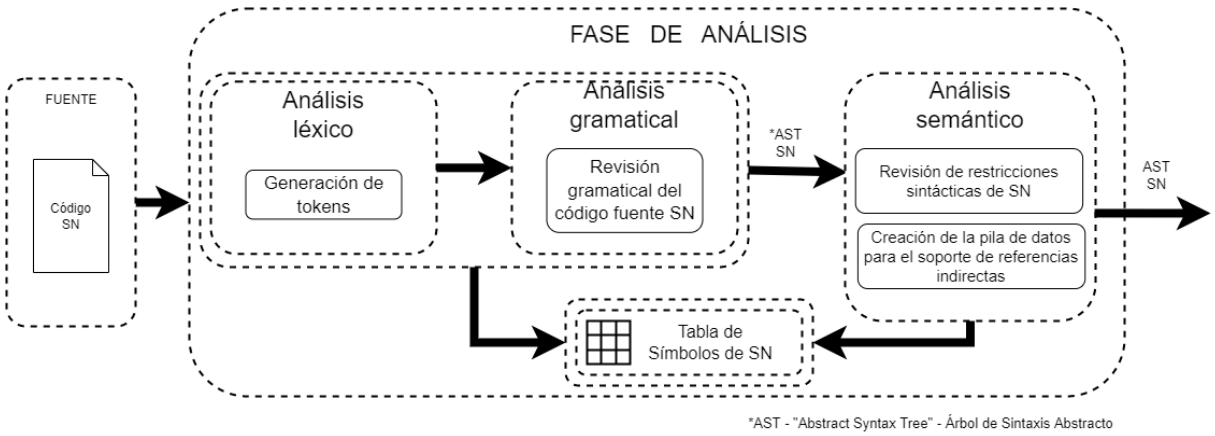

Figura 3.11: Arquitectura del compilador de SN - Fase de anaílisis

Fase de procesamiento intermedio

La fase de procesamiento intermedio comienza con el AST generado en la fase de análisis. En primera instancia de esta fase se genera la representación intermedia del programa SN: CompilationUnit.

Esta representación intermedia contiene diferentes estructuras de datos para gestionar los diversos componentes del programa SN. Entre las estructuras más destacables se encuentran:

1. *HashMap* ***importList*** usado para almacenar y gestionar la importación de los paquetes necesarios para el correcto funcionamiento del código intermedio.

2. *ArrayList* ***naturalisticAbstractions*** usado para contener todas las abstracciones naturalísticas.

3. *ArrayList* ***fileNames*** que contiene los nombres de los archivos de código intermerdio que será necesario generar.

Cada abstracción naturalística, sustantivos y adjetivos, se encargan de gestionar individualmente sus instrucciones. Adicionalmente la abstracción main se encarga de gestionar todas las instrucciones, y bloques de instrucciones, que no se encuentren contenidas en algún sustantivo o adjetivo. En la Figura 3.12 se presenta el esquema de la fase de procesamiento intermedio.

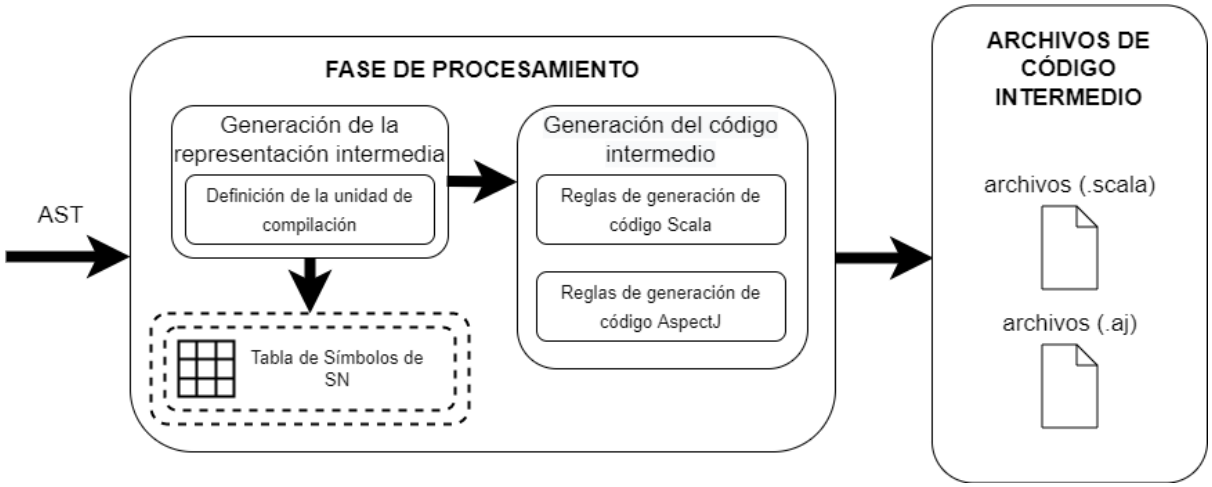

Figura 3.12: Arquitectura del compilador de SN - Fase de procesamiento intermedio

A partir de la representación intermedia, y por medio de las reglas de generacíon de código, se genera el código intermedio. Cada lenguaje, Scala y AspectJ, tienen sus propias reglas de generación individuales.

Como resultado de la fase de procesamiento se obtiene una serie de archivos Scala y AspectJ con los cuales comienza la etapa final del proceso de compilación de SN.

Fase de procesamiento a destino

La última fase del proceso de compilación de SN es la fase de procesamiento a destino. En esta fase se compilan, mediante comandos automatizados, los archivos Scala y AspectJ obtenidos en la fase de procesamiento intermedio.

Internamente tiene prioridad el código Scala el cual es el primero en compilarse, una vez que la compilación terminó correctamente, se aplican las condiciones descritas por las circunstancias.

Para las circunstancias el compilador utiliza programación orientada aspectos. El compilador traduce las circunstancias a código AspectJ.

Como resultado final de este proceso se obtienen varios archivos bytecode que se ejecutan mediante la Máquina Virtual de Java. En la Figura 3.13 se presenta el esquema de la fase de procesamiento a destino.

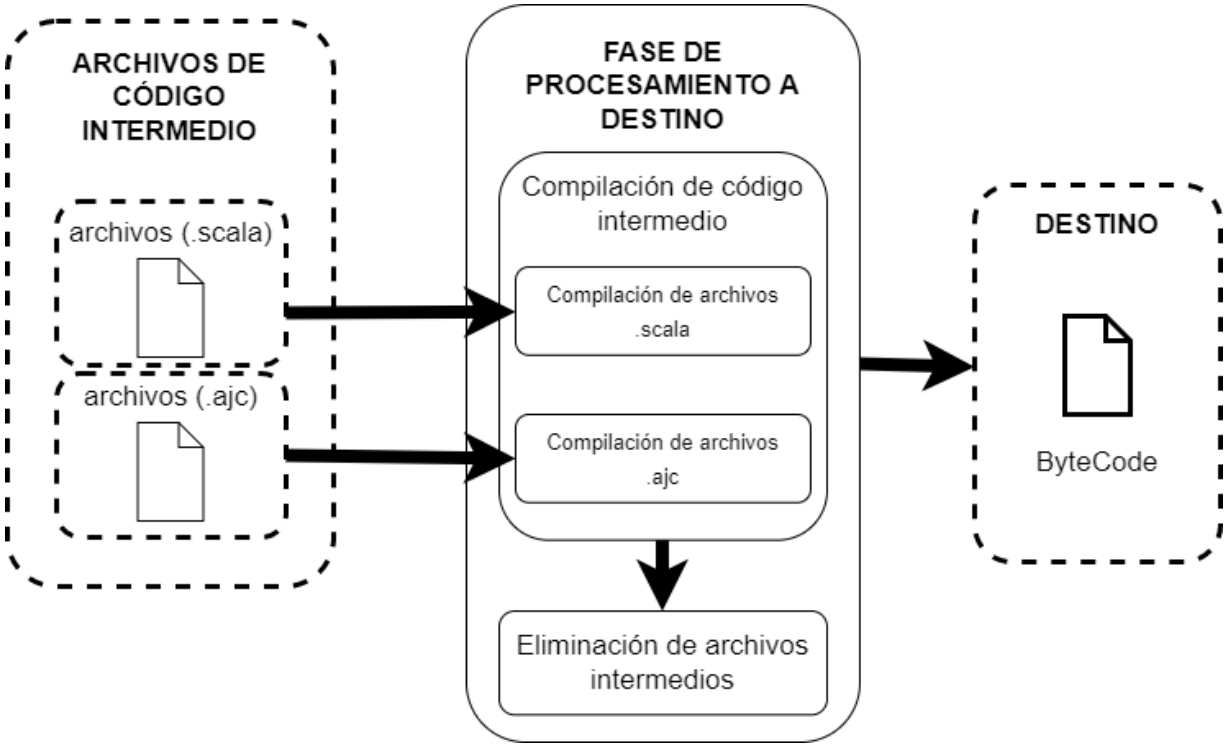

Figura 3.13: Arquitectura del compilador de SN - Fase de procesamiento a destino

Capítulo 4

Resultados

En este capítulo se presentan los resultados obtenidos en tres ejes principales: a) Refinamiento del código del compilador de SN; b) Mecanismos de difusión del lenguaje SN y c) exposición de las áreas de oportunidad y de trabajo a futuro para el lenguaje SN.

En el primer eje se destaca el propósito principal de este trabajo, el cual es presentar: una versión de SN que no solo permita su la difusión del lenguaje con la intención de dar a conocer las ventajas de la programación naturalística. Además se necesita una versión del compilador de la que se tenga información de su diseño a partir del cual identificar áreas de oportunidad y trabajo a futuro, esto con la intención de facilitar el continuo desarrollo del lenguaje para expandir sus capacidades. Para que esto sea posible es necesaria una versión de SN que cuente con una instalación sencilla; los problemas e inconsistencias en la ejecución del compilador SN suponen la primer barrera de entrada para quienes se interesen en trabajar con el lenguaje.

Como segundo eje de este capítulo se presentan todos los instrumentos generados para la difusión del lenguaje, entre los mecanismos generados se encuentran la web informativa de SN. Dicha página presenta un recorrido por las características del lenguaje, permite acceder a la descarga del compilador, muestra el proceso de instalación e incluye la documentación del compilador. Adicionalmente esta sección presenta elementos de apoyo en el aprendizaje y el entendimiento de SN.

El último eje muestra un análisis de la arquitectura y la gramática de SN, dicho análisis tiene como objetivo identificar las áreas de oportunidad para la mejora del lenguaje y determinar si la gramática logra la entendibilidad presupuesta de un lenguaje naturalístico. Además detalla el trabajo a futuro para continuar el desarrollo del lenguaje.

4.1. Refinamiento del código SN

En la sección 3.3 se abordó el resultado del proceso de ingeniería inversa aplicado sobre el código del compilador de SN. En dicha sección se describieron las etapas del proceso de ingenería inversa: la primer etapa consiste el limpiar el código fuente original para obtener un código correctamente documentado y organizado, el cual se usará en la segunda fase del proceso.

La segunda fase del proceso de ingeniería inversa consiste en extraer las abstracciones que conforman el software, dicho de otro modo, es obtener información de diseño y comportamiento, en este caso, del compilador SN. De esta fase se obtuvo tanto el diagrama de clases, diagrama de componentes y un esquema de funcionamiento del compilador.

Los productos de software obtenidos sirven para conocer el estado actual del compilador y a partir de dicha información establecer los puntos de mejora.

4.1.1. Principales problemas para la distribución de SN

En el marco de lo establecido por este tema de tesis, la mejora más relevante con respecto a la arquitectura del compilador corresponde a la fase de procesamiento a destino vista en la sección 3.13.

Al analizar el código responsable de compilar los archivos de los lenguajes intermedios destacan varios problemas:

1. La función *executeCommand* recibe una cadena de texto (instrucción de comando a partir de ahora) generada en otro punto del compilador, dividiendo de esa manera una sola responsabilidad en dos partes.

2. La estructura de la instrucción de comando solo es válida para el sistema operativo Windows.

3. Lo anterior limita a SN de ejecutarse en sistemas basados en Unix, como Linux o MacOs.

En el código 4.1 se muestra el método *executeCommand* original, en el cual se puede notar que solo funciona en un entorno de Windows porque el *ProcessBuilder* invoca una instancia de *CMD.exe*. Para aplicar correctamente el patrón de estrategia primero es necesario comprender, más allá de particularidades de cada sistema operativo, los pasos necesarios para compilar los archivos de código intermedio para comprender el proceso y el resultado esperado.

Al analizar el comportamiento del método se obtuvo la siguiente abstracción, que corresponde al funcionamiento general del método y del resultado esperado:

1. El método *executeCommand* recibe una variable final de tipo *String*, esta variable contiene la instrucción que debe ejecutarse para compilar el código intermedio, junto con una variable de tipo *boolean* que habilita la impresión de errores durante la compilación del código intermedio.

2. En primera instancia, el método distingue si el comando recibido pertenece a la compilación de archivos Scala o a la compilación de archivos AspectJ.

3. En ambos casos, procesa la cadena del comando para extraer el *String* que corresponde a la dirección donde se encuentra el archivo a compilar.

4. Con ambos elementos, el comando a ejecutar y la ruta donde se encuentra el archivo, se construye el *ProcessBuilder* que ejecutará el comando recibido como parámetro.

5. Se establece que el *ProcessBuilder* retorne los errores que se generen al ejecutar el comando de compilación y se construye el proceso dentro de una clausula *try-catch*.

6. Finalmente se lee el *Stream* de datos retornados por el proceso en busca de algún error en la ejecución, si se generaron errores en la compilación del código intermedio dichos errores se agregan al *Stack* de errores de Java, en caso contrario la compilación termina.

Código 4.1: Método *executeCommand* de la clase **snc.java**

```
1   BufferedReader r = null;
2   String line = null;
3   String aux = "";
4   String exe = "";
5   File batchDestiny = null;
6
7   if(command.contains("scalac.bat")) {
8     String auxDir = "";
9     exe = command.split("-d ")[1];
10    for(int i = 1; i < exe.length(); i++) {
11      auxDir += exe.charAt(i);
12      if(auxDir.endsWith("\" -nowarn")) {
13        auxDir = auxDir.substring(0, (auxDir.length()-9));
14        break;
15      }
16    }
17
18    aux = auxDir;
19    System.out.println(aux);
20    batchDestiny = new File(aux);
21
22  } else if(command.contains("ajc.bat")) {
23    String auxDir = "";
24    exe = command.split("-sourceroots ")[1];
25    for(int i = 1; i < exe.length(); i++) {
26      auxDir += exe.charAt(i);
27      if(auxDir.endsWith("\" -nowarn")) {
28        auxDir = auxDir.substring(0, (auxDir.length()-10));
29        break;
30      }
31    }
32    aux = auxDir;
```

```
33        System.out.println(aux);
34        batchDestiny = new File(aux);
35      }
36
37      System.out.println("Starting ProcessBuilder");
38      builder = new ProcessBuilder("cmd.exe", "/C", command);
39      System.out.println("cmd.exe" + "/C" +command);
40      builder.directory(batchDestiny);
41      System.out.println("Destiny of File" + batchDestiny);
42
43      builder.redirectErrorStream(true);
44      try {
45        pr = builder.start();
46        pr.waitFor();
47      } catch (IOException | InterruptedException e) {
48        System.out.println(e.getMessage());
49      }
50
51      if(print) {
52        r = new BufferedReader(new InputStreamReader(pr.getInputStream()))
    ;
53            while (true) {
54                try {
55            line = r.readLine();
56          } catch (IOException e) {
57            e.printStackTrace();
58          }
59                if (line == null) { break; }
60                main.ErrorStack.addError(line);
61            }
62      }
```

Otro elemento problemático es el método *compileFiles*, dicho método se encarga de invocar los elementos del compilador encargados de generar los archivos de código intermedio, es decir, los archivos con extensión *.scala* y *.aj*. Adicionalmente, y sobrepasando sus responsabilidades, el método genera la instrucción de comando utilizada por el método *executeCommand*. Para limitar las responsabilidades de cada método se decidió aislar los segmentos de código encargados de generar la instrucción de comando para compilar el código intermedio.

En el código 4.2 se muestra el método *compileFiles* original, este método construye la cadena de texto que posteriormente el método *executeCommand* utiliza para compilar los archivos del lenguaje intermedio.

Código 4.2: Método *compileFiles de la clase* **snc.java**

```
1
2     if(args[0].contains("\"")) {
3       args[0] = args[0].replaceAll("\"", "\\\\");
4     }
5
6     String[] classpathElements = NaturalisticLoader.getPaths();
7     NaturalisticLoader.seekClasses(classpathElements);
8     String localClasspath = ".";
9
10    for(String element : classpathElements) {
11      localClasspath += ";" + element;
12    }
13
14    String compiledSFile = "scalac.bat" + (" -classpath \"" +
    localClasspath + "\"") + " -encoding Cp1252 ";
15    String compiledAJFile = "ajc.bat" + (" -classpath \"" +
    localClasspath + "\"") + " -1.5 ";
16
17    String dir = "";
18    int length = 0;
19
```

```
20      if(args[0].endsWith(".sn")) {
21        length = args[0].split("\\\\").length-1;
22      } else {
23        length = args[0].split("\\\\").length;
24      }
25      for(int i = 0; i < length; i++) {
26        dir += args[0].split("\\\\")[i] + "\\";
27      }
28      if(dir.length() > 0) {
29        dir = dir.substring(0, dir.length()-1);
30      }
31
32      if(args[0].endsWith(".sn")) {
33        try {
34          readFile(args[0]);
35        } catch(RuntimeException | IOException e) {
36          e.printStackTrace();
37        }
38        compiledSFile += "-d \"" + dir + "\" -nowarn \"" + args[0].
    replaceAll("\\.sn", ".scala") + "\"";
39      } else {
40        try {
41          if(args[0].endsWith(".sn\\")) {
42            readFile(args[0].substring(0, args[0].length()-1));
43          } else if(args[0].endsWith(".sn")) {
44            readFile(args[0].substring(0, args[0].length()-1));
45          } else {
46            readFiles(args[0]);
47          }
48        } catch (RuntimeException | IOException e) {
49          // TODO Auto-generated catch block
50        }
51        compiledSFile += "\" -d \"" + dir + "\" -nowarn \"" + args[0] + "
```

```
    \\*.scala\"";
52   }
53
54   if(aspect) {
55     compiledAJFile += "-sourceroots \"" + dir + "\\.\" -nowarn -outxml
   ";
56     executeCommand(compiledAJFile, true);
57   }
58   executeCommand(compiledSFile, true);
59
60   System.out.println("***************************");
61       System.out.println(compiledSFile);
62       System.out.println("***************************");
63   System.out.println(compiledAJFile);
64   System.out.println("***************************");
65
66   // BORRAR LOS ARCHIVOS DE SCALA
67   if(/*!new File(classpath + "\\aux-file").exists()*/false) {
68     deleteClassFiles(cu, ".scala");
69     deleteClassFiles(cu, ".aj");
70   }
71 }
72
73 /**
74  * Clean the input path to create a valid format of path string
```

4.1.2. Implementación del patrón de diseño Estrategia

Para solucionar este problema se implementó el patrón de diseño: Estrategia. Dicho patrón sugiere tomar el elemento que necesita hacer algo de formas diferentes y se coloque en clases separadas llamadas *estrategias*.

La clase base, llamada contexto, debe tener un campo para almacenar una referencia a la

estrategia necesaria. El contexto delega el trabajo a un objeto de estrategia en específico vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase *Contexto*. De hecho, la clase *Contexto* no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, de tal forma es posible añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.

En el compilador de SN, cada algoritmo de ejecución de instrucciones de comando para la compilación de los archivos de lenguaje intermadio puede extraerse y ponerse en su propia clase con el único método *executeCommand*. Incluso contando con los mismos argumentos, cada clase de generación de comando devuelve un resultado diferente. La clase principal del compilador selecciona la estrategia activa a partir del sistema operativo detectado, la Figura 4.1 muestra el diagrama de clases de la implementación de este patrón de diseño, mientras tanto en la Figura 4.2 se observa el diagrama de clases del paquete main del compilador de SN con los cambios implementados.

La estructura de la solución propuesta es la siguiente:

1. La clase *snc* mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz *Strategy*.

2. La interfaz *Strategy* es común a todas las estrategias concretas. Declara un método que la clase *snc* utiliza para ejecutar una estrategia.

3. Las Estrategias Concretas implementan distintas variaciones de un algoritmo que la clase *snc* utiliza.

4. La clase *snc* invoca el método de ejecución en el objeto de estrategia vinculado cada vez

que necesita ejecutar el algoritmo. La clase *snc* no sabe con qué tipo de estrategia funciona o cómo se ejecuta el algoritmo.

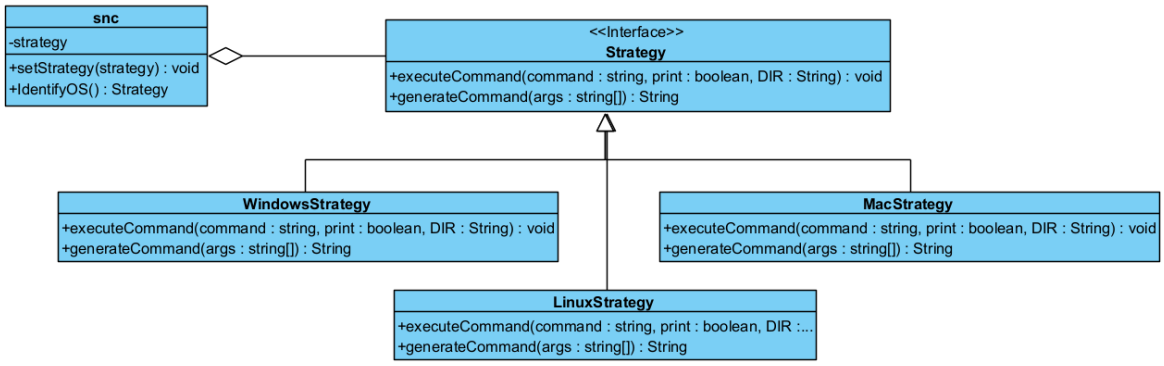

Figura 4.1: Diagrama de clases (simplificado) con la implementación del patrón *Estrategia*.

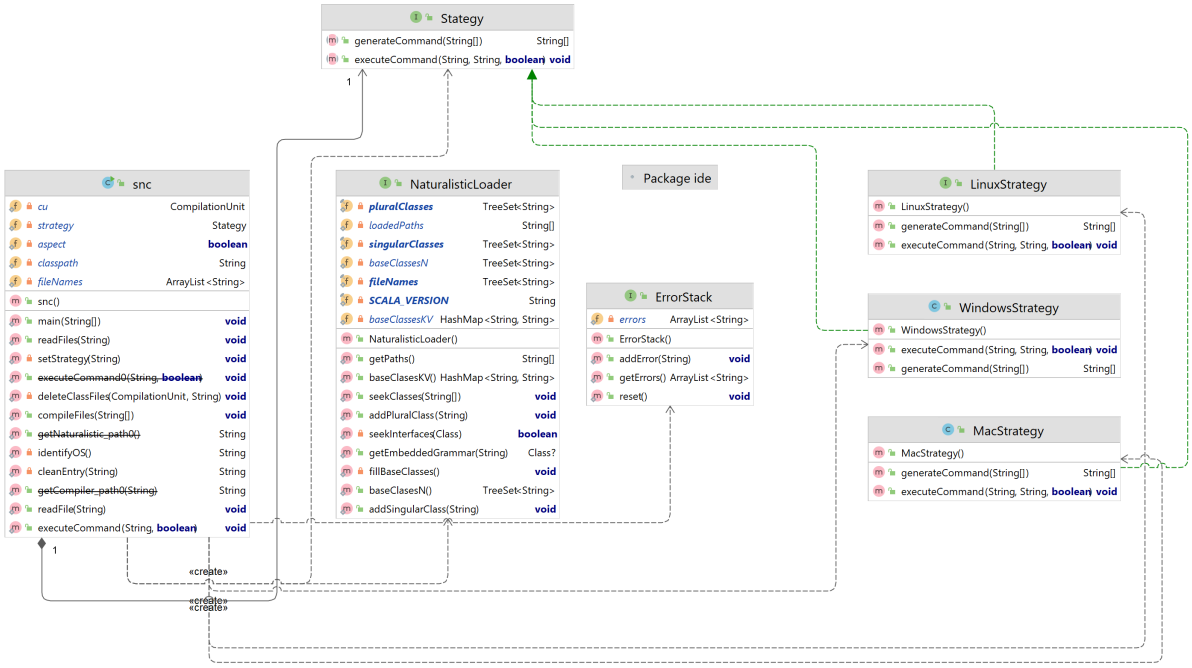

Figura 4.2: Diagrama de clases del paquete *main* con los cambios implementados *Estrategia*.

Además de la modificación necesaria al método *executeCommand* para adaptarse a los distintos ambientes de ejecución, se identificó la posibilidad de simplificar dicho método. Recibir como parametro la dirección, que el método tiene que obtener internamente, permite reducir el código aproximadamente un 40 %. El código 4.3 muestra el resultado de este proceso.

Código 4.3: Método *executeCommand* simplificado

```
1    public static void executeCommand(final String command, String Path,
     boolean print) {
2        Process pr = null;
3        ProcessBuilder builder = null;
4        BufferedReader r = null;
5        String line = null;
6        String aux = "";
7        File batchDestiny = null;
8
9        aux = Path + "\\";
10       batchDestiny = new File(aux);
11
12       builder = new ProcessBuilder("cmd.exe", "/C", command);
13       builder.directory(batchDestiny);
14
15       builder.redirectErrorStream(true);
16       try {
17           pr = builder.start();
18           pr.waitFor();
19       } catch (IOException | InterruptedException e) {
20           System.out.println(e.getMessage());
21       }
22
23       if(print) {
24           r = new BufferedReader(new InputStreamReader(pr.
   getInputStream()));
25           while (true) {
```

```
                try {
                    line = r.readLine();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                if (line == null) { break; }
                main.ErrorStack.addError(line);
            }
        }
    }
```

Se generó el método *generateCommand* y el método *recoverPath* a partir del código del método *compileFiles*, de este modo cada método se encarga únicamente de sus responsabilidades, lo que permite mantener el código mejor organizado y facilita su mantenimiento futuro.

Los códigos 4.4 y 4.5 muestran el resultado de la separación de funciones. Ahora cada método cumple con una función específica lo que simplifica su entendimiento y facilita su mantenimiento.

Código 4.4: Método *generateCommand* modificado

```
    public String[] generateCommand(String[] args, boolean aspect,
String dir){
        String[] classpathElements = NaturalisticLoader.getPaths();
        NaturalisticLoader.seekClasses(classpathElements);
        String localClasspath = ".";

        for(String element : classpathElements) {
            localClasspath += ";" + element;
        }

        String compiledSFile = "scalac.bat" + (" -classpath \"" +
localClasspath + "\"") + " -encoding Cp1252 ";
        String compiledAJFile = "ajc.bat" + (" -classpath \"" +
localClasspath + "\"") + " -1.5 ";
```

```
12
13      if(args[0].endsWith(".sn")) {
14          compiledSFile += "-d \"" + dir + "\" -nowarn \"" + args[0].
    replaceAll("\\.sn", ".scala") + "\"";
15      } else {
16          compiledSFile += "\" -d \"" + dir + "\" -nowarn \"" + args
    [0] + "\\*.scala\"";
17      }
18
19      if(aspect) {
20          compiledAJFile += "-sourceroots \"" + dir + "\\.\" -nowarn -
    outxml";
21      }
22
23      return new String[]{compiledSFile,compiledAJFile};
24   }
```

Código 4.5: Método *recoverPath*

```
1    public String recoverPath(String[] args){
2        String dir = "";
3        int length = 0;
4        String fileName = null;
5
6        if(args[0].endsWith(".sn")) {
7            length = args[0].split("\\\\").length-1;
8            fileName = args[0].split("\\\\")[args[0].split("\\\\").
    length-1];
9        } else {
10           length = args[0].split("\\\\").length;
11       }
12       for(int i = 0; i < length; i++) {
13           dir += args[0].split("\\\\")[i] + "\\";
14       }
```

```
15          if(dir.length() > 0) {
16              dir = dir.substring(0, dir.length()-1);
17          }
18
19          return dir;
20      }
```

Finalmente, el código 4.6 muestra al método *compileFiles* después de aplicar el patrón de diseño, se observa que ahora el método es mucho más compacto y que llama a los distintos métodos de la estrategia.

Código 4.6: Método *compileFiles* modificado

```
1           public static void compileFiles(String[] args) {
2
3               if(args[0].contains("\"")) {
4                   args[0] = args[0].replaceAll("\"", "\\\\");
5               }
6
7               String path = strategy.recoverPath(args[0]);
8               String[] Commands = strategy.generateCommand(args[0], aspect
    , path);
9
10              if(args[0].endsWith(".sn")) {
11                  try {
12                      readFile(args[0]);
13                  } catch(RuntimeException | IOException e) {
14                      e.printStackTrace();
15                  }
16              } else {
17                  try {
18                      if(args[0].endsWith(".sn\\")) {
19                          readFile(args[0].substring(0, args[0].length()
    -1));
20                      } else if(args[0].endsWith(".sn")) {
```

```
                        readFile(args[0].substring(0, args[0].length()
-1));
                } else {
                    readFiles(args[0]);
                }
            } catch (RuntimeException | IOException e) {
                //
            }
        }

        if(aspect) {
            strategy.executeCommand(Commands[1], path, true);
        }
        strategy.executeCommand(Commands[0], path, true);

        System.out.println("***************************");
        System.out.println(Commands[0]);
        System.out.println("***************************");
        System.out.println(Commands[1]);
        System.out.println("***************************");

        // BORRAR LOS ARCHIVOS DE SCALA
        if(!new File(classpath + "\\aux-file").exists()) {
            deleteClassFiles(cu, ".scala");
            deleteClassFiles(cu, ".aj");
        }
    }
```

4.2. Difusión del compilador SN

El segundo aspecto más importante para consolidar el paradigma naturalístico es: difundirlo. Para ello es necesario contar con una versión estable del compilador. Por lo tanto, el segundo eje

de este proyecto de tesis es establecer los elementos necesarios para difundir el lenguaje; entre los elementos generados se engloban: la web de SN, el repositorio que contiene el proyecto, ejemplos e información sobre el funcionamiento de SN y cómo crear software con el lenguaje.

4.2.1. Web de SN

Comenzando con el planteamiento de la Web de SN, este apartado muestra el diseño sugerido de la página del lenguaje, dejando de lado la implementación concreta dado que la implementación puede cambiar con el tiempo. Es por ello que en esta sección se establecen los criterios considerados para la construcción de la página web de SN. La Figura 4.3 muestra el mapa de sitio considerado para la Web de SN.

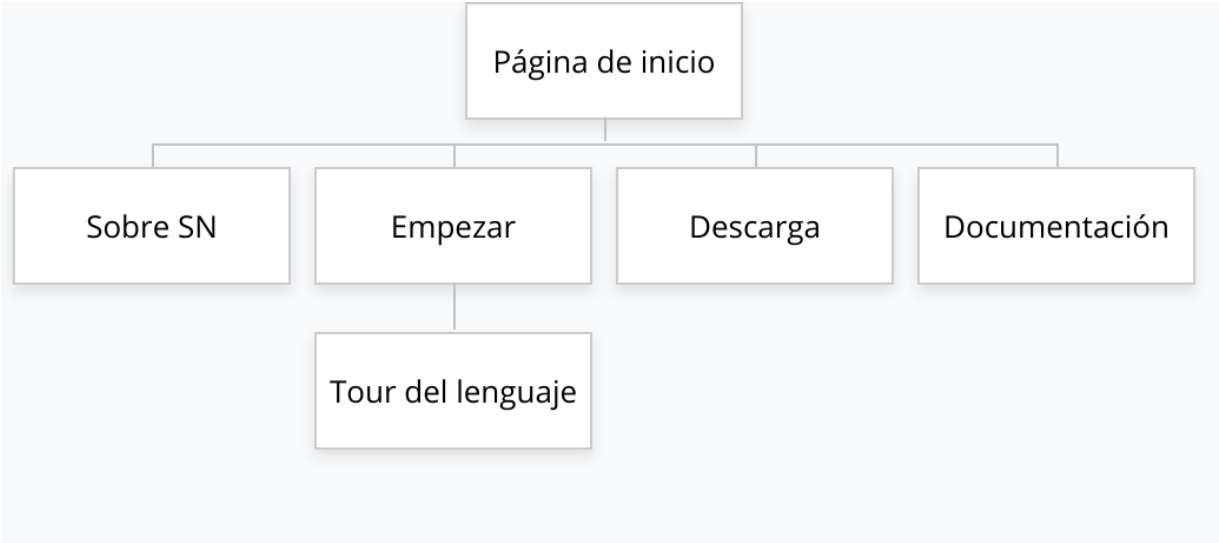

Figura 4.3: Mapa de sitio de la página web de SN.

Página de inicio

La intención de la web de SN es ser la puerta de entrada al lenguaje, por lo cual se considera que la página de inicio debe resaltar dos ejes importantes: aprender y comenzar. En otras palabras, la página de inicio debe procurar simplificar las opciones para el usuario; por un lado debe

invitar a empezar a conocer el lenguaje mediante la opción "Get Started", por el otro debe dirigir directamente a la descarga del compilador.

Simplificando de esta manera las opciones de la página principal, la página no distrae ni complica la navegación del usuario, por ejemplo, los usuarios nuevos pueden elegir la opción de empezar a conocer el lenguaje mientras que usuarios más avanzados pueden directamente descargar el compilador. En la Figura 4.4 se muestra el bosquejo de la página de inicio de la web de SN, considerando los elementos deseados de navegación.

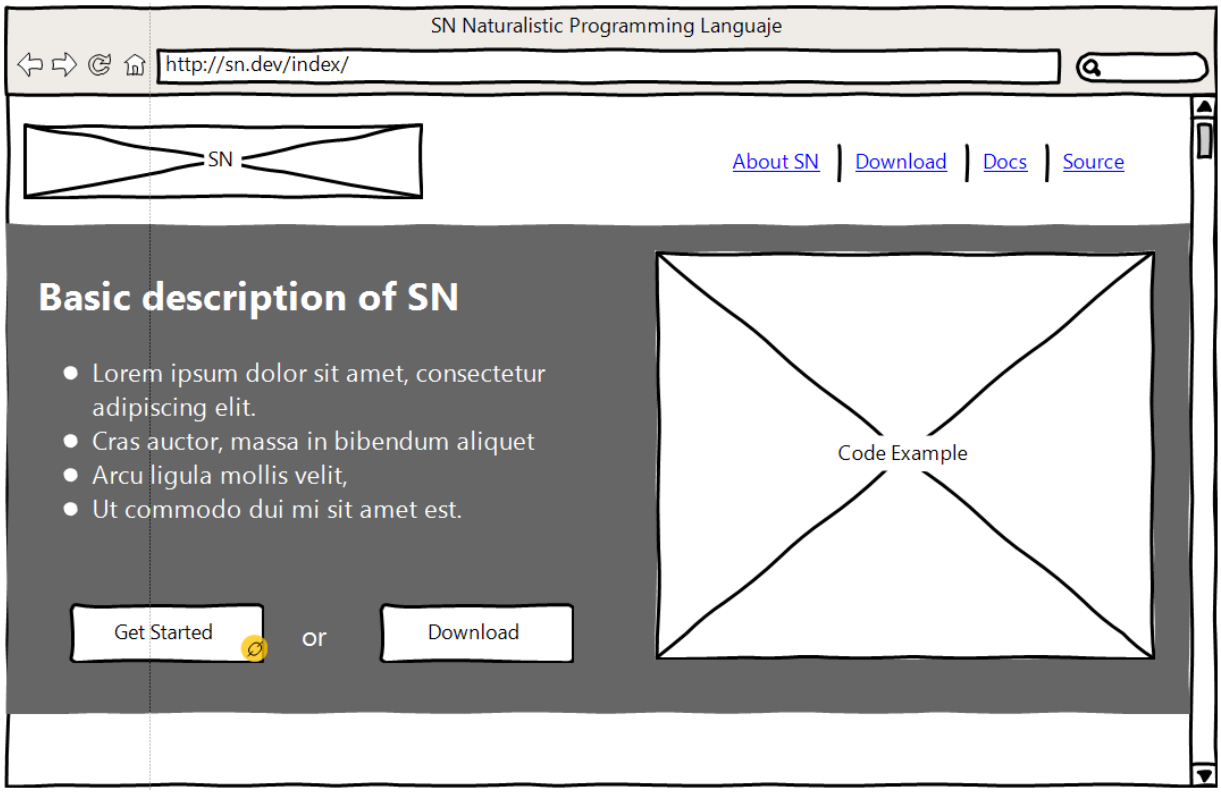

Figura 4.4: Mockup de la página de inicio de la Web de SN.

Página *Get Started*

La página *Get Started* está pensada para funcionar como la página que reúne todos los elementos que un usuario nuevo necesita para comenzar a trabajar con SN. Esta página ofrece el enlace de descarga del compilador de SN junto a las instrucciones de instalación y configuración

del mismo. Esta misma página debe poner a disposición del usuario el enlace a la documentación de SN y al *showroom* (el cual se detalla más adelante). De esta manera se pone a disposición de los nuevos usuarios distintos elementos de información sobre el lenguaje que le permiten aprender a utilizarlo, desde la instalación hasta la creación de software. En la Figura 4.5 se muestra el bosquejo de la página *Get Started* de la web de SN, considerando los elementos deseados de navegación.

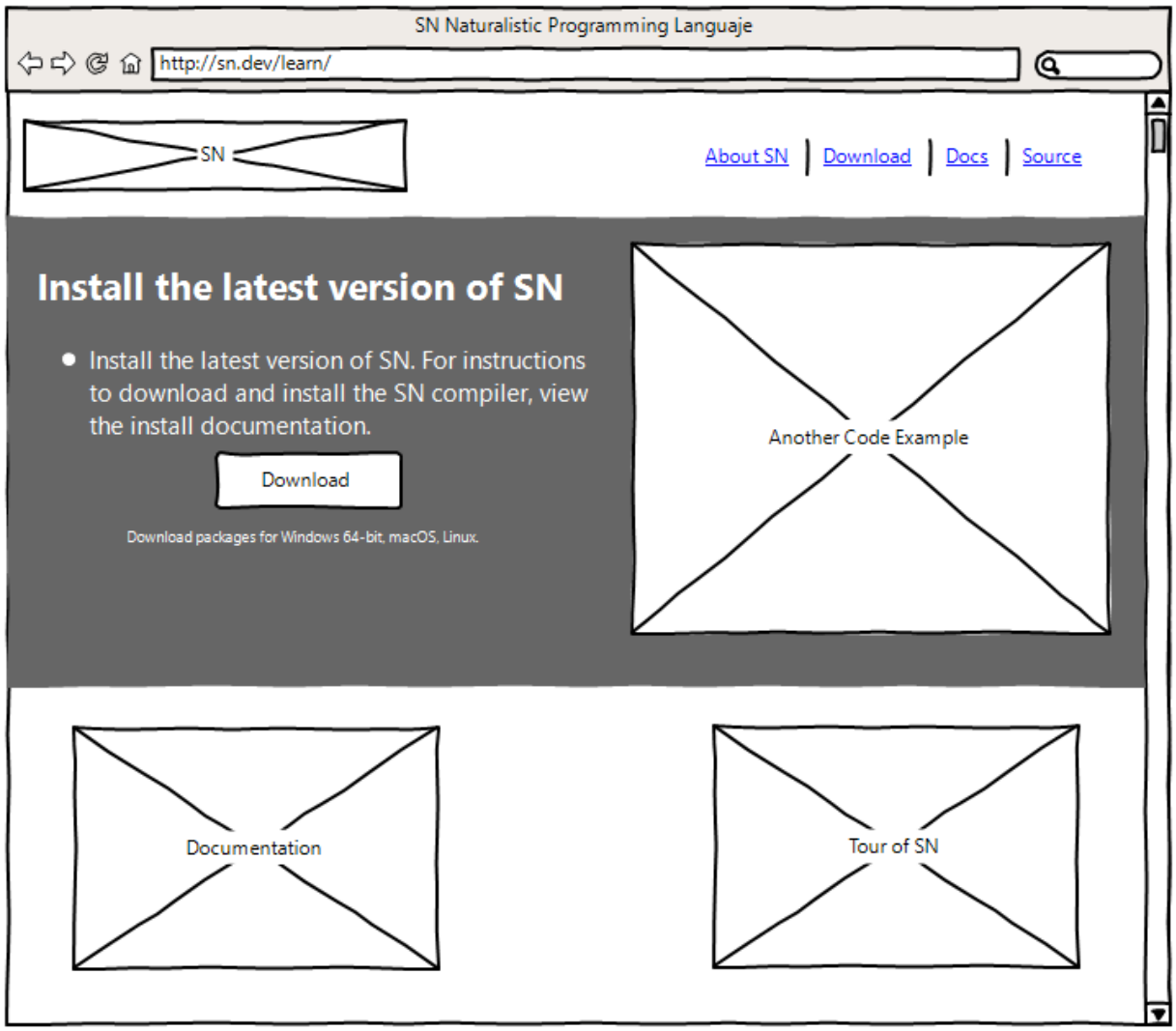

Figura 4.5: Mockup de la página *Get Started* de la Web de SN.

Página *About SN*

Esta página tiene como objetivo dar a conocer detalles relevantes de SN con un enfoque al concepto o proyecto en lugar de como lenguaje o compilador. En este sentido esta sección tiene como objetivo informar sobre los elementos teóricos que respaldan la creación de un lenguaje como SN, en esta página también se abordan temas como el objetivo del lenguaje. En otras palabras, esta sección está enfocada en hablar acerca del lenguaje SN, sus implicaciones y motivaciones, en lugar de hablar de su funcionamiento. En las Figuras 4.6 y 4.7 se muestra el bosquejo de la página *About SN* de la web de SN, considerando los elementos deseados de navegación.

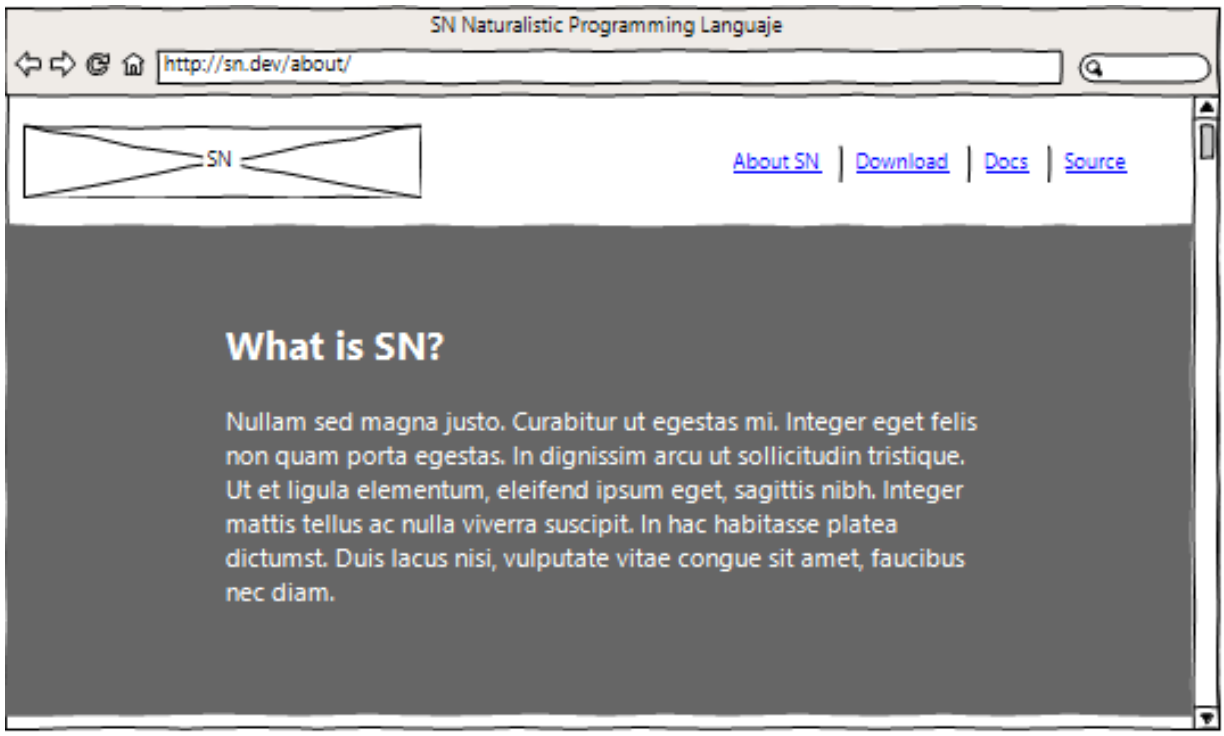

Figura 4.6: Mockup de la página *About SN* de la Web de SN (Elemento 1).

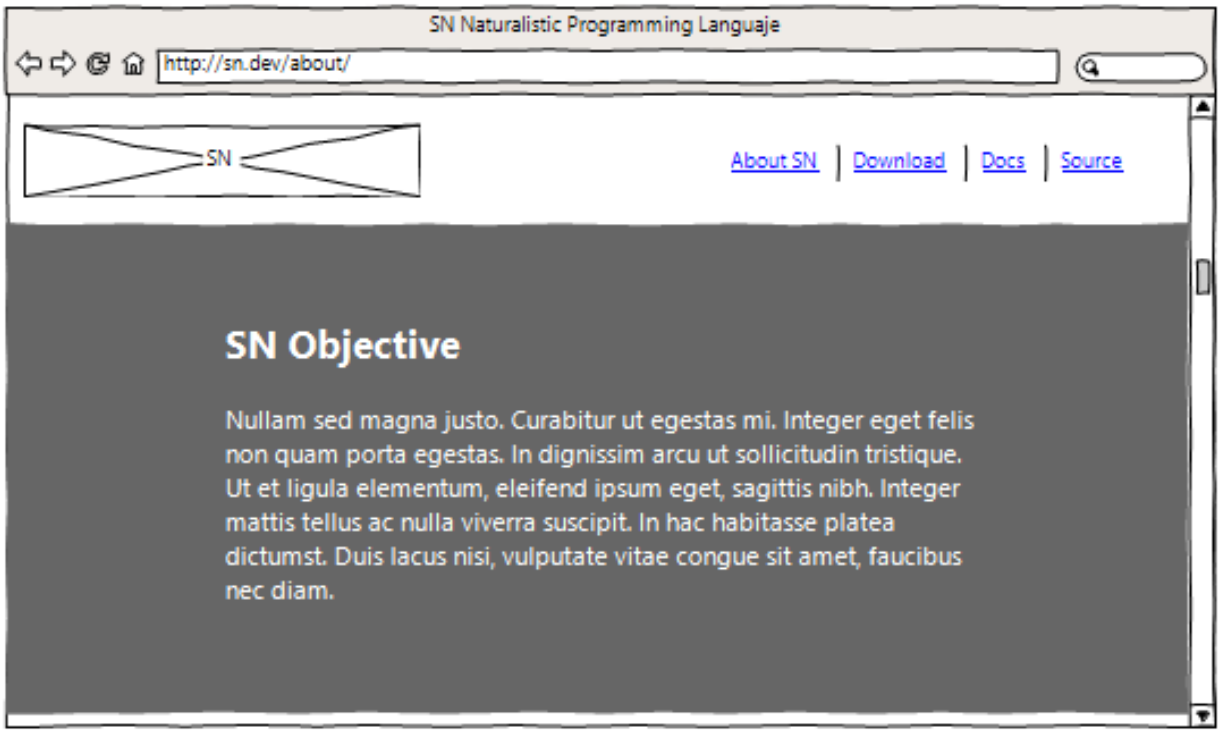

Figura 4.7: Mockup de la página *About SN* de la Web de SN (Elemento 2).

Página *Demo* o *Showroom*

Al contrario de la página *About SN*, la página *demo* o *showroom* tiene como objetivo mostrar elementos y características del lenguaje, en esta sección se coloca una exposición de ejemplos o fragmentos de código SN que destaquen las propiedades y características del lenguaje, entre los ejemplo que se pueden colocar se encuentran: el uso de referencias indirectas, la definición de sustantivos compuestos, la definición de los elementos para formar sustantivos compuestos, el uso de bloques condicionales y de repetición, entre otros. Puntualmente, esta página tiene el objetivo de familiarizar a los usuarios con la forma de desarrollar software con SN. En las Figuras 4.8 y 4.9 se muestra el bosquejo de la página *About SN* de la web de SN, considerando los elementos deseados de navegación.

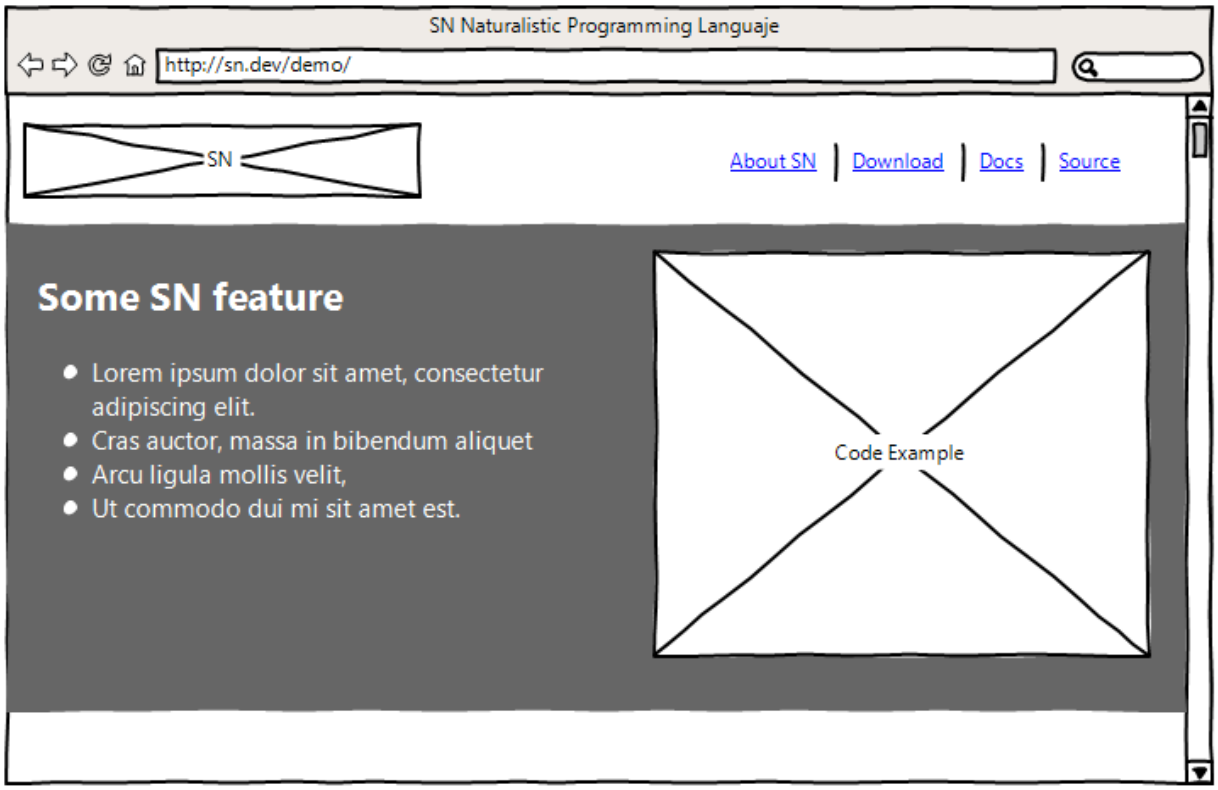

Figura 4.8: Mockup de la página *Demo* de la Web de SN (Elemento 1).

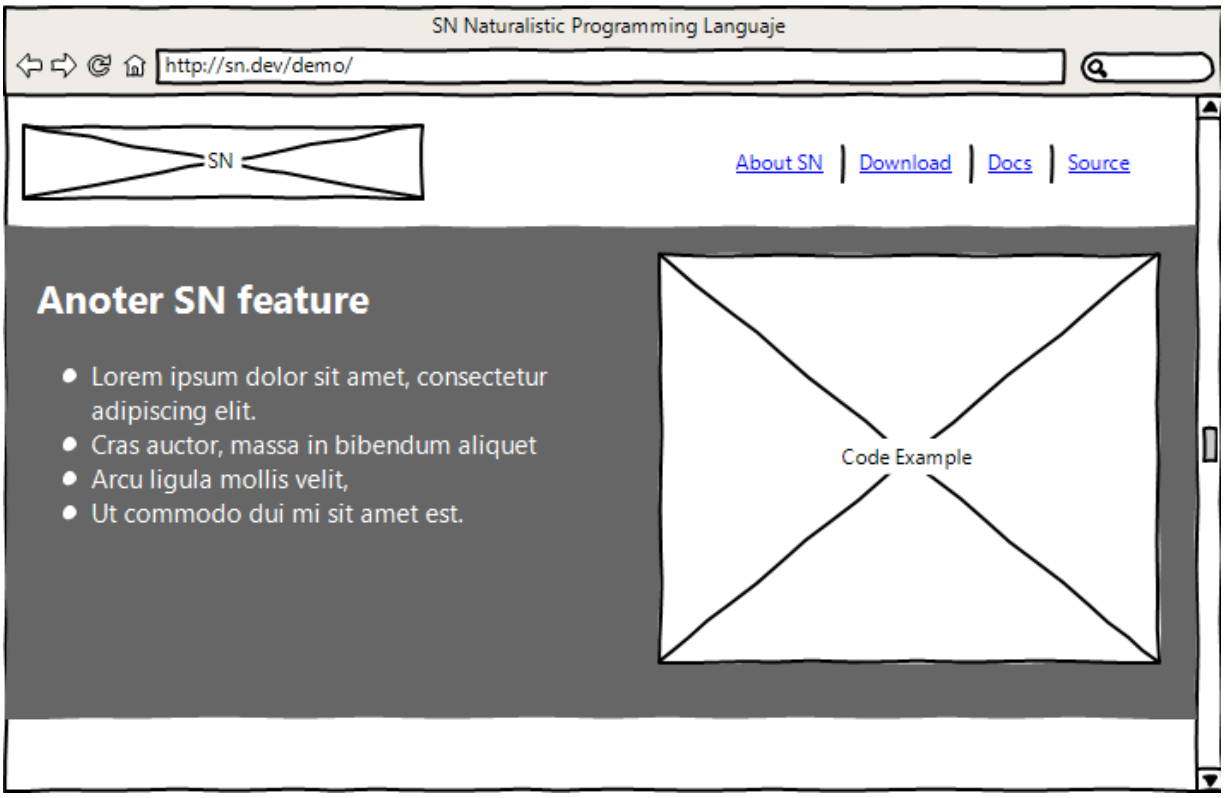

Figura 4.9: Mockup de la página *Demo* de la Web de SN (Elemento 2).

Página de Documentación de SN

Al referirse a la documentación de SN evoca dos tipos diferentes de documentación. La primera es la documentación del compilador de SN, esta documentación está enfocada en proporcionar información sobre cómo está implementado el lenguaje SN permitiendo así su modificación o mejora, por otro lado, el segundo tipo de documentación hace referencia a la documentación de la API de SN. La diferencia primordial entre ambos tipos de documentación es que una se enfoca a construir a SN y la otra se enfoca en construir con SN. En este caso la página de Documentación a la que dirigirá la web de SN corresponde al primer tipo de documentación mencionada. Esta documentación se genera automáticamente a partir de documentación interna del código del compilador de SN mediante la herramienta JavaDocs. En la Figura 4.10 se muestra la pantalla de la página de la documentación del compilador de SN.

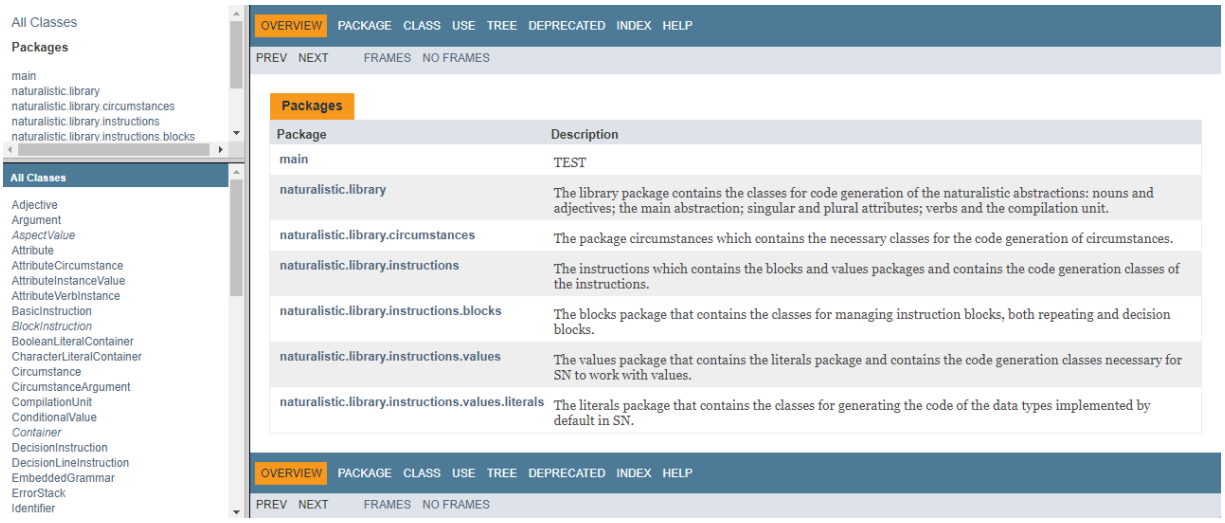

Figura 4.10: Pantalla de la documentación generada mediante la herramienta JavaDocs, del compilador de SN.

4.2.2. API de SN

En consideración de las herramientas para aprender a programar en SN se generó una representación visual de la API de SN, las cuales se ven en la Figuras 4.11 y 4.12 siendo la representación de la API de sustantivos y adjetivos respectivamente.

El objetivo principal de esta representación gráfica es comprender la jerarquía de los elementos implementados en SN. Ejemplifica de una forma sencilla y visual que la base de todo sustantivo en SN es la abstracción *Thing* ya que esta abstracción es la mínima representación de un “algo"que se busque describir. A partir de este ancestro se pueden definir los sustantivos singulares, en cambio para definir un sustantivo plural es necesario implementar la abstracción *Things* la cual brinda el soporte para los sustantivos plurales.

Todo este proceso de agregación es transparente para al usuario al momento de definir sus propios sustantivos, tanto plurales como singulares, sin embargo, es necesario señalarlo debido a la importancia conceptual de este proceso de agregación.

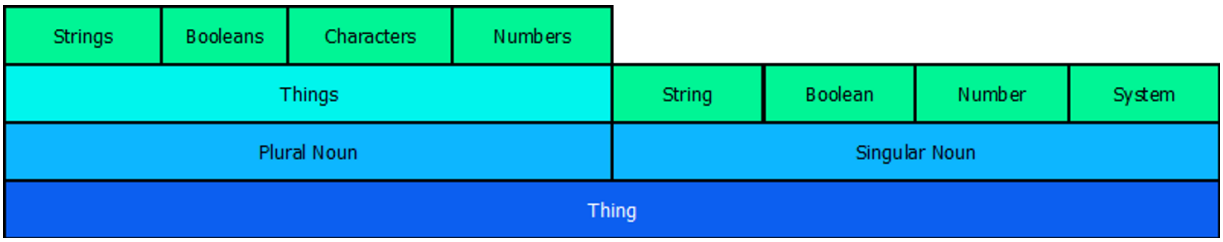

Figura 4.11: Representación gráfica de la API de SN - Sustantivos.

En el caso de los adjetivos, no tienen su origen en la abstracción *Thing* ya que no son una cosa en sí misma, más bien, representan variaciones en el comportamiento de los sustantivos. El origen común de todos los adjetivos es propiamente la abstracción adjetivo (*Adjective*) y como se observa en la Figura 4.12 las abstracciones se forman por medio de una mayor cantidad de capas de complejidad hasta que finalmente se especifica el adjetivo en sí mismo.

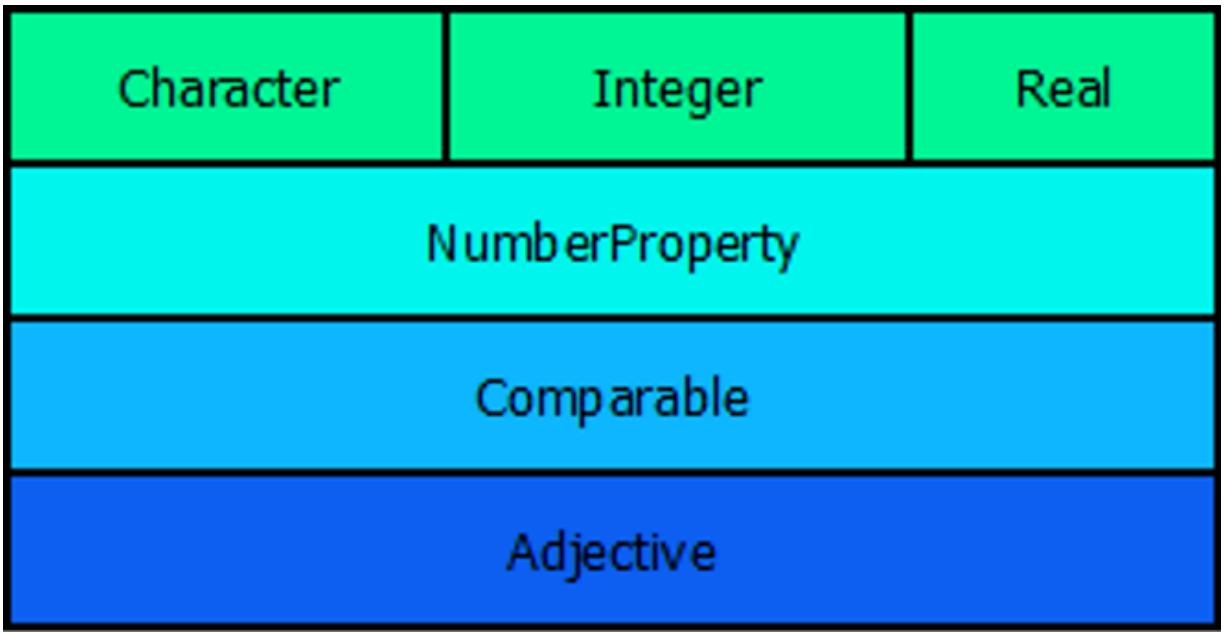

Figura 4.12: Representación gráfica de la API de SN - Adjetivos.

Comprender cuáles son las bases de los mecanismos de composición de SN permite visualizar de mejor manera la forma correcta de crear abstracciones personalizadas. Siendo el sustantivo la

definición de la cosa en sí misma, incluyendo atributos (como es la cosa) y verbos (que hace la cosa), por otro lado los adjetivos describen variaciones o modificaciones en el comportamiento original de los adjetivos.

4.2.3. Instalador de SN

Para facilitar la instalación y distribución del lenguaje en el sistema operativo Windows, se generó el instalador del compilador mediante la herramienta libre Inno Setup. El uso de un instalador previene errores producidos por copiar manualmente los archivos del instalador, en la Figura 4.13 se observa la primer pantalla del instalador, dicha pantalla contiene la licencia del lenguaje. Para ejemplificar su funcionamiento se colocó de forma temporal la licencia APACHE hasta definir la licencia del lenguaje SN.

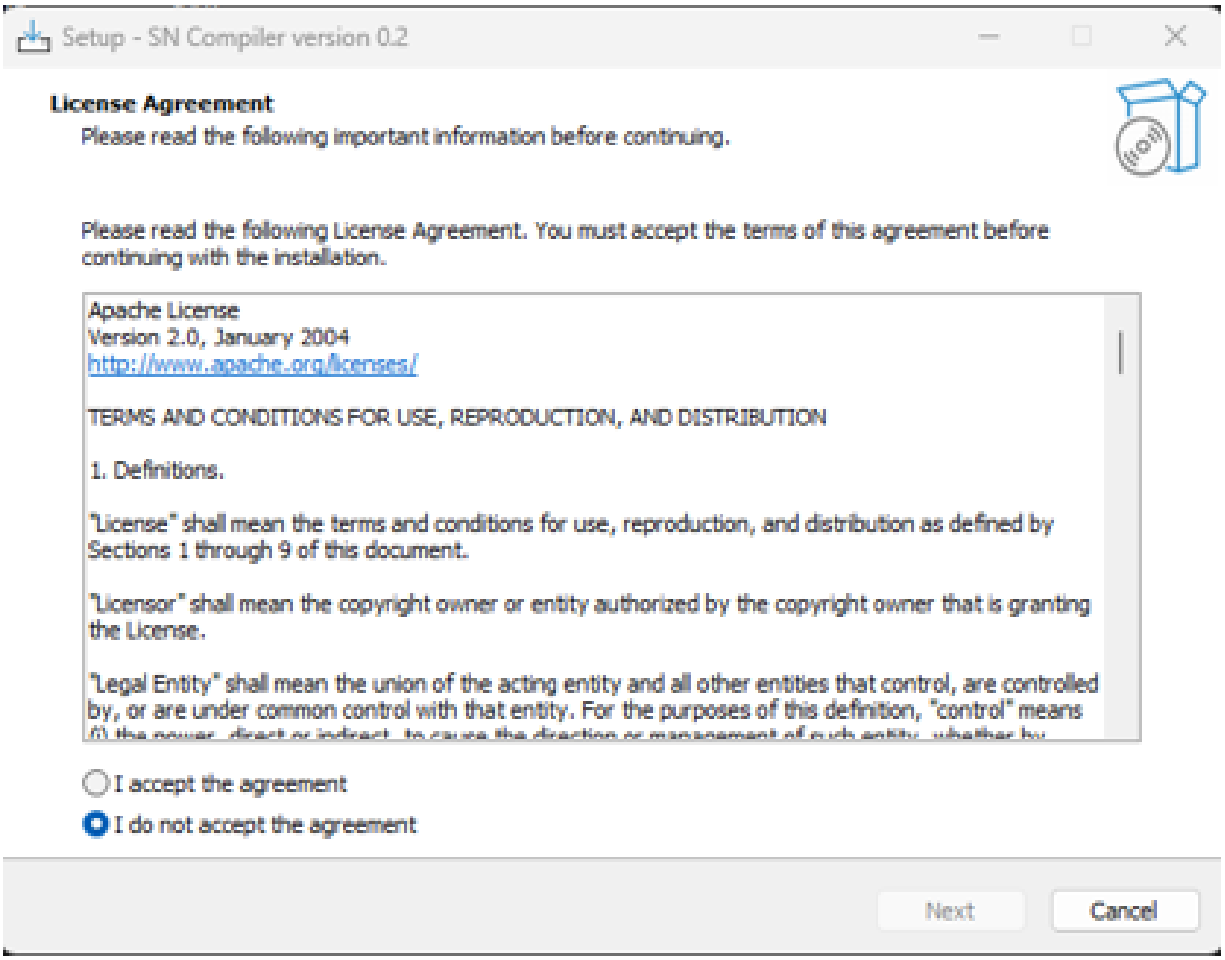

Figura 4.13: Pantalla de licencia del instalador del compilador de SN.

En la Figura 4.14 se observa la pantalla de selección de destino, la ruta por defecto es archivos de programa, sin embargo, esta ruta se puede modificar por el usuario.

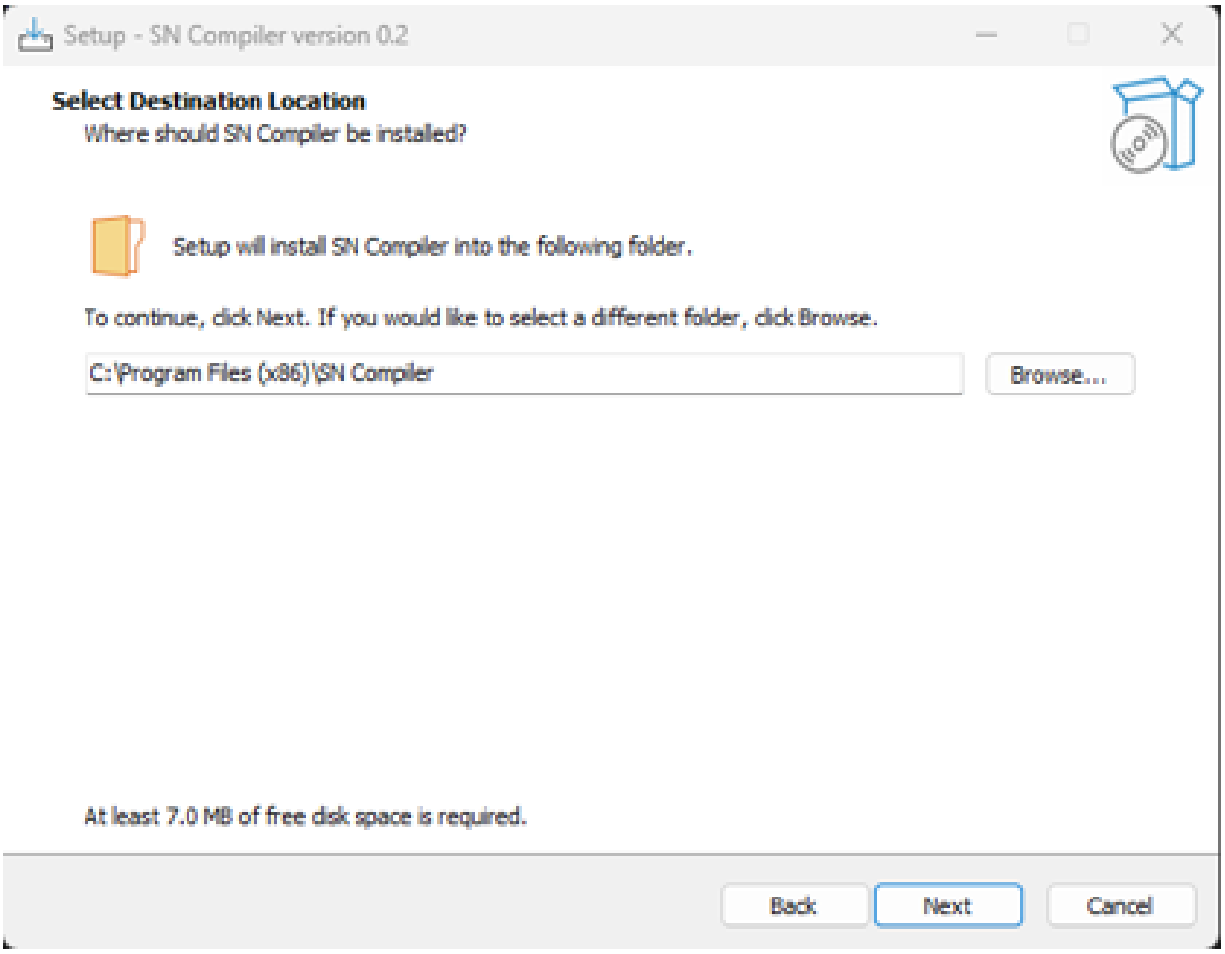

Figura 4.14: Pantalla de selección de ruta del instalador del compilador de SN.

En la figura 4.15 se muestra la pantalla de confirmación, aquí el usuario puede corroborar la ruta en la que se instalarán los archivos del compilador de SN.

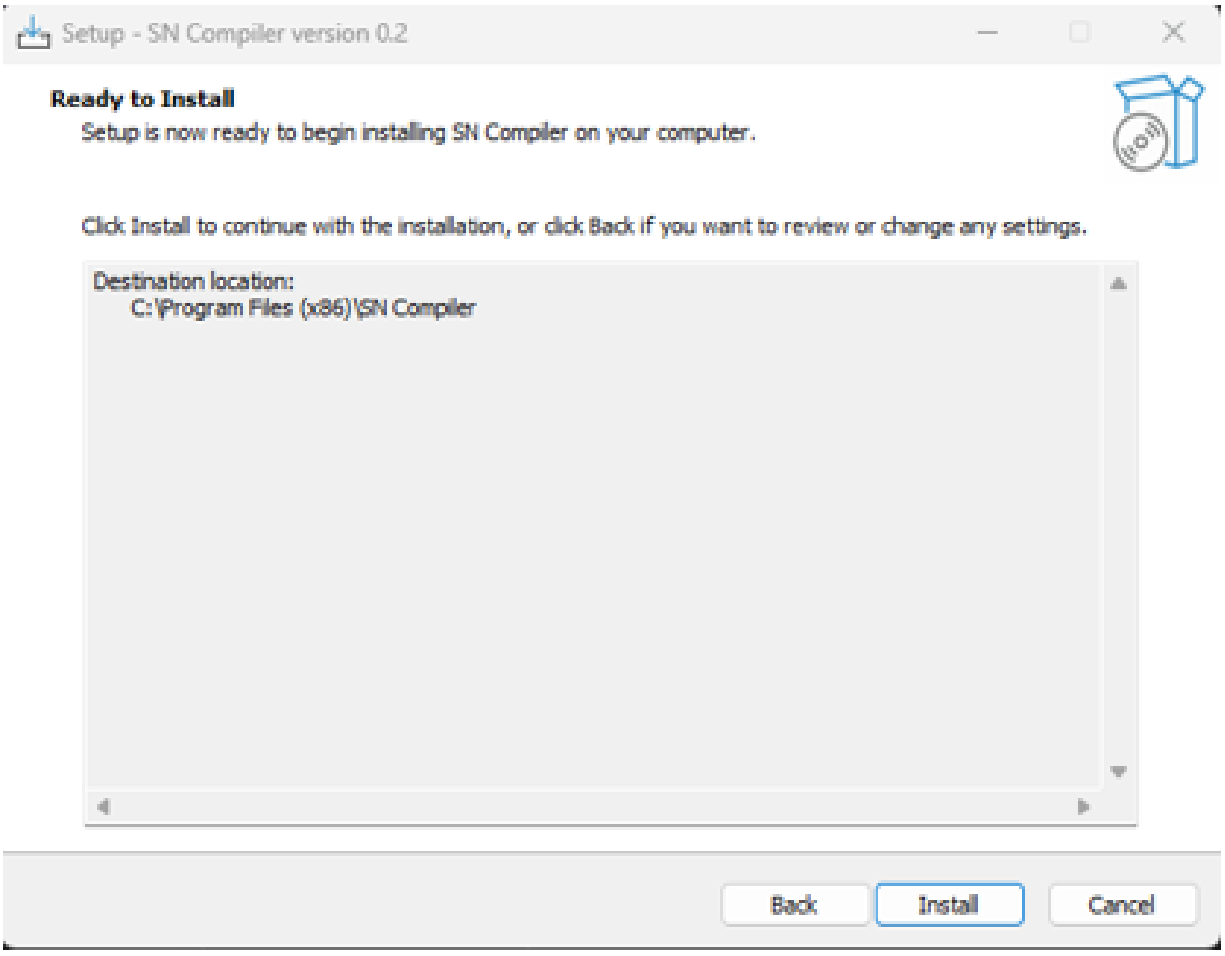

Figura 4.15: Pantalla de confirmación del instalador del compilador de SN.

Finalmente en la figura 4.16 se muestra la pantalla final del instalador de SN donde muestra que se completó la instalación del compilador y despliega un botón para finalizar el instalador.

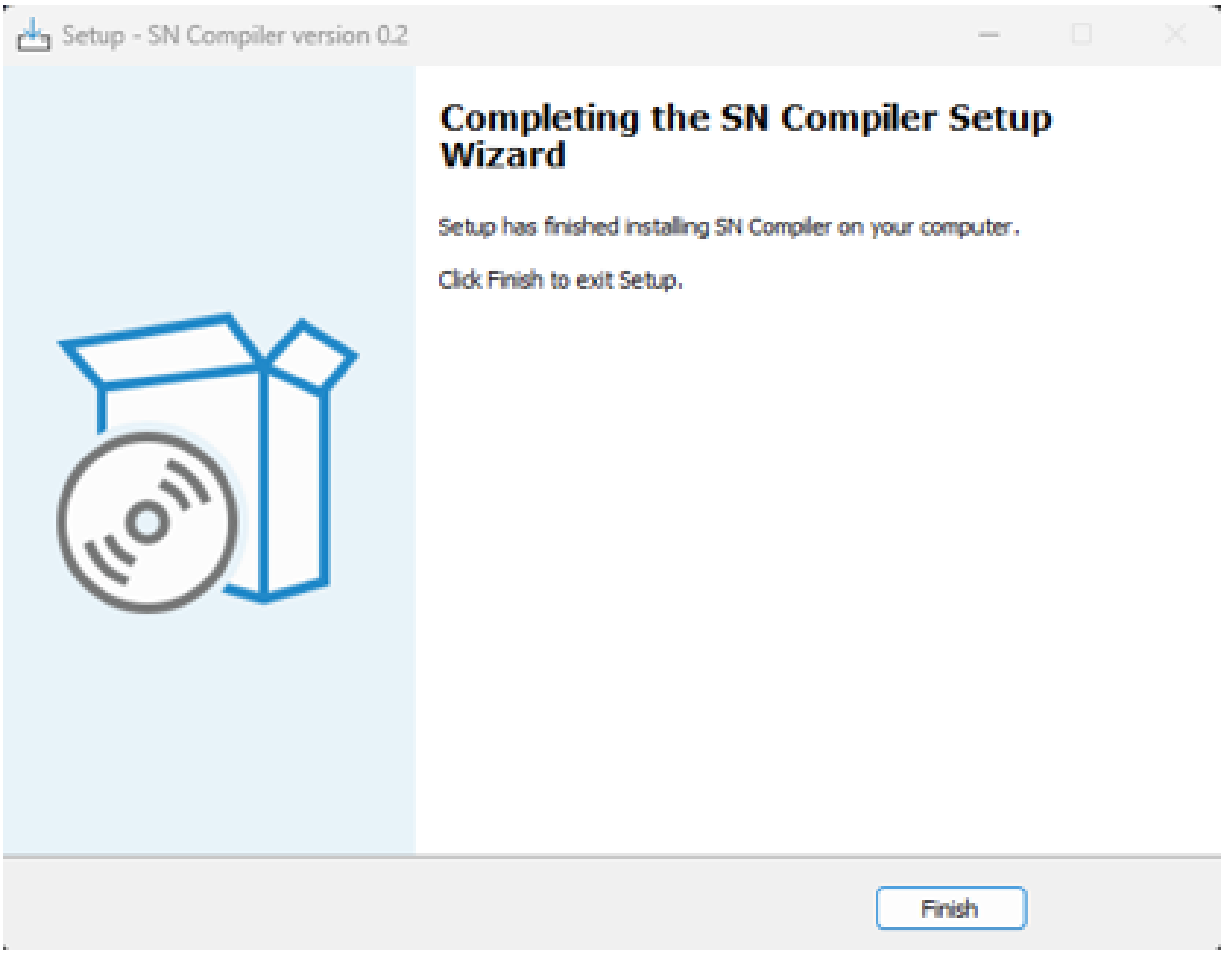

Figura 4.16: Pantalla de final del instalador del compilador de SN.

Como resultado de esta instalación en la Figura 4.17 se observan los archivos del compilador de SN colocados correctamente en la carpeta de destino, adicionalmente Inno Setup genera los archivos correspondientes para la desintalación del compilador.

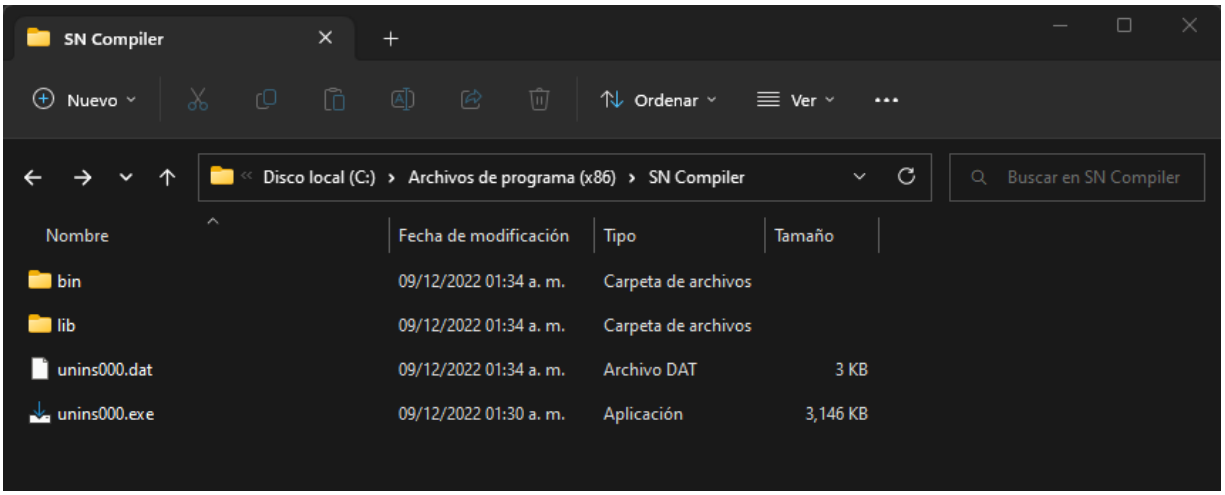

Figura 4.17: Archivos del compilador SN colocados en carpeta de destino.

Capítulo 5

Conclusiones y recomendaciones

En este último capítulo se presentan las conclusiones obtenidas de la realización de este proyecto de tesis, así como una reflexión de las implicaciones de los lenguajes naturalísticos y las recomendaciones de trabajo a futuro para la mejora del lenguaje SN.

5.1. Conclusiones

La principal contribución de esta tesis al lenguaje SN es preparar el camino para el trabajo a futuro para el desarrollo del lenguaje. Dada su naturaleza de prototipo, el lenguaje de SN se encuentra en una etapa muy temprana de su existencia, esto provoca que a pesar de ser un lenguaje de propósito general su alcance efectivo sea limitado. Sin embargo, no es posible negar que la programación naturalística tiene un enorme potencial de reducir la brecha entre el dominio del problema y el de la solución mediante la incorporación de diversos elementos del lenguaje natural, es por ello que proyectos como SN requieren de un continuo desarrollo para que eventualmente el potencial teórico del paradigma naturalístico se transforme en un poder tangible de simplificación de la resolución de problemas.

Para lograr que tal progreso se lleve a cabo, favoreciendo el crecimiento y fortalecimiento del paradigma, es necesario un continuo proceso de desarrollo y mejora de los lenguajes natu-

ralísticos para ampliar las capacidades de los mismos; siendo esta la única forma de demostrar empíricamente la fortaleza del paradigma en aspectos como la preservación de las ideas y la simplificación de los procesos de documentación de software.

El lenguaje SN, y el paradigma naturalístico en general, tiene un gran camino que recorrer antes de atraer a un segmento considerable de desarrolladores y un camino aún más largo para ser implementado en la construcción de software a gran escala, sin embargo, una empresa de tal magnitud es posible de llevar a cabo un paso a la vez, de tal forma que este trabajo de tesis es la punta de lanza que prepara el camino para el desarrollo y el fortalecimiento del lenguaje SN.

5.2. Recomendaciones de trabajo a futuro

A partir de los resultados obtenidos en este proyecto de tesis, se sugieren el siguiente trabajo a futuro:

1. Implementar un sistema de gestión de bibliotecas, con un enfoque naturalístico, con el proposito de asentar elementos que permitan el desarrollo de software de mayor escala.

2. A partir del sistema de gestión de bibliotecas identificar las bibliotecas prioritarias para la expansión de las capacidades del lenguaje. Ejemplos de bibliotecas:

   - Persistencia de datos en archivos.
   - Creación de interfaz gráfica.
   - Enlace para la obtención de datos.
   - Capacidades Web y/o móvil.

3. Finalmente, es necesario enfocar la construcción de SN con una filosofía modular que facilite la colaboración de diversos participantes en el desarrollo del lenguaje.

Productos Académicos

Bibliografía

[1] O. Pulido Prieto and U. Juárez Martínez, "Naturalistic programming: Model and implementation," *IEEE Latin America Transactions*, vol. 18, no. 07, pp. 1230–1237, 2020.

[2] G. Sartori, *Homo videns: la sociedad teledirigida*. Punto de lectura, 2006.

[3] L. Wittgenstein, *Investigaciones filosóficas*. Editorial Gredos, 2009.

[4] L.-M. Alducin-Francisco, U. Juarez-Martinez, S. G. Pelaez-Camarena, L. Rodriguez-Mazahua, M.-A. Abud-Figueroa, and O. Pulido-Prieto, "Perspectives for software development using the naturalistic language sn," in *2019 8th International Conference On Software Process Improvement (CIMPS)*, pp. 1–11, Oct 2019.

[5] F. Bellas, R. Unanue, and V. Fernández, *Lenguajes de programación y procesadores*. INGENIERÍA Y CIENCIAS, Editorial Centro de Estudios Ramon Areces SA, 2016.

[6] O. Pulido-Priedo, *Modelo conceptual para la implementación de lenguajes de programación naturalísticos de propósito general*. PhD thesis, Instituto Tecnológico de Orizaba, 2019.

[7] R. Laddad, *AspectJ in Action: Enterprise AOP with Spring Applications*. USA: Manning Publications Co., 2nd ed., 2009.

[8] S. Clarke and E. Baniassad, *Aspect-oriented analysis and design*. Addison-Wesley Professional, 2005.

[9] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, p. 154, 1996.

[10] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr, "Beyond aop: Toward naturalistic programming," *SIGPLAN Not*, vol. 38, pp. 34–43, dec 2003.

[11] R. Knöll and M. Mezini, "Pegasus: First steps toward a naturalistic programming language," OOPSLA '06, (New York, NY, USA), pp. 542–559, Association for Computing Machinery, 2006.

[12] R. Knöll, V. Gasiunas, and M. Mezini, "Naturalistic types," in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 33–48, 2011.

[13] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications," *Science of Computer Programming*, vol. 166, 05 2018.

[14] O. Pulido-Prieto and U. Juárez-Martínez, "A survey of naturalistic programming technologies," *Association for Computing Machinery*, vol. 50, sep 2017.

[15] O. Pulido-Prieto and U. Juárez Martínez, "A model for naturalistic programming with implementation," *Applied Sciences*, vol. 9, p. 3936, 09 2019.

[16] O. Pulido-Prieto, *SN Manual*. 2019.

[17] L. Francisco, Alducin María, "Estudio comparativo de los lenguajes sn y aspectj para la encapsulación de requerimientos no funcionales," 2021.

[18] R. S. Pressman, *Ingeniería del Software. Un Enfoque Práctico*. Sommerville Ian: Ingenieria del Software. Pearson, 2010.

[19] A. A. L. Ameneyro, "Gisweb: Reingenieria para la implementacion de un web feature service," May 2004.

[20] R. C.Martin, *Clean Code: A Handbook of Agile Software Craftsmanship [Book]*.

[21] Oracle, “How to write doc comments for the javadoc tool.” Accedido 28 de Febrero de 2022.