Instituto Tecnológico de Orizaba
Departamento de Comunicación y Difusión

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCION I.- TESIS

TRABAJO PROFESIONAL

"ESTUDIO COMPARATIVO DE LOS LENGUAJES SN Y ASPECTJ PARA LA ENCAPSULACIÓN DE REQUERIMIENTOS NO FUNCIONALES"

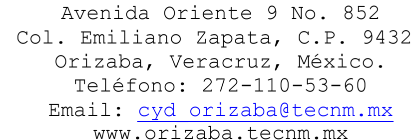
QUE PARA OBTENER EL GRADO DE:

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:
LUISA MARIA ALDUCIN FRANCISCO

DIRECTOR DE TESIS:
DR. ULISES JUAREZ MARTINEZ

ORIZABA, VER. MÉXICO FEBRERO 2021



Avenida Oriente 9 No. 852
Col. Emiliano Zapata, C.P. 94320
Orizaba, Veracruz, México.
Teléfono: 272-110-53-60
Email: cyd_orizaba@tecnm.mx
www.orizaba.tecnm.mx

Instituto Tecnológico de Orizaba
División de Estudios de Posgrado e Investigación

"2021: Año de la Independencia"

Orizaba, Ver., 16/junio/2021
Dependencia: **División de Estudios de Posgrado e Investigación**
Asunto: **Autorización de Impresión**
OPCION: **I**

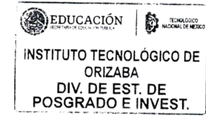
C. LUISA MARÍA ALDUCIN FRANCISCO
Candidato a Grado de Maestro en:
SISTEMAS COMPUTACIONALES
P R E S E N T E.-

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:

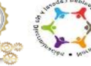
" Estudio comparativo de los lenguajes SN y AspectJ para la encapsulación de requerimientos no funcionales"

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

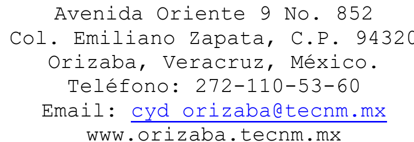
A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®



DR. MARIO LEONCIO ARRIOJA RODRÍGUEZ
JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN



Avenida Oriente 9 No. 852
Col. Emiliano Zapata, C.P. 94320
Orizaba, Veracruz, México.
Teléfono: 272-110-53-60
Email: cyd_orizaba@tecnm.mx
www.orizaba.tecnm.mx



Instituto Tecnológico de Orizaba
División de Estudios de Posgrado e Investigación

Orizaba, Ver., 18/mayo/2021

Asunto: Revisión del Trabajo Escrito

C. MARIO LEONCIO ARRIOJA RODRIGUEZ
JEFE DE LA DIVISION DE ESTUDIOS
DE POSGRADO E INVESTIGACION
P R E S E N T E.

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

LUISA MARÍA ALDUCIN FRANCISCO
No. DE CONTROL: M13011074

la cual lleva el título de:

"Estudio comparativo de los lenguajes SN y AspectJ para la encapsulación de requerimientos no funcionales"

y concluyen que se acepta.

A T E N T A M E N T E

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

EGRESADO (A) DE LA **MAESTRIA EN SISTEMAS COMPUTACIONALES**
OPCION: I Tesis

Avenida Oriente 9 No. 852
Col. Emiliano Zapata, C.P. 94320
Orizaba, Veracruz, México.
Teléfono: 272-110-53-60
Email: depi_orizaba@tecnm.mx
www.orizaba.tecnm.mx

AGRADECIMIENTOS

A Dios por haberme dado la oportunidad de terminar una etapa más de mi vida, por darme la fortaleza, la sabiduría e inteligencia necesaria para completar satisfactoriamente mi tesis.

A mis padres, José Luis Alducin López y Alicia Yazmin Francisco García por todo el apoyo y sacrifico entregado a lo largo de toda mi vida académica, por sus consejos, por sus palabras de aliento que me motivaron a no darme por vencida y recordarme que puedo lograr mis sueños

A mis hermanas, Guadalupe y Sara por motivarme a seguir adelante, por ser mis amigas y compañeras en momentos difíciles y alegres, además de constantemente decir que soy su ejemplo.

A mi abuelo Luis Alducin Muñoz, que en vida creyó en mí, me apoyo y se preocupó porque pudiera tener un soporte que permitiera seguir adelante con mis estudios.

A mi amigo Oscar Pulido Prieto por su apoyo incondicional en momentos clave de mi trabajo y por su amistad brindada.

A mi amiga, Xcaret Alburquerque Camarillo que siempre estuvo ahí para apoyarme y alentándome constantemente a cumplir este objetivo de vida.

A mi asesor, el Dr. Ulises Juárez Martínez por darme la oportunidad de llevar a cabo este trabajo, por aportarme sus conocimientos, por la paciencia y por todo el apoyo que me brindo a lo largo de estos dos años.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por la beca otorgada para poder financiar la estadía en la maestría, así como los congresos, estancias, entre otras cosas necesarias para concluir mis estudios.

Muchas Gracias

Índice general

Resumen xi

Abstract xiii

Introducción xv

1. Antecedentes 1

1.1. Marco teórico . . . 1

1.1.1. Requisitos . . . 1

1.1.1.1. Requisitos funcionales . . . 2

1.1.1.2. Requisitos no funcionales (NFR) . . . 2

1.1.2. Programación orientada a aspectos . . . 2

1.1.2.1. Aspecto . . . 2

1.1.2.2. Puntos de unión . . . 3

1.1.2.3. Corte en puntos . . . 3

1.1.2.4. Aviso . . . 3

1.1.2.5. Asunto . . . 3

1.1.3. Elementos lingüísticos de los lenguajes naturales . . . 3

1.1.3.1. Deíxis . . . 4

1.1.3.2. Endófora . . . 4

1.1.3.3. Sintagmas . . . 5

1.1.4. Contexto 6
1.1.5. Expresividad 7
1.1.6. Ambigüedad 7
1.1.7. Programación naturalística 8
1.1.7.1. SN 8
1.1.7.2. Circunstancias 9
1.2. Situación tecnológica, económica y operativa de la empresa 9
1.3. Planteamiento del problema 10
1.4. Objetivos 10
1.4.1. Objetivo general 11
1.4.2. Objetivos específicos 11
1.5. Justificación 11

2. Estado de la práctica 14
2.1. Trabajos relacionados 14
2.2. Análisis comparativo 27
2.3. Solución propuesta 36
2.3.1. Justificación de la solución seleccionada 36
2.3.2. Metodología para el desarrollo de la tesis 37
2.3.3. Justificación de la metodología para el desarrollo de la tesis 38

3. Aplicación de la metodología 39
3.1. Revisión de las características que ofrece AspectJ 39
3.1.1. Avisos 39
3.1.2. Primitivas de corte 40
3.1.2.1. call 40
3.1.2.2. execution 42
3.1.2.3. args 45

3.1.2.4. target . . . 48
3.1.2.5. this . . . 52
3.1.2.6. get . . . 56
3.1.2.7. set . . . 58
3.1.2.8. within . . . 60
3.1.2.9. cflow . . . 63
3.1.2.10. cflowbelow . . . 67
3.1.2.11. withincode . . . 70
3.1.2.12. preinitialization . . . 73
3.1.2.13. initialization . . . 76
3.1.2.14. staticiniatilization . . . 79
3.1.2.15. if . . . 81
3.1.2.16. adviceexecution . . . 83
3.1.2.17. handler . . . 85
3.2. Revisión de las características que ofrece SN . . . 88
3.2.1. Abstracciones de SN . . . 89
3.2.1.1. Noun (Sustantivo) . . . 89
3.2.1.2. Adjective (Adjetivo) . . . 89
3.2.1.3. Verb (Verbo) . . . 89
3.2.1.4. Attribute (Atributo) . . . 90
3.2.1.5. Main (Principal) . . . 90
3.2.1.6. Gramáticas embebidas . . . 90
3.2.1.7. Referencias indirectas . . . 91
3.2.1.8. Atributos derivados . . . 91
3.2.2. Contexto de ejecución . . . 92
3.2.3. Circunstancias . . . 92
3.2.3.1. Contexto de ejecución de verbos . . . 93

3.2.3.2. Exposición de contexto 93
3.2.3.3. Created 95
3.2.3.4. Assigned 98
3.2.3.5. Requires 99
3.2.3.6. Cannot 100
3.2.3.7. Mutually Excluded 101
3.3. Caso de estudio 102
3.3.1. Requerimientos del caso de estudio 102
3.3.2. Modelado (Enfoque de Temas) 105
3.3.2.1. Theme/Doc 106
3.3.3. Modelado (Enfoque experimental) 115

4. Resultados **119**
4.1. Implementación del caso de estudio 119
4.1.1. Implementación en Java y AspectJ 119
4.1.1.1. Aspecto para registros 120
4.1.2. Implementación en SN 134
4.1.2.1. Circunstancias para registros 134
4.2. Comparativa entre la implementación de solución de AspectJ y SN 141
4.2.1. Registro 141
4.2.2. Acceso 142
4.2.3. Revisión de licencias 143
4.3. Comparativa de las capacidades de AspectJ y SN 144
4.3.1. Comparativa de avisos y contexto de ejecución 145
4.3.2. Comparativa de primitivas y circunstancias 146
4.3.2.1. Equivalencias entre primitivas y circunstancias 147
4.3.2.2. Primitivas y circunstancias sin equivalencia 149

4.4. Ventajas y desventajas del uso de circunstancias y aspectos 151

5. Conclusiones y recomendaciones **153**

5.1. Conclusión 153

5.2. Recomendaciones 154

Bibliografía **156**

Índice de figuras

3.1. Descripción de la solución 108
3.2. Diagrama de vista aumentada registro 109
3.3. Diagrama de vista aumentada descarga 109
3.4. Diagrama de vista aumentada auditoria 110
3.5. Diagrama de vista aumentada concurrente 110
3.6. Diagrama de vista aumentada características 111
3.7. Diagrama de vista aumentada tiempo limitado 112
3.8. Diagrama de vista aumentada nombre registrado 112
3.9. Diagrama de vista aumentada nodo bloqueado 113
3.10. Diagrama de vista aumentada pago de uso 113
3.11. Diagrama de vista aumentada suscripción 114
3.12. Diagrama de vista aumentada ilimitado 114
3.13. Diagrama de vista aumentada de uso 115
3.14. Elementos propuestos por María Montessori 116
3.15. Elementos seleccionados para la representación de diagramas naturalísticos 117
3.16. Diagrama naturalístico del desarrollador 117
3.17. Diagrama naturalístico del cliente 118
3.18. Diagrama naturalístico de la aplicación 118

4.1. Diagrama UML de la aplicación 120

Índice de tablas

2.1. Análisis comparativo de los trabajos relacionados al tema de tesis 28

3.1. Requerimientos del sistema 103

4.1. Aplicación de aspectos y circunstancia en el caso: registro 142

4.2. Aplicación de aspectos y circunstancias en el caso: acceso 143

4.3. Aplicación de aspectos y circunstancia en el caso: revisión de licencias . . . 144

4.4. Comparativa de contextos de ejecución 145

4.5. Implementación de primitivas y circunstancias 147

4.6. Comparativa entre circunstancias y primitivas que tienen equivalencia . . . 149

4.7. Ventajas y desventajas de circunstancias y aspectos 152

Índice de códigos

3.1. Formula en Java. 41
3.2. Formula en Aspectos. 41
3.3. Generate DNI en Java. 43
3.4. Generate DNI en AspectJ. 45
3.5. Multiplication en Java. 46
3.6. Multiplication en Aspectos. 47
3.7. Game en Java. 48
3.8. Game en Aspect. 51
3.9. Random en Java. 53
3.10. Random en AspectJ. 54
3.11. Product en Java. 57
3.12. Product en AspectJ. 58
3.13. Person en Java. 59
3.14. Person en AspectJ. 59
3.15. Events en Java. 61
3.16. Events en AspectJ. 62
3.17. User en Java. 63
3.18. User en AspectJ. 65
3.19. Fibonacci en Java. 67
3.20. Fibonacci con AspectJ. 68

3.21. Account en Java. 70
3.22. Account con AspectJ. 72
3.23. Sports en Java. 74
3.24. Sports con AspectJ. 75
3.25. Toy en Java. 77
3.26. Toy con AspectJ. 78
3.27. Operations en Java. 79
3.28. Operations con AspectJ. 81
3.29. Weather en Java. 82
3.30. Weather con AspectJ. 83
3.31. Program en Java. 84
3.32. Program con AspectJ. 85
3.33. Exceptions en Java. 86
3.34. Exceptions con AspectJ. 87
3.35. Código Operations en SN 93
3.36. Código Created en SN 95
3.37. Código Assigned en SN 98
3.38. Código Requires en SN 99
3.39. Código Cannot en SN 101
3.40. Código Mutually Excluded en SN 102
4.1. Fragmento de código User en Java 121
4.2. Fragmento de código Developer en Java 122
4.3. Código para generar la clave del usuario en AspectJ 123
4.4. Fragmento de código License en Java 124
4.5. Fragmento de código Node en Java 125
4.6. Código que genera el identificador de las licencias 126
4.7. Fragmento de código descarga en Java 128

4.8. Código de acceso en AspectJ 129
4.9. Código que simula la ejecución de la aplicación 131
4.10. Código de licencia en AspectJ en SN 131
4.11. Código Operations en SN 132
4.12. Código Operations en SN 134
4.13. Fragmento de código aplicación en SN 135
4.14. Código Operations en SN 136
4.15. Fragmento de código licencia en SN 138
4.16. Fragmento de código Application en SN 139
4.17. Código Operations en SN 140

Resumen

El encapsulamiento de los requisitos no funcionales es un tema que no siempre se trata de forma adecuada en los desarrollos convencionales, es por esta razón que suelen encontrarse dispersos por el código, lo cual provoca que el posterior mantenimiento se dificulte al no haber una correcta documentación. La programación orientada a aspectos se dedica a limitar el problema de la dispersión, sin embargo, aunque lenguajes como AspectJ cuentan con los elementos mínimos necesarios para asemejarse al lenguaje natural, aún no son lo suficientemente expresivos; es por esto que se propuso que si se trabajara más en la sintaxis de AspectJ este podría ser más expresivo desde el punto de vista del lenguaje natural.

La propuesta de mejora de AspectJ, sugirió incluir más elementos del lenguaje natural para la creación de código más expresivo, pero la implementación de estos, genera problemas ya que por inercia el ser humano tiende a omitir información que provee de contexto a una instrucción.

Como solución al problema de la omisión de información se desarrollaron los lenguajes naturalísticos, los cuales toman elementos de los lenguajes naturales con los que permiten redactar instrucciones que provean lo necesario para evitar la ambigüedad. SN es un lenguaje naturalístico que permite generar programas por medio de un subconjunto controlado del idioma inglés. Este lenguaje cuenta con un mecanismo que permite, por medio de un control de eventos, encapsular los Requisitos no funcionales. Actualmente no se reporta un análisis comparativo que permita estudiar las diferencias y similitudes

que existen entre los aspectos de AspectJ y las circunstancias de SN, como métodos que permiten encapsular los Requisitos no funcionales, desde dos paradigmas diferentes.

Por lo anteriormente expuesto se propuso realizar, por medio de ejercicios, un estudio que permita identificar las ventajas y desventajas que se presentan tanto en los aspectos como en las circunstancias para el encapsulamiento de requisitos no funcionales.

Abstract

The encapsulation of non-functional requirements is an issue that is not always adequately addressed in conventional developments, which is why they are often scattered throughout the code, making subsequent maintenance difficult due to the lack of proper documentation. Aspect-oriented programming is dedicated to limit the problem of dispersion, however, although languages such as AspectJ have the minimum of required elements to resemble natural language, they are still not expressive enough; this is why it was proposed that if more work was done on the syntax of AspectJ, it could be more expressive from the point of view of natural language.

The proposal to improve AspectJ, suggested including more natural language elements for the creation of more expressive code, but the implementation of these, generates problems since by inertia the human being tends to omit information that provides context to an instruction.

As a solution to the problem of omitting information, naturalistic languages were developed, which take elements from natural languages to write instructions that provide what is necessary to avoid ambiguity. SN is a naturalistic language that allows to generate programs by means of a controlled subset of the English language. This language has a mechanism that allows, by means of an event control, to encapsulate non-functional requirements. Currently, there is no comparative analysis reported that allows studying the

differences and similarities that exist between the aspects of AspectJ and the circumstances of SN, as methods that allow encapsulating non-functional requirements, from two different paradigms.

For the above mentioned, it was proposed to carry out, by means of exercises, a study that allows identifying the advantages and disadvantages that are presented both in the aspects and in the circumstances for the encapsulation of non-functional requirements.

Introducción

La necesidad de creación y mantenimiento de software llevó a los investigadores a desarrollar nuevas técnicas de programación que permitan generar software mejor estructurado para facilitar el mantenimiento del código. Sin embargo, el paradigma orientado a objetos encapsula los requisitos funcionales de forma que no es necesario redactarlos más de una vez, mientras que los requisitos no funcionales se encuentran dispersos a través de las diferentes abstracciones que componen el sistema. Es por esto que surgieron los lenguajes orientados a aspectos, cuyo propósito es limitar la dispersión de código, por medio de entidades bien definidas.

Por otra parte, la transformación de la especificación de requisitos a código es compleja, puesto que estos se transforman a partir de un lenguaje natural a un lenguaje formal. Por lo tanto, la brecha entre el dominio del problema y la solución sigue siendo amplia, ya que los requisitos que estableció el cliente sufren una transformación para que se traduzcan al lenguaje de programación que se implemente. Aunque de que la solución más eficiente sería el uso de lenguajes naturales para el desarrollo de sistemas, este tipo de lenguajes presentan el problema de la ambigüedad, ya que una computadora solo procesa instrucciones que no consideran el contexto en el que se plantea la solución, por esta razón se propone el uso de lenguajes naturalísticos para que la programación sea más expresiva desde la perspectiva humana, a partir de un desarrollo cuyo lenguaje sigue con la forma-

lidad necesaria en la programación, pero que se asemeja a los lenguajes naturales.

En ambos paradigmas, existen lenguajes que permiten encapsular los requisitos no funcionales, por lo cual se propone analizar por medio de ejercicios y un caso de estudio las ventajas, desventajas, similitudes y diferencias entre los mecanismos que proporcionan AspectJ y SN.

El presente documento se conforma por cuatro capítulos: en el primer capítulo se definen los conceptos básicos necesarios para la comprensión del dominio del problema, así como el planteamiento del mismo, el objetivo general, objetivos específicos y la justificación que respaldan el presente proyecto; en el segundo capítulo se encuentran los artículos más relevantes en el estado del arte relacionados con los temas de requisitos no funcionales, programación orientada a aspectos y programación naturalística, también se indica la metodología que se utilizó durante el desarrollo de la tesis y la justificación de la misma; en el tercer capítulo se muestra el desarrollo de la tesis; y por último en el cuarto capítulo se muestran los resultados que se obtuvieron del caso de estudio.

Capítulo 1

Antecedentes

En este capítulo se abordan los conceptos fundamentales para la comprensión del desarrollo del tema de tesis, así como la problemática que se trató, los objetivos para la solución de la misma y la justificación del trabajo que se realizó.

1.1. Marco teórico

En esta sección, se presentan los términos más relevantes para el tema de tesis.

1.1.1. Requisitos

Especificación que describe el comportamiento del sistema, además de expresar las restricciones propias de la empresa, para satisfacer las necesidades del cliente [1].

1.1.1.1. Requisitos funcionales

Los requisitos funcionales describen las actividades a realizar en el sistema, dicho de otra manera, es el comportamiento o funciones del software y consideran las interacciones que tendrá el cliente con el programa [1].

1.1.1.2. Requisitos no funcionales (NFR)

Son aquellos que complementan a la funcionalidad principal del sistema, permiten definir políticas que el cliente establece, de modo que se cumplan las condiciones particulares tales como: restricciones de tiempo, restricciones en formato de texto, entre otras [1].

1.1.2. 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 que estos se encuentren dispersos 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 se considera ortogonal y complementario del segundo [2].

1.1.2.1. Aspecto

Un aspecto (del inglés *Aspect*) es una funcionalidad transversal, la cual se repetirá a lo largo del sistema, que se implementa de forma modular y que se encuentra separada del resto del sistema [2].

1.1.2.2. 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 [2].

1.1.2.3. Corte en puntos

Los cortes en puntos (*point cut*), también se les conoce como cortes, son la combinación de varios puntos de unión en la que el aspecto lleva a cabo su comportamiento. Los *join point* se unen por medio de operadores lógicos, los cuales son *AND* y *OR* [3].

1.1.2.4. Aviso

Un aviso (*advice*) implementa la funcionalidad transversal, es decir, una acción que se realiza como parte del aspecto. Un aviso define cómo se va a modificar el código en el contexto del corte en puntos [3].

1.1.2.5. Asunto

Un asunto es algo que sucede o interesa, sobre el cual se realizarán gestiones o acciones. En programación se entiende como un conjunto de información que tiene un efecto en el código de un programa [4].

1.1.3. Elementos lingüísticos de los lenguajes naturales

En esta sección se explicarán los términos necesarios para la comprensión adecuada de los elementos lingüísticos que se emplearon durante el desarrollo del lenguaje SN.

1.1.3.1. Deíxis

En el *Oxford English Dictionary* se 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:

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

Con el término deíxis se hace referencia al sentido de una oración o frase, el cual depende del contexto en el que se trabaja. La *deixis* se encuentra presente en todos los lenguajes naturales, ya que permite expresar ideas, reduciendo el número de palabras, por ejemplo, la oración: *"tú tomaras clases por la tarde y yo tomaré la misma clase"*, se reduce a *"tú tomaras clases por la tarde y yo también"* sin que se pierda su significado. En algunas ocasiones, las frases resultantes, de la omisión de palabras en una frase, son ambiguas, sin embargo, esto se resuelve con el proceso cognitivo humano.

1.1.3.2. Endófora

A una referencia indirecta que se realiza dentro del mismo texto [5], se le conoce como endófora, se clasifican según el momento en el que se define el objeto al que se hace referencia.

Anáfora: Se le conoce como anáfora, a la referencia indirecta que se realiza después de declarar el referente [5]. Ejemplo: *Toma la memoria de la mesa, después conéctala a la computadora.*

Catáfora: Cuando la referencia indirecta se realiza antes que se declare el objeto referente, se conoce como catáfora [5]. Ejemplo: *Después de tomarla de la mesa, conecta la memoria a la computadora.*

Exófora: Cuando se reemplaza una palabra, cuyo concepto se definió previamente en otro texto, por otra que hace una referencia indirecta, se le conoce como exófora. 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.

1.1.3.3. 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 le 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: sintagmas nominales, sintagmas verbales, sintagmas preposicionales y sintagmas adjetivales, por mencionar los más importantes.

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*): En 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."*. En 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 termino.

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.

1.1.4. 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, se entiende que 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.

De acuerdo a lo que se menciona en la literatura, algunas herramientas naturalísticas se centran en contextos particulares, dejando a los demás de lado, aunque se requiere que se abarquen todos los contextos para que tengan abstracciones y sintaxis independiente del dominio. Debido a esto, si se requiere otro contexto, el programador integra el código necesario; sin embargo, el código previamente escrito podría sufrir alteraciones y en el proceso perder el contexto. Para tratar este problema, los diseñadores de un lenguaje proporcionan una clara definición de las abstracciones para evitar la pérdida de contexto, o utilizar únicamente los elementos requeridos por el contexto.

1.1.5. 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 que se centra 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, se entiende a la expresividad como el nivel de descripción que tiene el código, esto con la intención de que los usuarios comprendan el objetivo para el cual se desarrolló el programa dentro del dominio en el que se aplica.

1.1.6. Ambigüedad

Según el diccionario de la real academia española:

"Cualidad de ambiguo"

Ambiguo:

"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 que se presenta en palabras, ideas o sintagmas, cuyo significado se vuelve susceptible a tener más de una interpretación, cada una sujeta al pensamiento del receptor del mensaje.

1.1.7. Programación naturalística

Se define a la programación naturalística como al paradigma que toma elementos de los lenguajes naturales para diseñar lenguajes formales de programación más expresivos desde la perspectiva humana, la intención de esto es reducir la brecha existente entre el planteamiento del problema y la solución del mismo.

La programación naturalística se relaciona con elementos lingüísticos tomados del lenguaje natural para proporcionar un nivel más alto de expresividad, aunque este tipo de programación no logra erradicar la ambigüedad, la resuelve formalizando y restringiendo al lenguaje natural para lograr un subconjunto libre de ambigüedades [6].

1.1.7.1. SN

SN [1] es un lenguaje de programación naturalístico y de propósito general que se diseñó en el Tecnológico Nacional de México, Campus Orizaba. Este lenguaje presenta un subconjunto controlado del idioma inglés, lo que le permite diseñar código que sea más expresivo desde el punto de vista del lenguaje natural.

[1] Se describirá más a fondo en el capítulo 3

1.1.7.2. 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 pemiten 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 qué adjetivos se permiten para la composición, qué adjetivos se requieren o qué adjetivos no se permiten [6].

1.2. Situación tecnológica, económica y operativa de la empresa

El instituto tecnológico de Orizaba (ITO) es una institución que se fundó 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. Se encuentra ubicada en Oriente 9, Colonia Emiliano Zapata, CP. 94320 en la ciudad de Orizaba, Veracruz [7].

A partir del año 2016, el ITO pasó a formar parte del Tecnológico Nacional de México. Actualmente este plantel ofrece siete especialidades a nivel de licenciatura: Ingeniería Eléctrica, Electrónica, Mecánica, Industrial, Química, Sistemas Computacionales e Informática. 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, habiendo consolidado

la enseñanza a nivel superior; el Instituto Tecnológico de Orizaba implanta sistema de calidad en busca de la excelencia educativa.

1.3. Planteamiento del problema

Actualmente en el desarrollo de aplicaciones, los requisitos no funcionales (NFR) suelen encontrarse dispersos entre las diversas abstracciones del código, lo cual vuelve complejo el control y la documentación de estos. Es por esto, que surgió el paradigma orientado a aspectos, el cual permite encapsular los NFR en entidades bien definidas. El principal exponente del paradigma orientado a aspectos es AspectJ, el cual presenta los elementos mínimos necesarios para tener similitudes con un lenguaje naturalístico.

Ya que el lenguaje naturalístico SN es una nueva propuesta, no se reporta un marco de referencia que permita analizar el comportamiento de las circunstancias como método de encapsulación para requisitos no funcionales, por otro lado, AspectJ permite encapsular requisitos no funcionales de forma eficiente, aunque su mecanismo se basa en la sintaxis y no en el contexto, de modo que posee la limitante de la fragilidad.

Es por esto que se propone un análisis comparativo, por medio de ejercicios y un caso de estudio, que permita estudiar las similitudes, diferencias, ventajas y desventajas que se presentan en ambos métodos de encapsulamiento de los NFR.

1.4. Objetivos

En esta sección se describen el objetivo general y los objetivos específicos de esta tesis.

1.4.1. Objetivo general

Estudiar las propiedades naturalísticas del lenguaje SN y las capacidades de corte del lenguaje AspectJ para la encapsulación de requisitos no funcionales por medio de un caso de estudio que permita establecer las ventajas y limitaciones entre ambos lenguajes.

1.4.2. Objetivos específicos

- Identificar las ventajas y limitaciones de las circunstancias del lenguaje SN en la encapsulación de requisitos no funcionales por medio de la comparación con los cortes de AspectJ.
- Mostrar las capacidades de expresividad que tienen los cortes en AspectJ.
- Identificar las capacidades de expresividad en el lenguaje SN por medio de ejemplos equivalentes a los cortes en AspectJ.
- Desarrollar un caso de estudio que permita implementar requisitos no funcionales en ambos lenguajes.
- Analizar las diferencias y similitudes entre la solución propuesta con circunstancias del lenguaje SN y cortes en AspectJ.

1.5. Justificación

Los lenguajes de programación orientados a objetos tienen como limitante que son incapaces de encapsular los requisitos no funcionales, los cuales se encuentran dispersos por todas las abstracciones que conforman el sistema, lo que dificulta el mantenimiento. Como solución a este problema, surgió la programación orientada a aspectos (POA), la cual permite encapsular los requisitos no funcionales en entidades bien definidas y reutilizables, lo que limita la dispersión del código. AspectJ es el mayor exponente de la POA, el cual es un

lenguaje de programación de propósito general y extensión orientada a aspectos para Java.

Sin embargo, AspectJ presenta desventajas, como por ejemplo la especificación de cortes se basa en la sintaxis de Java, particularmente en las firmas de los métodos de dicho lenguaje, lo que provoca que al momento de cambiar un requisito que afecte a la estructura o la firma de los métodos de Java, se requiera adaptar la especificación en AspectJ para evitar una propagación incorrecta de la funcionalidad complementaria. La expresividad en los cortes de AspectJ presenta los elementos básicos, deseables en otros lenguajes, que permiten programar con mayor proximidad en un lenguaje natural.

La programación naturalística es una técnica de programación que ofrece la capacidad de reducir la brecha entre el dominio del problema y la solución, lo que permite generar código con un nivel de expresividad que se asemeja al de los lenguajes naturales. Además, los lenguajes naturalísticos permiten que el código sea autodocumentado, y al mismo tiempo mantienen la formalidad de los lenguajes de programación.

El lenguaje de programación naturalístico SN posee un mecanismo que permite agregar funcionalidad complementaria, las circunstancias, las cuales se basan en la sintaxis. SN permite un mayor nivel de variación en las firmas, ya que se basa en un subconjunto del idioma inglés.

AspectJ permite encapsular de forma correcta lo requisitos no funcionales para evitar la dispersión en el código, sin embargo, la sintaxis que presenta es poco expresiva. Además de que las especificaciones de cortes son frágiles, ya que dependen de la firma y son susceptibles a los cambios del código en Java. Mientras que las circunstancias de SN, aunque se basan en sintaxis, no presentan la fragilidad, ya que, el problema se evita al asociar directamente

la firma del verbo que provee la funcionalidad con otro verbo que provee su contexto, y al ser SN un lenguaje naturalístico, el nivel de expresividad es mayor.

Capítulo 2

Estado de la práctica

En este capítulo se dan a conocer los trabajos relacionados a los temas de requisitos no funcionales, así como el trabajo que se realiza con tecnologías orientadas a aspectos y naturalísticas.

2.1. Trabajos relacionados

Se conoce a los requisitos no funcionales como aquellos que definen el comportamiento del sistema. Estos requisitos suelen documentarse de forma diferente a los requisitos funcionales, ya que no tienen medidas cuantitativas y se considera que sus descripciones suelen ser vagas. En [8] los autores explicaron que los NFR son importantes para el desarrollo de software, sin embargo, el problema es que su manejo se basa principalmente en lograr diferenciarlos de los requisitos funcionales. Por otra parte, se mencionó que, ya que los NFR describen comportamientos, deberían contar con una documentación similar a los funcionales para permitir su estudio. En el artículo se reportó un experimento, en el cual se tomaron en cuenta 530 NFR de 11 especificaciones industriales, como resultado se obtuvo que el 75 % de los requisitos que se clasificaron como no funcionales describían el comportamiento del sistema y solo el 25 % la representación del sistema. Como con-

clusiones resultantes del análisis de los 530 casos de NFR, los autores presentaron que si bien los requisitos no funcionales no son iguales a los requisitos funcionales y no deben recibir el mismo tratamiento, sin embargo, deberían manejarse de una manera similar a los funcionales para integrarse de manera adecuada en el software.

En la actualidad, muchos de los sistemas se ven en la necesidad de migrar su forma de trabajo para que se logre trabajar desde diferentes partes del mundo. El uso de la nube es cada vez más popular, pues múltiples servicios, como por ejemplo, los que son para pedidos, ya se utilizan en este medio. Sin embargo, en [9] propusieron que las aplicaciones industriales ya existentes tendrían que beneficiarse también de la nube. Un enfoque de migración para aplicaciones heredadas es la configuración de la máquina virtual (MV) para cada cliente, esto incluido el programa completo, el servidor a la base de datos, pero este enfoque es costoso tanto en recursos como operativamente. La tenencia múltiple sería la opción más viable en cuanto a costo, ya que los clientes comparten los recursos, sin embargo, es complejo para la recomposición de las aplicaciones heredadas. Como solución se propone agregar componentes que se encarguen de los problemas que se presenten en la tenencia múltiple, tales como son los aislamientos del cliente, la autenticación y la personalización, sin la necesidad de recomponer la aplicación. Para los componentes, se utilizaron aspectos, específicamente AspectJ. AspectJ es una tecnología que permite desarrollar por medios sistemáticos software de manera modular. AspectJ es un lenguaje que complementa con aspectos a programas escritos en Java, esto para evitar la dispersión de código, además de que permite cambiar la estructura dinámica de un programa al interceptar ciertos puntos del flujo del programa, llamados puntos de unión. Ejemplos de puntos de unión son las llamadas o ejecuciones de método y constructor, los accesos a campos y las excepciones. Los puntos de unión se especifican sintácticamente mediante corte en puntos.

La evolución de los lenguajes de programación se impulsó, ya que existe una necesidad de lograr una mejor separación de asuntos (del ingles *Separation of Concerns*, SoC). Los expertos en ingeniería de software sugieren que la mejor manera de tratar la SoC es con la correcta descomposición del sistema en módulos que sean capaces de acoplarse entre sí, pero al mismo tiempo sean capaces de un aislamiento relativo. Con lo que se mencionó anteriormente, se espera una mejor compresión y un mayor potencial para la reutilización de código. En [10] los autores explicaron que lo sistemas son cambiantes, ya que durante su ciclo de vida los usuarios tienen diferentes necesidades, y por ellos los requisitos cambian. El contenido funcional de un sistema debe aumentarse continuamente o el sistema se vuelve cada vez menos útil. El problema resultante de los cambios en aumento de funcionalidades es que los módulos se cargan de responsabilidades, lo cual reduce la facilidad de adaptar nuevos requisitos, además de que el costo de mantenimiento aumenta. Se abordó el tema de las pruebas que se realizaron con la programación orientada a aspectos, ya que los resultados son diversos y por lo tanto, no concluyentes. Es por esta razón que los autores propusieron un experimento, mediante el uso de Telecom. Telecom es un simulador de sistema de telefonía que permite a los clientes llevar acabo actividades como realizar, aceptar, fusionar y colgar llamadas. Los autores construyeron varios módulos compuestos por código en Java y AspectJ para cada una de las funcionalidades que se mencionaron anteriormente.

Actualmente los sistemas de software son complejos, pues capturar de manera correcta la estructura y procesos que son necesarios para el contexto en el que se utilizan, resulta una tarea difícil, ya que para lograrlo es necesario contar con experiencia previa en el dominio. Videira et. al. [11] señalaron que: “el núcleo del problema es que los mecanismos de descripción existentes (lenguajes de programación) son inadecuados cuando se trata de transmitir información relevante a las personas sobre sistemas de software”, además mencionan que otro de los problemas centrales en el desarrollo de software es la falta de apoyo

para la comprensión del programa entre las personas que colaboran en el proyecto. La propuesta para solucionar los problemas que se presentan en el software, es ampliar el trabajo que se realiza en la programación orientada a aspectos (POA), ya que con los lenguajes de programación actuales, los programadores tienen que expresar ideas con un límite en las referencias estructurales y reflexivas, sin tomar en cuenta las referencias temporales. Mientras que los lenguajes naturales poseen formas referenciales más temporales que las que la POA proporciona actualmente. Por ello los autores consideran que el siguiente paso en la POA es incluir las referencias temporales que se utilizan en los lenguajes naturales.

Las computadoras realizan sus operaciones cuando reciben código que esté en lenguaje máquina, el cual consiste en una serie de números binarios, pero este lenguaje es difícil de comprender desde la perspectiva humana, por lo tanto, la expresividad era nula. Como solución para el problema de la expresividad, se generaron lenguajes intermediarios, tales como ensamblador, pero aún la expresividad era poca. Los lenguajes de tercera generación se crearon para solucionar este problema mediante el uso de verbos. Tiempo después los lenguajes de programación orientada a objetos (POO) utilizaron sustantivos como identificadores de instancia. Los objetos permiten a los programadores definir abstracciones del mundo real que realizan tareas específicas definidas por verbos. En [12] se mencionó que sin embargo, aún con la transformación de los lenguajes, de código binario a lenguajes de alto nivel que permiten que la expresividad sea mayor, es necesario modelar los requisitos al estilo del paradigma en el cual se proponga la solución. Una vez que se desarrolló el modelo, los requisitos se traducen a código, lo cual provoca que se mantengan o se deformen. Por lo tanto, la expresividad de los requisitos es muy baja. Como solución al problema de la baja expresividad se propuso el uso de lenguaje natural (inglés), aunque el uso de este lenguaje presenta otra problemática, la ambigüedad, ya que las computadoras no son capaces de interpretar el contexto en el que una oración se presenta. Así que como alternativa se presenta el uso de los lenguajes naturalísticos (los cuales se basan en elementos

de los lenguajes naturales), esta opción da como resultado la programación naturalística. Se define a la programación naturalística como el paradigma que toma elementos de los lenguajes naturales para diseñar lenguajes de programación más expresivos desde la perspectiva humana, la intención de esto es reducir la brecha existente entre el planteamiento del problema y la solución del mismo. Se relaciona con elementos lingüísticos tomados del lenguaje natural para proporcionar un nivel más alto de expresividad, aunque este tipo de programación no logra erradicar la ambigüedad del lenguaje natural, la resuelve formalizando y restringiendo al lenguaje natural para lograr un subconjunto libre de ambigüedades.

Los usuarios finales que deben representar datos tabulares en hojas de cálculo no suelen tener los conocimientos de programación para realizar estas tareas de manera automática. Como solución al problema que se planteó, Gulwani et. al. [13] propusieron una metodología que implica el diseño de un lenguaje específico de dominio tipificado (DSL) que soporte un álgebra de mapa expresiva, además de filtrar, reducir, unir y formatear capacidades a un nivel de abstracción apropiado para usuarios no expertos. El componente clave es un algoritmo de traducción para convertir una especificación de lenguaje natural en el contexto de una hoja de cálculo dada a un conjunto de programas probables. El DSL se estructura en torno a un álgebra central, filtro y mapa que tienen inspiración en el lenguaje SQL (Structured Query Language; Lenguaje de Consulta Estructurada), para programar en hojas de cálculo. Un programa en el DSL lee y actualiza la hoja de cálculo sobre la cual se ejecuta, esta hoja se modeló como tablas, donde cada una de estas es un conjunto de filas y columnas (que se encuentran etiquetadas y tipificadas de forma única), cada celda tiene atributos de formato que incluyen atributos booleanos, así como otras características que permiten dar formato (negrita, subrayado, color y tamaño de letra). Para traducir la entrada del lenguaje natural de un usuario en un conjunto de programas probables en el DSL, los autores del artículo describen su algoritmo y los sub-algoritmos

necesarios. El algoritmo principal toma como entrada una oración en inglés y una hoja de cálculo y devuelve una lista ordenada de expresiones de alto nivel, programas, que interpreten la oración. Este algoritmo se basa en programación dinámica e iterativa que calcula el conjunto de todas las expresiones que se producen para sub-secuencias contiguas mayores, luego devuelve un conjunto de expresiones ordenadas por puntuación. El algoritmo de síntesis es el tipo de Synth que genera todas las composiciones de tipo seguro de las expresiones que se preparan de forma recursiva a partir de fragmentos pequeños. La regla del algoritmo de traducción se basa en un conjunto de reglas de patrones que construyen expresiones basadas en palabras coincidentes en la entrada de usuario y los conjuntos de expresiones previamente calculadas.

Knöll y Mezini [14] mencionaron que existe una brecha entre las expectativas que mantienen los desarrolladores y las técnicas que se utilizan para programar, lo cual se debe a cuatro problemas que son:

- **Problema mental:** los desarrolladores tienen una percepción inicial de la solución que se pretende dar al programa, sin embargo, esta idea se transforma a manera de que sea comprensible para el compilador del lenguaje en que se llevará a cabo el software. Esta doble transformación obliga al desarrollador a deformar su idea principal, hasta que se ajuste al lenguaje de programación.

- **Problema del lenguaje de programación:** la evolución de los lenguajes se vuelve compleja, ya que hay que traducir algoritmos ya existentes a nuevas tecnologías, pero en algunos casos, gente ajena al área informática se ve en la necesidad de realizar pequeños programas para sus investigaciones y sin el conocimiento de los lenguajes modernos, desarrollan su trabajo en lenguajes antiguos, lo cual provoca que estos aún tengan una fuerte presencia.

- **Problema del lenguaje natural:** actualmente los equipos de desarrollo suelen tener integrantes de diferentes partes del mundo, lo cual provoca que su idioma nativo sea diferente, inclusive que se encuentre regionalismos que impidan que haya una correcta comunicación.

- **Problema técnico:** si bien lo más recomendable para el desarrollo es que los participantes en el proyecto aporten ideas originales que se implementen en el código, actualmente esto no es así; ya que mucho del tiempo se invierte en implementar y depurar el programa.

Pegasus es una herramienta que facilita el desarrollo de software a partir del lenguaje natural, que permite introducir texto en lenguaje natural, luego realiza el análisis que devuelve código ejecutable. Pegasus usa tres características básicas para funcionar: lee el texto que se expresó en lenguaje natural, genera código fuente y por último expresa lenguaje natural. Actualmente esta herramienta permite generar programas simples en inglés y en alemán. Pegasus analiza la estructura gramatical de una oración para obtener sus ideas y relaciones, y luego identifica el contexto de la oración. Una idea aparece como una entidad, acción o propiedad. Pegasus analiza una idea comparando el significado de la idea de oración abstracta con las ideas almacenadas en un diccionario de entidades. Luego, Pegasus descompone la idea en sub-ideas para resolver su significado. Una vez que se resuelve una idea, genera código Java, aunque no se limita a este lenguaje. Para generar texto en otro lenguaje natural, Pegasus compara la entrada con el diccionario de ideas y genera una salida.

En [15] se describió a la programación como una actividad compleja, ya que para desarrollarla, el programador debe especificar a detalle, por medio de instrucciones, el comportamiento general del sistema y cualquier error en el código generará un fallo. Los autores presentaron Macho, una tecnología que permite el desarrollo de programas simples

con lenguaje natural, esto por medio del análisis de texto y posterior transformación en consultas. Una vez que se obtienen las consultas, se seleccionan los fragmentos de código fuente para la construcción del programa. Macho cuenta con subsistemas que permiten llevar a cabo el proceso que se explicó anteriormente, los cuales son:

- **Analizador sintáctico del lenguaje natural:** el cual revisa los verbos que se traducen como acciones, y nombres que implican objetos.
- **Base de datos:** Una biblioteca que almacena más de 1,200 fragmentos de código a partir del cual los fragmentos candidatos se seleccionan.
- **Stitcher(engrapadora)**: Combina los fragmentos de código para generar programas candidatos.
- **Depurador automático:** Depura los fragmentos de código por medio de pruebas dinámicas que le permiten el acceso al comportamiento del programa.

Macho analiza dos expresiones por medio de una variable y combina fragmentos de código, si la salida coincide con el tipo de entrada o por medio del subsistema. Macho provee un limitado uso de control, donde el analizador sintáctico de lenguaje natural genera expresiones `if` y el sintetizador infiere, generando ciclos si el sistema lo sugirió.

Para cumplir con la seguridad en el software es necesario tomar en cuenta dos cosas, que son: los requisitos y las amenazas de seguridad. Mai et al. [16] mencionaron que las pruebas de seguridad se dividen en dos categorías: las pruebas funcionales, con las cuales se valida que las propiedades de seguridad se aplicaron de manera correcta y las pruebas de vulnerabilidad, en las cuales se trata de identificar las posibles debilidades de la seguridad. Por medio del uso de lenguaje natural (NL), los autores proponen MCP (Misuse Case Programming, Programación de casos de uso indebido), con este enfoque se generan pruebas de vulnerabilidad para especificaciones de casos de uso que son incorrectas. El prototipo de

MCP genera casos de pruebas, que son simulaciones de ataques cuyo objetivo es encontrar vulnerabilidades en el sistema. El prototipo se enfoca únicamente en casos en los que los requisitos de seguridad se manifiestan con lenguaje natural. MCP requiere como entrada un conjunto de especificaciones de casos de mal uso y una API (*Application Programming Interface*, Interfaz de Programación de Aplicaciones) de controlador de prueba y genera automáticamente un conjunto de casos de prueba ejecutables que simulan las actividades descritas en las especificaciones de casos de mal uso. MCP es una solución de programación en lenguaje natural que traduce automáticamente cada paso en las especificaciones del caso de uso incorrecto en instrucciones ejecutables.

Se considera que el uso del lenguaje natural facilitaría a los seres humanos aprender a programar con la capacitación mínima. Aunque de igual manera se presenta el problema de que los humanos no suelen mantener una secuencia para la ejecución de las órdenes, pero sí proporcionan una narrativa que expresa lo más importante. Si bien los humanos son capaces de, por medio del conocimiento empírico, identificar qué actividad se realiza primero, la computadora ejecuta según el orden que se le indica, esto se tiene que evitar para permitir la expresión natural. Los lenguajes naturales permiten expresar de forma más entendible, para los seres humanos, las instrucciones que se desean procesar. Landhäuÿer y Hug [17] desarrollaron el proyecto AliceNLP (*Natural Language Programming*). El objetivo principal de AliceNLP es introducir a los niños a la programación orientada a objetos, por medio de una biblioteca de objetos tridimensionales, los cuales con *drag and drop*, se utilizan para armar animaciones. La idea del método de trabajo que presenta el proyecto es la comprensión de lo que desea el usuario sin importar la enseñanza de un lenguaje de programación. Para solucionar el problema de la falta de indicaciones secuenciales, AliceNLP enumera los pasos individuales y los usuarios describen en qué parte de la secuencia deben colocarse acciones específicas.

Los lenguajes de programación en la actualidad tienen su inspiración en los lenguajes naturales, ya que, por ejemplo, la POO (Programación Orientada a Objetos) tiene su base en la perspectiva del mundo, y POA (Programación Orientada a Aspectos) se basa en la definición de reglas. Los lenguajes naturales cuentan con ciertas desventajas, tales como la ambigüedad y la redundancia, sin embargo, es posible reducir estos problemas por medio de la combinación de otros términos lingüísticos. En [18] se definió un tipo naturalista como un conjunto de cualidades, representables por predicados lógicos, que todas las instancias, del tipo respectivo, deben cumplir para pertenecer a ese tipo. Los autores mencionaron que los identificadores en el lenguaje natural se describen por las cualidades que se le atribuyen, por ejemplo, en el idioma inglés se definen sustantivos (conceptos del mundo real), los cuales se describen con mayor detalle por adjetivos. El estudio del lenguaje natural permite la creación de técnicas que eviten la redundancia, lo que permitiría mejorar la programación naturalística en sistemas pequeños. Ya que actualmente los lenguajes de programación se centran en problemas de diseño medianos o grandes. A una escala mayor, se cree que el lenguaje permite asegurar la correcta estructuración de información.

La documentación de los programas de software resulta una actividad tediosa, además de que suele tomar mucho tiempo, por otro lado, también suele ser compleja, sobre todo cuando hay varias modificaciones, hechas por diferentes personas que se involucran en el proyecto. La documentación se considera muy importante, ya que, describe el software en sus diferentes atributos, sin embargo, mucha de la documentación existente en la actualidad, se centra principalmente en el usuario final, y no explica de forma clara el comportamiento interno de la aplicación. Para solucionar el problema de la escasa documentación del código, en [19] se propuso el uso de un subconjunto controlado del idioma inglés, en el cual por medio de declaraciones, se describiera la estructura y el entorno del software. El nombre de este lenguaje es Attempto Controlled English (ACE). La idea

central de ACE es documentar el código mediante declaraciones, semejantes al lenguaje natural, que se controlan en los niveles léxico, sintáctico y semántico para permitir la traducción automática y no ambigua a notaciones lógicas formales. Idealmente se espera que las especificaciones propuestas en ACE se encuentren al lado del código que describen, en forma de comentarios pero que al mismo tiempo, el IDE (*Integrated Development Environment*, Entorno de Desarrollo Integrado) sea capaz de procesarlo, así los cambios en el código que generen incongruencias, serían detectados más fácilmente. Los autores identifican tres desafíos que se presentan para permitir un uso de ACE efectivo y confiable:

- **Validación:** dentro del enfoque que se presentó, esta se logra ya que la documentación se verifica de forma automática e incremental cada vez que se agrega o elimina algo.

- **Retroalimentación inmediata:** ACE realiza la carga inicial del código fuente y la documentación, y a partir de esto solo permite modificaciones incrementales. Se toma a consideración que el tiempo entre modificaciones es menor al de procesamiento, lo que permite obtener los comentarios de manera rápida, cada vez que se presente una modificación.

- **Impacto mínimo en los ingenieros de software:** ya que ACE es un conjunto controlado del idioma inglés, cuando se cuenta con un conocimiento previo del lenguaje, requiere menos esfuerzo para la comprensión de lo que expresa en él. La interacción con las computadoras actualmente utiliza tareas repetitivas para el desarrollo de programas pequeños. Los programadores tienen que mantenerse en constante aprendizaje del uso de lenguajes de dominio específicos (DSL) para realizar las tareas

necesarias de forma efectiva.

La síntesis tradicional de programas es la tarea de simplificar automáticamente un programa en algún DSL a partir de especificaciones completas. Este tipo de síntesis es difícil ya que no es posible verificar que la síntesis concuerde correctamente con la especificación que se proporcionó. Desai et al. [20] presentaron un marco general para la construcción de sintetizadores de programas. Este marco toma entradas en lenguaje natural y definiciones DSL a partir de las cuales crea un sintetizador al aprender de las clasificaciones de los resultados de una traducción basada en programación de palabras clave. En el artículo se abordaron los problemas de la síntesis de programas de un DSL a un lenguaje natural. Ya que los lenguajes naturales son imprecisos, no es posible garantizar que el programa que se sintetizó sea corregible. Es por esta razón que se genera un conjunto de programas clasificados que permite al usuario inspeccionar el código fuente del programa, además de permitir realizar pruebas de ejecución y posteriormente seleccionar el programa que mejor se ajuste a la especificación que desea. Sin embargo, esta herramienta aún presenta algunas limitantes, sobre todo, la dificultad de realizar la traducción cuando el lenguaje natural no tenga correspondencia directa con el DSL, pues el traductor se verá con una reducción en la eficacia. Además de que el DSL debe ser funcional y no debe contener variables temporales.

Los lenguajes de programación tienen el problema de la implementación de ideas de diferentes maneras según el lenguaje de programación. Otro problema que hay es que cuando los grupos de trabajo son internacionales, la documentación se debe realizar en inglés, lo cual provoca que muchas veces esta actividad sea lenta y poco productiva. Por último está el problema de que los desarrolladores pasan la mayor del tiempo en la descripción del programa. Es importante conocer la teoría que proveen los lenguajes de programación, para lograr expresar de manera adecuada los requisitos, pero el aprendizaje de su sintaxis y estructuras no debería ser un problema para la productividad. Es por esto que se propuso

un enfoque que genera código a partir de un lenguaje independiente. Además de que para proporcionar la independencia del lenguaje, el enfoque utiliza un enriquecimiento de un modelo semántico, el cual representa ideas similares en diferentes lenguajes naturales. El enriquecimiento implementa código XML que representa casos de uso. El enfoque que se planteó en [21] soporta casos de uso en los idiomas inglés y francés. La experimentación se planteó con cinco casos de estudio pertenecientes a sistemas diferentes, la evaluación mostró cuantitativamente la capacidad de extraer entradas que se ajusten a las solicitudes del experto para generar elementos en código XML que se ajusten a la entrada.

La programación naturalística pretende proveer a los usuarios la facilidad de escribir código en su lenguaje nativo y traducirlo directamente a una especificación de un programa. Aunque aún este tipo de programación es una actividad desafiante, ya que el compilador tiene la necesidad de detectar errores, sintácticos, gramaticales, además de problemas de ambigüedad. En [22] se describió un prototipo de sistema de programación en lenguaje natural para juegos de computadora / video, en el cual, se presentan oraciones escritas en inglés que se convierten automáticamente en un código de juego funcional y ejecutable en JavaScript. Los errores inherentes a un lenguaje, tales como el error sintáctico o semántico, se informan al usuario, al momento de la compilación, con el fin de mejorar el juego. Los autores afirmaron que con menos de 20 oraciones simples en inglés, se pueden generar hasta 1800 líneas de código de juego. Las restricciones clave que se colocan en el lenguaje es que cada cláusula debe incluir objetos. Un objeto es una entidad identificable utilizada en un videojuego, como los personajes específicos, la puntuación, entre otros. Se tiene en cuenta que si bien estos objetos son sustantivos en NL, no todos los sustantivos en NL son objetos válidos en un videojuego. Por lo tanto, los autores llamaron a este lenguaje natural orientado a objetos CNL, o simplemente OONL.

2.2. Análisis comparativo

Los trabajos que se reportan, presentan problemas que se relacionan a los requisitos no funcionales, además de abordar una tecnología que permite encapsular los anteriormente mencionados. De igual forma se documentan lenguajes emergentes cuyo objetivo es programar de forma que las expresiones sean similares a las del lenguaje natural. Sin embargo, ninguno de los artículos, realiza un enfoque comparativo entre los lenguajes naturalísticos y el lenguaje AspectJ, como se observa en la tabla 2.1.

Tabla 2.1: Análisis comparativo de los trabajos relacionados al tema de tesis

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Eckhardt et al. [8]	La documentación de los requisitos no funcionales no siempre está completa.	Estudio de diferentes casos de estudio que permiten verificar el comportamiento de los NFR.	No se mencionan.	Los requisitos no funcionales deberían manejarse de forma similar a los funcionales para integrarse de manera adecuada en el software.	Finalizado
Hohenstein, y Koka [9]	Se desea realizar la migración de una aplicación heredada a la nube, lo cual implica reconfigurar diversos elementos a cada cliente.	El uso de aspectos para la creación de componentes: aislamiento del cliente, la autenticación, y la personalización.	▪ AspectJ	Componentes que se encarguen de los problemas que se presenten en la tenencia múltiple, sin la necesidad de recomponer la aplicación.	Finalizado

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Przybylek, Adam [10]	La evolución de los lenguajes de programación se impulsó ya que existe una necesidad de lograr una mejor separación de preocupaciones (SoC).	Un experimento por medio de Telecom para verificar si el uso de aspectos provee resultados concluyentes.	▪ AspectJ ▪ Telecom	Evaluación empírica cuyo primer experimento contó con 35 sujetos, a los que se les pidió comprendieran código de objetos y código de aspectos. El tiempo de finalización fue un 29 % más largo para el grupo AO que para el grupo OO.	Pruebas
Videira et al. [11]	Los lenguajes de programación actuales son inadecuados para transmitir información relevante para los clientes.	Análisis de la sintaxis de AspectJ y propuesta para ampliar el trabajo de la programación orientada a aspectos.	▪ AspectJ	Se concluye que si se trabajara más en la sintaxis de AspectJ, este podría ser más expresivo.	Finalizado

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Pulido-Prieto y Juárez-Martínez [12]	La poca expresividad existente en los lenguajes de programación.	Se propone el uso de lenguajes naturalísticos como solución al problema de expresividad.	▪ Pegasus ▪ Metafor ▪ Macho ▪ HyperTalk ▪ Nlyze	Se presenta un modelo de programación naturalística.	Finalizado
Gulwani, y Marron [13]	Los usuarios deben representar datos tabulares, sin embargo, no siempre tienen los conocimientos para realizar estas tareas.	Lenguaje específico de dominio tipado (DSL) que soporte un álgebra de mapa expresiva.	▪ Hojas de cálculo ▪ SQL	Modelo de interacción con soporte para resolución de ambigüedades, secuenciación de programas de comunicación DSL e integración con técnicas de programación.	Mejoras

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Knöll y Mezini [14]	Brecha entre las expectativas que mantienen los desarrolladores y las técnicas que se utilizan para programar.	La herramienta Pegasus que facilita el desarrollo de software en lenguaje natural.	No se mencionan.	Pegasus que soporta el código en dos lenguajes, inglés y alemán.	Mejoras
Cozzie et al. [15]	El programador tiene que especificar a detalle el comportamiento general del sistema.	La tecnología Macho que permite el desarrrollo de programas simples con lenguaje natural.	No se mencionan.	La tecnología Macho.	Finalizado.

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Mai et al. [16]	La seguridad del software debe tomar en cuenta los requerimientos no funcionales y las vulnerabilidades de seguridad.	MCP que genera casos de prueba que simulan ataques para encontrar vulnerabilidades.	No se mencionan.	La tecnología MCP.	Mejoras.
Landhäuÿer y Hug [17]	Las computadoras no cuentan con la capacidad de identificar el orden en que las actividades se realizan.	La tecnología AliceNLP que se utiliza para armar animaciones por medio de una biblioteca de objetos tridimensionales.	No se mencionan.	Método de trabajo, que presenta el proyecto, es la comprensión de lo que desea el usuario sin importar la enseñanza de un lenguaje de programación.	Finalizado

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Knöll et al. [18]	Los lenguajes naturales cuentan con ciertas desventajas, tales como la ambigüedad y la redundancia.	La reducción de estos problemas, por medio de la combinación de otros términos lingüísticos.	No se mencionan.	Especificación de términos importantes para un lenguaje naturalístico.	Finalizado.
Kuhn y Bergel [19]	La documentación de software se considera una actividad tediosa y muchas veces no se presenta de forma correcta.	Lenguaje Attempto Controlled English (ACE) que es un subconjunto del idioma inglés.	▪ AceWiki.	El lenguaje ACE es un subconjunto controlado del idioma inglés, en el cual por medio de declaraciones, se describe la estructura y el entorno del software.	Mejoras.

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Desai et al. [20]	La síntesis tradicional es difícil, ya que no es posible verificar que concuerde correctamente con la especificación que se proporcionó.	Un marco general para la construcción de sintetizadores de programas.	No se mencionan.	Marco que toma entradas en lenguaje natural y definiciones DSL a partir de los cuales crea un sintetizador.	Mejoras.
Mefteh et al. [21]	Es necesario aprender la sintaxis y estructuración para expresar de forma adecuada los requerimientos.	Evaluación que mostró cuantitativamente la capacidad de extraer entradas que se ajusten a las solicitudes del experto para generar elementos en código XML que se ajusten a la entrada.	▪ XML	Enfoque que soporta casos de uso en los idiomas inglés y francés.	Mejoras.

Continúa en la siguiente página

Tabla 2.1 – *Continuación de la página anterior*

Artículo	**Problema**	**Contribución**	**Tecnologías**	**Resultado**	**Estado**
Hsiao [22]	La programación naturalística es una actividad desafiante, ya que el compilador tiene la necesidad de detectar errores, sintácticos, gramaticales, además de problemas de ambigüedad.	Prototipo de sistema de programación en lenguaje natural para juegos de computadora / video, en el cual, se presentan oraciones escritas en inglés.	▪ JavaScript	Lenguaje natural orientado a objetos CNL, o simplemente OONL.	Finalizado.

2.3. Solución propuesta

En esta sección se presenta una descripción de la propuesta de solución al problema, así como las tecnologías que se analizaron para el diseño de la metodología a emplear para el correcto desarrollo de la tesis.

2.3.1. Justificación de la solución seleccionada

Se seleccionó la siguiente propuesta, tomando en cuenta la compatibilidad, desempeño y las capacidades que ofrece la metodología de software.

Lenguaje orientado a aspectos: AspectJ [23], es el lenguaje de POA que cuenta con soporte actualizado. Además, de que los avances en cuestión de aspectos se realizan para este lenguaje.

Lenguaje naturalístico: SN es un lenguaje naturalístico que cuenta con un mecanismo que permite la encapsulación, mediante el manejo de eventos de los requerimientos no funcionales [24].

IDE: Eclipse [25] a pesar de ser ligero, su trabajo por medio de *plug-ins* lo vuelve una herramienta robusta. Además, Eclipse permite trabajar con las actualizaciones de AspectJ.

Metodología de desarrollo: *Theme Approach* permite identificar comportamientos en forma de "acciones" en los requisitos. Si una acción incide sobre un requisito de forma que la misma indique una característica, entonces dicha acción es un tema. Además, si un requisito incide sobre más de un tema y uno de los temas provee (o corta) la funcionalidad de otros temas, entonces dicho tema es un aspecto [26].

2.3.2. Metodología para el desarrollo de la tesis

En esta sección se detalla de forma secuencial la lista de actividades que se realizaron durante el desarrollo de la tesis. Las actividades que son necesarias para el correcto desarrollo de la tesis se detallan a continuación:

1. Análisis del estado del arte de los trabajos que se relacionan con los aspectos.
2. Análisis del estado del arte de trabajos relacionados a la programación naturalística.
3. Revisión de la documentación de AspectJ respecto a las características que ofrece para la encapsulación de requerimientos no funcionales.
4. Revisión de la documentación de SN respecto a las características que proveen las circunstancias para la encapsulación de requerimientos no funcionales.
5. Revisión de la documentación de Theme Approach.
6. Realización de un conjunto de ejercicios en AspectJ.
7. Realización del conjunto de ejercicios de AspectJ en SN.
8. Análisis comparativo de los ejercicios que se realizaron en AspectJ y SN
9. Selección de un caso de estudio industrial que permita analizar el encapsulamiento de requerimientos no funcionales.
10. Realización de pruebas con el caso de estudio seleccionado.
11. Escritura del análisis comparativo entre las circunstancias y los aspectos.

2.3.3. Justificación de la metodología para el desarrollo de la tesis

La metodología que se seleccionó para el desarrollo de la tesis tuvo como base un plan de trabajo que permite identificar las diferentes capacidades que proveen AspectJ y SN para el encapsulamiento de los requerimientos no funcionales, para posteriormente probar cada una de ellas mediante ejercicios y un caso de estudio con el fin de encontrar las similitudes, diferencias, ventajas y desventajas.

Capítulo 3

Aplicación de la metodología

En este capítulo se presenta de forma detallada el desarrollo de la metodología de la tesis, por medio de un conjunto de ejercicios que permitieron verificar las características de ambos lenguajes que se aplicaron en el caso de estudio.

3.1. Revisión de las características que ofrece AspectJ

AspectJ es un lenguaje de programación orientado a aspectos para Java. AspectJ permite encapsular requerimientos no funcionales en entidades bien definidas, aspectos.

3.1.1. Avisos

Un aviso es un fragmento de código que se ejecuta al momento de detectar un punto de unión. Los avisos son de tres tipos:

- **Before:** Introduce comportamiento antes de que se realice la ejecución que se establece en el punto de unión.

- **Around:** Permite realizar la intervención y sustitución del comportamiento del punto de unión. Es necesario indicar el mismo valor de retorno del elemento del cual se sustituirá la ejecución, también es posible invocar el comportamiento original, mediante el uso de la directiva *proceed()*.

- **After:** Agrega comportamiento luego de que se realiza la ejecución del punto de unión.

3.1.2. Primitivas de corte

Una primitiva de corte es un elemento con el cual es posible definir el *joinpoint* en donde se intervendrá la ejecución del programa. AspectJ provee diecisiete primitivas con las cuales es posible satisfacer la implementación de requerimientos no funcionales.

3.1.2.1. call

La primitiva de corte *call* permite definir el punto de unión necesario para el momento en el que se realiza la llamada de un constructor o método; dicho de otra manera, el momento en el que se invoca un constructor o método. La sintaxis necesaria para el uso de esta primitiva es la siguiente:

```
call(* package.method(..));
```

Para ejemplificar el uso de la primitiva `call`, se presenta el ejercicio `Formula`. El código **??** en Java, presenta una clase de nombre `Formula`, en la línea 5 se define un método público que recibe dos argumentos: distancia y tiempo. En la línea 6 se imprime el resultado del cálculo de velocidad el cual es: `distancia/tiempo`. De igual manera se declara una segunda clase, línea 10, la cual tiene por nombre *main*, en este caso solo se declara el método `Main`, en la línea 12, se instancia el objeto `Formula` y en la número 13 se llama a `Speed`.

Código 3.1: Formula en Java.

label

```
package callExample;

public class Formula {

  public void Speed(int distance, int time){
    System.out.print(distance/time);
  }
}

class Main{
  public static void main(String [] args){
    Formula formula = new Formula();
    formula.Speed(10, 2);
  }
}
```

Por lo anteriormente descrito, la salida del programa será: 5.

En el código 3.2 se presenta el programa de AspectJ que modificará el comportamiento del programa en Java. En la línea 4 se define el corte en puntos en el método `speed()` que recibe dos valores enteros y se encuentra en la clase `Formula` del paquete `callExample`. Para complementar el comportamiento que se introducirá de forma dinámica, se usa el aviso `before` (antes), línea 6, sobre el corte que se definió previamente. En este aviso, se imprimen dos mensajes.

Código 3.2: Formula en Aspectos.

```
1 package callExample;
2
3 public aspect FormulaA {
4   pointcut corte():call(public void callExample.Formula.Speed
      (int, int));
5
6   before():cut(){
7     System.out.println("--- Speed calculation ---");
8     System.out.print("The Speed is: ");
9   }
10 }
```

La salida del programa, luego de aplicar el aspecto es:

```
--- Speed calculation ---
The speed is: 5
```

3.1.2.2. execution

La primitiva *execution* realiza el corte en el momento que se ejecutan los métodos o constructores. Esta primitiva cuenta con soporte para trabajar con los tres tipos de avisos. La sintaxis necesaria para definir este tipo de aviso es:

```
execution(* package.method(..));
```

Para el ejemplo de aplicación de esta primitiva, se presenta un programa que registra personas, estas deben contar con un DNI (Documento Nacional de Identificación) por lo cual se espera que se genere un número de identificación cada vez que una persona sea

inscrita. La clase persona del código 3.4, tiene tres métodos: `addPerson()` en la línea 5, en el que se recibe el nombre de la persona y posteriormente se imprime. El método `generateDNI()`, utiliza la biblioteca de `Math.random` para obtener el valor numérico que compone el DNI; en la línea 16 se llama al método `generateLetter()`, el cual recibe el número aleatorio para seleccionar una letra. En `main` se crea la instancia de `Person` y luego se agrega una persona.

Código 3.3: Generate DNI en Java.

```
1  package executionExample;
2
3  public class Person{
4
5    public void addPerson(String name){
6      System.out.println("The␣name␣is:␣"+name);
7    }
8
9     public void generateDni() {
10      String DNI="";
11            final int divisor = 23;
12
13            int numDNI = ((int) Math.floor(Math.random() *
                 (100000000 - 10000000) + 10000000));
14            int res = numDNI - (numDNI / divisor * divisor);
15
16            char letter = generateLetter(res);
17
```

```
18            DNI = Integer.toString(numDNI) + letter;
19
20            System.out.println(DNI);
21        }
22
23        private char generateLetter(int res) {
24            char letters[] = {'T', 'R', 'W', 'A', 'G', 'M', 'Y'
                 ,
25              'F', 'P', 'D', 'X', 'B', 'N', 'J', 'Z',
26              'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E'};
27
28            return letters[res];
29        }
30
31        public static void main (String [] args){
32          Person p = new Person();
33          p.addPerson("Luisa");
34        }
35
36  }
```

En pantalla es posible observar la siguiente salida:

```
The name is: Luisa
```

Sin embargo, aún es necesario que se genere el DNI luego de que se agregue a una persona, por esta razón el código 3.4 define el punto unión en la ejecución del método `addPerson()`, en la línea 7 el aviso `after` genera e imprime el DNI.

Código 3.4: Generate DNI en AspectJ.

```
package executionExample;

public aspect ExecAspect {

  pointcut cut():execution(* executionExample.Person.
     addPerson(String));

  after():cut(){
    System.out.println("The DNI generated is: ");
    Person p = new Person();
    p.generateDni();
  }
}
```

Por lo cual la salida en pantalla sería la siguiente:

```
The name is: Luisa
The DNI generated is:
12739469E
```

Tras la ejecución de los programas con los que se ejemplificaron el uso de las primitivas `call` y `execution`, pareciera que el comportamiento que se presenta es igual, sin embargo, la diferencia se encuentra en el momento sobre el que actúa el corte.

3.1.2.3. args

La primitiva `args` permite realizar la exposición de contexto de los valores correspondientes al método sobre el que interviene el corte. Por lo cual permite manipular los valores que se reciben como argumentos. La sintaxis que es necesaria para el uso de esta primitiva:

```
pointcut argument(int x): execution(void package.m(int))&& args(x);
```

Como escenario de prueba se presenta el ejercicio de tablas de multiplicar. En el código 3.5, se define una clase de nombre `Multiplication`, en la línea 5, se encuentra el método multiplicar que recibe un valor de tipo entero, mediante este se muestra completa la tabla de multiplicar.

Código 3.5: Multiplication en Java.

```
1  package argsExample;
2
3  public class Multiplication {
4
5    public void multiplication(int val1){
6      for(int i=1;i<=10;i++){
7        System.out.print(val1+"X"+i+"= ");
8        System.out.println(val1*i);
9      }
10   }
11
12   public static void main(String [] args){
13     new Multiplication().multiplication(6);
14   }
15 }
```

La impresión resultante es:

```
6X1= 6
6X2= 12
6X3= 18
```

```
6X4= 24
6X5= 30
6X6= 36
6X7= 42
6X8= 48
6X9= 54
6X10= 60
```

En el aspecto `MultiplicationAspect` línea 5, el corte en puntos que se define recibe un valor de tipo entero (al igual que el método que se interviene), en este se combinan las primitivas `execution` y `args` para que el corte se ejecute y sea posible manipular la variable que recibe. En la línea 7, el aviso `before` imprime un mensaje que indica el valor de la tabla de multiplicar, código 3.6.

Código 3.6: Multiplication en Aspectos.

```
1  package argsExample;
2
3  public aspect MultiplicationAspect {
4
5    pointcut cut(int val):execution(void argsExample.
        Multiplication.multiplication(int))&&args(val);
6
7    before(int a):cut(a){
8      System.out.println("Multiplication table is: "+a);
9
10   }
11 }
```

Luego de aplicar el aspecto, en pantalla se muestra lo siguiente:

```
Multiplication table is: 6
6X1= 6
6X2= 12
6X3= 18
6X4= 24
6X5= 30
6X6= 36
6X7= 42
6X8= 48
6X9= 54
6X10= 60
```

3.1.2.4. target

La primitiva `target` expone el contexto de un objeto que realiza una invocación, mediante la cual se hace referencia a métodos o variables que pertenecen a este.

```
pointcut corte(Objeto x): execution(void package.m(int)) && target(x);
```

El caso que se presenta requiere que cada vez que un juego se registre, se almacene en un `ArrayList<>`, cada vez que esto se realice se debe imprimir. Se declara la clase `Game` la cual tiene como atributos `title, category` y `clasification`, la clase cuenta con los métodos `get` y `set` de cada uno de estos. En la línea 36, el método `addGame()`, recibe un argumento del tipo `Game` y se agrega al `ArrayList<>`. En `main` se instancia la clase `Game` y se registran dos juegos.

Código 3.7: Game en Java.

```
1  package targetExample;
2
3  import java.util.ArrayList;
4
5  public class Game {
6
7    private String title;
8    private String category;
9    private String clasification;
10   ArrayList<Game> games = new ArrayList<Game>();
11
12   public ArrayList<Game> getgames(){
13     return games;
14   }
15
16   public String getTitle(){
17     return title;
18   }
19
20   public String getCategory(){
21     return category;
22   }
23
24   public String getclasification(){
25     return clasification;
26   }
```

```
27
28    public Game(){}
29
30    public Game(String title, String category, String
        clasification){
31      this.title = title;
32      this.category = category;
33      this.clasification = clasification;
34    }
35
36    public void addGame(Game game){
37      games.add(game);
38    }
39
40    public static void main (String [] arsg){
41      Game play = new Game();
42
43      Game game = new Game("Dungeon␣Defenders:␣Awakened","RPG",
          "+13");
44      play.addGame(game);
45      Game game2 = new Game("Roki","Aventura","+15");
46      play.addGame(game2);
47    }
48
49  }
```

De la ejecución de este programa no se obtiene ningún resultado en pantalla, mientras que se solicita que se vea la impresión de la lista de juegos cada que se agregue uno. Por lo

cual, se agrega el aspecto `GameAspect` (código 3.8), en la línea 10 se define el aviso `before`, para realizar la exposición de contexto es necesario combinar la primitiva `target` con `call` o `execution`, esto con el fin de especificar que el aviso se ejecutará cuando se cree una nueva instancia de la clase con el constructor que no recibe argumentos; de igual manera se asigna el valor del objeto que recibe `target` a la variable del mismo tipo que se definió la línea 7, para manipularla y utilizar los valores en otros avisos. Para complementar la funcionalidad que se requiere es necesario un corte que se aplique sobre `addGame()` esto se indica en la línea 14. En el aviso `after` que se encuentra en la línea 16, aquí es donde se hace uso del objeto que obtuvo para acceder a los datos que se almacenaron.

Código 3.8: Game en Aspect.

```
1  package targetExample;
2
3  import java.util.ArrayList;
4
5  public aspect GameAspect {
6
7    Game game;
8    ArrayList<Game> games = new ArrayList<Game>();
9
10   before(Game j):execution(targetExample.Game.new()) &&
       target(j){
11     game=j;
12   }
13
14   pointcut cut():execution(* targetExample.Game.addgame(Game)
       );
```

```
15
16   after():cut(){
17     games = game.getgames();
18     System.out.println("
          ---------------------------------------------------");
19     System.out.println("Add game");
20     for(Game i : games){
21       System.out.println("Game title: "+i.getTitle());
22     }
23   }
24 }
```

La impresión final, luego de que se aplicara el aspecto es la siguiente:

```
----------------------------------------------------
Add game
Game title: Dungeon Defenders: Awakened
----------------------------------------------------
Add game
Game title: Dungeon Defenders: Awakened
Game title: Roki
```

3.1.2.5. this

La primitiva this permite encontrar los puntos de unión pertenecientes a un objeto que contiene un método y las asociaciones que son necesarias para la carga y ejecución de la clase. En el código 3.9 se declara un `ArrayList<>`, al cual se le agregan datos enteros que se generaron de forma aleatoria en el método `generateSerie()`, línea 13; al final de la asignación de los números se muestran los datos desde `showSeries()`.

Código 3.9: Random en Java.

```
1 package thisExample;
2
3 import java.util.ArrayList;
4 import java.util.Random;
5
6 public class Randomnumbers {
7   private ArrayList<Integer> serie;
8
9   public Randomnumbers () {
10          serie = new ArrayList<Integer> ();
11      }
12
13      public void generateSerie () {
14
15          Random numAleatorio = new Random();
16          for (int i=0; i < 10; i++) {
17          serie.add(numAleatorio.nextInt(1000) );
18           }
19          showSeries();
20      }
21
22      public void showSeries() {
23        System.out.println("Impresion");
24          for (Integer tmpObjeto :  serie) {
25              System.out.println (tmpObjeto);    }
26      }
```

```
27
28     public static void main (String [] args){
29       Randomnumbers r = new Randomnumbers();
30       r.generateSerie();
31     }
32 }
```

La salida en pantalla variará según la invocación, un ejemplo es:

```
977
314
710
```

La primitiva `this` se aplica sobre el objeto `RandomNumbers`, esto se define en el punto de unión, línea 4. El aviso `before`, imprime un mensaje al cual se le concatena la variable `thisJoinPoint` en el que se encuentra.

Código 3.10: Random en AspectJ.

```
1 package thisExample;
2
3 public aspect RandomAspect {
4   pointcut general(): this(thisExample.Randomnumbers);
5
6   before(): general() {
7     System.out.println("JoinPoint:␣" + thisJoinPoint);
8   }
9 }
```

La impresión comienza con los puntos de unión que son necesarios para la carga de clase, lo cual incluye los `import` a las bibliotecas que utilizará el programa. De igual manera,

cuando se utiliza la biblioteca `Java.IO` para la impresión de los números que se generaron, se muestran los puntos de unión de carga para esto. Lo que se verá en pantalla será lo siguiente:

```
Puntos de unión en this  : initialization(thisEjemplo.RandomNumeros())
Puntos de unión en this  : execution(thisEjemplo.RandomNumeros())
Puntos de unión en this  : call(Java.util.ArrayList())
Puntos de unión en this  : set(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : execution(void thisEjemplo.RandomNumeros.
                           generarSerieDeAleatorios())
Puntos de unión en this  : call(Java.util.Random())
Puntos de unión en this  : get(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : call(int Java.util.Random.nextInt(int))
Puntos de unión en this  : call(Integer Java.lang.Integer.valueOf(int))
Puntos de unión en this  : call(boolean Java.util.ArrayList.add(Object))
Puntos de unión en this  : get(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : call(int Java.util.Random.nextInt(int))
Puntos de unión en this  : call(Integer Java.lang.Integer.valueOf(int))
Puntos de unión en this  : call(boolean Java.util.ArrayList.add(Object))
Puntos de unión en this  : get(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : call(int Java.util.Random.nextInt(int))
Puntos de unión en this  : call(Integer Java.lang.Integer.valueOf(int))
Puntos de unión en this  : call(boolean Java.util.ArrayList.add(Object))
Puntos de unión en this  : get(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : call(int Java.util.Random.nextInt(int))
Puntos de unión en this  : call(Integer Java.lang.Integer.valueOf(int))
Puntos de unión en this  : call(boolean Java.util.ArrayList.add(Object))
```

```
Puntos de unión en this  : get(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : call(int Java.util.Random.nextInt(int))
Puntos de unión en this  : call(Integer Java.lang.Integer.valueOf(int))
Puntos de unión en this  : call(boolean Java.util.ArrayList.add(Object))
Puntos de unión en this  : call(void thisEjemplo.RandomNumeros.mostrarSerie())
Puntos de unión en this  : execution(void thisEjemplo.RandomNumeros.mostrarSerie())
Puntos de unión en this  : get(ArrayList thisEjemplo.RandomNumeros.serieAleatoria)
Puntos de unión en this  : call(Iterator Java.util.ArrayList.iterator())
Puntos de unión en this  : call(boolean Java.util.Iterator.hasNext())
Puntos de unión en this  : call(Object Java.util.Iterator.next())
Puntos de unión en this  : get(PrintStream Java.lang.System.out)
Puntos de unión en this  : call(void Java.io.PrintStream.println(Object))
186
Puntos de unión en this  : call(boolean Java.util.Iterator.hasNext())
Puntos de unión en this  : call(Object Java.util.Iterator.next())
Puntos de unión en this  : get(PrintStream Java.lang.System.out)
Puntos de unión en this  : call(void Java.io.PrintStream.println(Object))
267
Puntos de unión en this  : call(boolean Java.util.Iterator.hasNext())
Puntos de unión en this  : call(Object Java.util.Iterator.next())
Puntos de unión en this  : get(PrintStream Java.lang.System.out)
Puntos de unión en this  : call(void Java.io.PrintStream.println(Object))
Puntos de unión en this  : call(boolean Java.util.Iterator.hasNext())
```

3.1.2.6. get

La primitiva get permite intervenir al momento en el que se realiza la lectura del valor de un campo que se define en una clase. Para el ejercicio se solicita que al realizar la

venta de un electrodoméstico, se tome el precio base y se agregue un valor para mostrar el precio final. La clase `Product` se compone de los atributos `priceBase` y `weigth`, con sus respectivos métodos `get`; en el método `main` se desea imprimir el valor final de la venta, código 3.11.

Código 3.11: Product en Java.

```
1  package getExample;
2
3  public class Product {
4    private double priceBase=100.00;
5    private int weight = 5;
6
7    public double getprice(){
8      return priceBase;
9    }
10
11   public int getWeight(){
12     return weight;
13   }
14
15   public static void main (String [] args){
16     Product prod = new Product();
17     System.out.println("The final price is: " + prod.getprice
         () );
18   }
19 }
```

Por lo que es posible apreciar en pantalla:

```
The final price is: 100.0
```

Para agregar el valor extra necesario que permita obtener el precio final, se define un *pointcut* que interviene el comportamiento en el momento en que se consulte el valor de `priceBase` (código 3.12). En la línea 7, se define un aviso `around` el cual se especifica que el tipo de retorno será `int`, ya que este aviso devolverá el valor del `priceBase` + 150, para mostrar en pantalla el precio final.

Código 3.12: Product en AspectJ.

```
1  package getExample;
2
3  public privileged aspect ProdAspect {
4
5    pointcut cut(): get(double getExample.Product.priceBase);
6
7    int around():cut(){
8      System.out.println(proceed());
9      return 100+150;
10   }
11 }
```

Una vez que se aplica el aspecto, la impresión cambia por la siguiente:

```
100
The final price is: 250.0
```

3.1.2.7. set

Permite identificar el momento en que se asigna valor a un campo. Cuenta con soporte para los tres tipos de avisos. Se propone que cada vez que se asigne la edad de una persona,

se verifique si es menor o mayor de edad. El código 3.13 define la clase `Person` la cual solo cuenta con un atributo `age` y su correspondiente método `set`. En `main` se crea la instancia de `Person` y se asigna la edad de 21.

Código 3.13: Person en Java.

```
1  package setExample;
2  
3  public class Person {
4    int age;
5  
6    public void setAge(int age){
7      this.age=age;
8    }
9  
10   public static void main (String [] args){
11     Person p = new Person();
12     p.setAge(21);
13   }
14 }
```

Este código no genera ninguna salida. Para verificar la edad, se define el `pointcut` en el código 3.14, en el cual se realiza la exposición de contexto en conjunto con el uso de la primitiva `set` para que la acción se realice cada vez que se asigne un valor a edad. Por el uso de `args`, es posible manipular el valor que se asigna, y con ello es posible verificar si la persona es menor o mayor de edad.

Código 3.14: Person en AspectJ.

```
1  package setExample;
2  
```

```
3  public aspect PersonAspect {
4
5    pointcut write(int a):set(int setExample.Person.age)&& args
       (a);
6
7    after(int age):write(age){
8      System.out.println("The␣age␣is:␣"+ age);
9      if(age>17)
10       System.out.println("Older");
11     if(age<18)
12       System.out.println("Younger");
13   }
14
15 }
```

La salida luego de la intervención del aspecto es:

```
The age is: 21
Older
```

3.1.2.8. within

La primitiva *within* interviene durante la ejecución en todos los puntos de unión, desde el proceso de construcción, de un mismo objeto; igualmente si se combina con alguna otra primitiva, permite que el corte se aplique únicamente cuando el punto se encuentre en la clase que se especificó, sin tomar en cuenta si en el aspecto también se encuentra algo que haga referencia al patrón de firma. Para mostrar la capacidad de esta primitiva, el ejercicio que se presenta en el código 3.15 declara la clase `Events`, la cual tiene un constructor que

no recibe argumentos, línea 5 y un bloque estático que se encuentra en la línea 9.

Código 3.15: Events en Java.

```
package withinExample;

public class Events {

  public Events(){
    System.out.println("Events constructor");
  }

  static{
    System.out.println("Static");
  }

  public static void main (String [] args){
    System.out.println("Primitiva Within");
    Events e = new Events();
  }
}
```

El resultado de la ejecución es:

```
Static
Primitiva Within
Events constructor
```

En el código 3.16 se utiliza la primitiva within para encontrar los puntos de unión que se contienen en el objeto *Eventos*, como se indica en el `pointcut` de la línea 4.

Código 3.16: Events en AspectJ.

```
package withinExample;

public aspect EventsAspect {
  pointcut w():within(withinExample.Events);

  before():w(){
    System.out.println(thisJoinPoint);
  }
}
```

El resultado en pantalla muestra los puntos de unión de cada evento que se llevó a cabo durante la ejecución:

```
staticinitialization(withinEjemplo.Eventos.<clinit>)
get(PrintStream Java.lang.System.out)
call(void Java.io.PrintStream.println(String))
Static
execution(void withinEjemplo.Eventos.main(String[]))
get(PrintStream Java.lang.System.out)
call(void Java.io.PrintStream.println(String))
Primitiva Within
call(withinEjemplo.Eventos())
preinitialization(withinEjemplo.Eventos())
initialization(withinEjemplo.Eventos())
```

```
execution(withinEjemplo.Eventos())
get(PrintStream Java.lang.System.out)
call(void Java.io.PrintStream.println(String))
Events constructor
```

3.1.2.9. cflow

La primitiva cflow provee un mecanismo que identifica un método, que cumple con un patrón de firma, y el flujo de control que se derive de este. Un flujo de control es el orden en el que se ejecutan las llamadas entre métodos, de modo que si se ejecuta el método `a` y se invoca al método `b` y a su vez a `c`, se deberán cumplir todas las instrucciones del último método que se invocó, para regresar el control al segundo y luego al primero.

En el caso de prueba se solicita que el usuario tenga un nombre y una contraseña, durante tres pasos diferentes es necesario verificar que la contraseña y el usuario sea correcto para que tenga acceso autorizado. En el código 3.17 la clase `User` tiene los atributos `user` y `password` cada uno con su respectivo método `get`. En la línea 20 se define el método `firstStep()`, el cual en la línea 23 invoca a `secondStep()` y este a su vez en la línea 28 llama al método `thirdStep()`. En `main` se realiza la instancia de `User` y se invoca a `firstStep()`.

Código 3.17: User en Java.

```
1 package cflowExample;
2
3 public class User {
4   private String user;
5   private String password;
6
```

```
7    public User(String user, String password){
8      this.user = user;
9      this.password = password;
10   }
11
12   public String getUser(){
13     return user;
14   }
15
16   public String getPassword(){
17     return password;
18   }
19
20   void firstStep(){
21     this.user="Luisa";
22     System.out.println("First step");
23     secondStep();
24   }
25
26   void secondStep(){
27     this.user="Luisa";
28     System.out.println("Second step");
29     thirdStep();
30   }
31
32   void thirdStep(){
33     this.user="Luisa";
```

```
34      System.out.println("Third␣step");
35    }
36
37    public static void main (String [] args){
38      Usuario usuario = new Usuario("Luisa","abc");
39      usuario.firstStep();
40    }
41  }
```

Por lo que en pantalla se verá lo siguiente:

```
First step
Second step
Third step
```

Sin embargo, no se verifica si los datos son iguales en cada entrada para que se autorice el acceso. Por esta razón en el código 3.18 se especifica en la línea 5 que el punto de unión es sobre el flujo de control que se deriva de la llamada del método `firstStep()`. En la línea 13 se realiza la exposición de contexto del objeto `Usuario`, esto para acceder a los valores que se enviaron de usuario y contraseña. En el aviso `before` de la línea 17 se revisa si los datos que se proporcionaron son iguales o cambian.

Código 3.18: User en AspectJ.

```
1 package cflowExample;
2
3 public aspect UserAspect {
4
5   pointcut control(): set(String cflowExample.User.user)
```

```
6                 && cflow(call(void firstStep()))
7                 && !within(UserAspect);
8
9     User user;
10    String us="Luisa";
11    String pass="abc";
12
13    before(User u): execution(cflowExample.User.new(String,
          String)) && target(u){
14      user = u;
15    }
16
17    before():control(){
18      if(user.getUser().compareTo(us)== 0 && user.getPassword()
            .compareTo(pass)==0){
19        System.out.println("Authorized␣access");
20      }else{
21        System.out.println("Access␣deny");
22      }
23    }
24  }
```

Una vez que el aspecto se aplica la ejecución cambia y ahora la impresión en pantalla es:

```
Authorized access
First step
Authorized access
```

```
Second step
Authorized access
Third step
```

3.1.2.10. cflowbelow

La primitiva `cflowbelow` permite identificar e intervenir el comportamiento que se deriva de un flujo de control, sin tomar en cuenta al método que comienza el flujo. Por lo general esto es más fácil de observar en el caso de la recursividad. Para ejemplificar el uso de esta primitiva, se toma como ejercicio la serie de Fibonacci. En el código 3.19, en la línea 10 se especifica el método recursivo que permite obtener el valor del `Fibonacci()`. En el método `main` se invoca a `Fibonacci` con el valor de 6.

Código 3.19: Fibonacci en Java.

```
1  package cflowbelowExample;
2
3  public class Fibonacci {
4     public static void main(String[] args) {
5        Fibonacci fib = new Fibonacci();
6           int result = fib.fibonacci(6);
7           System.out.println("Result is: "+result);
8        }
9
10       public  int fibonacci(int n) {
11          if (n == 0) {
12             return 0;
13          } else if (n == 1) {
```

```
14                  return 1;
15              } else {
16                  return fibonacci(n - 1) + fibonacci(n - 2);
17              }
18          }
19  }
```

Al ejecutar se mostrara en pantalla lo siguiente:

```
Result is: 8
```

Para observar el trabajo que realiza `cflowbelow` se definen dos puntos de unión, el primero, línea 5, indica que el `corte` se realizará cada vez que se llame al método `fibonacci()` y la exposición de contexto del valor que se recibe en la llamada. El segundo indica que la intervención de código será sobre el corte anteriormente descrito completando la sentencia con la primitiva `cflowbelow` de la ejecución del `pointcut invocation`. En el aviso `before` se imprime el valor que recibe `fibonacci` en cada llamada.

Código 3.20: Fibonacci con AspectJ.

```
1  package cflowbelowExample;
2
3  public aspect cflowbelowAspect {
4
5    pointcut invocation(int x):
6      call(int cflowbelowExample.Fibonacci.fibonacci(int)) &&
7      args(x);
8
9    pointcut principal(int x):
10     invocation(x) && cflowbelow(invocation(int));
```

```
11
12    before(int x): principal(x) {
13      System.out.println("Invocation␣value␣is␣:␣"+x);
14    }
15  }
```

La salida resultante es la siguiente, esta no incluye el valor de la invocación original, ya que la primitiva no toma en cuenta esta.

```
Invocation value is: 5
Invocation value is: 4
Invocation value is: 3
Invocation value is: 2
Invocation value is: 1
Invocation value is: 0
Invocation value is: 1
Invocation value is: 2
Invocation value is: 1
Invocation value is: 0
Invocation value is: 3
Invocation value is: 2
Invocation value is: 1
Invocation value is: 0
Invocation value is: 1
Invocation value is: 4
Invocation value is: 3
Invocation value is: 2
Invocation value is: 1
```

```
Invocation value is: 0
Invocation value is: 1
Invocation value is: 2
Invocation value is: 1
Invocation value is: 0
Result is: 8
```

3.1.2.11. withincode

Esta primitiva permite intervenir los eventos que se derivan de la implementación de métodos y constructores, esta primitiva no cuenta con soporte para aviso `around`. En el ejemplo se realiza la simulación de un cajero en el cual es posible revisar el saldo y retirar dinero. En el código 3.21, la clase `Account` tiene los atributos `numAccount` y `balance` ambos con sus respectivos métodos `get`. El constructor de la clase recibe los dos valores para inicializarlos de clase, línea 14. El método `showBalance()` imprime el número de cuenta y el saldo disponible, línea 19; mientras que en la línea 24, `remove()`, actualiza el saldo de la cuenta e imprime el monto de retiro, y el saldo actual.

Código 3.21: Account en Java.

```
1 package withincodeExample;
2
3 public class Account {
4   private String numAccount;
5   private double balance;
6
7   public void setbalance(double balance){
8     this.balance = balance;
```

```
9     }
10
11    public double getBalance(){
12      return balance;
13    }
14    public Account(String num, double sal){
15      this.numAccount=num;
16      this.balance=sal;
17    }
18
19    public void showBalance(){
20      System.out.println("Account␣number:␣" + this.numAccount);
21      System.out.println("Balance:␣" + this.balance);
22    }
23
24    void remove(double quantity){
25      this.setbalance(this.getBalance()-quantity);
26      System.out.println("Quantity:␣" + quantity);
27      System.out.println("Balance:␣" + this.balance);
28    }
29
30    public static void main(String args []){
31      Account Account = new Account("1234-9274", 1250.00);
32      Account.showBalance();
33      Account.remove(350.00);
34    }
35  }
```

La salida del código por lo valores que se definieron para los métodos, será la siguiente:

```
Account number: 1234-9274
Balance: 1250.0
Quantity: 350.0
Balance: 900.0
```

Al aplicar la primitiva `withincode` se intervendrá la ejecución del método `showBalance()` que muestra los puntos de unión necesarios para la carga de las asociaciones necesarias (código 3.22). En la línea 8 el aviso especifica que se imprimirá la variable `thisJoinPoint` antes del corte.

Código 3.22: Account con AspectJ.

```
1  package withincodeExample;
2
3  public aspect AccountAspect {
4
5      pointcut jpm():
6        withincode(* withincodeExample.Account.showBalance());
7
8      before(): jpm() {
9        System.out.println(thisJoinPoint);
10     }
11   }
```

La impresión que aparece en pantalla luego de la intervención del comportamiento muestra la llamada y la obtención de las bibliotecas necesarias para imprimir antes de cada valor que se indique en el método `showBalance()`:

```
get(PrintStream Java.lang.System.out)
call(Java.lang.StringBuilder(String))
get(String withincodeEjemplo.Cuenta.numCuenta)
call(StringBuilder Java.lang.StringBuilder.append(String))
call(String Java.lang.StringBuilder.toString())
call(void Java.io.PrintStream.println(String))
Account number: 1234-9274
get(PrintStream Java.lang.System.out)
call(Java.lang.StringBuilder(String))
get(double withincodeEjemplo.Cuenta.saldo)
call(StringBuilder Java.lang.StringBuilder.append(double))
call(String Java.lang.StringBuilder.toString())
call(void Java.io.PrintStream.println(String))
Balance: 1250.0
Quantity: 350.0
Balance: 900.0
```

3.1.2.12. preinitialization

La primitiva `preinitialization` permite exponer los puntos de unión que conciernen a los super constructores, es decir, cuando hay herencia. En el caso de no haber herencia explícita de todos modos es posible aplicar, ya que es necesario alcanzar la clase `Object`. En el código 3.23 se presentan 3 clases, la primera en la línea 6 de nombre `Sport`, tiene un bloque de inicialización y un constructor que recibe un número e imprime el valor. En la línea 15 se declara la segunda clase: `Tennis` que hereda de `Sport`, esta tiene un constructor que recibe un valor numérico el cual pasa al super constructor, y en la línea 21 se declara un bloque estático que imprime un mensaje. La última clase `Game` hereda de `Tennis`, línea 26, esta también cuenta con un constructor, este recibe una valor de cadena y un entero.

En `main` se instancia `Game`.

Código 3.23: Sports en Java.

```
package preinializationExample;

class Sport{
  int num;

  public Sport(int num){
    System.out.println("The sport's code: "+num);
    this.num = num;
  }

  {
    System.out.println("Sport initialization");
  }
}
public class Tennis extends Sport{

  public Tennis(int code){
    super(code);
  }

  static{
    System.out.println("Tennis static");
  }
```

```
24 }
25
26 class Game extends Tennis{
27
28   public Game(String date, int num){
29     super(num);
30     System.out.println("The␣date␣of␣game:␣"+date);
31   }
32
33   public static void main(String [] args){
34     new Game("10-02-2019", 2);
35   }
36 }
```

La salida en pantalla que se muestra es la siguiente:

```
Tennis static
Sport initialization
Sport code: 2
The date of game: 10-02-2019
```

El código 3.24 aplica la primitiva `preinitialization` en el `pointcut` de la línea 4, el cual especifica que será sobre el constructor de `Game`. Con el aviso `before`, de la línea 7, se imprime la variable `thisJoinPoint`, al igual que en el aviso `after` de la línea 10.

Código 3.24: Sports con AspectJ.

```
1 package preinializationExample;
2
```

```
3 public aspect TennisAspect {
4   pointcut preini():
5     preinitialization(preinializationExample.Game.new(String
        ,int));
6
7   before(): preini() {
8     System.out.println("before:␣" + thisJoinPoint);
9   }
10  after(): preini() {
11    System.out.println("after:␣" + thisJoinPoint);
12  }
13 }
```

La impresión resultante es la siguiente:

```
Tennis static
before: preinitialization(preinializationEjemplo.Partido(String, int))
after: preinitialization(preinializationEjemplo.Partido(String, int))
Sport initialization
Sport code: 2
The date of game: 10-02-2019
```

3.1.2.13. initialization

El proceso de construcción de un objeto en Java no solo implica la ejecución de constructores, el proceso incluye otros elementos: bloques de inicialización de instancia y la asignación de valores de instancia. La primitiva `initialization` interviene los tres elementos que se mencionaron anteriormente; cuenta con soporte para los avisos `before` y `after`, mientras que no es posible implementar `around`.

La clase `Toy` del código 3.25, tiene únicamente un constructor que recibe un número entero. La clase de la línea 10 `Toys`, tiene un bloque de inicialización en el cual se asignan valores a los atributos de tipo `Toy`. Dentro de `main` solamente se crea una instancia anónima de `Toys`.

Código 3.25: Toy en Java.

```
1  package initializationExample;
2
3  class Toy{
4
5    Toy(int code){
6      System.out.println("Toy code: "+code);
7    }
8  }
9
10 public class Toys {
11   Toy j1;
12   Toy j2;
13
14   {
15     j1 = new Toy(1);
16     j2 = new Toy(2);
17     System.out.println("It creates toys");
18   }
19
20   public static void main(String [] args){
```

```
21      new Toys();
22    }
23  }
```

La salida en pantalla luego de la ejecución es la siguiente:

```
Toy code: 1
Toy code: 2
It creates toys
```

En el aspecto del código 3.26 se define el `pointcut` en la línea 5, en este se aplica la primitiva `initialization` sobre el constructor sin argumentos de la clase `Toys`. Con los avisos `before` y `after` se imprime la variable `thisJoinPoint` de antes y después de que el corte se aplique.

Código 3.26: Toy con AspectJ.

```
1  package initializationExample;
2
3  public aspect ToyAspect {
4
5    pointcut objectini():
6      initialization(initializationExample.Toys.new());
7
8    before(): objectini(){
9      System.out.println("Start" + thisJoinPoint);
10   }
11
12   after(): objectini(){
```

```
13      System.out.println("End" + thisJoinPoint);
14    }
15  }
```

La salida en pantalla de esta intervención será la siguiente:

```
Start(initializationEjemplo.Toys())
Toy code: 1
Toy code: 2
It creates toys
End(initializationEjemplo.Toys())
```

3.1.2.14. staticiniatilization

Un bloque estático es un fragmento de código que permite inicializar ciertas condiciones necesarias para el trabajo que realizará la clase. La primitiva `staticiniatilization` identifica el punto de unión en los bloques estáticos pertenecientes a una clase; cuenta con soporte para los tres avisos `before, around` y `after`. El ejercicio que se presenta en el código 3.27 muestra la clase `Operations()` la cual tiene un atributo estático, que se inicializa dentro del bloque estático que se encuentra en la línea 8, mediante un llamado al método estático `started()`.

Código 3.27: Operations en Java.

```
1  package staticinializationExample;
2  import java.util.Random;
3
4  public class Operations {
5    static int number;
6
```

```
7        static{
8             number = Operations.started(1000);
9        }
10
11       public static int started(int number) {
12           if (number > 100 && number < 0){
13                System.out.println("Your␣number␣must␣be␣between␣0
                    ␣and␣100");
14           }
15           return new Random().nextInt(number);
16       }
17
18       public static void main(String[] args) {
19                System.out.println("Value␣" + Operations.number)
                     ;
20       }
21 }
```

La salida depende del número aleatorio, en una de las posibles ejecuciones el resultado es el siguiente:

```
Your number must be between 0 y 100
Value 570
```

Para intervenir el comportamiento del programa, en el código 3.28 se aplica la primitiva `staticinitialization` sobre el objeto `Operations`, línea 4. El aviso `around`, que se encuentra en la línea 7, imprime la variable `thisJoinPoint` además de crear una nueva instancia de `Operations` y se altera el valor de número.

Código 3.28: Operations con AspectJ.

```
1 package staticinializationExample;
2
3 public aspect OperationsAspect {
4   pointcut clinit():
5     staticinitialization(staticinializationExample.Operations
        );
6
7   void around(): clinit() {
8     System.out.println("around:␣" + thisJoinPoint);
9     new Operations().number=Operations.started(150);
10  }
11 }
```

Luego de aplicar el aspecto, la salida tendrá como variación que el mensaje de la condición del método `inicializar`, no aparecerá. El valor de número depende de la generación del número aleatorio. Una posible salida es:

```
around: staticinitialization(staticinializationExample.Operations.<clinit>)
Value 62
```

3.1.2.15. if

La primitiva `if` ayuda a que el aviso solo se ejecute sobre el corte que se especificó, si se cumple una condición. Esta primitiva debe combinarse con alguna otra para su uso. En el ejemplo se espera recibir un valor de `humidity` para hacer una predicción de si hará calor o lloverá. En el código 3.29 la clase `Weather` tiene como atributos `temperature` y `humidity`, este último es estático. En el método `main` se asigna el valor de la `temperature`

y posteriormente se muestra en pantalla el valor de la `humidity`.

Código 3.29: Weather en Java.

```
1  package ifExample;
2
3  public class Weather {
4    int temperature;
5    static int humidity=25;
6
7    public void setTemperature(int temp){
8      temperature = temp;
9    }
10
11   public static void main(String [] args){
12     Weather weather = new Weather();
13     weather.setTemperature(30);
14     System.out.println("Humidity␣in␣the␣environment:␣" +
         humidity);
15   }
16 }
```

La salida en pantalla será:

```
Humidity in the enviroment: 25
```

Para mostrar la predicción de si lloverá o hará calor, se aplica el aspecto del código 3.30; en la línea 5 se define el `pointcut` en la ejecución del método `main`. En el aviso `before` de la línea 8 se complementa la instrucción con la condición `if`, en la cual se indica que si el

valor de la `humidity` es mayor de 20, entonces se mostrará el mensaje `Rain`, mientras que en el aviso `after` la primitiva `if` indica que si la `humidity` es menor de 20, se imprime el mensaje `Hot day`.

Código 3.30: Weather con AspectJ.

```
package ifExample;

public privileged aspect WeatherAspect {
  pointcut condition():
    execution(* Weather.main(..));

  before():condition() && if(Weather.humidity>20){
    System.out.println("Rain");
  }

  after():condition() && if(Weather.humidity<20){
    System.out.println("Hot day");
  }
}
```

En pantalla se mostrará:

```
Humidity in the enviroment: 10
Hot day
```

3.1.2.16. adviceexecution

Esta primitiva permite que el corte se aplique únicamente cuando el patrón de firma se encuentre en el aspecto, de forma que ignora si el patrón se cumple en algún punto

del objeto; sin embargo, esta primitiva no se aplica por si sola. En el ejercicio del código 3.31 se define la clase programa que tiene un atributo estático `name` cuyo valor es `"Planet in danger"`; el método `show` de la línea 7 imprime un mensaje junto con el nombre del programa.

Código 3.31: Program en Java.

```
1  package adviceExample;
2
3  public class Program {
4
5    static String name="Planet␣in␣danger";
6
7    void show(){
8      System.out.println("The␣program␣ends:␣␣"+nombre);
9    }
10
11   public static void main(String args []){
12     Program p = new Program();
13     p.show();
14   }
15 }
```

El resultado de la ejecución de este programa es el siguiente:

```
The program ends: Planet in danger
```

Para ejemplificar el uso de esta primitiva se aplica en el código 3.32, se define el punto de unión, en la línea 4, con la primitiva `get` para el valor de `name` y se complementa

con la negación de `cflowbelow`; se aplica el aviso `before` en la línea 6, en este se imprime la variable `thisJoinPoint` y luego se imprime un mensaje con el nombre del programa.

Código 3.32: Program con AspectJ.

```
1 package adviceExample;
2
3 public aspect ProgramAspect {
4
5   pointcut asp(): get(String adviceExample.Program.name)&& !
      cflow(adviceexecution());;
6
7   before(): asp() {
8     System.out.println(thisJoinPoint);
9     System.out.println("The program starts: " + Program.name)
        ;
10  }
11 }
```

La impresión en pantalla muestra solo la variable `thisJoinPoint` de la consulta del dato que se realizó en el aspecto por la indicación de la primitiva `adviceexecution`.

```
get(String adviceExample.Program.name)
The program starts: Planet in danger
The program ends: Planet in danger
```

3.1.2.17. handler

La primitiva `handler` permite intervenir sobre el manejador de excepciones. En el código 3.33 se definen tres clase, la primera `MyException` que hereda de la clase `Exception`,

cuenta con un método que devuelve un mensaje. La segunda clase `AnException` también hereda de `Exception` y en la línea 14 define un método que devuelve un mensaje. En el método `main` en un sentencia `try-catch` se verifica una edad, en el caso de ser menor a lo que se establece, se arrojará la excepción.

Código 3.33: Exceptions en Java.

```
1  package handlerExample;
2
3  public class MyException extends Exception{
4    private static final long serialVersionUID = 1L;
5
6        public String getMenssage() {
7          return "Younger";
8       }
9  }
10
11 class AnException extends Exception{
12   private static final long serialVersionUID = 1L;
13
14   public String getMenssage(){
15    return "Is␣not␣correct";
16   }
17
18 }
19
20 class PropiaClaseExcepcion {
```

```
21      public static void main(String[] args) {
22        int age = 15;
23         try {
24               if(age<18)
25                 throw new MyException();
26               if(age>50)
27                 throw new AnException();
28       } catch(MyException e) {
29             System.out.println("Exception:␣" + e.getMenssage()
                  );
30             e.printStackTrace();
31       }catch(AnException c){
32          System.out.println("Exception:␣" + c.getMenssage());
33       }
34            }
35        }
```

La salida derivada de la ejecución es:

```
Exception: Younger
handlerEjemplo.MyException
at handlerEjemplo.PropiaClaseExcepcion.main(MExcepcion.Java:25)
```

La primitiva `handler` se aplica en el código 3.34 en el cual el punto de unión que se define en la línea 4 indica que se intervendrá cuando se ejecute alguna de las excepciones que se crearon. En la línea 7 el aviso `before` indica que se imprimirá la variable `thisJoinPoint`.

Código 3.34: Exceptions con AspectJ.

```
1 package handlerExample;
```

```
2
3 public aspect ExcepcionAspect {
4   pointcut catcher():
5     handler(handlerExample.MyException) || handler(
        handlerExample.AnException);
6
7   before(): catcher() {
8     System.out.println("before:␣" + thisJoinPoint);
9   }
10 }
```

La impresión resultante es:

```
before: handler(catch(MExcepcion))
excepción: Menor de edad
handlerEjemplo.MExcepcion
at handlerEjemplo.PropiaClaseExcepcion.main(MExcepcion.Java:25)
```

3.2. Revisión de las características que ofrece SN

SN es un lenguaje de programación naturalístico y de propósito general que se diseñó en el Tecnológico Nacional de México, Campus Orizaba. Este lenguaje se basa en el idioma inglés, permite trabajar con referencias indirectas de acuerdo a la posición en la que se establece una instancia, además de que permite realizar composición de un sustantivo y varios adjetivos para complementar una entidad y darle un comportamiento más especializado. SN soporta ciclos y condicionales a partir de la posición relativa de las instrucciones (ya sea antes o después), además permite trabajar con circunstancias que asocian a una oración con otra de forma indirecta para cubrir un comportamiento no funcional, es decir:

definir el contexto de una oración. La sintaxis del lenguaje permite trabajar con diversas abstracciones que lo asemejan más a un lenguaje natural.

3.2.1. Abstracciones de SN

SN cuenta con un conjunto de abstracciones, que permiten que su sintaxis se asemeje a los lenguajes naturales, que se describen a continuación.

3.2.1.1. Noun (Sustantivo)

Los sustantivos son la representación concreta y modular de alguna abstracción del mundo real, cuentan con atributos y realizan acciones que se describen en forma de verbos. En SN se emplea la palabra en inglés `noun` para definir un sustantivo, su sintaxis es la siguiente:

```
noun Nombre de la abstracción:
```

3.2.1.2. Adjective (Adjetivo)

Se define como la parte de una oración que complementa al sustantivo, describe una característica que se comparte por abstracciones del mundo real. Se utiliza la palabra en inglés `adjective` la sintaxis para definir un adjetivo es la siguiente:

```
adjective Nombre del adjetivo:
```

3.2.1.3. Verb (Verbo)

Un verbo es una palabra que indica una acción que complementa al sustantivo y de este modo se conforma una oración. En SN, se utiliza la palabra `verb` para describir conjuntos de instrucciones de manera específica, los verbos en este lenguaje tienen un nombre y varios tipos de sintaxis con base en la posición del sustantivo que contiene a dicho verbo.

```
itself nombre del verbo:
```

Es posible agregar argumentos que permitan realizar la correcta ejecución del verbo, para lo cual la sintaxis sería:

```
verb itself nombre del verbo nombre del argumento tipo del argumento:
```

3.2.1.4. Attribute (Atributo)

Un atributo es una propiedad de un sustantivo, en SN, se utiliza la palabra reservada `attribute` para definirlos. La sintaxis para definir atributos es la siguiente:

```
attribute Nombre.
```

3.2.1.5. Main (Principal)

`main` es una abstracción que funciona como punto de inicio del programa, la sintaxis para declarar la abstracción main, es la siguiente:

```
main Nombre del Main:
```

3.2.1.6. Gramáticas embebidas

Dado que la naturalidad de SN dificulta trabajar con vocabulario de un dominio particular, SN permite trabajar con gramáticas de dominio específico embebidas, este mecanismo es extensible, lo que significa que el programador es capaz de cargar su propia gramática.

```
(grammars.embedded.math.Math) {:
  area = (b * h)/2
:}
```

En este ejemplo se presenta el mecanismo para integrar una gramática embebida. En el cual los operadores {: y :} indican el espacio donde se ejecutará la gramática embebida, por tanto se recomienda no definir gramáticas con dichos operadores.

```
(xmlhandler) {:
   <biblioteca>
   <libro titulo=?La divina comedia? isbn=?12839? paginas=314/>
   <libro titulo=?La iliada? isbn=?8293? paginas=120/>
   </ biblioteca >
:}
```

Donde la gramática embebida permitiría procesar código XML para traducirlo a abstracciones de SN, con base en un contexto particular.

3.2.1.7. Referencias indirectas

En SN, las referencias indirectas son aquellas con las que se llama a instancias según la posición en la que se crearon. Por ejemplo: se referencia al primer número, a la tercera cadena o a la última instrucción que se ejecutó; además, se utilizan las palabras reservadas `it` y `these` para hacer referencia al último singular y plural que se utilizó, respectivamente.

```
System prints it and newline.
```

3.2.1.8. Atributos derivados

Un atributo derivado no existe por sí mismo, sino que depende de la existencia de otros atributos. Los atributos derivados se comportan como un verbo, pero se accede a ellos como si fueran atributos. Su sintaxis consiste en la palabra `derived` que se antepone a la palabra reservada `attribute`, pero además requiere definir el tipo y un conjunto de instrucciones.

`derived attribute result as a real number.`

3.2.2. Contexto de ejecución

En SN el contexto de ejecución permite especificar el momento en el que se ejecutará un comportamiento independiente al que realiza el verbo que se requería. Los contextos de ejecución son cuatro:

- **Before:** Permite agregar la ejecución de un verbo complementario antes de que se realice la ejecución del verbo que se efectuó.
- **Instead:** Realiza la ejecución de un verbo en lugar del verbo que se invocó originalmente.
- **After:** Complementa la ejecución del verbo original con la ejecución posterior de un verbo.
- **When:** Según la forma en la que se plantee funciona como *before* o *after*, de forma que es posible especificar una condición que indique el momento en el que este intervendrá.

En la sección 3.2.3, se ejemplificará el uso de estos contextos de ejecución.

3.2.3. 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. Las circunstancias se definen en el mismo sustantivo que las requiere, además de que el comportamiento que se agrega o se cambia debe encontrarse en un verbo. SN permite trabajar con diferentes tipos de circunstancias, las cuales son:

3.2.3.1. Contexto de ejecución de verbos

En SN todas las circunstancias se realizan antes, después o en un lugar de la ejecución de un verbo o una asignación. La circunstancia que define la ejecución se encuentra implícita, por esto que no es necesario utilizar una palabra reservada que indique esta acción. Un ejemplo de la sintaxis necesaria para esto es:

```
circumstance: exit it after show something.
```

Esta indica que se ejecutará el verbo `exit` luego de que se ejecute el verbo `show`.

3.2.3.2. Exposición de contexto

Las circunstancias de SN proponen la exposición de contexto para argumentos y sustantivos de forma implícita, no existe una palabra reservada. En el código 3.35 se presenta un ejemplo de exposición de contexto en el caso de argumentos, mediante un problema que permite imprimir un valor y su tabla de multiplicar. El ejercicio se resuelve con el uso del sustantivo `Operation`, en la línea 2 se especifica el verbo `print_table` cuya función es imprimir un mensaje que indique el valor de la tabla de multiplicar. En la línea 6, el verbo `multiplication`, ejecuta de forma repetitiva la multiplicación e impresión para que sea posible ver la tabla de multiplicar completa. La circunstancia se declara en la línea 15, esta indica que se ejecutará el verbo `print_table` antes del verbo `multiplication`. Como se observa, no es necesario el uso de alguna palabra reservada para hacer la manipulación del valor que requieren ambos verbos.

Código 3.35: Código Operations en SN

```
1 noun Operation:
2   verb itself print_table table as Number:
```

```
3     System prints "The␣table␣of␣multiplication␣is:␣" and
        table.
4     System prints newline.
5
6   verb itself multiplication table as Number:
7     counter is 1.
8     repeat the next 5 instructions while counter is lesser
        than 11.
9     System prints table and "x".
10    System prints counter and "=␣".
11    table * counter.
12    System prints it and newline.
13    add 1 to counter.
14
15  circumstance: it print_table number before something
      multiplication number.
16
17 main Argument:
18   an Operation.
19   the Operation multiplication 5.
```

La impresión en pantalla resultante del programa es la siguiente:

```
The table of multiplication is: 5


5x1= 5
5x2= 10
5x3= 15
```

```
5x4= 20

5x5= 25

5x6= 30

5x7= 35

5x8= 40

5x9= 45

5x10= 50
```

3.2.3.3. Created

`Created` identifica el momento en el que se crea la referencia a un sustantivo. Para ejemplificar el caso de este ejemplo se presenta como ejercicio la inscripción de un alumno en diversos cursos, por lo cual se pretende que cada vez que "crea" el alumno, se inscriba en el curso. En el código 3.36 se define un sustantivo `Student` el cual cuenta con el atributo `number_control` y el atributo `course`. En la línea 8, se declara el atributo derivado que permite agregar estudiantes al plural; la circunstancia que se indica en la línea 13, indica que se ejecutará el verbo `register` cuando se cree el `Student`. En la abstracción `main` se imprime el nombre de tres cursos para que el estudiante escoja uno para inscribirse; en la línea 29 se indica que se deberán repetir las siguientes nueve instrucciones mientras el valor de la opción sea diferente de 0. En la línea 34 se crea al estudiante, por lo que será la instrucción sobre la que se ejecute la circunstancia.

Código 3.36: Código Created en SN

```
1 noun Student with plural as Students:
2   attribute number_control as a Number.
3   attribute course as a Number.
4
```

```
5    verb itself register:
6      System prints "Your␣register␣is␣complete" and newline.
7
8    overrided derived attribute string as a String:
9      a String with "" as value.
10     add "␣number␣control:␣" to it; and add number_control to
          it.
11     add "␣course:␣" to it; add course to it; and return the
          String.
12
13   circumstance: it register when this is created.
14
15 main Inscription:
16   option is -1.
17   other is 0.
18   number is 0.
19   a Student.
20   some Students.
21   System prints "1.-␣NodeJS␣nivel␣intermedio" and newline.
22   System prints "2.-␣Desarrollador␣profesional␣de␣Software"
        and newline.
23   System prints "3.-␣Bases␣de␣datos␣para␣desarrollos␣web" and
         newline.
24   System prints "0.-␣Salir." and newline.
25   System prints "Teclea␣tu␣numero␣de␣control" and newline.
26   System reads.
27   number is the integer of it.
```

```
28
29    repeat the next 9 instructions until option is equal to 0.
30    System prints "¿En␣que␣curso␣deseas␣registrarte?␣".
31    System reads.
32    other is the integer of it.
33    execute the next 5 instructions when other is distinct to
         0.
34    a Student with number as number_control and other as course
         .
35    add it to the Students.
36    System prints "¿Deseas␣inscribirte␣en␣otro␣curso?␣1.-␣Si␣
         0.-␣No" and newline.
37    System reads.
38    option is the integer of it.
39
40    System prints the string of the Students and newline.
```

La ejecución en pantalla se verá de la siguiente forma:

```
1.- NodeJS nivel intermedio
2.- Desarrollador profesional de Software
3.- Bases de datos para desarrollos web
0.- Salir.
Teclea tu numero de control
13011074

¿En que curso deseas registrarte? 1
 Your register is complete
```

```
¿Deseas inscribirte en otro curso? 1.- Si 0.- No
 0
 [ number control: 13011074 course: 1 ]
```

3.2.3.4. Assigned

Como su nombre indica, esta circunstancia interviene el comportamiento del programa cuando se asigna un valor a un atributo o variable. En el código 3.37 se presenta un ejercicio en el que se desea registrar a una persona y cada vez que esto se realice, se verifique que sea mayor de edad. En la línea 1 se define el sustantivo persona, este tiene como atributo la edad, para verificar si la persona es mayor o menor de edad, en la línea 3, el verbo `verify` recibe el valor de una edad, si es mayor de 17 se imprime como resultado que la persona es adulta, en el caso contrario que es joven. La circunstancia, línea 11, indica que se ejecutará el verbo `verify` luego de que se asigne el valor de la edad de la persona.

Código 3.37: Código Assigned en SN

```
1  noun Person:
2    attribute age as a Number.
3    verb itself verify age as Number:
4      System prints "The␣age␣is:␣" and age.
5      System prints newline.
6      execute the next instruction when age is greater than 17.
7      System prints "Adult" and newline.
8      execute the next instruction when age is lesser than 18.
9      System prints "Younger" and newline.
10
```

```
11    circumstance: it verify age after the age of something is
         assigned.
12
13  main Asign:
14    a Person with 15 as age.
15
16    the age of the Person is 21.
```

La impresión en pantalla sería la siguiente:

```
The age is: 15
The person is younger
```

3.2.3.5. Requires

La circunstancia `requires` implica que un sustantivo tiene que combinarse con uno o más adjetivos. Esta circunstancia permite utilizar modificadores `and` y `or`. El modificador `and` indica que la abstracción requiere forzosamente de todos los adjetivos que se especifican, mientras que `or` implica que requiere de al menos uno. A continuación se presenta un ejemplo:

```
noun House:
circumstance: this requires Habitable.
```

En el código 3.38 se indica un sustantivo `Academic` en la línea 1, en este hay varios adjetivos: `Student, Scholar` y `Resident` (especializaciones de universitario). En la línea 2 la circunstancia indica que se requiere que `Student` sea también `Scholar`.

Código 3.38: Código Requires en SN

```
1 noun Academic:
2   circumstance: This requires Student and Scholar.
3
4 adjective Student.
5 adjective Scholar.
6 adjective Resident.
7
8 main Prueba:
9   a Student.
```

En este caso no hay una impresión en pantalla, ya que solamente verifica que no se cree el becario sin ser estudiante.

3.2.3.6. Cannot

`Cannot` restringe el tipo de composición que se realiza entre un sustantivo y uno o más adjetivos. La circunstancia `cannot` posee dos modificadores; `and` indica que la instancia no debe poseer alguno de los adjetivos, mientras que `or` permite la existencia de los adjetivos siempre y cuando no se combinen entre sí. A continuación se presenta un ejemplo:

```
noun Buliding:
circumstance: this cannot be Office.
```

El uso de esta circunstancia en el código 3.39, crea un sustantivo `Person` el cual se especializa como `Worker, Unemployed` y `Intern`; en la línea 6 se especifica que una persona no puede ser `Worker` (trabajador) y `Unemployed` (desempleado) al mismo tiempo. La circunstancia de la línea 7 especifica que el `Worker` tampoco puede ser `Intern`.

Código 3.39: Código Cannot en SN

```
1  noun Person.
2  adjective Worker.
3  adjective Unemployed.
4  adjective Intern:
5  
6    circumstance: this cannot be Worker and Unemployed.
7    circumstance: this cannot be Worker and Intern.
8  
9  main prueba:
10   System prints "pruebas"
11   a Worker Person.
```

3.2.3.7. Mutually Excluded

La circunstancia `mutually excluded` se utiliza para indicar exclusión mutua entre adjetivos, de modo que se restringe el uso de adjetivos para evitar construcciones cuyo contexto es contradictorio. A continuación se presenta un ejemplo:

```
circumstance: Habitable and Ofice are mutually excluded.
```

Es posible definir una construcción similar con el uso de *requires* y *cannot*, sin embargo, esto las asocia a un sustantivo particular o a definirlas dentro de uno de los adjetivos involucrados en la composición, mientras que `mutally excluded` define la exclusión independientemente de dónde se defina, lo cual permite desacoplar su implementación para reutilización donde un sustantivo se combina con dos o más adjetivos que no necesariamente sean mutuamente excluyentes para su contexto particular.

En el código 3.40 se indica un sustantivo que se especializa con dos adjetivos `Car` y `Airplane`. En la línea 5 la circunstancia indica que `Car` es mutuamente excluyente con `Airplane`.

Código 3.40: Código Mutually Excluded en SN

```
noun Transport.
adjective Car.
adjective Airplane.

  circumstance: Car and Airplane are mutually excluded.

noun main:
  a Car.
```

3.3. Caso de estudio

El software es un producto que se encuentra protegido mediante el uso de una licencia de software, la cual permite establecer un contrato entre el desarrollador y el cliente. El caso de estudio de licenciamiento de software surge de la necesidad de autorizar o impedir el uso de software que se encuentra protegido por los derechos de propiedad intelectual.

3.3.1. Requerimientos del caso de estudio

Los requerimientos del caso de estudio de licenciamiento de software se encuentran originalmente redactados en el idioma inglés. Ya que SN es un lenguaje naturalístico cuya sintaxis es similar a la del inglés, se consideró que lo más adecuado sería no realizar una

traducción de los requerimientos, con el fin de facilitar la transformación a código. Los requerimientos se listan en la tabla 3.1.

Tabla 3.1: Requerimientos del sistema

Clave	**Descripción**
R1	Application developers must register applications with the server.
R2	Applications registering results in the assigment of unique number generated to identify clients.
R3	After application registration, developers and vendors can register applications for distribution, including registration of application details, system requirements, and usage rights models for a particular application.
R4	Application registering includes entering authentication information and contact details.
R5	After application registration, the developer uploads the application and the usage rigth enforcement code.
R6	The enforcement code or disallows usage of an application according to a particular usage rigth model.
R7	A user can browse through the applications and download applications and particular licenses.
R8	Downloading an application packages the application with the chosen usage rights model.
R9	The usage rigths management system (server) must be used to verify and enforce user rights according to specific models.
R10	Whenever an application is launched, the usage rights should be checked according the relevant model.
R11	Several usage rigths models may be enforced.

Continúa en la siguiente página

Tabla 3.1 – *Continuación de la página anterior*

Clave	**Descripción**
R12	Unlimited-usage: with this model, the server retains evidence of a single payment by the client, which when found, allows usage.
R13	Named-user license, if a client is registered for use, usage is allowed.
R14	Under a named user license, if a client is registered for use, usage is allowed.
R15	Time-limited: the client retains a log of when the license started and its expiry date.
R16	Once the time limit is reached, usage is disallowed.
R17	Featured-based: when a feature is invoked, the client checks usage rigths from the server.
R18	In feature-based, usage is allowed based on identity.
R19	Suscription-based: The user pays a monthly fee and checks at application launch whether the fee has been received.
R20	Pay-per-use or audit-based: each time an application is launched, the client is billed.
R21	Node-locked: usage is allowed based on node identification such as IP address.
R22	In an audit-based model, users must send the logs of the previous application usage to the server for billing.
R23	Concurrent usage: A certain number of nodes or users are allowed to access the application at a time.
R24	In the concurrent model, users can satisfy the usage rigths for the application only if the number of users who have concurrent usage rigths at any one time does not exceed the maximun number of granted rigths.

Continúa en la siguiente página

Tabla 3.1 – *Continuación de la página anterior*

Clave	Descripción
R25	The audit-based model, pay-per-use model, and concurrent usage model require server contact to determinate usage allowance.
R26	For pay-per-use, audit-based, and feature-based models, usage data is logged at application launch and, additionaly, according to the usage model.
R27	For the audit-based model, the logged usage data is sent to the server once a month for customer billing purposes.
R28	The time-limited model can be extended by purchasing a new license with a new expiry date.

3.3.2. Modelado (Enfoque de Temas)

El enfoque de temas [27] provee soporte al desarrollo de software orientado a aspectos en dos niveles del ciclo de desarrollo: requerimientos (Theme/Doc), en el cual se obtienen las vistas que muestran las relaciones y diseño (Theme/UML).

El enfoque de temas requiere de tres actividades:

1.- Análisis

- Realizar el análisis de requisitos con el fin de la identificación de temas, mediante la selección de términos clave.
- Theme/Doc muestra las relaciones entre comportamientos para facilitar la identificación de aspectos.

2.- Diseño

- Utilizar los temas que se localizaron en la fase de análisis para identificar posibles clases.
- Complementar y ajustar los detalles necesarios para el diseño.
- Verificar si surgen nuevos temas de corte.

3.- Composición

- Recombinar los modelos.
- Determinar las relaciones entre los temas.

3.3.2.1. Theme/Doc

Una vez que se identificaron los requerimientos del caso de estudio que se presentó en 3.3, se procede a realizar las siguientes actividades:

1. **Identificación de temas.**

   A partir de características, servicios o casos de uso, se realiza un análisis, con base en los requerimientos que se presentaron en la sección 3.3.1 que permite identificar las entidades potenciales, tanto de objetos (clases) como de temas (asuntos de base y de corte).

2. **Refinar el conjunto de temas.**

   Para realizar el refinamiento es necesario desarrollar un diagrama de vistas de relaciones (acciones). Una vista se compone de dos elementos: la acción (se representa con rombos) y los requerimientos asociados a esta (rectángulo redondeado).

En la figura 3.1, se muestran las relaciones que se obtienen del análisis de los requerimientos, a partir de estas relaciones, se obtienen los siguientes temas:

- 9 temas de licencias de software, una para cada tipo de licencia: *audit, concurrent, featured, named, node, payper, subscription, limited* y *unlimited.*
- 2 temas para realizar registros: de usuario y de aplicaciones.
- 1 tema para facturar.
- 1 tema para la descarga.
- 1 tema para verificar el modelo de la licencia.

En el diagrama 3.1 se aprecia que algunos requerimientos, se asocian a más de un tema, por ejemplo: el R11 que se asocia a todos los temas de licencia.

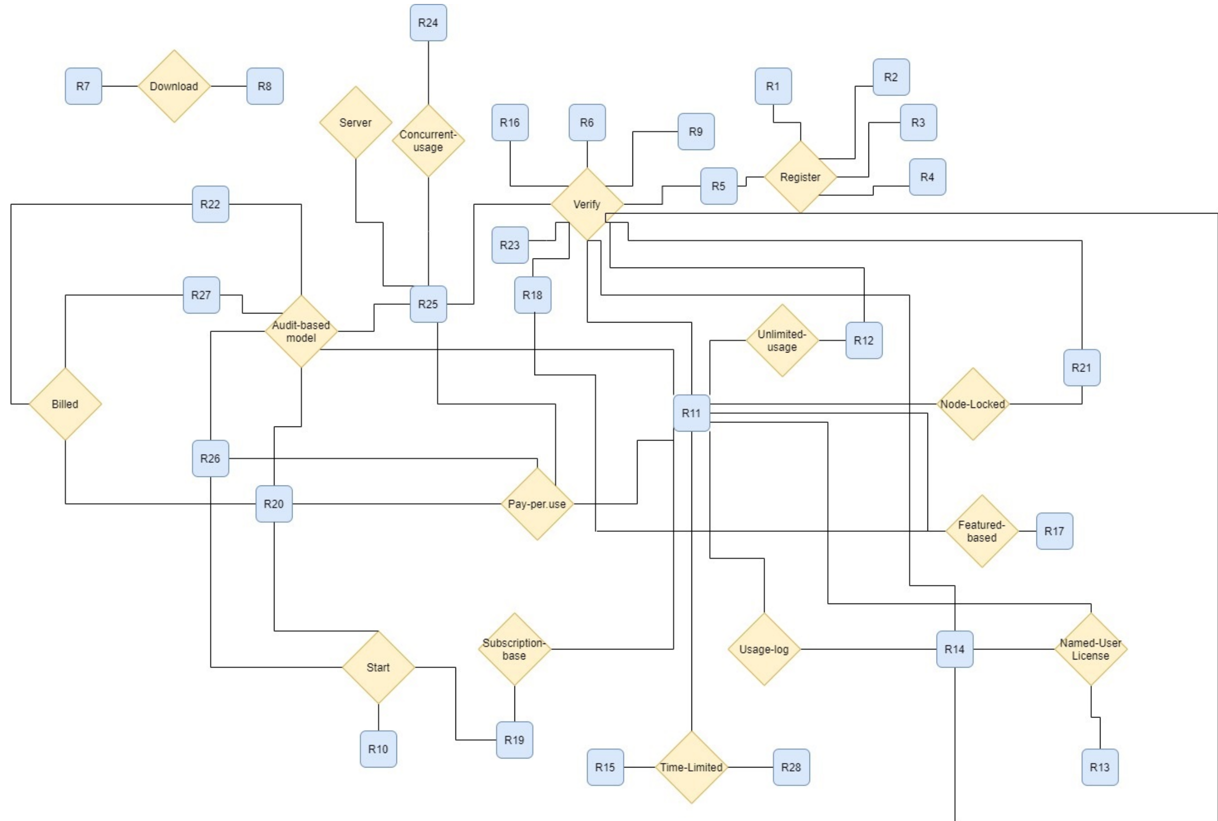


Figura 3.1: Descripción de la solución
:

3. **Revisión de temas usando el aumento de vista del tema.**

La revisión de temas por medio del aumento de vista (vistas individuales) permite determinar qué objetos son necesarios de modelar en Theme/UML. En estas vistas se muestra la relación específica entre temas, requisito y entidades. A continuación se muestran las vistas individuales.

En la figura 3.2, se muestra el tema *register* que mantiene relación con los requerimientos: R1, R2, R3, R4 y R5; estos requerimientos se encuentran asociados a posibles

entidades que son: *database, developer, client, application* y *license.*

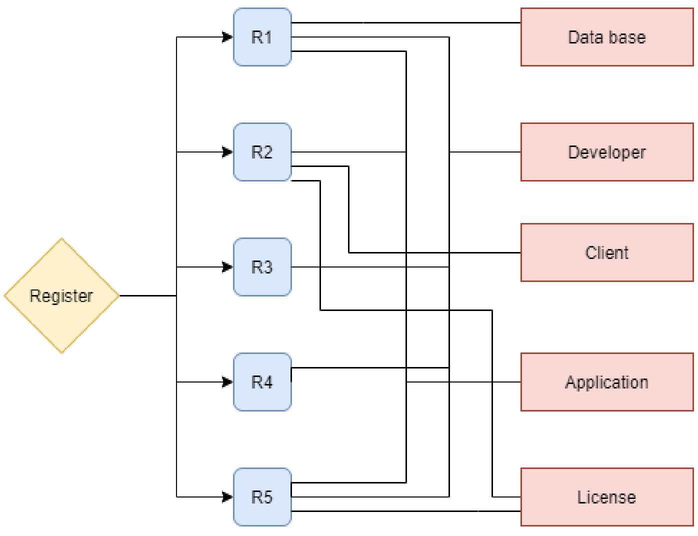


Figura 3.2: Diagrama de vista aumentada registro
:

En el caso del tema *download* 3.3, se asocia a los requerimientos: R7 y R8 al mismo que tiempo se muestran que las entidades que se relacionarían a estos requerimientos serían: *license* y *application.*

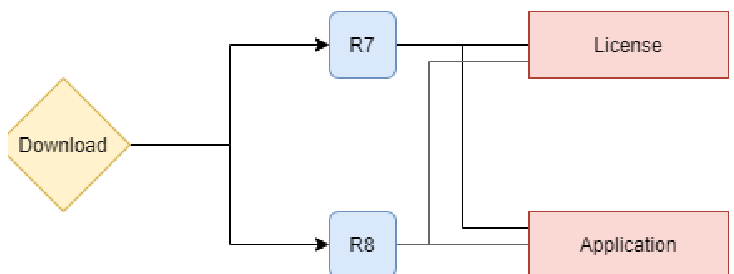


Figura 3.3: Diagrama de vista aumentada descarga
:

Para el modelo de licencia de auditoria 3.4 se asocia el tema *audit* con los requerimientos: R22 y R27, los cuales se asocian a las entidades: *data base, client, license* y *audit*.

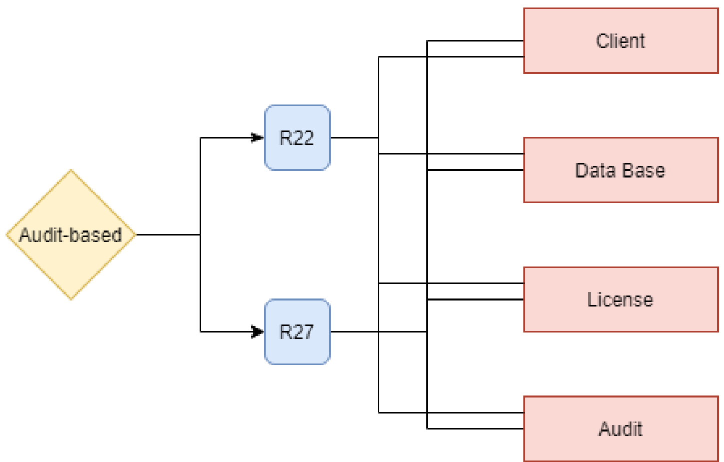


Figura 3.4: Diagrama de vista aumentada auditoria
:

El tema que hace referencia al tipo de licencia concurrente, figura 3.5, se enlaza con los requerimientos: R23, R24 y R25, de igual forma mantiene relación con las entidades: *license, concurrent* y *data base*.

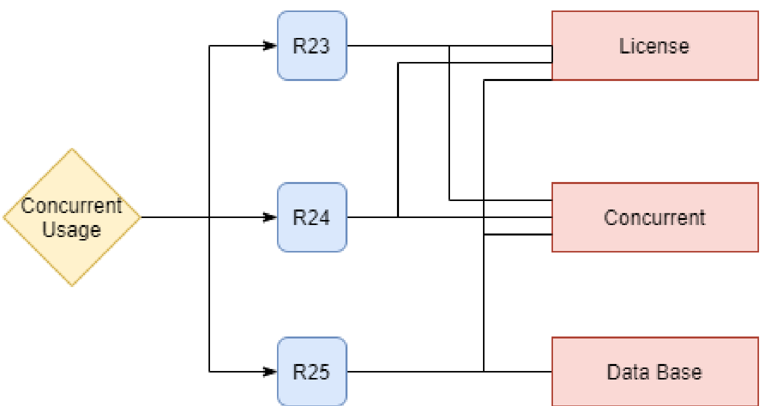


Figura 3.5: Diagrama de vista aumentada concurrente
:

En la figura 3.6, se presenta la relación del tema *featured-based* con los requerimientos: R17 y R18, los cuales requieren de la interacción entre las entidades: *client, license, featured* y *data base.*

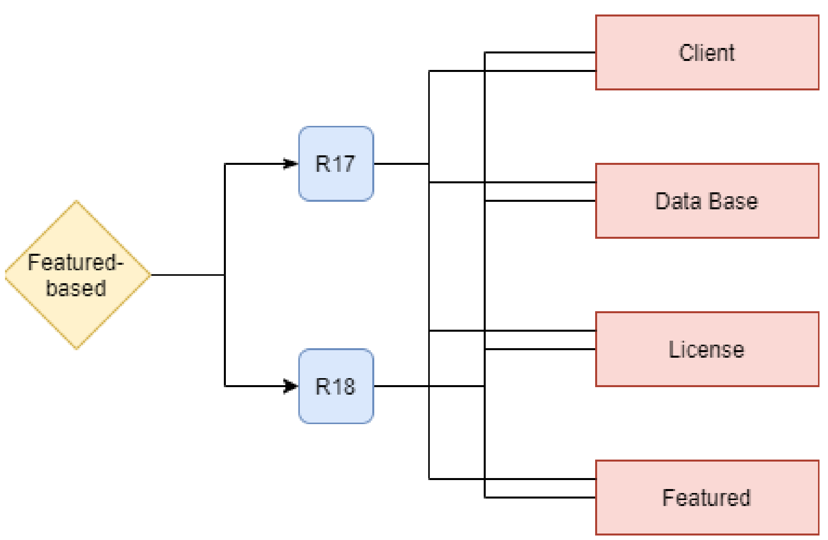


Figura 3.6: Diagrama de vista aumentada características
:

El tema limitado 3.7 muestra la asociación del tema *limited* con los requerimientos: R15, R16 y R28, y con las entidades: *client, license* y *limited.*

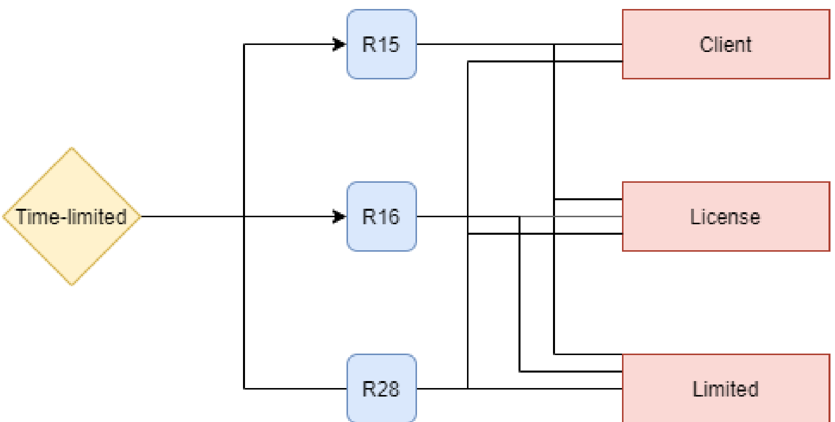


Figura 3.7: Diagrama de vista aumentada tiempo limitado
:

El tema *named* 3.8, se asocia a los requerimientos: R13 y R14, a su vez estos mantienen una interacción con: *client* y *data base*.

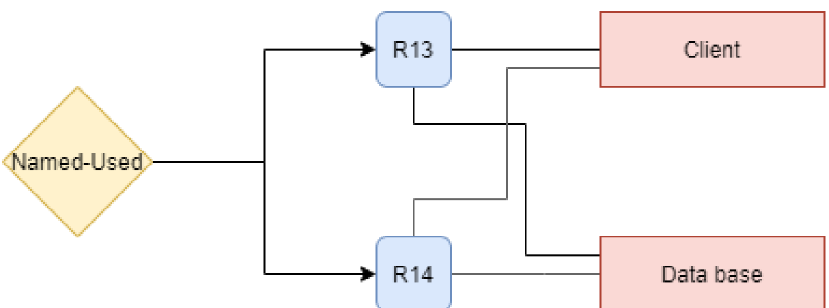


Figura 3.8: Diagrama de vista aumentada nombre registrado
:

La representación del enlace entre el tema *node*, con el requerimiento R21 se muestra en la figura 3.9, este requerimiento mantiene comunicación con las entidades: *license* y *node*.

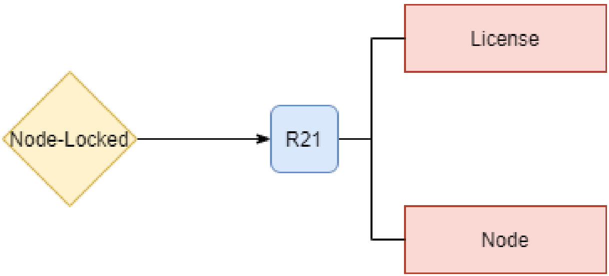


Figura 3.9: Diagrama de vista aumentada nodo bloqueado
:

En la figura 3.10, se muestra la vista individual del tema *payper-use* y su asociación con los requerimientos: R25 y R29, los cuales requieren de la interacción entre las entidades: *client*, *license*, *payper* y *data base*.

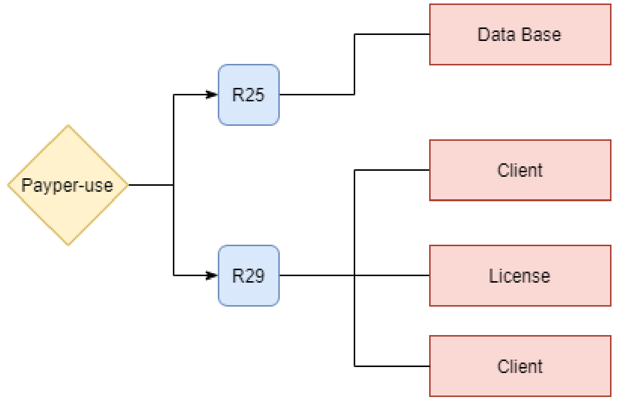


Figura 3.10: Diagrama de vista aumentada pago de uso
:

El tema *suscription* se asocia al requerimiento: R19, el cual requiere de la interacción con las entidades: *license* y *suscription*, como se muestra en la figura 3.11.

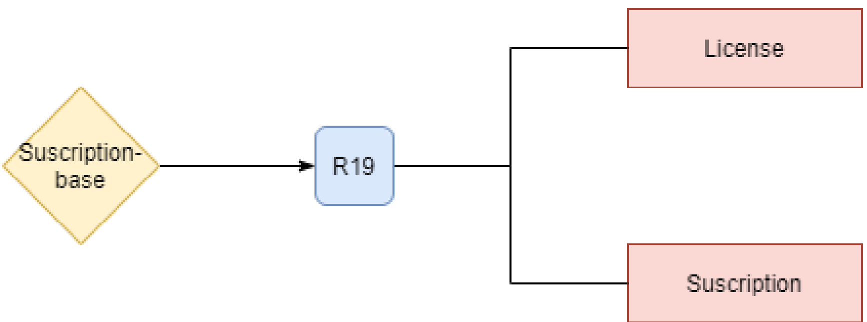


Figura 3.11: Diagrama de vista aumentada suscripción
:

En la figura 3.12, se presenta la relación del tema *unlimited usage* con el requerimiento: R12, que interactua con las entidades: *client, license, unlimited* y *data base.*

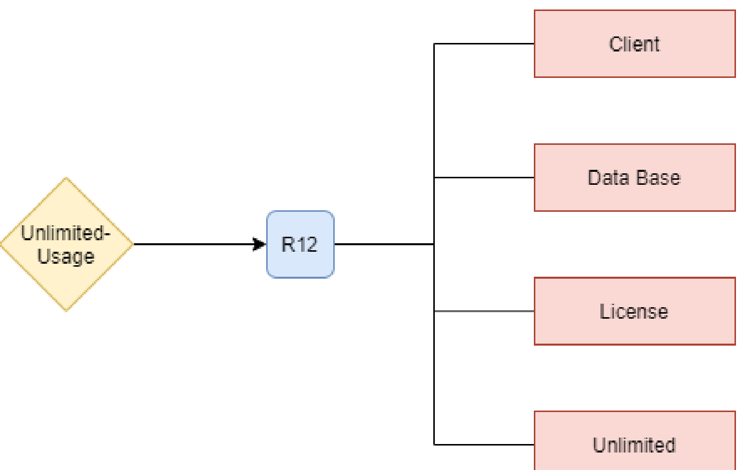


Figura 3.12: Diagrama de vista aumentada ilimitado
:

El tema *usage-log* se enlaza con el requerimiento: R14, y este mantiene relación con las entidades: *client* y *data base*, como se muestra en la figura 3.13.

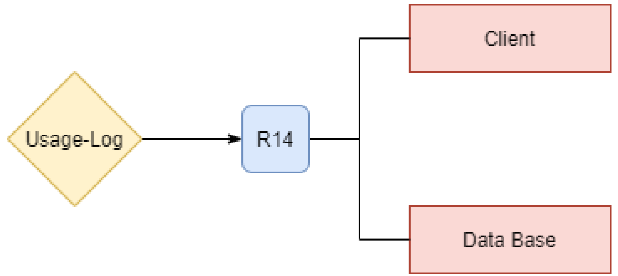


Figura 3.13: Diagrama de vista aumentada de uso
:

3.3.3. Modelado (Enfoque experimental)

Para el enfoque de programación naturalístico no se reporta una metodología que permita guiar el desarrollo, además, ya que uno de sus principales objetivos es reducir la brecha entre el dominio del problema y de la solución, se desconoce cuál debería ser el proceso para la etapa de diseño. Por lo anterior, se propone una simbología para diagramas, la cual se basa en el método Montessori [28]. María Montessori propone un modelo educativo [29] que permite la enseñanza de los elementos de la oración por medio de la asociación con ciertas figuras geométricas 3.14.

Sustantivo		Noun
Artículo		Article
Adjetivo		Adjective
Verbo		Verb
Adverbio		Adverb
Pronombre		Pronoun

Figura 3.14: Elementos propuestos por María Montessori
:

Sin embargo, no todos los elementos que se consideran en el método Montessori son necesarios para la representación de entidades en el desarrollo naturalístico con SN, por lo que se seleccionaron los que se asociaban a elementos de SN 3.15 y adecuaron algunos de los símbolos propuestos para detallar diagramas sencillos.

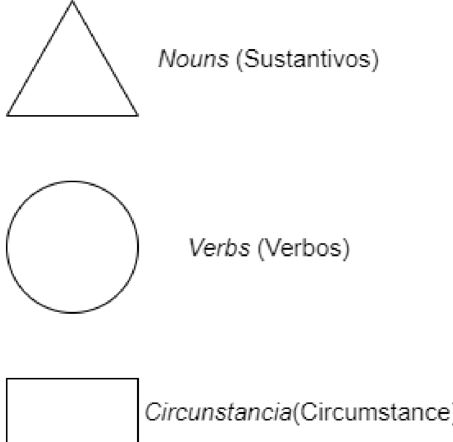


Figura 3.15: Elementos seleccionados para la representación de diagramas naturalísticos
:

Una vez que se definió cuáles serían los elementos a utilizar en los diagramas naturalísticos, se procedió a desarrollar los necesarios para el caso de estudio que se presentó en 3.3.

Según los requerimientos que se listan en 3.3.1, el desarrollador registra las aplicaciones y sube la aplicación al servidor. En la figura 3.16, se presenta un diagrama con los dos *nouns* (sustantivos) necesarios, en el caso de *developer* porque es quien realizará las actividades, mientras que *application* es sobre quien se realizarán los cambios. Antes de las actividades: *register* y *upload* se especifica la circunstancia que indica que antes de que el usuario pueda realizar estas actividades, debe haber accedido al sistema.

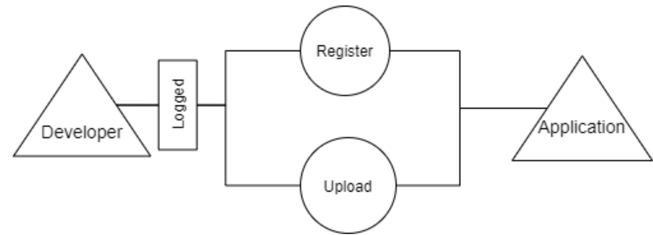


Figura 3.16: Diagrama naturalístico del desarrollador
:

Para ilustrar las actividades que realizará el cliente, se desarrollo un diagrama naturalístico, imagen 3.17, en el cual se ilustra que el cliente puede descargar o buscar aplicaciones, sin embargo, es necesario que el cliente haya accedido a la aplicación antes de poder realizar la descarga de aplicaciones.

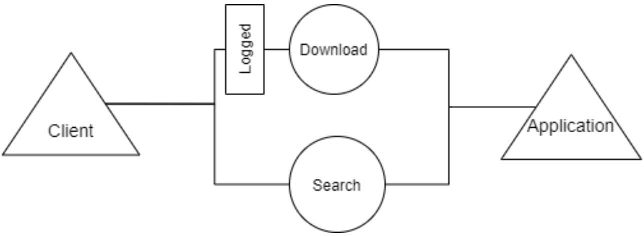


Figura 3.17: Diagrama naturalístico del cliente
:

En la figura 3.18 se especifica que una vez que la aplicación se ejecute, es necesario hacer una revisión de los derechos de usuario que están asociados a la compra de la aplicación, antes de permitir que se prosiga con el uso normal.

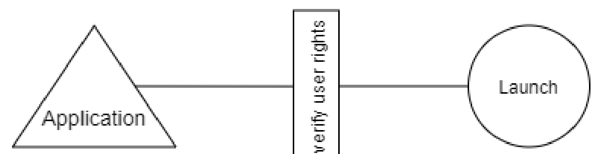


Figura 3.18: Diagrama naturalístico de la aplicación
:

Capítulo 4

Resultados

En este capítulo se describirán los resultados que se obtuvieron durante el desarrollo del caso de estudio propuesto para la comparativa.

4.1. Implementación del caso de estudio

Una vez que se realizó el análisis de requisitos y la transformación de estos mediante el diseño que se llevó a cabo en el capítulo anterior, se procedió a realizar la implementación a código Java y AspectJ para el paradigma orientado a aspectos, y SN para el paradigma naturalístico; por lo que en esta sección se explicarán algunos de los aspectos y circunstancias que modifican el comportamiento de los requisitos funcionales.

4.1.1. Implementación en Java y AspectJ

Para la implementación a código en AspectJ, se guió mediante un diagrama de clases, para el cual se tomaron en cuenta las entidades identificadas en el capítulo 3, durante el modelado bajo el enfoque de temas.
En la figura 4.1 se muestran las clases *application, user, client, developer, purchase, license,*

audit, concurrent, featured, named, node, payper, subscription, limited y *unlimited*. Los usuarios se especializan en dos tipos que son: desarrollador y cliente

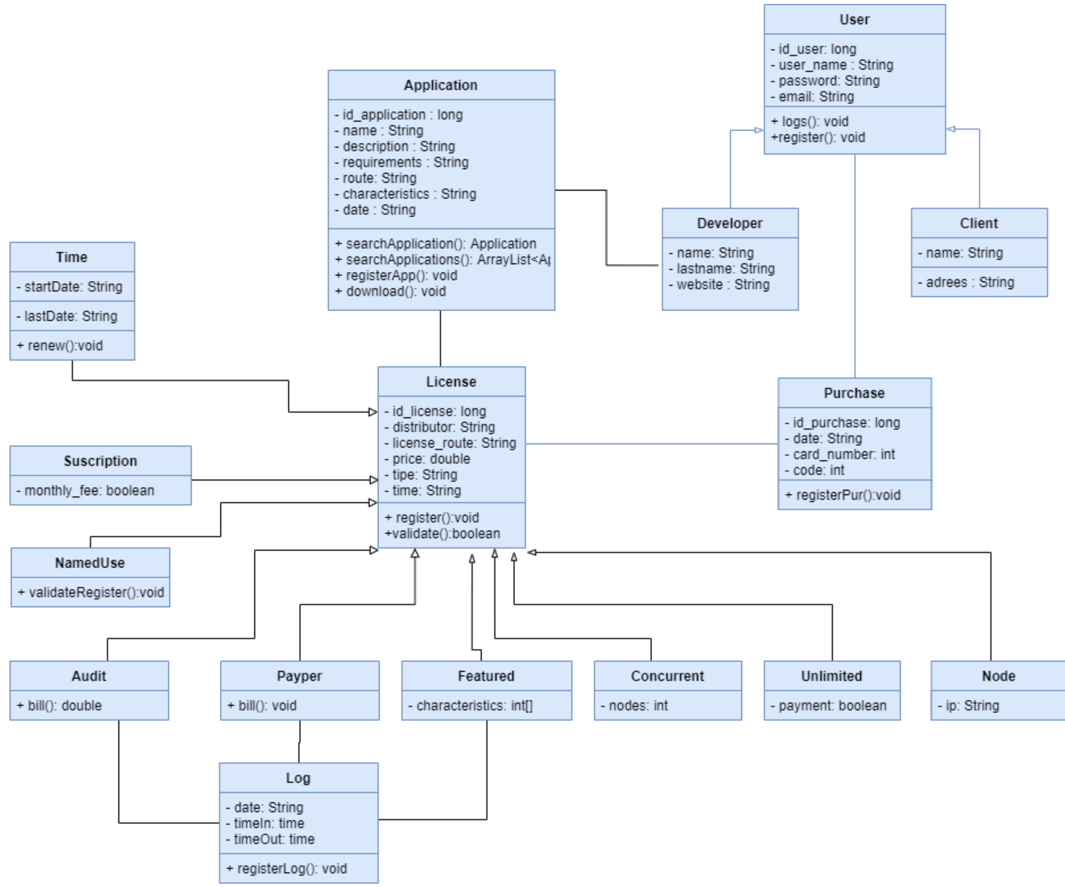


Figura 4.1: Diagrama UML de la aplicación

4.1.1.1. Aspecto para registros

En los requisitos del sistema se especificaba que al registrar alguna de las entidades, `license`, `purchase`, `user` y `application`, que se establecieron, se debe generar un número de identificación para que se almacene. A continuación se detallan dos casos del caso de estudio:

Registro de usuarios

El sistema genera un `id_user`, cuando un usuario se registra, tanto `Client` como `Developer`, por lo que se creó una clase abstracta llamada `User`, código 4.1, en la cual se implementan dos métodos abstractos, uno para el ingreso `logs` y otro para el registro de usuario `register`.

Código 4.1: Fragmento de código User en Java

```java
public abstract class User {
   long id_user;
   String user_name;
   String password;
   String email;

   public User(String user_name,String password, String email
      ){
    this.user_name = user_name;
    this.password = password;
    this.email = email;
  }

   public abstract void logs(String user_name, String
      password);

   public abstract void register();
   }
```

La clase `Developer` hereda de `User`, esto permite que implemente el método `register`, código 4.2, en este se genera la consulta necesaria para la inserción, luego de la creación de un objeto de tipo `Developer`, sin embargo, al crear el usuario no se genera el número de identificación.

Código 4.2: Fragmento de código Developer en Java

```
1  public class Developer extends User{
2    private String name;
3    private String lastname;
4    private String website;
5    private String cad = "";
6
7    public Developer(String name, String lastname, String
       website, String user_name, String password, String mail){
8      super(user_name, password, mail);
9      conection = new Connect();
10     this.name = name;
11     this.lastname = lastname;
12     this.website = website;
13   }
14
15     public void register(){
16     cad = "INSERT␣INTO␣developer(user_name,␣password,␣name,␣
         lastname,␣mail,␣id_developer)"
```

```
17          + " VALUES ('"+this.user_name+"','" +this.password+"
              ','"+this.nam+"','"+this.lastname+"','"+this.email+
              "','"+this.id_user+"');";
18      conection.insert(cad);
19
20    }
```

Por lo tanto, para complementar el objeto `Developer`, se define un corte en la línea 4 de código 4.3, en el cual se especifica que se ejecutará cuando se llama al método `register` de cualquier clase que herede `User`. En la línea 6, se realiza la exposición de contexto de las clases que hereden de `User` lo que permite manipular el objeto que se creó. Por último en la línea 10 se indica que se intervendrá antes de la ejecución del método `register`; se genera un valor de tipo `long` y se asigna al usuario.

Código 4.3: Código para generar la clave del usuario en AspectJ

```
1  public aspect registerAspect {
2    User user = null;
3
4    pointcut registerUser(): execution(void project.User+.
       register());
5
6    before(User us): execution(project.User+.new(..))&& target(
       us){
7      user = us;
8    }
9
10   before(): registerUser(){
```

```
    Random random = new Random();
    id=random.nextInt(1000000)*System.currentTimeMillis();
    user.setId_user(id);
  }
```

Registro de licencia de software

Existen varios tipos de licencia, sin embargo, todas comparten características, por lo que se declaró una clase abstracta llamada `License`, código 4.4, en la cual se especifican dos métodos abstractos, `register` y `validate` que se implementarán en las clases que heredan las diferentes licencias.

Código 4.4: Fragmento de código License en Java

```

public abstract class License {
  private String id_license;
  private String distributor;
  private double price;
  private String tipe;
  private String route;
  private String time;

  public License(String distributor, double price, String
     tipe, String route, String time) {
        this.distributor = distributor;
        this.price = price;
        this.tipe = tipe;
```

```
14          this.route = route;
15          this.time = time;
16          conection = new Connect();
17      }
18
19    public abstract void register();
20
21    public abstract boolean validate();
22  }
```

Una de las licencias que heredaron de `License` es `Node`, por lo que en esta clase, código 4.5, se implementan los métodos abstractos, al igual que en el caso del usuario, el identificador que sirve igual como cadena para verificar la licencia no se agrega al objeto.

Código 4.5: Fragmento de código Node en Java

```
1  public class Concurrent extends License{
2    private int nodes;
3
4    public Node(String distributor, double price, String tipe,
         String route, String time,String ip){
5      super(distributor,  price,  tipe, route,  time);
6      this.ip = ip;
7    }
8
9    @Override
10   public void register() {
```

```
11      cad ="INSERT␣INTO␣license(id_license,␣distribuitor,␣
           license_route,␣price,␣tipe)"
12          + "␣VALUES␣('"+this.getId_license()+"','"+this.
               getDistributor()+"','"+this.getRoute()
13          +"',"+this.getPrice()+ ",'"+this.getTipe()+"');";
14      conection.insert(cad);
15      cad="INSERT␣INTO␣concurrent(id_license,␣price,␣nodes)"
16          + "VALUES␣('"+this.getId_license()+"',"+this.getPrice
               ()+","+this.getNodes()+");";
17      conection.insert(cad);
18    }
19  }
```

El número de identificación se genera por medio de un aspecto, código 4.6, en la línea 3 se define el corte para el registro de licencias, especificando que se intervendrá al momento de ejecución del método `register` de cualquier clase que herede de `License`. En la línea 7 realiza la exposición de contexto de las licencias. El aviso que se especifica en la línea 11, indica la generación de un `id` alfa-numérico antes de que se ejecute el método `register`.

Código 4.6: Código que genera el identificador de las licencias

```
1  public aspect registerAspect {
2
3    License  license = null;
4
5    pointcut registerLic():  execution(void project.License+.
        register());
6
```

```
7   before(License lic):execution(project.License+.new(..))&&
       target(lic){
8     license=lic;
9   }
10
11  before():registerLic(){
12    String id="";
13      Random rng = new Random();
14        int dig3 = rng.nextInt(9000) + 10000;
15        int dig5 = rng.nextInt(9000000) + 1000000;
16        id = decimalAHexadecimal(dig5) + "-" +
              decimalAHexadecimal(dig3) + "-" +
              decimalAHexadecimal(System.currentTimeMillis());
17        JOptionPane.showMessageDialog(null, id);
18        license.setId_license(id);
19
20        System.out.println(id);
21  }
22
23   public String decimalAHexadecimal(long decimal) {
24          String hexadecimal = "";
25          String caracteresHexadecimales = "0123456789abcdef"
               ;
26          while (decimal > 0) {
27              int residuo =(int)decimal % 16;
28              hexadecimal = caracteresHexadecimales.charAt(
                   residuo) + hexadecimal;
```

```
29                  decimal /= 16;
30              }
31              return hexadecimal;
32          }
33  }
```

Acceso al sistema

Existen algunas actividades que no pueden llevarse a cabo si el usuario no ha ingresado al sistema, tales como la descarga, el registro y compra de licencias, código 4.7. En este caso se ejemplifica que el método al que se interviene es `download` de la clase `Application`, en este el usuario que realice la descarga debe seleccionar la carpeta en la que desea guardar la aplicación.

Código 4.7: Fragmento de código descarga en Java

```
1  public class Application {
2    public void download(String nameApp){
3      Application a = searchApplication(nameApp);
4      File file = null;
5      String opath=a.getRoute();
6      String dpath = "";
7          JFileChooser fileChooserCargar = new JFileChooser();
8          fileChooserCargar.setDialogTitle("Choose");
9          fileChooserCargar.setCurrentDirectory(new java.io.
             File("."));
10         fileChooserCargar.setFileSelectionMode(JFileChooser.
             DIRECTORIES_ONLY);
```

```
11          int seleccion = fileChooserCargar.showOpenDialog(new
               JPanel());
12
13          if (seleccion == JFileChooser.APPROVE_OPTION) {
14              file = fileChooserCargar.getSelectedFile();
15          }
16      ...
17  }
```

Sin embargo, para que pueda proceder con la acción, el usuario ya debe haber accedido al sistema, por lo que se revisa con el aspecto, código 4.8; en la línea 4 se define un corte que indica la intervención cuando algún tipo de usuario realice la ejecución del método `logs`. En la línea 7, en aviso *after* se revisa que el usuario no se encuentre sin datos, y que el `id` exista, en caso de que esto no se cumpla, se lanza una excepción que indica que el usuario no se encuentra registrado. En la línea 17, se declara otro corte, el cual especifica los métodos que necesitan verificar el acceso.

Código 4.8: Código de acceso en AspectJ

```
1
2 public aspect LoginAspect {
3   User user = null;
4
5   pointcut login(User user) : execution(public void project.
       User+.logs(..)) && target(user);
6
7   after(User user) : login(user) {
8     System.out.println(user.getId_user());
```

```
9       if(user != null && user.getId_user() != 0.0) {
10        this.user = user;
11      } else {
12        throw new RuntimeException("User␣is␣not␣registered");
13      }
14
15    }
16
17    pointcut options(User user) : call(public * project.License
        +.register())
18                    && call(public * project.Purchase.
                        registerPur(..))
19                    && call(public * project.Application.
                        download(..))
20                                && call(public * project.User
                                    +.*(..))
21                                && !within(project.User+)
22                                && !within(project.LoginAspect
                                    +)
23                                && !call(* project.User+.set
                                    *(..))
24                                && !call(* project.User+.get
                                    *(..))
25                                && !call(public void project.
                                    User+.logs())
26                                && target(user);
27
```

```
28    Object around(User user) : options(user) {
29      if(this.user != user) {
30        throw new RuntimeException("El␣usuario␣debe␣estar␣
            identificado");
31      } else {
32        return proceed(user);
33    }
34  }
```

Revisión de los derechos de usuario

Los requisitos solicitan que una vez que se ejecute la aplicación, se revise cuáles son los derechos de usuario que están asociados, para el caso que se presenta, solo se realizó una simulación mediante un método de la clase `Application`, código 4.9, el cual recibe el nombre de la aplicación.

Código 4.9: Código que simula la ejecución de la aplicación

```
1    public void launch(String nameApp){
2      Application app = searchApplication(nameApp);
3      JOptionPane.showMessageDialog(null, app.getName() + "␣" +
           app.getDeconsultaion());
4    }
```

Para la revisión de licencias, se realizó herencia de aspectos, por lo que se declaró un aspecto abstracto, código 4.10, en el cual en la línea 6 se define que el corte se aplicará sobre la ejecución del método `launch` que recibe un valor de tipo `String`. Este corte podrá ser usado por los aspectos que hereden de este.

Código 4.10: Código de licencia en AspectJ en SN

```
1 public abstract aspect LicenseAspect {
2
3   String license = "";
4   ResultSet rs= null;
5
6   pointcut launchApp(): execution(public void project.
       Application.launch(String));
7 }
```

Lo primero que es necesario realizar una vez que se ejecuta la aplicación, es solicitar la cadena de identificación de la licencia. En el código 4.11 se define el aspecto `NodeAspect` el cual hereda de `LicenseAspect`. En la línea 3, se define un aviso que se ejecuta en vez del método `launch`, esta será la primer licencia que se verifique si existe, por lo que después de realizar la búsqueda de la licencia, en la línea 5 se comprueba que la licencia no esté vacía. En el caso de que la variable licencia no se encuentre vacía, se hace una comparación para saber si el tipo de licencia coincide con el tipo `Node`, el sistema obtiene la ip del computador en el que se está ejecutando. Si el tipo de licencia coincidió y la IP también se procede con la ejecución normal, en caso contrario, se lanza una excepción que indica que la licencia no está asociada a esa IP. Si el tipo de licencia no coincide, entonces se continúa con la ejecución para que los demás aspectos asociados a licencias realicen la revisión.

Código 4.11: Código Operations en SN

```
1 public aspect NodeAspect extends LicenseAspect{
2
3   void around(): launchApp(){
```

```
4      license = search(JOptionPane.showInputDialog("Write␣your␣
          license"));
5      if(license == null){
6        new Exception("Your␣license␣if␣doesn't␣exist");
7      }else{
8      if(license.compareTo("Node")==0){
9         InetAddress address = null;
10             try {
11                 address = InetAddress.getLocalHost();
12             } catch (UnknownHostException ex) {
13
14             }
15             System.out.println("IP␣Local␣:" + address.
                  getHostAddress());
16        Node n = new Node();
17        n.setId_license(license);
18        n.setIp(address.getHostAddress());
19        if(n.validate() == true){
20          proceed();
21        }
22        else{
23          new Exception("Your␣license␣if␣doesn't␣exist");
24        }
25     }else{
26        proceed();
27     }
28   }}
```

En el caso del último aspecto de los modelos de licencia, ya no verifica si es el tipo de licencia coincide, ya que en este último caso, debió verificar primero si la licencia existía y si pertenecía alguna de las otras licencias.

Código 4.12: Código Operations en SN

```
1 public aspect UnlimitedAspect extends LicenseAspect {
2
3   void around():launchApp(){
4     proceed();
5   }
6 }
```

4.1.2. Implementación en SN

Ya que en el desarrollo naturalístico no hay un método para el desarrollo, además de que su objetivo es reducir la brecha entre el dominio del problema y la solución; por lo que uno de sus objetivos principales es disminuir la traducción de requisitos a código, por esta razón no se desarrollaron más diagramas.

4.1.2.1. Circunstancias para registros

El lenguaje SN, al momento de redacción de esta tesis, no cuenta con la capacidad de generar números aleatorios, por lo que no se consideró parte para el registro de ningún sustantivo.

Registro de aplicaciones.

En el código 4.13 se define el sustantivo `Application`, el cual tiene como atributos: `name_application`, `description`, `requirements`, `route`, `upload_date`, `idApplication`

y `id_developer`. En la línea 15 se especifica el verbo *register*, el cual permite armar el consulta sql que inserta los datos. En la línea 30, la circunstancia indica que cuando se cree una aplicación, se realizará el registro.

Código 4.13: Fragmento de código aplicación en SN

```
1  noun Application with plural as Applications:
2
3    attribute application_name as a String.
4    attribute description as a String.
5    attribute requirements as a String.
6    attribute route as a String.
7    attribute upload_date as a String.
8    attribute idApplication as a Long Number.
9    attribute id_developer as a String.
10
11   attribute developer as a Developer.
12   attribute client as a Client.
13   attribute license as a License.
14
15     verb itself register:
16     idApplication is a Persistent and Long Number with the
          idApplication of this as value.
17     app is a Connectable Application with idApplication as
          idApplication and the id_developer of developer as
          id_developer.
18
```

```
    rows are some Strings.
    add "idapplication" to rows.
    add "application_name" to rows.
    add "description" to rows.
    add "requirements" to rows.
    add "route" to rows.
    add "upload_date" to rows.
    add "id_developer" to rows.

    insert rows into app.

  circumstance: it register when this is created.
```

Registro de usuarios

En el caso de los usuarios, SN permite establecer una jerarquía de sustantivos. En la línea 7 se define que `Developer` es un tipo de usuario, en la línea 14 se define el verbo `register`. Mientras que en la línea 26 se define la circunstancia que indica que una vez que se cree el desarrollador, se realice el verbo registrar.

Código 4.14: Código Operations en SN

```
abstract noun User:

  attribute user_name as a String.
  attribute password as a String.
  attribute mail as a String.

noun Developer is User:

  attribute id_developer as a Long Number.
  attribute name as a String.
```

```
11    attribute lastname as a String.
12    attribute website as a String.
13
14    verb itself register:
15
16      user_name is a Persistent String with the user_name of
          this as value.
17      password is a Persistent String with the password of this
           as value.
18
19      name is a Persistent String with the name of this as
          value.
20      lastname is a Persistent String with the name of this as
          value.
21      mail is a Persistent String with the mail of this as
          value.
22      id_developer is a Persistent String with id_developer of
          this as value.
23
24      developer is a Connectable Developer with user_name as
          user_name, password as password, name as name, lastname
           as lastname, mail as mail and id_developer as
          id_developer.
25
26    circumstance: it register when this is created.
```

Composición de licencias de software

Además de permitir una jerarquía similar a la herencia, SN permite la especialización por medio de adjetivos, en el código 4.15, se muestran algunos de los adjetivos que representan a los tipos de licencias que se encuentran especificadas en los requisitos. En la línea 11 se define un sustantivo abstracto, el cual cuenta con los atributos de `id_license`,

distributor, license_route, price, time y tipe. Sin embargo, es necesario el uso de circunstancias para especificar que no pueden ser dos tipos de licencia al mismo tiempo, a partir de la línea 20 hasta la línea 27 se presenta un extracto de las circunstancias de exclusión mutua, que indican que una licencia no puede ser, por ejemplo, de tipo limited y unlimited, al mismo tiempo.

Código 4.15: Fragmento de código licencia en SN

```
1
2  adjective Limited:
3
4    attribute startDate as a String.
5    attribute endDate as a String.
6
7  adjective Named.
8
9  adjective Node:
10
11 abstract noun License:
12
13   attribute id_license as a Long Number.
14   attribute distributor as a String.
15   attribute license_route as a String.
16   attribute price as a Float Number.
17   attribute tipe as a String.
18   attribute time as a String.
```

```
19
20  circumstance: Limited and Unlimited are mutually excluded.
21  circunstance: Audit and Concurrent are mutually excluded.
22  circunstance: Concurrent and Featured are mutually excluded
      .
23  circumstance: Featured and Named are mutually excluded.
24  circunstance: Named and Node are mutually excluded.
25  circunstance: Node and Payper are mutually excluded.
26  circumstance: Payper and Subscription are mutually excluded
      .
27  circunstance: Subscription and Unlimited are mutually
      excluded.
```

Acceso al sistema

Para verificar el acceso al sistema, es necesario especificar la circunstancia en cada sustantivo que contenga un verbo que no se permita ejecutar sin que el usuario ingrese primero al sistema. En el código 4.16 se ejemplifica el caso de la descarga de una aplicación. En la línea 17, la circunstancia define que deberá ejecutarse el verbo `validate` antes de que se realice la ejecución del verbo `download`.

Código 4.16: Fragmento de código Application en SN

```
1 noun Application with plural as Applications:
2
3 verb download nameApp as String from itself:
4
```

```
5     idApp is a Persistent String with nameApp as value.
6     idDeveloper is a Persistent String with the id of
         developer as value.
7
8     app is a Connectable Application with idApp as id and
         idDeveloper as id_developer.
9
10    rows are some Strings.
11    add "idapplication" to rows.
12    add "id_developer" to rows.
13
14    select the Strings from app.
15    return the first Application from these.
16
17    circumstance: validate the user of this before download
         nameApp something.
```

Revisión de Licencias de software

La revisión de las licencias de software se lleva a cabo una vez que se ejecuta la aplicación (verbo `launch`), para revisar si pertenece a un tipo de licencia el identificador que el usuario proporciona, se realiza la comparativa en el método `verify`, en este caso se comienza por el tipo de licencia ilimitado.

Código 4.17: Código Operations en SN

```
1 noun Application with plural as Applications:
```

```
2
3   verb itself launch application as String:
4     System prints "Launch␣Application" and app.
5
6   verb itself verify application as String:
7     System prints "Please␣write␣your␣license␣number" and new
        line.
8     System reads.
9
10    circumstance: it verify application before something
        launch application.
```

4.2. Comparativa entre la implementación de solución de AspectJ y SN

En esta sección se explicarán, las similitudes y diferencias que se encontraron entre la solución propuesta con AspectJ y SN.

4.2.1. Registro

En el caso de los registros, los aspectos y circunstancias no se pudieron implementar de la misma forma (Tabla 4.1); en AspectJ, el aviso intervenía antes de la ejecución del registro para generar un valor de identificación; en SN no se cuenta con un mecanismo que permita la generación de números para crear el id, sin embargo, se declaró en la circunstancia que

luego de que se creara el sustantivo (*User, Application, Purchase, License* y *Log*), se realizara el registro en base de datos.

Tabla 4.1: Aplicación de aspectos y circunstancia en el caso: registro

Corte y Aviso	`pointcut registerUser():`
	`execution(void project.User+.register());`
	`before(User us): execution(project.User+.new(..))`
	`                 && target(us){`
	` user = us; }`
	`before(): registerUser(){`
	`   Random random = new Random();`
	`   id=random.nextInt(1000000)*System.currentTimeMillis();`
	`   user.setId_user(id);  }`

Circunstancia	`circumstance: it register when this is created.`

4.2.2. Acceso

Fue necesario controlar que no todos los métodos pudieran ejecutarse sin que el usuario ingresara previamente al sistema. Las diferencias que se presentan ante la solución propuesta en ambos lenguajes, se muestran en la Tabla 4.2, son: en AspectJ se declaró un aspecto que obtenía la instancia del `User`, de igual forma por medio de un corte se especificaron los diferentes métodos (que se encuentran en distintas clases del sistema) sobre los que actuaría la verificación de usuario; por otra parte, en SN fue necesario que la circunstancia, que indicaba que se debía validar el acceso del usuario al sistema, se encontrara en cada uno de los sustantivos que contenía uno de los verbos a intervenir.

Tabla 4.2: Aplicación de aspectos y circunstancias en el caso: acceso

Corte y Aviso	pointcut options(User user) :
	call(public * License+.register())
	&& call(public * project.Purchase.registerPur(..))
	&& call(public * project.Application.download(..))
	&& call(public * project.User+.*(..))
	&& !within(project.User+)
	&& !within(project.LoginAspect+)
	&& !call(* project.User+.set*(..))
	&& !call(* project.User+.get*(..))
	&& !call(public void project.User+.logs())
	&& target(user);
	
	Object around(User user) : options(user) {
	if(this.user != user) {
	throw new RuntimeException("You need to log in");
	} else {
	return proceed(user);
	}

Circunstancia	circumstance: validate the user of this before download nameApp something. circumstance: validate the user of this before register something. circumstance: validate the user of this before something register_purchase.

4.2.3. Revisión de licencias

Una de las actividades principales en el sistema es verificar que una aplicación se encuentre asociada a una licencia válida antes de su ejecución, por lo que en AspectJ se realizaron nueve aspectos que se encargaban de validar a qué tipo de licencia pertenece. En SN no se realizó esta misma implementación, ya que en este caso se declaró un verbo

que permite realizar por medio de comparativas si la licencia existe o a cual tipo pertenece, por lo que la circunstancia solo especifica que este verbo se debe llevar a cabo antes de la ejecución de la aplicación. La diferencia entra la implementación en código con ambos lenguajes, se encuentra en la Tabla 4.3.

Tabla 4.3: Aplicación de aspectos y circunstancia en el caso: revisión de licencias

Corte y Aviso	`pointcut launchApp():`
	`execution(public void project.Application.launch(String));`
	`public aspect NodeAspect extends LicenseAspect{..}`
	`public aspect UnlimitedAspect extends LicenseAspect{..}`
	`public aspect AuditAspect extends LicenseAspect{..}`
	`public aspect ConcurrentAspect extends LicenseAspect{..}`
	`public aspect FeaturedAspect extends LicenseAspect{..}`

Circunstancia	`circumstance: it verify application` `before something launch application.`

4.3. Comparativa de las capacidades de AspectJ y SN

AspectJ provee diecisiete primitivas de corte que permiten encapsular los requisitos no funcionales, mientras que SN cuenta con siete tipos de circunstancias que proveen la capacidad de trabajo para dar un tratamiento correcto a los NFR. Una vez que se analizaron estas características de cada uno de los lenguajes se encontraron algunas posibles equivalencias que permitirían tener un comportamiento semejante en ambos lenguajes.

4.3.1. Comparativa de avisos y contexto de ejecución

Los avisos y los contextos de ejecución sirven para especificar en que momento se realizará una acción ajena al comportamiento original, pero que es necesaria para que el funcionamiento se encuentre completo. AspectJ documenta tres tipos de avisos: *before*, *around* y *after*; por otra parte SN reporta cuatro contextos de ejecución: *before*, *instead*, *after* y *when*. En la Tabla 4.4 se muestra el momento, tanto de circunstancias como de avisos, en el que permite agregar o cambiar comportamiento.

Tabla 4.4: Comparativa de contextos de ejecución

Tiempo en el que se ejecuta	**Avisos**	**Circunstancias**
Antes	before	before
Después	after	after
Cambia la ejecución original	around	instead
Depende de la definición	No tiene	when

Con base en lo anterior se concluye que en ambos lenguajes es posible definir al menos tres tiempos básicos de ejecución, que se explican como: *antes*, *en lugar de* y *después*. Sin embargo, SN cuenta con un contexto más que AspectJ, *when* el cual según su definición es posible que se utilice como *before* o *after*.

4.3.2. Comparativa de primitivas y circunstancias

Al analizar ambos lenguajes y las capacidades que presentaban ambos para el encapsulamiento de los requisitos no funcionales se encontraron diferencias significativas en la implementación de código, las cuáles se muestran en la Tabla 4.5; AspectJ declara un corte que según la definición y primitiva que se implemente, es capaz de intervenir: variables, métodos, constructores, objetos, bloques de inicialización y estáticos, siempre que cumplan con un patrón de firma, si el comportamiento ya se encuentra en el código de algún método, para accederlo solo es necesario realizar la exposición de contexto de un objeto o crear una instancia, en el caso de que este no codificara anteriormente es posible agregar las instrucciones como parte del aviso, por lo tanto, no es necesario que el aspecto se encuentre en el mismo archivo que la clase. En SN las circunstancias operan sobre: verbos, atributos y adjetivos; las instrucciones que se añaden o sustituyen a la ejecución original deben indicarse en un verbo, por lo cual no es posible que se agreguen como fragmento de código externo; la circunstancia se debe definir dentro del sustantivo sobre el cual va actuar.

Tabla 4.5: Implementación de primitivas y circunstancias

	Aspectos	**Circunstancias**
Propiedades en las que actúa	1. Variables 2. Métodos 3. Argumentos 4. Constructores 5. Atributos 6. Objetos 7. Inicializadores estáticos 8. Inicializadores de instancia	1. Verbos 2. Atributos 3. Adjetivos 4. Sustantivos
Definición de instrucciones	Independiente de lo que se establece en la clase de Java	Cualquier instrucción que ejecute la circunstancia debe especificarse como parte de un verbo
Separación de asuntos	El aspecto se declara y programa desde un archivo independiente	La circunstancia se especifica dentro del sustantivo que hará uso de ella

4.3.2.1. Equivalencias entre primitivas y circunstancias

Se encontraron tres equivalencias entre primitivas y circunstancias, las cuáles intervienen ciertos comportamientos como:

- **Ejecución:** Mientras que en AspectJ es necesario especificar mediante la palabra reservada *execution*, en SN no es necesario indicarlo, ya que se encuentra de manera explícita en todas las circunstancias.

- **Asignación:** Cuando se realiza la asignación de una variable en Java, la primitiva de AspectJ que permite intervenir es *set*. Mientras que en SN la circunstancia se define con el uso de la palabra *assigned*.

- **Exposición de contexto:** En ambos lenguajes es posible que se realicen dos tipos de exposición de contexto

    - *Argumentos:* Para lograr intervenir los argumentos con los que trabaja un método (en el caso de AspectJ) o un verbo (en el caso de SN). Para el primer caso es necesario utilizar la palabra reservada *args*, por su parte en el segundo caso esta se encuentra implícita en la definición de la circunstancia.

    - *Objetos o Sustantivos (Noun):* Esto con el fin de trabajar con las propiedades que cuenta la abstracción sobre la cual se está trabajando; para realizar la exposición de un objeto de modo que sea posible trabajar con él a lo largo del aspecto se define el corte con la primitiva *target*, en cambio SN, al igual que en el caso de los argumentos, lo deja de forma explícita en sus instrucciones.

En la Tabla 4.6 se ejemplifican las diferentes sintaxis de ambos lenguajes para cada uno de los puntos anteriormente descritos.

Tabla 4.6: Comparativa entre circunstancias y primitivas que tienen equivalencia

Comportamiento en el que intervienen	Implementación en SN	Implementación en AspectJ
Ejecución	Implícita en la definición circumstance: enter it before show something.	pointcut corte(): execution(*package.method(..));
Asignación	circumstance: it verify age after the age of something is assigned.	pointcut escribeCampo(int a): set(int setEjemplo.Persona.edad) && args(a);
Exposición de argumentos	circumstance: it replace message instead something displays message.	pointcut argument(int x): execution(voidpackage.m(int)) && args(x);
Exposición de objetos o sustantivos	Implícito en la definición circumstances: execute System print it when this is created.	pointcut argument(Object x): execution(void package.m(int)) && target(x);

4.3.2.2. Primitivas y circunstancias sin equivalencia

Como se mencionó al principio de la sección, AspectJ cuenta con diecisiete primitivas, mientras que SN solo reporta siete circunstancias, por lo cual no es posible encontrar una equivalencia entre todas las primitivas y circunstancias.

Primitivas sin equivalencia en las circunstancias

Luego de analizar las propiedades de ambos lenguajes para encontrar las similitudes y equivalencias entre ambos, se encontró que en algunos casos las circunstancias no tenían forma de implementar una ejecución similar a la que provee AspectJ, de forma que no es posible comparar su sintaxis, ni su ejecución. Las primitivas que no tienen un similar en SN son:

- *call*
- *this*
- *get*
- *within*
- *withincode*
- *cflow*
- *cflowbelow*
- *preinitialization*
- *initialization*
- *staticinitialization*
- *if*
- *adviceexecution*
- *handler*

Circunstancias sin equivalencia con las primitivas

Por el contrario, algunas de las construcciones que provee SN en las circunstancias, no son posibles de poner en práctica de forma correspondiente por medio del uso de primitivas, por esta razón no existe la posibilidad de equiparar estos elementos. Las circunstancias que no poseen un semejante en AspectJ:

- *requires*
- *cannot*
- *mutually excluded*
- *created*

Sin embargo, si bien *created* no tiene una primitiva correspondiente, es posible definir algo similar mediante la primitiva *call* o *execution* e indicar en el patrón de firma el constructor de una clase. De modo que la instrucción:

```
noun Student:

circumstance: it System prints it when this is created.
```

Sería equivalente a:

```
pointcut corte():execution(* Student.new());
```

4.4. Ventajas y desventajas del uso de circunstancias y aspectos

Con el desarrollo de ejercicios y un caso de estudio, fue posible encontrar algunas de las ventajas y desventajas que presentan cada uno de los lenguajes, estas se muestran en la Tabla 4.7.

Tabla 4.7: Ventajas y desventajas de circunstancias y aspectos

	AspectJ	SN
Ventajas	▪ Permite una mejor separación de asuntos al mantener el aspecto separado de la clase Java. ▪ Reduce la dispersión de código. ▪ Su comportamiento se implementa fuera de la clase Java. ▪ Es posible intervenir métodos que se encuentren en diferentes clases, mientras que estos cumplan con el patrón de firma.	▪ Permite un mayor control de las entidades sobre las que actúa. ▪ Delimitan la especialización entre adjetivos. ▪ No es necesario realizar la exposición de contexto de manera explícita.
Desventajas	▪ Se presenta fragilidad, ya que dependen del patrón de firma del método en Java. ▪ Se requiere aprender sintaxis adicional.	▪ Las circunstancias se encuentran dispersas según el sustantivo donde sean necesarias. ▪ Es necesario aprender sintaxis adicional.

Capítulo 5

Conclusiones y recomendaciones

En el presente capítulo se presentarán las conclusiones y recomendaciones que se infirieron a partir del desarrollo de la presente tesis.

5.1. Conclusión

Los mecanismos de encapsulamiento de requerimientos no funcionales son necesarios de contemplar y especificar dentro del desarrollo de software de ambos paradigmas, que se presentaron en esta tesis, ya que permite tener un mejor control sobre los eventos que complementan al sistema, además de que estos mecanismos apoyan a tener una mejor documentación de lo que facilita el mantenimiento del sistema.

AspectJ permite encapsular los requerimientos no funcionales de forma que limita la dispersión de código,sin embargo, aún presenta problemas como la fragilidad, ya que depende del patrón de firma del método en Java, lo que puede tener como consecuencia, que si no especifica correctamente el patrón de firma completo, que el aviso intervenga en otro

método, lo que provoque que se altere el comportamiento regular del sistema.

Las circunstancias del lenguaje naturalístico SN, no se encapsulan fuera de la entidad que las requiere, lo cual provoca que se declaren dentro del mismo sustantivo que contiene el verbo, atributo o adjetivo al que se intervendrá; pero esta necesidad es la que permite mantener un mejor control sobre en "donde" actuara, ya que si algún elemento que se encuentre en otro sustantivo, mantiene alguna similitud no hay posibilidad de que la circunstancia lo afecte.

Aunque no todas las circunstancias cuentan con un equivalente en aspectos (lo cual no es necesario, ya que algunos aspectos se requieren para actuar sobre elementos con lo que SN no cuenta, tales como: inicializadores de instancia e inicializadores estáticos es capaz de intervenir de forma similar a sus elementos.

5.2. Recomendaciones

Con base en los resultados que se obtuvieron a partir del desarrollo de los ejercicios y el caso de estudio que se presentaron en el capítulo 4 de la presente tesis, se recomienda lo siguiente:

- Desarrollar más casos de estudio que permitan verificar el comportamiento de las circunstancias en diferentes escenarios.

- Realizar un análisis comparativo con otros lenguajes que contemplen un método de encapsulamiento de requerimientos no funcionales con el fin de encontrar cuales son

los escenarios que SN no contempla para solucionar.

- Documentar las circunstancias en un manual para lograr mantener un mejor control sobre cual es el comportamiento que tiene cada una de ellas.

Bibliografía

[1] A. Affleck and A. Krishna, "Supporting quantitative reasoning of non-functional requirements: A process-oriented approach," in *2012 International Conference on Software and System Process (ICSSP)*, pp. 88–92, June 2012.

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

[3] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming* (M. Akşit and S. Matsuoka, eds.), (Berlin, Heidelberg), pp. 220–242, Springer Berlin Heidelberg, 1997.

[5] S. Lohmeier, *Shaping statically resolved indirect anaphora for naturalistic programming: a transfer from cognitive linguistics to the Java programming language*. PhD thesis, 2011.

[6] P. P. Oscar, *Modelo naturalístico para la implementación de lenguajes de programación de propósito general.* PhD thesis, Instituto Tecnologíco de México, The address of the publisher, 7 1993. An optional note.

[7] orizaba.tecnm, “Instituto tecnologico de orizaba,” 2019.

[8] J. Eckhardt, A. Vogelsang, and D. M. Fernández, “Are "non-functionalrequirements really non-functional? an investigation of non-functional requirements in practice,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 832–842, May 2016.

[9] U. Hohenstein and P. Koka, “Reusable components for adding multi-tenancy to legacy applications,” in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 187–194, June 2017.

[10] A. Przybyłek, “An empirical study on the impact of aspectj on software evolvability,” *Empirical Software Engineering*, vol. 23, no. 4, 2018.

[11] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr, “Beyond aop: Toward naturalistic programming,” in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’03, (New York, NY, USA), pp. 198–207, ACM, 2003.

[12] O. Pulido-Prieto and U. Juárez-Martínez, “A survey of naturalistic programming technologies,” *ACM Comput. Surv.*, vol. 50, pp. 70:1–70:35, Sept. 2017.

[13] S. Gulwani and M. Marron, “Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 803–814, ACM, 2014.

[14] R. Knöll and M. Mezini, “Pegasus: first steps toward a naturalistic programming language,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 542–559, ACM, 2006.

[15] A. Cozzie, M. Finnicum, and S. T. King, “Macho: Programming with man pages.,” in *HotOS*, 2011.

[16] P. F. G. A. y. B. Mai, X. P., “A natural language programming approach for requirements-based security testing.,” p. 12, 2018.

[17] M. Landhaeusser and R. Hug, “Text understanding for programming in natural language: Control structures,” in *2015 IEEE/ACM 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pp. 7–12, May 2015.

[18] 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*, Onward! 2011, (New York, NY, USA), pp. 33–48, ACM, 2011.

[19] y. B. A. Kuhn, T., “Verifiable source code documentation in controlled natural language.,” in *Science of Computer Programming,*, pp. 2944–2949, 2014.

[20] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, “Program synthesis using natural language,” in *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, (New York, NY, USA), pp. 345–356, ACM, 2016.

[21] 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, pp. 89 – 119, 2018.

[22] M. S. Hsiao, "Automated program synthesis from object-oriented natural language for computer games," in *Controlled Natural Language-Proceedings of the Sixth International Workshop, CNL 2018, Maynooth, Co. Kildare, Ireland, August 27-28*, pp. 71–74, 2018.

[23] Eclipse.org, "The aspectj project," 2019.

[24] O. Pulido-Prieto and U. Juárez-Martínez, "A model for naturalistic programming with implementation," *Applied Sciences*, vol. 9, no. 18, p. 3936, 2019.

[25] Eclipse.org, "What is eclipse?," 2019.

[26] S. BANIASSAD, Elisa; CLARKE, "Finding aspects in requirements with theme/doc," *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, vol. 166, pp. 15 – 22, 2004.

[27] E. Baniassad and S. Clarke, "Theme: An approach for aspect-oriented analysis and design," in *Proceedings. 26th International Conference on Software Engineering*, pp. 158–167, IEEE, 2004.

[28] A. Guio and A. Catalina, "Propuesta didáctica no parametral para la enseñanza de la gramática,"

[29] M. MONTESSORI, "La mente absorbente del niño,"