



DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCIÓN 1.- TESIS

TRABAJO PROFESIONAL

**“MÉTODO PARA EL ANÁLISIS, DISEÑO Y CONSTRUCCIÓN DE
SOFTWARE BASADO EN EL PARADIGMA NATURALÍSTICO”**

**QUE PARA OBTENER EL GRADO DE:
DOCTOR EN CIENCIAS
DE LA INGENIERÍA**

PRESENTA:

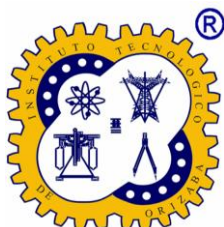
MIS. LIZBETH ALEJANDRA HERNÁNDEZ GONZÁLEZ

DIRECTOR DE TESIS:

DR. ULISES JUÁREZ MARTÍNEZ

CODIRECTOR DE TESIS:

DR. JEZREEL MEJÍA MIRANDA



ORIZABA, VERACRUZ, MÉXICO

MAYO 2024

Orizaba, Veracruz, 26/04/2024
Dependencia: División de Estudios de
Posgrado e Investigación
Asunto: Autorización de Impresión
OPCION: I

C. LIZBETH ALEJANDRA HERNÁNDEZ GONZÁLEZ
Candidato a Grado de Doctor en:
CIENCIAS DE LA INGENIERÍA
P R E S E N T E.-

De acuerdo con el Reglamento de Titulación vigente de los Centros e Institutos Tecnológicos Federales del Tecnológico Nacional de México, 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:

"MÉTODO PARA EL ANÁLISIS, DISEÑO Y CONSTRUCCIÓN DE SOFTWARE BASADO EN EL PARADIGMA NATURALÍSTICO."

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

ATENTAMENTE

Excelencia en Educación Tecnológica®
CIENCIA - TÉCNICA - CULTURA®



DRA. OFELIA LANDETA ESCAMILLA
JEFA DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN



Orizaba, Veracruz, **Marzo 22, de 2024.**
Asunto: **Revisión de trabajo escrito**

C. OFELIA LANDETA ESCAMILLA
JEFA DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN
P R E S E N T E.-

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

LIZBETH ALEJANDRA HERNÁNDEZ GONZÁLEZ

La cual lleva el título de:

**"MÉTODO PARA EL ANÁLISIS, DISEÑO Y CONSTRUCCIÓN DE SOFTWARE BASADO EN EL
PARADIGMA NATURALÍSTICO."**

Y concluyen que se acepta.

ATENTAMENTE
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

PRESIDENTE: DR. ULISES JUÁREZ MARTÍNEZ


FIRMA

SECRETARIO: DRA. LISBETH RODRÍGUEZ MAZAHUA


FIRMA

VOCAL: DR. ALFONSO AGUILAR LASSERRE


FIRMA

VOCAL: DR. EDUARDO ROLDÁN REYES


FIRMA

VOCAL: DRA. MARÍA KAREN CORTÉS VERDÍN


FIRMA

VOCAL
SUPLENTE: DR. JEZREEL MEJÍA MIRANDA


FIRMA

TA-09 -18

Declaración de originalidad y cesión de derechos

Orizaba, Veracruz, el día 02 del mes de Mayo del año 2024.

El(la) que suscribe

C HERNÁNDEZ GONZÁLEZ LIZBETH ALEJANDRA

Declaro que esta tesis, que tiene una extensión de 153 cuartillas, ha sido escrita por mí y constituye el registro escrito del trabajo de la tesis titulada

“MÉTODO PARA EL ANÁLISIS, DISEÑO Y CONSTRUCCIÓN DE SOFTWARE BASADO EN EL PARADIGMA NATURALÍSTICO”

del programa: **DOCTORADO EN CIENCIAS DE LA INGENIERÍA** bajo la asesoría y dirección del (la) **DR. ULISES JUÁREZ MARTÍNEZ Y DR. JEZREEL MEJÍA MIRANDA** y no ha sido sometida en ninguna otra institución previamente.

Todos los datos y las referencias a materiales ya publicados están debidamente identificados con su respectivo crédito e incluidos en las notas bibliográficas y en las citas que se destacan como tal y, en los casos que así lo requieran, cuento con las debidas autorizaciones de quienes poseen los derechos patrimoniales. Por lo tanto, me hago responsable de cualquier litigio o reclamación relacionada con derechos de propiedad intelectual, exonerando de toda responsabilidad al Tecnológico Nacional de México campus Orizaba.

También declaro que, al presentar esta tesis, cedo los derechos del trabajo al Tecnológico Nacional de México campus Orizaba para su difusión, con fines académicos y de investigación, bajo las regulaciones propias de la institución y que si existe algún acuerdo de confidencialidad de la información lo haré saber en forma escrita para que se omitan las secciones correspondientes.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y del director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección: depi_orizaba@tecnm.mx. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente de este.



Lizbeth Alejandra Hernández González

Agradecimientos

A tí mamá, que desde el cielo me sigues cuidando. Gracias por hacer de mí lo que soy. Te extraño

A mí esposo Román, gracias por apoyarme incondicionalmente en todos mis proyectos. Éste es un logro más de los dos.

A mis hijos, el motor de mi vida y mi más grande inspiración. Me dan y me enseñan tanto.

A mí director, el Dr. Ulises Juárez Martínez, por su paciencia y por compartir sus conocimientos y sabiduría conmigo. Por su calidad humana y profesional.

A mí codirector, el Dr. Jezreel Mejía Miranda, por su apoyo y paciencia, por compartir su tiempo, conocimientos y experiencia conmigo.

A la Dra. Karen Cortés Verdín, por todos los consejos que me dio antes y durante mis estudios de doctorado, gracias amiga por todo tu apoyo.

Al Dr. Alberto Aguilar Lasserre y al Dr. Eduardo Roldán Reyes, por su apoyo en la elaboración de este trabajo.

A mis amigos de la FEI, por que me inspiran cada día a ser mejor, son un gran ejemplo para muchos, gracias por su amistad. Trabajar junto a ustedes es un honor.

Gracias a Dios, por rodearme de todos ellos.

Índice general

Resumen	1
Abstract	2
Introducción	3
Planteamiento del problema	4
Justificación	5
Hipótesis	6
Objetivos	6
Contribución al conocimiento	7
1. Antecedentes	8
2. Estado del arte	14
2.1. Pegasus	14
2.2. Modelo conceptual naturalístico	15
2.3. SN	17
2.4. Cal-4700	18
2.5. Comparación de características entre lenguajes	18
2.6. Ejemplos de programas naturalísticos	20
2.6.1. Pegasus	21
2.6.2. Cal-4700	21
2.6.3. SN	21
2.7. Discusión	22
3. Aplicación de la metodología	25
3.1. Revisión del estado del arte	26
3.2. Revisión de los lenguajes naturalísticos encontrados	26
3.3. Prueba de los lenguajes naturalísticos encontrados	27
3.4. Análisis de las características de las metodologías actuales	28
3.5. Realización de la propuesta de Análisis y Diseño naturalístico	28
3.6. Determinación de los las características que tendrán los problemas para probar el método, para una futura evaluación	28
3.7. Evaluación del método de desarrollo	29
3.8. Realización de estancias	30
3.9. Realización de ajustes y generación de la versión final	30

4. Estándar ISO/IEC TR 29110	31
5. Principios de Análisis y Diseño de software	43
5.1. Análisis de Requerimientos	45
5.1.1. Casos de Uso 2.0 (CU 2.0)	46
5.1.2. Otras herramientas y notaciones de modelado	49
5.1.2.1. Diagrama de actividades	50
5.1.2.2. BPMN (<i>Business Process Model and Notation</i>)	50
5.2. Diseño de software	51
5.2.1. Diseño arquitectónico	52
5.2.2. Diseño detallado	54
5.2.2.1. Principios de diseño	55
5.2.2.2. Atributos de calidad (requisitos no funcionales)	56
5.2.2.3. Abstracción	56
5.2.2.4. Formas de representación del diseño	57
6. Método de desarrollo de software naturalístico	61
6.1. Análisis de Requerimientos	61
6.2. Diseño y Construcción	65
7. Pruebas de concepto del método	66
7.1. Sistema de Administración de Cursos (CMS)	66
7.1.1. Caso de Uso Inscribir estudiante	67
7.1.2. Caso de Uso Calificar estudiante	73
7.1.3. Caso de Uso Borrar inscripción de estudiante	78
7.1.4. Aspecto Registrar seguimiento	81
7.2. Aplicación del método en un contexto lúdico	83
7.2.1. Ideas atómicas (sustantivos)	84
7.2.2. Casos de uso y aspectos	85
7.2.3. Persistencia de datos	85
7.2.4. Estructura del proyecto en Cal-4700	86
7.2.5. Caso de Uso Terminar juego	87
7.3. Prueba de aplicación del método con alumnos de maestría	90
7.4. Discusión	92
Conclusiones	94
Productos Académicos	96
Apéndice A. Revisión Sistemática de la Literatura	100
A.1. Método de revisión	101
A.1.1. Estrategia de búsqueda	101
A.1.2. Fuentes de búsqueda	102
A.1.2.1. Herramientas	102
A.1.3. Criterios	103
A.1.3.1. Años de selección	104

A.1.3.2. Criterios de inclusión	104
A.1.3.3. Criterios de exclusión	105
A.1.4. Análisis de los datos	105
A.1.4.1. Extracción de datos	105
A.1.4.2. Síntesis de datos	105
A.2. Resultados	106
A.3. Discusión	112
Apéndice B. Tabla de datos extraídos	114
Apéndice C. Pruebas guiadas de Cal-4700	116
Apéndice D. Artefactos de modelado del caso de aplicación lúdico	119
D.1. Caso de Uso Seleccionar nivel	119
D.2. Caso de Uso Seleccionar imagen	122
D.3. Caso de Uso Iniciar juego	125
D.4. Caso de Uso Consultar puntuaciones	130
D.5. Aspecto Registrar datos de puntuaciones	135
D.6. El archivo interfaz	137
D.7. El archivo puzzle	139
Referencias bibliográficas	140

Índice de figuras

1.1. Jerarquía de clases de cuentas bancarias	9
2.1. Representación naturalística de una lotería primitiva en Cal-4700.	22
2.2. Simulador de Lotería Naturalística en SN.	22
2.3. Uso de gramáticas embebidas en SN.	23
4.1. Procesos del estándar 29110 Perfil Básico	33
4.2. Diagrama del proceso Gestión del Proyecto	34
4.6. Productos de trabajo identificados por colores	35
4.3. Diagrama del proceso de Implementación de Software	37
4.4. Tareas del Proceso Gestión del Proyecto	38
4.5. Tareas del Proceso Implementación del Software	39
4.7. IS - Análisis de Requisitos	40
4.8. IS - Arquitectura y Diseño detallado	41
4.9. IS - Construcción de Software	42
5.1. Modelo de ciclo de vida en cascada.	44
5.2. Diagrama de CU de un sistema de biblioteca.	48
5.3. Planeación de iteraciones utilizando el mapa de historias.	49
5.4. Definición simple BDD con Gherkin.	49
5.5. Diagrama de actividades préstamo de software.	50
5.6. Proceso de mantenimiento de stock de productos.	51
5.7. Modelo del proceso de Diseño (Budgen, 2003).	52
5.8. Diagrama de objetos distribuidos (Taylor et al., 2009).	53
5.9. Instrucciones de ensamblaje en el mundo real (proporcionada por Gerry Rzeppa).	57
5.10. DFD Verificar reporte de gastos (ISO/IEC/IEEE, 2017).	58
5.11. Descripción de transición de estados del editor vi de Unix (Budgen, 2003).	58
5.12. DER de dos clases de entidad reconocibles (Budgen, 2003).	59
5.13. DEJ que describe la operación de una máquina de té (Budgen, 2003).	59
6.1. Método de Desarrollo Naturalístico.	62
6.2. Notación de <i>idea</i>	63
6.3. Ejemplo de una idea en Pegasus.	63
7.1. Diagrama de CU Registro de cursos.	67
7.2. Prototipo CU Inscribir	68

7.3. Ideas atómicas identificadas.	69
7.4. Diagrama Entidad-Relación	70
7.5. Diagramas de análisis(a) y diseño(b) del CU Inscribir.	70
7.6. Salida CU Inscribir	74
7.7. Prototipos CU Calificar.	76
7.8. Diagramas de análisis(a) y diseño(b) del CU Calificar.	77
7.9. Prototipo CU Borrar inscripción de estudiante	79
7.10. Diagramas de análisis(a) y diseño(b) del CU Borrar inscripción de estudiante.	80
7.11. Diagramas de análisis(a) y diseño(b) del Aspecto Seguimiento.	82
7.12. Idea atómica jugador.	84
7.13. Idea atómica nivel.	84
7.14. Idea atómica puntuación.	85
7.15. Diagrama de casos de uso de la aplicación lúdica.	86
7.16. Esquema de persistencia de datos del sistema.	86
7.17. Prototipo del CU Terminar juego.	88
7.18. DFD del CU Terminar juego.	89
7.19. Diagrama de estructura del CU Terminar juego.	89
7.20. Interfaz de usuario del CU Terminar juego.	90
7.21. Artículo JCR publicado	96
7.22. Artículo Evolution of NP: a Need	98
7.23. Artículo Caso de estudio lúdico utilizando el PN	99
 A.2. Número de publicaciones por año	 108
A.1. Flujo de información a través de las diferentes fases de la revisión sistemática (Moher et al., 2009).	109
A.3. Número de publicaciones por campo de investigación	110
A.4. Mapa bibliométrico de términos relacionados.	111
A.5. Conjunto de clústers de términos relacionados	111
 D.1. Prototipo del CU Seleccionar nivel.	 120
D.2. DFD del CU Seleccionar nivel.	121
D.3. Diagrama de estructura del CU Seleccionar nivel.	121
D.4. Interfaz de usuario del CU Seleccionar nivel.	122
D.5. Prototipo del CU Seleccionar imagen.	122
D.6. DFD del CU Seleccionar imagen.	124
D.7. Diagrama de estructura del CU Seleccionar imagen.	124
D.8. Interfaz de usuario del CU Seleccionar imagen - nivel fácil.	125
D.9. Prototipo solicitar nombre de jugador.	127
D.10. Prototipo inicia juego.	128
D.11. DFD del CU Iniciar juego.	128
D.12. Diagrama de estructura del CU Iniciar juego.	129
D.13. Interfaz de usuario para solicitar nombre del jugador.	130
D.14. Interfaz de usuario del CU Iniciar juego en el nivel fácil.	131
D.15. Prototipo para el CU Consultar puntuaciones.	132
D.16. DFD del CU Consultar puntuaciones.	132

D.17.Diagrama de estructura del CU Consultar puntuaciones.	133
D.18.Interfaz para la consulta de puntuaciones a través del nivel de dificultad. . .	134
D.19.Interfaz de usuario que muestra las listas de puntuaciones del nivel fácil. . .	134
D.20.DFD del aspecto Registrar datos de puntuaciones.	136
D.21.Diagrama de estructura del aspecto Registrar datos de puntuaciones.	137

Índice de tablas

1.1. Ejemplos representativos de PLN.	12
2.1. Comparación de Lenguajes Naturalísticos.	19
4.1. Abreviaturas de los Productos de trabajo utilizados.	35
5.1. Historias de CU y porciones del sistema de biblioteca.	47
5.2. Correspondencia entre elementos de CU y elementos del mapa de historias. .	48
7.1. Especificación del CU Inscribir.	68
7.2. Tabulación de ideas CU Inscribir.	69
7.3. Especificación del CU Calificar.	74
7.4. Tabulación de ideas CU Calificar.	75
7.5. Especificación del CU Borrar inscripción de estudiante.	78
7.6. Tabulación de ideas CU Borrar inscripción de estudiante.	79
7.7. Especificación del aspecto Registrar seguimiento.	81
7.8. Tabulación de ideas aspecto Registrar seguimiento.	81
7.9. Descripción del CU Terminar juego.	88
7.10. Tabulación de ideas del CU Terminar juego.	89
7.11. Encuesta de evaluación del método con resultados	93
A.1. Número de resultados por término de búsqueda.	106
A.2. Número de Publicaciones por fuente	107
B.1. Publicaciones exclusivamente relacionadas con el paradigma naturalístico. . .	114
C.1. Encuesta de evaluación Cal-4700 con resultados.	118
D.1. Descripción del CU Seleccionar nivel.	119
D.2. Tabulación de ideas del CU Seleccionar nivel.	120
D.3. Descripción de CU Seleccionar imagen.	123
D.4. Tabulación de ideas del CU Seleccionar imagen.	123
D.5. Descripción del CU Iniciar juego.	126
D.6. Tabulación de ideas del CU Iniciar juego.	127
D.8. Tabulación de ideas del CU Consultar puntuaciones.	131
D.7. Descripción del CU Consultar puntuaciones.	133
D.9. Descripción del aspecto Registrar datos de puntuaciones.	136

D.10.Tabulación de ideas del aspecto Registrar datos de puntuaciones.	136
---	-----

Resumen

En este trabajo se plantea la necesidad de elaborar un método de desarrollo de software basado en el paradigma naturalístico, éste paradigma busca incorporar elementos del lenguaje natural en la construcción de lenguajes de programación de propósito general, con la finalidad de reducir la brecha entre el dominio del problema y el dominio de la solución. La intención es lograr un mayor acercamiento al usuario a través de lenguajes más expresivos. El lenguaje natural es inherentemente ambiguo, sin embargo su riqueza permite expresar con mayor claridad las ideas.

En la actualidad se cuenta con tres lenguajes de programación naturalística de propósito general: Pegasus, SN y Cal-4700, los cuales cumplen con la filosofía del paradigma naturalístico, ser reflexivos (capaces de hacer referencias a elementos descritos previamente), ser de propósito general y generar código directamente ejecutable.

El objetivo del trabajo es proponer un método de desarrollo, del cual no se encuentra reporte en la literatura, utilizando los lenguajes de programación naturalística de propósito general. Como primer paso, los lenguajes se compararon con base en un modelo conceptual existente, además de la ejecución de algunos programas ejemplo, con la finalidad de comprender su filosofía, intención y sintaxis. Como segundo paso, se analizaron los métodos de desarrollo actuales con base en los principios que tienen en común, desde el punto de vista del proceso de desarrollo de software, específicamente para las fases de análisis y diseño de software. Lo anterior con la finalidad de identificar los elementos mínimos necesarios que todo método de desarrollo de software debe tener, además de identificar algunos artefactos que pudieran incorporarse al paradigma naturalístico.

Con base en lo anterior, se propone un método de desarrollo naturalístico que incluye las fases de Análisis, Diseño y Construcción de Software, además de dos Pruebas de concepto como ejemplos de su aplicación: uno elaborado por la autora de este trabajo y otro elaborado por un estudiante de la Maestría en Sistemas Computacionales del ITO, como parte de su trabajo de tesis. Se presenta avance de la documentación del método, con base en el estándar ISO/IEC 29110, el cual es aplicable a cualquier empresa de desarrollo de software. Esto permitirá verificar la trazabilidad de los requerimientos del problema mientras se aplica el método.

Palabras clave: Programación Naturalística, paradigmas de programación, Métodos de Ingeniería de Software, Procesos de Ingeniería de Software, Expresividad de lenguajes de programación, Método de Programación Naturalística

Abstract

This work proposes the need to elaborate a software development method based on the naturalistic paradigm, this paradigm seeks to incorporate elements of natural language in the construction of general purpose programming languages, in order to reduce the gap between the domain of the problem and the domain of the solution. The intention is to achieve a closer approach to the user through more expressive languages. Natural language is inherently ambiguous, yet its richness allows ideas to be expressed more clearly.

At the present time there are three general purpose naturalistic programming languages: Pegasus, SN and Cal-4700, which comply with the philosophy of the naturalistic paradigm, being reflective (able to make references to previously described elements), being general purpose and generating directly executable code.

The objective of the work is to propose a development method, of which there is no report in the literature, using general purpose naturalistic programming languages. As a first step, the languages were compared based on an existing conceptual model, in addition to the execution of some example programs, in order to understand their philosophy, intent and syntax. As a second step, the current development methods were analyzed based on the principles they have in common, from the point of view of the software development process, specifically for the software analysis and design phases. The above with the purpose of identifying the minimum necessary elements that every software development method must have, besides identifying some artifacts that can be incorporated to the naturalistic paradigm.

In accordance with the above, a naturalistic development method that includes the Analysis, Design and Construction phases is proposed, as well as two proof concept as examples of its application: one of them was elaborated by the author of this document and the other one was elaborated by a master's student as part of his thesis work. Progress of the documentation of the method is presented, according to the ISO/IEC 29110 standard, which is applicable to any software development company, to verify the traceability of the requirements by following the method.

Keywords: Naturalistic programming, Programming paradigms, Software Engineering methods, Software Engineering processes, Programming languages expressiveness, Naturalistic Programming method

Introducción

Dentro de la Ingeniería de Software se encuentran múltiples y diversas propuestas para desarrollar software de calidad, métodos, modelos de procesos, técnicas, estándares, entre otros. Sin embargo, la falta de comprensión en las abstracciones utilizadas durante las diferentes fases del proceso de desarrollo, ocasionan que la información se pierda o deforme en el paso de una fase a otra (ISO/IEC, 2016). Muchas veces los requerimientos del usuario se deben adaptar a los formalismos de algún paradigma de programación, en lugar de transmitirse de manera más transparente. Esto ocasiona que siga existiendo una brecha entre el dominio del problema y el de la solución, debido a la falta de expresividad de los paradigmas de programación actuales, además de insatisfacción del cliente así como altos costos de retrabajo (Pulido Prieto & Juárez Martínez, 2020).

La falta de expresividad¹, es decir, la falta de claridad para establecer o transmitir correctamente las ideas y permitir un acercamiento al dominio del experto, ha sido un problema latente en el desarrollo de software. Un lenguaje expresivo sería aquel que tiene la capacidad de comunicar de manera transparente, lo que el usuario desea a través de los requerimientos. El lenguaje debería estar lo más cerca posible al dominio del experto, ya que muchas veces lo que el analista entiende o establece en un modelo, no concuerda con lo que el experto del dominio necesita. En el paso de los requerimientos a la codificación es posible perder o deformar la información. Los lenguajes de programación muchas veces obligan a establecer ideas que estén alejadas de la mejor solución, ya que se hace lo que permite el lenguaje.

El Paradigma Naturalístico propone incorporar elementos del lenguaje natural en la construcción de lenguajes de programación con el objetivo de lograr una mayor expresividad al programar, y por lo tanto, de disminuir la brecha entre dominios. La intención es contar con lenguajes de propósito general, los cuales empleen elementos del lenguaje natural como la reflexividad (capacidad de hacer referencia a elementos mencionados previamente) y que generen código directamente ejecutable, sin forzar otro tipo de abstracciones. Con el uso de estos lenguajes la construcción de software sería más natural, se enfocaría en la intención, más que en aspectos técnicos, además se reduciría la carga mental del programador.

El paradigma ha mostrado resultados en la solución de algunos problemas. Actualmente se cuenta con un modelo naturalístico (Pulido-Prieto & Juárez-Martínez, 2019) y tres lenguajes naturalísticos que cumplen con la filosofía del paradigma: Pegasus (Knöll & Mezini, 2006), SN (Pulido Prieto & Juárez Martínez, 2020) y Cal-4700 (Rzeppa & Rzeppa, 2019b). Aunque este nuevo paradigma cuenta con lenguajes de programación de propósito general, actualmente no se reporta evidencia de un método aplicable al proceso de desarrollo de software, el cual

¹"The quality of expressing somebody's thoughts and feelings", The Oxford English Dictionary

permita verificar los beneficios que promete.

El propósito de este trabajo es proponer un método de desarrollo utilizando los lenguajes existentes (Pegasus a nivel teórico ya que no se cuenta con compilador disponible), de manera que se promueva el uso del paradigma en el desarrollo de software.

Los capítulos están organizados de la siguiente manera: en el Capítulo 1 se presentan los antecedentes que justifican la motivación del trabajo. El Capítulo 2 presenta el Estado del Arte del paradigma de acuerdo a lo obtenido en la Revisión Sistemática de la Literatura (incluida en el Apéndice A), el Capítulo 3 presenta la aplicación de la metodología. El Capítulo 4 describe el estándar ISO/IEC 29110 y su relevancia para documentar el método resultante. El Capítulo 5 presenta los fundamentos de Análisis y Diseño que sirvieron como base para crear el método. El Capítulo 6 describe el método naturalístico propuesto, el Capítulo 7 presenta dos ejemplos como prueba de concepto del método, y finalmente, se presentan las conclusiones.

Planteamiento del problema

La riqueza del lenguaje natural permite expresar ideas de manera clara, en un lenguaje de programación tradicional, la expresividad se ve comprometida dado que la traducción de requerimientos se hace con base en una gramática limitada de alto nivel. A mayor expresividad, mayor claridad y entendimiento habrá. Logrando esto, se obtendría una mayor concordancia con los requerimientos y se facilitaría el mantenimiento de software, minimizando los costos de retrabajo. De acuerdo a Pulido-Prieto (2019) los paradigmas de programación actuales limitan la expresividad del lenguaje, dando pie a la necesidad de explorar descripciones del lenguaje natural, el cual es ambiguo, pero implícitamente expresivo.

El Diccionario de Inglés de Cambridge define *Naturalistic* como: “*Similar to what exists in nature*”, por lo que un lenguaje de programación naturalístico sería aquel que se asemeje al lenguaje natural. De acuerdo con Pulido-Prieto & Juárez-Martínez (2017), el lenguaje natural debe restringirse a una representación formal que pueda procesarse en la computadora, pero que también tenga el nivel de abstracción más alto posible, para que sea entendida por cualquier persona involucrada.

De acuerdo al estudio reportado en (Pulido-Prieto & Juárez-Martínez, 2017), actualmente se cuenta con lenguajes naturalísticos específicos del dominio, viéndose limitada su aplicación a diferentes contextos. Al respecto de lenguajes naturalísticos de propósito general, el primer lenguaje reportado fue Pegasus (Knöll & Mezini, 2006), basado en el concepto de representación de “ideas”. Pegasus permite traducir entre diferentes idiomas y a diferentes lenguajes de programación (Knöll, 2019), logrando una mayor independencia del proceso de traducción.

En (Pulido-Prieto & Juárez-Martínez, 2019) se propone un modelo para la programación naturalística, el cual indica los elementos mínimos necesarios para la construcción de lenguajes naturalísticos de propósito general, a la vez que se presenta un prototipo de lenguaje naturalístico de propósito general basado en el modelo propuesto. El lenguaje, llamado SN (por sus siglas en latín, *Sicut Naturali*, Como Natural), acepta declaraciones de un subconjunto del idioma inglés y las traduce a código ejecutable que la computadora puede procesar.

En la literatura gris se encuentra Cal-4700 ([Rzeppa & Rzeppa, 2019a](#)), un lenguaje de programación en inglés plano que incorpora frases imperativas, procedimientos y funciones en lenguaje natural para escribir sin necesidad de traducciones intermedias, como si se estuviera escribiendo un pseudocódigo.

Actualmente no se reporta evidencia de un método con enfoque naturalístico, aplicable al proceso de desarrollo de software, por lo que es necesario contar con uno, con la formalidad que dicta la disciplina de Ingeniería de Software. El siguiente paso entonces es probar la aplicación de lenguajes naturalísticos en uno o más problemas de desarrollo de software, siguiendo un enfoque metodológico. La intención es proponer cómo incluir las fases del proceso de desarrollo de software: Análisis, Diseño y Construcción, como primeros pasos, utilizando el paradigma naturalístico.

Justificación

De acuerdo a los autores de Pegasus, la desventaja con otros paradigmas es que el programador ajusta la idea a un lenguaje de programación específico, cuando lo natural debería ser escribir los programas en el lenguaje nativo del programador, sin necesidad de pensar en tecnicismos o abstracciones de otro idioma o lenguaje de programación, de tal manera que diferentes programadores puedan leer el mismo programa en su idioma particular. Éste concepto de “idea” se formaliza en ([Knöll & Mezini, 2006](#)) para hacerla accesible a la computadora, en donde los autores definen el concepto *Naturalistic Programming* como “*writing computer programs with the help of natural language*” ([Knöll & Mezini, 2006](#)). Con Pegasus, el proceso de traducción entre idiomas y lenguajes de programación de alto nivel queda a cargo de sus traductores ([Knöll, 2019](#)). El objetivo de lo anterior es que el programador exprese las ideas en la manera en la que las piensa. Las reflexiones anteriores dan lugar a definir las siguientes ventajas, que [Knöll \(2019\)](#) establece claramente al respecto del paradigma naturalístico:

- Reducción de la carga mental del programador, ya que no se preocuparía por un paradigma en particular o por aspectos técnicos.
- Mejora en la documentación, ya que el código se autodocumenta. Si es necesario que alguien entienda en su idioma, se traduce con Pegasus, con esto se reduce la ambigüedad y malos entendidos.
- Los programas se escribirían una sola vez, no habría necesidad de reescribirlos en diferentes lenguajes de programación. Por ejemplo, los algoritmos de búsqueda.
- Reducción en la curva de aprendizaje de la tecnología de desarrollo. Con la tecnología actual, se tienen demasiadas variantes y es muy complejo dominarlas todas.
- El programador se enfocaría en resolver el problema, más que en abstracciones o aspectos técnicos.
- Se facilitaría la reutilización de software.
- Si existen dudas, la propuesta es que Pegasus pregunte para resolverlas.

Como todo paradigma, la evolución de la programación naturalística lleva tiempo, generalmente van surgiendo lenguajes de programación alrededor, para posteriormente concretar propuestas de análisis y diseño, como fue el caso de la POO. En este caso, a pesar de que ya se cuenta con lenguajes naturalísticos de propósito general, no se encuentra reporte en la literatura acerca de métodos aplicables a este paradigma.

Es necesario contar con un método aplicable al proceso de desarrollo de software naturalístico, de manera que el resultado sea un conjunto de oraciones naturalísticas que eviten desviar la atención en otro tipo de abstracciones. En primera instancia, se necesitan propuestas para las fases de Análisis de Requerimientos y Diseño, pues éstas guían el resto del proceso de desarrollo. En este caso, también se cuenta con lenguajes naturalísticos de propósito general, por lo que los artefactos generados se deben poder trasladar hacia alguno de estos lenguajes. Esto permitirá verificar o bien rectificar las ventajas del paradigma, definir campos de aplicación y motivar la mejora de los lenguajes existentes y métodos propuestos, es decir, que el paradigma evolucione. No se trata de eliminar los conceptos anteriores, sino de contar con propuestas específicas al enfoque naturalístico con pasos bien definidos.

Es importante evitar caer en el impulso de considerar innecesarias las fases del proceso de desarrollo de software, como todo proyecto, un proyecto de software debe ser analizado, diseñado y probado. Si el analista comete un error en el levantamiento y especificación de requerimientos, la representación de código será incorrecta.

Según Roger Pressman, se puede entender a la Ingeniería de Software como una disciplina que tiene bases ingenieriles para proveer un conjunto de métodos, herramientas y procesos definidos para construir software de calidad. Una definición más formal se propone en ([ISO/IEC/IEEE, 2017](#)):

Se trata de “la aplicación de un método cuantificable, disciplinado y sistemático para el desarrollo, operación y mantenimiento de software, esto es, la aplicación de la Ingeniería de Software”.

Por lo tanto, para lograr un producto de software de calidad, no es suficiente contar con conocimientos tecnológicos actuales (lenguajes naturalísticos en este caso), es necesario aplicar metodologías, métodos, técnicas de administración, verificación y validación, entre otros; que permitan seguir un proceso de calidad, lo que asegurará en gran medida la calidad del producto final. Como primer paso, este proyecto propondrá un método de desarrollo basado en el paradigma naturalístico, para promover su uso y evolución, de esta manera se comprobarán o bien rectificarán las ventajas del mismo.

Hipótesis

Es posible proponer un método utilizando el paradigma naturalístico, que permita el desarrollo de software de una forma más clara y comprensible para los interesados.

Objetivos

A continuación, se presentan el objetivo general y los objetivos específicos del trabajo.

Objetivo general

Proponer un método para el Análisis, Diseño y Construcción de software, basado en el paradigma naturalístico, utilizando como base lenguajes de programación naturalística de propósito general.

Objetivos específicos

- Dar a conocer los lenguajes naturalísticos de propósito general existentes, con base en el modelo propuesto por [Pulido-Prieto & Juárez-Martínez \(2019\)](#), para verificar su adaptación al paradigma y promover su uso en el desarrollo de software.
- Analizar las características de las propuestas metodológicas actuales, con base en los principios que tienen en común, mediante la identificación de elementos naturalísticos que permitan incluirse en la propuesta.
- Realizar la propuesta de Análisis y Diseño naturalístico, mediante la inclusión de pasos concretos para su aplicación al proceso de desarrollo de software.
- Evaluar el método naturalístico mediante su aplicación en casos prácticos.

Contribución al conocimiento

El trabajo aportará concretamente en contar con un método de desarrollo de software basado en el paradigma naturalístico, que siga los principios de Análisis y Diseño de Software, y que utilice lenguajes de programación naturalísticos de propósito general. Otros aportes serán:

- La identificación de contextos o áreas en las que el enfoque puede ser aplicado.
- Contar con una propuesta de análisis y diseño naturalístico.
- Contar con elementos que permitan mejorar el prototipo del lenguaje naturalístico SN (producto del ITO), el trabajo en progreso Cal-4700, o bien para diseñar nuevos y mejores lenguajes.

Capítulo 1

Antecedentes

Entre los paradigmas de programación de propósito general más usados, destaca el Paradigma Orientado a Objetos (POO), el cual ha encabezado la manera en la que se desarrolla el software desde hace más de 20 años. El POO trata de representar el problema como lo concibe el ser humano, identificando clases y objetos, con sus propiedades y acciones, así como la interacción entre ellos. Con el POO se ha avanzado hacia una descomposición más natural que la que se hacía utilizando solamente bases algorítmicas. Esta evolución se fue dando de forma gradual, como todo nuevo paradigma. El POO es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son todos miembros de una jerarquía de clases unidas por relaciones de herencia (Booch et al., 2007).

De acuerdo a (Booch et al., 2007) el modelo de objetos abarca los principios de abstracción, encapsulamiento, modularidad, jerarquía, tipificación, concurrencia y persistencia. Lo importante del modelo de objetos reside en que estos elementos se unen de forma sinérgica. El desarrollo de lenguajes de programación ha complementado estos avances, contando en la actualidad con un gran número de opciones tecnológicas para su implementación.

La expresividad que provee el modelo de objetos, al representar el problema en términos de clases y objetos, con sus propiedades, operaciones y relaciones entre sí, se puede observar en la Figura 1.1, en donde se presenta una jerarquía de clases de una aplicación que administra cuentas bancarias, utilizando el Lenguaje Unificado de Modelado¹. Cada clase incluye sus atributos y métodos aplicados a ellas, los cuales se reflejarán en las instancias correspondientes (objetos). De acuerdo al ejemplo, existen dos tipos de cuentas, de **Cheques** y de **Inversión**. La cuenta de cheques permite al cliente acceder diariamente a sus fondos, además de disponer de una chequera; la cuenta de inversión permite acceder a los fondos en determinadas fechas, dependiendo del plazo seleccionado, el cual determina la tasa de interés a ganar. Ambas cuentas heredan la definición y comportamiento (métodos) de la superclase **Cuenta**.

Aún con las bondades que el POO brinda, Kiczales et al. (1997) encontraron muchos problemas de programación para los cuales ni las técnicas de programación orientadas a objetos ni los procedimientos, eran suficientes para capturar claramente algunas de las decisiones de diseño importantes que un programa debía implementar. La meta de la POA es dar soporte al programador en separar claramente los componentes de los aspectos

¹<https://www.uml.org/>

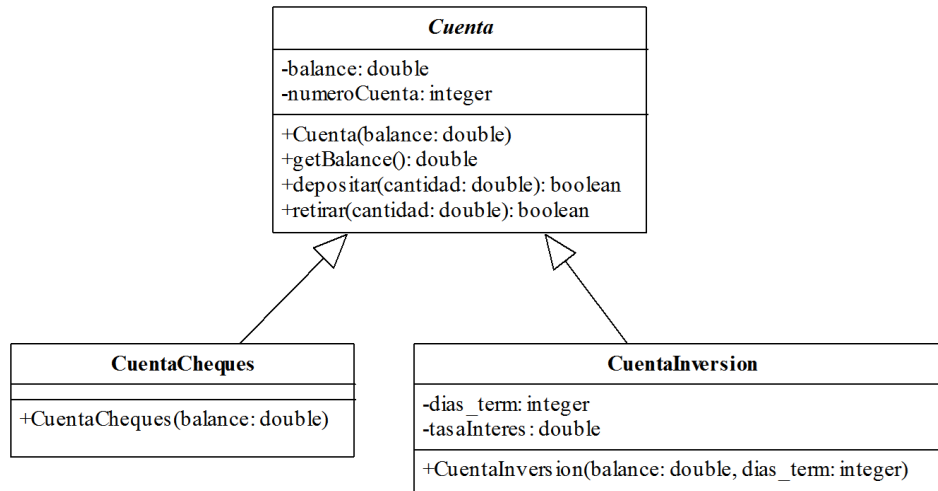


Figura 1.1: Jerarquía de clases de cuentas bancarias.

(componentes uno de otro, aspectos uno de otro y componentes de aspectos).

La mayoría de los lenguajes que Kiczales y Videira llaman generales descomponen por funciones, pero existen algunos aspectos que son transversales a esta descomposición funcional, p. e. el manejo de excepciones, optimización de memoria, disminución del tráfico de la red, entre otros, es decir, cierto tipo de requerimientos no funcionales. La POA vino a dar una oportunidad de crear programas más legibles y mantenibles, al igual que eficientes, separando los componentes de los aspectos ortogonales de un sistema (Kiczales et al., 1997).

De manera ilustrativa más no exhaustiva, en el Listado 1.1 se muestra la implementación en Java de la clase **Cuenta** de la Figura 1.1, se incluye sólo la definición de atributos así como el método **retirar** (línea 5), con la intención de enfocar el ejemplo en un aspecto que validará que la cantidad a retirar sea mayor que 100. Supóngase que no se puede modificar la clase **Cuenta** para manipular la validación o excepción correspondiente. El Listado 1.2 muestra el aspecto **AspectoCuenta**, el cual va a intervenir la llamada del método **retirar** (línea 3) y exponer para su uso el argumento **cantidad** (línea 4), el cual representa la cantidad a retirar. El aviso **around** (línea 6) significa que el método será intervenido durante su llamada (no antes ni después); si la cantidad a retirar es menor que 100, se dará aviso al usuario y el método no se ejecutará, de lo contrario, se procederá con la ejecución del método original (línea 10).

Listado de código 1.1: Implementación de la clase *Cuenta* en Java.

```

1 public abstract class Cuenta {
2     private double balance;
3     private int numeroCuenta;
4
5     public void retirar(double cantidad) {
6         setBalance(balance - cantidad);
7     }
8 }
  
```

Listado de código 1.2: Aspecto en Java para una aplicación bancaria.

```
1 public aspect AspectoCuenta {  
2   @pointcut retirarCantidad(double cant):  
3     call (void Cuenta.retirar(double)) && args(cant);  
4  
5   void around(double cant) : retirarCantidad(cant) {  
6     if (cant < 100) {  
7       System.out.println( "La cantidad no es suficiente " );  
8     } else {  
9       proceed(cant);  
10    }  
11  }  
12 }
```

De acuerdo a [Lopes et al. \(2003\)](#), el éxito del POA se debe, en parte, al reconocimiento en la industria del software de que los problemas de expresividad en los lenguajes de programación son serios; existen lenguajes de aspectos cuyo enfoque hace que el código sea más expresivo, más legible y más confiable, además de abordar la necesidad de mejorar la confiabilidad y reducir el tiempo de desarrollo. Los autores mencionan que una de las principales características de los lenguajes naturales, que los distingue de la mayoría de los lenguajes formales, es el uso diverso de relaciones anafóricas², ésta característica se soporta en los lenguajes del POA, ya que ofrecen una arquitectura reflexiva. La combinación de elementos lingüísticos identificados, además de la capacidad reflexiva de los lenguajes orientados a aspectos, dio pie a considerar al lenguaje natural como alternativa de programación, proponiendo el paradigma naturalístico, el cual toma en cuenta el rol de los lenguajes naturales en el diseño de lenguajes de programación, con el objetivo de lograr mayor expresividad y mayor acercamiento al dominio del experto, con menor distorsión de la información. Aunque el lenguaje natural es inherentemente ambiguo, su riqueza permite lograr una mayor expresividad.

El interés por la capacidad expresiva de los lenguajes de programación existe desde tiempo atrás, de acuerdo a [Felleisen \(1991\)](#) desde la década de los 70. En su trabajo, el autor desarrolló una notación formal para comparar la expresividad de diferentes lenguajes de programación, dada la gran cantidad de afirmaciones informales que se encontraban en la literatura. En el mismo trabajo menciona a ([Paterson & Hewitt, 1970](#); [Chandra & Manna, 1975](#)) pioneros en el estudio de la expresividad, mediante la propuesta de una esquematología comparativa, además de ([Hoare, 1989](#)), cuyo enfoque se relaciona con la formalización y comparación de la expresividad. Otras propuestas han tratado de dar respuesta a ello desde diferentes enfoques, el más reportado en la literatura es el uso del lenguaje natural como herramienta de programación, cuyo interés también data de mucho tiempo atrás. De acuerdo a [Lieberman & Liu \(2006\)](#), esto ha sido un viejo sueño en las ciencias de la computación. Sin embargo, dada la complejidad del lenguaje natural, el objetivo ha sido muy difícil de lograr en su totalidad.

La mayoría de las propuestas publicadas que intentan dar respuesta a la inquietud de programar en lenguaje natural se reportan como:

- Creación de Lenguajes Específicos de Dominio (LED) ([Mernik et al., 2005](#)) para lograr

²The use of words that refer to and replace other words used earlier in a text (TheCambridge English Dictionary).

la expresividad de acuerdo con un determinado contexto. En (Felleisen et al., 2018), los autores proponen el uso del lenguaje Racket³, el cual permite crear LEDs embebidos en un lenguaje base, con el objetivo de programar en el lenguaje específico creado, el cual está estrechamente relacionado con el dominio del usuario.

- Programación Orientada al Lenguaje, de acuerdo a Felleisen et al. (2018), en un mundo ideal los desarrolladores de software analizarían cada problema en el idioma de su dominio y luego articularían soluciones en términos coincidentes, por lo que ellos proponen la creación de lenguajes específicos de acuerdo a las necesidades.

Aunque estos lenguajes son expresivos, están hechos para un dominio específico. Los lenguajes de propósito general se aplican a todos los dominios con relativa eficiencia, pero carecen de expresividad (Pulido-Prieto & Juárez-Martínez, 2017). Esto representa un dilema.

- Propuestas que estudian lenguajes de paradigmas que por su naturaleza matemática son inherentemente más expresivos, como es el caso de la programación lógica temporal reportado en (Baudinet, 1989).
- Trabajos de Procesamiento de Lenguaje Natural (PLN), rama de la Inteligencia Artificial (IA) que ha representado de manera sobresaliente el estudio del lenguaje humano. Sus inicios se remontan a la década de los 60's y ha madurado significativamente a lo largo de los años (Rajput, 2020). El PLN incluye una variedad de técnicas computacionales motivadas por la teoría del análisis automático y la representación del lenguaje humano. Las dificultades y la complejidad de la representación son grandes, inclusive para llegar a un acuerdo entre diferentes personas sobre la comprensión de un determinado lenguaje (Fu, 2019). El PLN permite que una máquina procese un lenguaje humano natural y lo traduzca a un formato que la máquina entienda (Rajput, 2020).

Algunas aplicaciones de PLN son: resumen de textos (Awasthi et al., 2021), traducción automática de idiomas (Lample et al., 2018), análisis de textos (Guetterman et al., 2018), análisis de sentimientos (Ni et al., 2021), identificación de discursos de odio (Parihar et al., 2021), sistemas de apoyo a la toma de decisiones clínicas (Vasilakes et al., 2021) y detección de problemas de salud mental (Rajput, 2020).

En cuanto a la programación de computadoras por medio del lenguaje natural, muchas de las propuestas se enfocan en áreas específicas como la robótica, considerando la Programación por parte de Usuarios Finales. Un ejemplo se reporta en (Schlegel et al., 2019).

En la literatura se encuentran innumerables aplicaciones de propósito específico relacionadas con el desarrollo de software, algunos ejemplos representativos se reportan en la Tabla 1.1.

³<https://racket-lang.org/>

Tabla 1.1: Ejemplos representativos de PLN.

Trabajo	Aportación/propuesta
Slack as a UML modeling tool with Natural Language ⁴	Vinculación de Slack (herramienta de trabajo colaborativo) con una plataforma UML y un chat bot. El robot entiende instrucciones para realizar diagramas de UML.
From UML and OCL to natural language ⁵	Desarrollo de un método que parafrasea los diagramas de clase UML/OCL para generar una descripción de los contenidos y las reglas del modelo en lenguaje natural. Los interesados pueden utilizar esta descripción para validar los modelos antes de continuar con el proceso de desarrollo.
From Natural Language to Programming Language (Liu & Wu, 2018)	El marco de trabajo PiE, es capaz de sintetizar programas en el lenguaje de programación LOGO, para dibujar gráficos a partir de la conversación en inglés entre el programa Eliza y los usuarios.
A Natural Language Programming Approach for Requirements-based Security Testing (Mai et al., 2018)	Los autores se centran en generar casos de prueba a partir de requerimientos negativos (comportamiento no deseable). Identifican palabras clave para soportar la extracción de información de flujo de control.
Automatic Generation of System Test Cases from Use Case Specifications: an NLP-based Approach (Wang et al., 2019)	Herramienta UMTG que automatiza la generación de casos de prueba de sistema, en función de las especificaciones de casos de uso y el modelo de dominio del sistema bajo prueba.

Es importante comprender que la programación naturalística no busca implementar el idioma inglés como lenguaje natural, ya que los lenguajes naturales son inherentemente ambiguos y habría demasiadas variantes. Tampoco se trata de que los no-programadores escriban programas de computadora, puesto que ellos no se preocupan por criterios de calidad o aspectos técnicos. Se trata de una alternativa, como punto intermedio, que considere elementos del lenguaje natural en el diseño de los lenguajes de programación, especialmente aquellos que consideran la reflexividad, para crear lenguajes que sean tanto expresivos como ejecutables (Lopes et al., 2003).

El paradigma naturalístico no considera a la inferencia en la construcción de lenguajes, tampoco se pretende que los lenguajes aprendan, etiqueten, clasifiquen o se entrenen. La construcción de los lenguajes naturalísticos de propósito general actuales se basa en una gramática de un subconjunto del idioma Inglés.

El paradigma naturalístico se vio respaldado y a la vez directamente favorecido por Roman Knoll y Mira Mezini, quienes reportaron en 2006, el primer lenguaje naturalístico de propósito general identificado en la literatura, Pegasus, el cual genera código directamente ejecutable (Knöll & Mezini, 2006). En su propuesta, los autores defienden que el programador debería programar como si contara una historia, por lo que diseñaron una interfaz en donde se puede escribir una “idea” en lenguaje natural; en sus primeros pasos, Pegasus permitía escribir oraciones en Alemán e Inglés.

En el Listado 1.3 se puede observar un ejemplo de un programa naturalístico para implementar el requerimiento: “Closing the open files” (Knöll et al., 2011). En el Listado 1.4 se muestra la representación del mismo requerimiento en el lenguaje artificial Java.

⁴<https://modeling-languages.com/slack-uml-modeling/>

⁵<https://modeling-languages.com/?s=natural+language>

Listado de código 1.3: Ejemplo del código naturalístico “Closing the open files”.

```
1 if there are open files , close them and inform the user
```

Listado de código 1.4: Ejemplo del código artificial “Closing the open files”.

```
1 boolean filesOpen = false ;
2 for (File f : application . getFiles ( ) )
3   if ( f . isOpen ( ) )
4   {
5     filesOpen = true ;
6     f . close ( ) ;
7   }
8 if ( filesOpen )
9   user . inform ( ) ;
```

Como se puede observar, la representación naturalística permite considerar al código mismo como una aproximación a la auto documentación, ya que el código es muy parecido a la oración que define el requerimiento, como si se escribiera un pseudocódigo, es decir la brecha entre dominios es casi nula. Cualquiera que tenga conocimientos de programación puede identificar lo que hace el Listado 1.4, sin embargo, entre más complejos sean los requerimientos, los programas serán más difíciles de entender y mantener.

No es suficiente representar los problemas con objetos o aspectos, las necesidades van más allá. Se necesitan lenguajes que no limiten la expresividad, sino que permitan representarla de manera que se obtengan programas acordes a lo que el usuario necesita. [Kazman \(2017\)](#) al hablar de Software Engineering afirma:

“A medida que aumenta la importancia del software en nuestro mundo, también aumentan los deberes, las habilidades y los conocimientos que se requieren de los ingenieros de software. Los desafíos están ahí y nosotros, como comunidad, debemos estar a la altura para enfrentarlos”.

En este trabajo se busca proponer un método naturalístico para el desarrollo de software de propósito general, con base en los lenguajes de programación naturalística de propósito general existentes, los cuales son reflexivos, expresivos y generan código directamente ejecutable.

Capítulo 2

Estado del arte

El paradigma naturalístico ha evolucionado lentamente desde que fue propuesto. En (Pulido-Prieto & Juárez-Martínez, 2017), los autores presentan un estudio de las tecnologías de programación naturalística, en donde exponen los lenguajes específicos de dominio y los lenguajes de propósito general existentes. En el estudio destacan a Pegasus (Knöll & Mezini, 2006) como el único que genera código que puede considerarse como directamente ejecutable, ya que como paso intermedio construye código fuente en uno de diferentes lenguajes de programación de alto nivel, el cual no necesita completarse. Dado que, dentro del paradigma naturalístico lo que se busca es contar con lenguajes de propósito general, que incluyan elementos del lenguaje natural, que sean reflexivos y ejecutables, este capítulo se centra en presentar los avances al respecto, destacando los lenguajes más representativos y la existencia de un modelo conceptual que apoya el fortalecimiento de este paradigma. El modelo sirvió como punto de referencia para analizar y comparar las características de los lenguajes, con el objetivo de conocer sus alcances y limitaciones. Al exponer cada lenguaje, se expone su fundamento naturalístico y un ejemplo representativo, lo que permite una mejor comprensión de cada propuesta y del paradigma en general.

2.1. Pegasus

El primer lenguaje naturalístico de propósito general reportado en la literatura es Pegasus (Knöll & Mezini, 2006). De acuerdo con sus autores, la desventaja con otros paradigmas es que el programador ajusta la idea a un lenguaje de programación específico, cuando lo natural debería ser escribir los programas en el lenguaje nativo del programador, sin necesidad de pensar en tecnicismos o abstracciones de otro idioma o lenguaje de programación, de tal manera que diferentes programadores puedan leer el mismo programa en su idioma particular. Con Pegasus, el proceso de traducción entre diferentes idiomas y lenguajes de programación de alto nivel queda a cargo de sus traductores (Knöll, 2019). El objetivo de lo anterior es que el programador exprese las ideas en la manera en la que las piensa. En (Knöll & Mezini, 2006), los autores expresan las desventajas de la programación actual:

1. Las ideas de un programador deben ser traducidas a un lenguaje de programación específico, lo que podría ocasionar pérdida de semántica.

2. La curva de aprendizaje es muy alta, considerando las opciones que hay en la actualidad.
3. La reutilización de código se ve limitada, porque no existe un conjunto de datos uniforme y certificado para dicho código.

En opinión de los autores de Pegasus, las principales características distintivas de los lenguajes naturales en comparación con los lenguajes de programación actuales son: referencia implícita, compresión y dependencia del contexto. Las tres características ayudan a reducir en gran medida la redundancia. En los lenguajes naturales se depende del contexto para dar significado a una frase, los lenguajes artificiales son independientes del contexto, lo que dificulta su capacidad expresiva.

La unidad mínima de Pegasus es la “idea”, formada por percepciones de acuerdo con el pensamiento humano. Este concepto se formaliza en (Knöll & Mezini, 2006) para hacerla accesible a la computadora. Los autores establecen la conveniencia de usar ambos, lenguaje natural y notación formal, para evitar los inconvenientes de expresar las ideas de diferente manera. También definen el concepto **Naturalistic Programming** como: “*writing computer programs with the help of natural language*”.

2.2. Modelo conceptual naturalístico

Un aporte significativo al paradigma naturalístico es el modelo conceptual propuesto en (Pulido-Prieto & Juárez-Martínez, 2019), de acuerdo a los autores, un paradigma de programación debe ser lo suficientemente genérico como para expresar abstracciones independientemente del dominio y, al mismo tiempo, describir suficientes elementos expresivos para permitir lenguajes de uso general.

En (Pulido-Prieto & Juárez-Martínez, 2017), los autores identifican los siguientes problemas en la construcción de lenguajes naturalísticos de propósito general:

1. Referencias indirectas ambiguas. Es necesario fortalecer el modo en el cual se implementan las referencias indirectas en gramáticas formales, para incrementar la expresividad desde una perspectiva humana.
2. Carencia de iteradores naturalísticos. Los lenguajes de programación implementan las iteraciones de una manera explícita, es poderosa y expresiva, pero no lo suficiente para describir cosas en términos del lenguaje natural.
3. Carencia de bifurcaciones naturalísticas. Una bifurcación naturalística que funciona en una forma similar al iterador naturalístico, descrito anteriormente, podría permitir la descripción de las instrucciones en términos de “los próximos diez elementos”, por ej., sin utilizar estructuras antinaturales.
4. Carencia de abstracciones con características naturalísticas. Una abstracción que admite una cantidad y una combinación dinámica similar a un adjetivo podría simplificar el uso de listas en lenguajes de programación.

El modelo conceptual identifica los elementos mínimos necesarios que todo lenguaje naturalístico de propósito general debe poseer, considerando los siguientes elementos que están presentes en el lenguaje natural¹:

- Deixis: Señalamiento a una persona, un lugar o un tiempo, o a una expresión lingüística mediante ciertos elementos gramaticales. La deixis está incluida en todos los idiomas:
 - Deixis anafórica: Deixis que se produce mediante anáfora.
 - Deixis catafórica: Deixis que se produce mediante catáfora.
- Endófora: Es una referencia indirecta definida en el mismo texto, se divide en dos clasificaciones:
 - Anáfora: Referencia indirecta cuyo referente se definió previamente en el mismo texto, p.ej. “Toma la llave, úsala en la cerradura”.
 - Catáfora: Referencia indirecta cuyo referente es subsecuentemente definido en el mismo texto, p.ej. “Después de tomarla, usa la llave en la cerradura”.
- Sintagma: Palabra o conjunto de palabras que se articula en torno a un núcleo y que puede ejercer alguna función sintáctica.

El sintagma más común es la frase imperativa, carece de sujeto y predicado, pero tiene significado en el lenguaje.

Derivado de lo anterior, los elementos propuestos en el modelo reportado son los siguientes (Pulido-Prieto & Juárez-Martínez, 2019):

1. Sustantivo como abstracción base que es singular o plural.
2. Adjetivo como complemento al sustantivo.
3. Verbo, que es la acción que realiza un sustantivo.
4. Circunstancia como condicionante que responde a eventos.
5. Sintagma para definir instrucciones con una complejidad más allá del sujeto y el predicado.
6. Instrucciones en términos de elementos que se describieron con anterioridad (anáforas). Las referencias catafóricas se dejan como un elemento opcional.
7. Tipos que se definan de forma explícita y estática, de modo que se utilicen en la construcción de instrucciones que se conformen con sintagmas.
8. La implementación del modelo es un lenguaje textual que ofrece un nivel de expresividad similar a un lenguaje natural, pero que se formalizó para evitar la ambigüedad.

¹Definiciones tomadas del Diccionario de la Real Academia Española, <https://www.rae.es>

De acuerdo a los autores, el objetivo del modelo propuesto es que los lenguajes que lo implementen soporten un mecanismo de composición en tiempo de instanciación, por medio de la combinación de sustantivos con adjetivos; que dichos lenguajes permitan el uso de circunstancias para definir contextos relativos a eventos que se describan. Por último, que dichos lenguajes posean un nivel de expresividad que se base en el uso de sintagmas complejos. El modelo es una contribución hacia la disminución de la brecha que existe entre el dominio del problema y el dominio de la solución.

2.3. SN

En ([Pulido-Prieto & Juárez-Martínez, 2019](#)), junto con el modelo naturalístico, los autores reportan la construcción del lenguaje SN (*Sicut Naturali*: como natural, por sus siglas en latín), como prueba de concepto del modelo propuesto. El lenguaje soporta el uso de referencias indirectas y la temporalidad, además de ser suficientemente formal para eliminar la ambigüedad de modo que se describan abstracciones en términos de elementos propios del inglés.

SN surgió de un objetivo: contar con un lenguaje de programación de propósito general, basado en lenguaje natural, como una evolución de la Programación Orientada a Aspectos, que dé mayor expresividad, que sea comprendido por parte del equipo de desarrollo y el cliente o usuario; que permita traducir los requerimientos en declaraciones que no pierdan o deformen la información, llegando hasta la implementación.

SN incluye los siguientes elementos, derivados del modelo propuesto por sus autores:

1. Uso de sustantivos (singular y plural).
2. Uso de adjetivos para abstraer características.
3. Atributos para describir los elementos que componen a un sustantivo.
4. Verbos para definir acciones de los sustantivos.
5. Circunstancias para establecer condiciones a las que reaccionarán los sustantivos, los adjetivos y los verbos.
6. Atributos derivados, cuyo valor depende de otros atributos o verbos.
7. Sintagmas nominales, construcciones gramaticales cuyo núcleo es el sustantivo.
8. Oraciones.
9. Iterador naturalístico, SN permite trabajar con instrucciones reflexivas (referencia anafórica o catafórica) para realizar iteraciones.
10. Decisiones naturalísticas.

2.4. Cal-4700

Otro lenguaje naturalístico (encontrado en literatura gris) es Cal-4700 ([Rzeppa & Rzeppa, 2019b](#)), cuya propuesta es escribir en inglés plano, partiendo de la analogía de escribir un pseudocódigo, omitiendo el paso de traducción a una sintaxis artificial. Los autores, padre e hijo, se propusieron desarrollar un compilador de inglés simple con el interés de responder a las siguientes preguntas:

1. ¿Es más fácil programar cuando no se tienen que traducir los pensamientos a una sintaxis alternativa?
2. ¿Se pueden analizar los lenguajes naturales de una manera relativamente “descuidada” (como los humanos aparentemente los analizan) y aun así proporcionar un entorno lo suficientemente estable para una programación productiva?
3. ¿Pueden los programas de bajo nivel (como compiladores) escribirse conveniente y eficientemente en idiomas de alto nivel (como el inglés)?

Los autores dan respuesta afirmativa a estas preguntas, mediante la propuesta de un analizador que funciona de acuerdo a como funcionan teóricamente, los centros de análisis gramatical en el cerebro humano.

2.5. Comparación de características entre lenguajes

En la Tabla 2.1 se presenta un cuadro comparativo de las características de los lenguajes naturalísticos encontrados: Pegasus, SN y Cal-4700. Dicha comparación se reportó en ([Hernández-González et al., 2020](#)). Se consideran estos lenguajes dado que cumplen con la filosofía del paradigma naturalístico, cuentan con elementos del lenguaje natural, son reflexivos (capaces de identificar elementos que hayan sido mencionados previamente y hacer referencia a ellos) y generan código directamente ejecutable a partir de oraciones escritas en lenguaje natural, sin necesidad de forzar abstracciones de otros paradigmas.

La comparación se hizo reconociendo el cumplimiento de las características expuestas en el modelo conceptual de [Pulido-Prieto & Juárez-Martínez \(2019\)](#), agregando otros elementos con base en las propuestas de los mismos lenguajes. Los lenguajes SN y Cal-4700 fueron probados funcionalmente, ya que se cuenta con la documentación y archivos ejecutables correspondientes, SN por ser un producto del Instituto Tecnológico de Orizaba y Cal-4700 por proporcionar su documentación en su página oficial². La revisión de Pegasus se hizo con base en la información y ejemplos contenidos en su sitio oficial³, ya que no se encuentra compilador disponible para su prueba. Se marca con el símbolo “x” el cumplimiento de cada característica y con un “-” la ausencia.

²<http://www.osmosian.com/>

³<http://www.pegasus-project.org/en/Welcome.html>

Tabla 2.1: Comparación de Lenguajes Naturalísticos.

Elementos del modelo	Pegasus	SN	Cal-4700
1. Sustantivos singular y plural	x Implícito	x	x
2. Adjetivo	x Implícito	x	x
3. Verbo	x	x	x
4. Circunstancia (eventos)	x	x	x
5. Sintagma ⁴	x	x	x
6. Referencia anafórica	x	x	x
7. Definición explícita de tipos	-	x	x
Otros elementos			
8. Asignaciones naturalísticas	x	x	x
9. Atributos	x	x	x
10. Oraciones	x	x	-
11. Iterador naturalístico/ciclos	x	x	x
12. decisiones naturalísticas (condicionales)	x	x	x
13. Referencia catafórica (opcional)	x En ciclos	x En ciclos	x En ciclos
14. Uso de archivos externos para su ejecución	x	-	x

De manera general se puede observar que los tres lenguajes permiten crear declaraciones comunes en lenguaje natural, es decir, que utilicen sustantivos, verbos, adjetivos y atributos, además de definir sintagmas, los más comunes como frases imperativas. Permiten hacer referencias anafóricas, a la vez de ejecutar instrucciones que se esperan de todo lenguaje de programación, como es el caso de la respuesta a eventos, definición de condicionales y el uso de ciclos, iteradores o repetidores.

De acuerdo a la revisión de los ejemplos de Pegasus, éste no permite la definición de tipos de forma explícita, probablemente los tipos estén definidos en su diccionario. La definición de adjetivos o sustantivos tampoco se hace de manera explícita, pero en los ejemplos se ve claramente el uso de los mismos.

La documentación de Cal-4700 indica que no permite el uso de oraciones con la estructura gramatical que lo exige, pero sí el uso de sintagmas como frases imperativas. Resalta el hecho de que, por el momento, ninguno de los tres lenguajes implementa las referencias catafóricas, característica opcional del modelo conceptual que se tomó como base para la comparación (Pulido-Prieto & Juárez-Martínez, 2019).

La ausencia de la característica 14. Uso de archivos externos para su ejecución, en realidad es una ventaja, ya que Pegasus necesita una base de datos para su ejecución, Cal-4700 necesita que su archivo *the noodle* esté presente al momento de compilar los programas, pero SN sólo se tiene que instalar como cualquier programa de computadora y seguir el proceso de compilación similar a cualquier lenguaje de programación.

Una característica importante por resaltar es que Pegasus permite elegir el lenguaje de programación en el que se desea generar el código, mientras que SN y Cal-4700 generan el código ejecutable directamente. De manera individual, se puede resaltar de cada uno lo siguiente:

⁴Instrucciones con una complejidad más allá del sujeto y del predicado.

- De acuerdo a la documentación de Pegasus, éste traduce entre diferentes idiomas: Alemán, Inglés, Chino, Español, Hindi, Ruso y otros. Además de traducir a diferentes lenguajes de programación: Java, C, C++, Ruby, Python, Haskell y Prolog (Knöll, 2019).
- Cal-4700 permite el uso de banderas (*Flags*), rutinas de conversión de tipos, el uso de posesivos y definición de registros. No permite condicionales anidadas, ciclos anidados, objetos, números reales ni ecuaciones, ya que cuando se habla en lenguaje natural no se utilizan estos conceptos (Rzeppa & Rzeppa, 2019b). Aunque los autores no están de acuerdo en utilizar comentarios al programar, entienden la necesidad e incluyen tres tipos de comentarios: generales, observaciones dentro de las instrucciones de código y cualificadores, los cuales permiten especificar detalles que cambian el curso de ejecución del programa. Los comentarios generales se podrían utilizar para aclarar algún referente, ya que no debería ser necesario repetir lo que representa el código en sí.
- SN no admite comentarios, puesto que la intención al programar en un lenguaje naturalístico es evitar el uso de los mismos, ya que el código se auto documenta. El lenguaje es único en su tipo ya que fue construido con base en una gramática, sin necesidad del uso de archivos externos, ontologías o bases de datos. Una característica importante por resaltar de SN es que implementa propiedades no funcionales (aspectos) por medio de las circunstancias, además de que permite el uso de gramáticas embebidas para, por ejemplo, realizar cálculos matemáticos.

Además, incluye el uso de sintagmas nominales⁵ y atributos derivados, atributos cuyo valor depende de otros atributos o verbos, cuya implementación es similar a un verbo, pero su sintaxis e invocación es la de un atributo (Pulido-Prieto & Juárez-Martínez, 2019).

Esta comparación permitió exponer de manera más concreta lo que ofrecen los lenguajes naturalísticos de propósito general actuales, a la vez que detectar oportunidades de mejora; la ejecución de programas construidos con SN y Cal-4700 permitió visualizar la interacción del programador con los lenguajes. Este acercamiento motiva a reflexionar la necesidad de que el paradigma evolucione dadas las ventajas que promete, para que se utilice como una propuesta formal en el desarrollo de software. En la siguiente sección se incluyen ejemplos de programas de los lenguajes revisados.

2.6. Ejemplos de programas naturalísticos

En esta sección se incluyen programas que ejemplifican el uso de los lenguajes presentados en la sección anterior, Pegasus, SN y Cal-4700; con el objetivo de lograr una mejor comprensión del paradigma. Los ejemplos de Pegasus y Cal-4700 fueron tomados de sus sitios oficiales, (Knöll, 2019) y (Rzeppa & Rzeppa, 2019b) respectivamente. Los ejemplos de SN fueron construidos como parte de un ejercicio escolar.

⁵Frase cuyo núcleo es el sustantivo y se asocia al sujeto gramatical.

2.6.1. Pegasus

El Listado 2.1 muestra el uso de consultas de datos en lenguaje natural, el uso de macros y sistemas de bases de datos. Como se puede observar, pareciera que en el código se está contando una historia, más que codificando un programa estructurado. Por esta razón, se omiten los números de línea. De igual manera, se observa que no hay una definición explícita de tipos, sustantivos, adjetivos o verbos, como se comentó en la sección anterior.

Listado de código 2.1: Sistema con acceso a Base de Datos en Pegasus.

```
“Computer, take all employees who have been
working for us longer than 20 years and whose
last name starts with A to O and send them an
invitation to the celebration of the jubilee.”

“Sending an invitation to a ceremony to
someone (to a person): Create a new letter
with the address of the person as addresses.
Write “Dear (the first name of the person)!”
Then [...]”

“Delete contact data of an employee five
years ater this employee has left the university ,
but his Internet user account at once.”
```

2.6.2. Cal-4700

El listado de la Figura 2.1 se escribió usando el editor de Cal-4700, disponible para MS-Windows, y ejemplifica la representación naturalística de una lotería primitiva. Se deja al lector decidir el nivel de expresividad de tales instrucciones. Como se puede observar, el programa utiliza un generador de números aleatorios, un ciclo, una suma y un condicional.

Como Pegasus, Cal-4700 define varios tipos de sustantivos en su archivo de soporte para ayudar a los programadores. Además, Cal-4700 solo compila archivos sin extensiones y usa su propio editor para invocar al compilador y generar el programa. Los archivos deben almacenarse en un directorio único que también contiene el archivo *the noodle*, el cual actúa como el cerebro de Cal-4700. Finalmente, el nombre del archivo ejecutable es el de la carpeta que almacena los archivos.

Utilizando Cal-4700, los programadores pueden construir aplicaciones de consola o aplicaciones de pantalla (de forma gráfica). Su propia documentación se creó con el editor de documentos de Cal-4700 “*What You See Is What You Get*”.

2.6.3. SN

El listado incluido en la Figura 2.2 presenta un código ejemplo escrito con SN usando su editor básico, también disponible para MS-Windows. El código está escrito para realizar también un simulador de lotería. Se observa que no es necesario utilizar las instrucciones de ejecución, inicio, apagado y cierre de ciclo, obteniendo un código más corto. SN genera código binario en Java (*bytecode*), el nombre de la clase principal (ejecutable) es el nombre



Figura 2.1: Representación naturalística de una lotería primitiva en Cal-4700.

designado en el código fuente, después de la instrucción *main*. En la Figura 2.2, la clase ejecutable generada es *Lottery.class*.

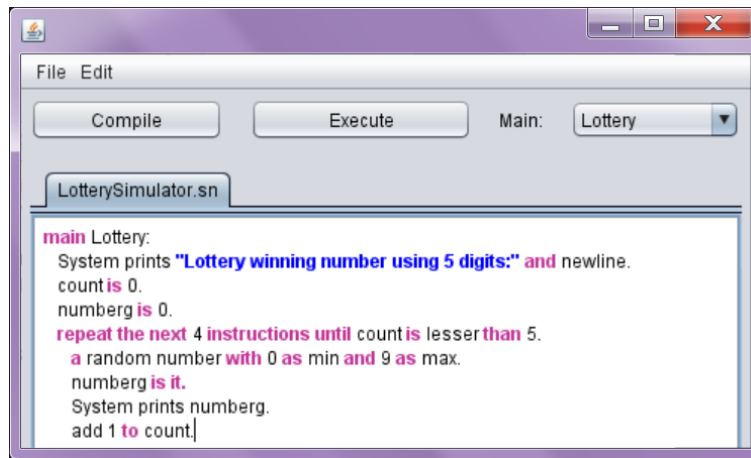


Figura 2.2: Simulador de Lotería Naturalística en SN.

El listado en la Figura 2.3 realiza el cálculo promedio de tres números. En este caso, SN utiliza una gramática embebida para representar las instrucciones matemáticas de forma sencilla, respetando así su expresividad inherente.

Para mayor referencia sobre SN y su funcionalidad, en (Alducin-Francisco et al., 2019) los autores presentan una discusión exhaustiva de un conjunto de programas de computadora desarrollados con este lenguaje.

2.7. Discusión

Como se observó en las secciones de este capítulo, los lenguajes expuestos ofrecen más allá de lo indispensable para ser considerados lenguajes naturalísticos de propósito general. Los ejemplos se incluyeron con el objetivo de comprender la intención del paradigma y comprobar sus características. Sin embargo, surgen varias interrogantes en cuanto a su uso, entre ellas ¿qué tan flexible es Pegasus en la definición de sustantivos por parte del usuario? ¿Se

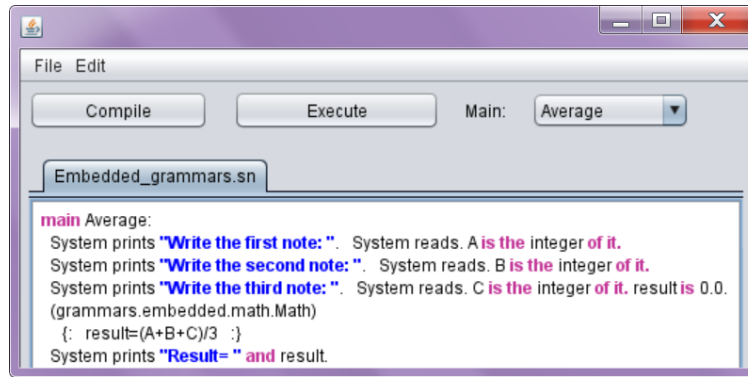


Figura 2.3: Uso de gramáticas embebidas en SN.

eliminaron completamente las instrucciones imperativas? ¿Cuál lenguaje es más declarativo o expresivo?

En el caso de instrucciones matemáticas, autores como Knöll (2019) sugieren que se escriban de manera literal, dado que se trata de una disciplina formal cuyas representaciones son expresivas en sí mismas, de lo contrario supondría un retroceso en la misma. Por su parte, SN permite el uso de gramáticas embebidas, conservando la representación formal que tienen las operaciones matemáticas, pero contempla también la representación naturalística. Por casos como éste es conveniente conservar instrucciones imperativas, con la probabilidad de que haya otros casos que lo ameriten.

La respuesta a preguntas específicas de lo que permite o no un lenguaje o sobre cuál es más declarativo o expresivo, no es sencilla de realizar, ya que no basta con ejecutar ejercicios aislados o revisar la documentación del lenguaje. Otro aspecto importante para analizar es la curva de aprendizaje inherente al uso de un lenguaje de programación. En el caso del paradigma naturalístico, la intención es que la programación sea más natural y expresiva, sin embargo, qué tan natural es un lenguaje para un desarrollador tendría que probarse dentro de un estudio de caso controlado.

Actualmente no se reporta ningún estudio de caso en donde se utilicen estos u otros lenguajes naturalísticos dentro de un proceso de desarrollo de software, o una propuesta de método que dicte los pasos a seguir. Aún con las ventajas de programar utilizando elementos del lenguaje natural, es importante considerar las dimensiones que pueden llegar a tener los programas, y cómo estos se deben analizar y estructurar para resolverlos.

Todas estas interrogantes pueden promover trabajos de investigación para que el paradigma evolucione. Se cuenta con un modelo naturalístico y lenguajes que lo soportan, a opinión de los autores, la primera consideración para su evolución sería incorporar el uso de estos lenguajes dentro de un proceso de desarrollo de software, para ello deberán proponerse ciertas pautas de análisis y diseño como primeros pasos, ya que es la base del proceso de desarrollo, por ejemplo, cómo se identificarían y representarían los sustantivos.

También es importante considerar de qué manera el paradigma naturalístico podría mejorar el proceso de desarrollo de software. Contar con métodos de análisis y diseño es el primer paso para la aceptación del paradigma, las implicaciones en el proceso de desarrollo podrían incluir para su mejora, la agilidad, la consecuente reducción de tiempos y la disminución de la densidad de defectos, entre otras actividades involucradas.

Teóricamente, se puede considerar que en el paso de una fase a otra se perderá menos información y se evitarán o disminuirán costos de retrabajo, pero estas suposiciones deben comprobarse. La resolución es que se necesita de un método que permita verificar y validar las ventajas del paradigma, con todos los retos que conlleva, esto permitirá considerarlo seriamente como propuesta de desarrollo, de lo contrario, no se cuenta con datos concretos que promuevan su uso.

Capítulo 3

Aplicación de la metodología

La historia indica que cuando surge un nuevo paradigma, surgen también uno o más lenguajes de programación que lo soportan. Las fases del proceso de desarrollo de software se van resolviendo hacia atrás, es decir, se resuelve el diseño, el análisis y al final, la especificación de requerimientos. Lo consecuente en este caso fue, de igual manera, establecer un camino hacia atrás, desde la fase de codificación, hasta la fase que inicia con el desarrollo de los requerimientos. Tomando en cuenta lo anterior, y para dar cumplimiento a los objetivos del trabajo de investigación, se propusieron los siguientes pasos específicos para la obtención del método:

1. Revisión y actualización, en su caso, del estado del arte del paradigma naturalístico. Identificando lenguajes, métodos, estudios de caso y propuestas en general.
2. Revisión de la documentación correspondiente (artículos, ejemplos, manuales, sitios web) de los lenguajes naturalísticos de propósito general encontrados, cuya salida sea código directamente ejecutable. Esto permitirá seleccionar los lenguajes que puedan incluirse en las pruebas de los casos de estudio.
3. Como complemento del paso anterior, prueba de los lenguajes naturalísticos seleccionados, mediante programas representativos de su gramática, con el objetivo de verificar su funcionalidad y concretar la decisión de incluirlos o no.
4. Análisis de las características de las diferentes propuestas metodológicas más utilizadas en la actualidad, para identificar si cuentan con elementos naturalísticos que puedan aplicarse a la propuesta.
5. A partir del punto anterior, realización de la propuesta de Diseño naturalístico.
6. Realización de la propuesta de Análisis naturalístico.
7. Determinación de los las características que tendrán los problemas para probar el método, para una futura evaluación.
8. Prueba de la propuesta.
9. Evaluación de los resultados de la aplicación del método propuesto.

10. Realización de estancia.
11. Realización de los ajustes necesarios.
12. Generación de la versión final de la propuesta metodológica.

En las siguientes secciones se resumen las actividades realizadas. Los pasos 5 y 6 (realización de la propuesta de diseño y análisis naturalístico) se realizaron en paralelo, por lo que se reportan juntos, al igual que el diseño de los casos de estudio (pasos 7 y 8), la prueba como parte de la validación y evaluación del método (pasos 9 y 10), así como la realización de ajustes y generación de la versión final (pasos 12 y 13).

3.1. Revisión del estado del arte

Para obtener las publicaciones relevantes del paradigma naturalístico, se realizó la Revisión Sistemática de la Literatura, de acuerdo a [Turner \(2010\)](#); [Kitchenham & Charters \(2007\)](#); [Moher et al. \(2009\)](#); [Galster et al. \(2018\)](#); [Siddaway \(2014\)](#). En el Apéndice A se especifica la estrategia que se siguió para realizar la búsqueda, las fuentes de búsqueda, los criterios de inclusión y exclusión que se aplicaron para filtrar los resultados, así como la manera de extraer y sintetizar los datos. En A.2 se presentan los resultados obtenidos y una discusión sobre ellos.

Dentro de los resultados encontrados, resalta el modelo naturalístico propuesto por [Pulido-Prieto & Juárez-Martínez \(2019\)](#), el cual determina los elementos mínimos a considerar en la construcción de lenguajes naturalísticos de propósito general. Se encontraron tres lenguajes naturalísticos: Pegasus ([Knöll & Mezini, 2006](#)), SN ([Pulido Prieto & Juárez Martínez, 2020](#)) y Cal-4700 ([Rzeppa & Rzeppa, 2019b](#)). Los tres cumplen con las características del enfoque naturalístico: son de propósito general, son reflexivos (que permiten hacer referencia a elementos definidos previamente) y generan código directamente ejecutable. No se encontró reporte en la literatura de algún método de desarrollo naturalístico, sólo la aplicación del enfoque en la traducción de oraciones en inglés a partir de descripciones de Casos de Uso, hacia código fuente en Java ([Mefteh et al., 2018](#)). La desventaja de este trabajo es que sigue vinculado a abstracciones no naturalísticas (objetos), además de utilizar un patrón predeterminado en la fase (oculta) del diseño.

3.2. Revisión de los lenguajes naturalísticos encontrados

De los lenguajes naturalísticos de propósito general encontrados (Pegasus, SN y Cal-4700), se revisaron los artículos, sitios web y manuales correspondientes, con el objetivo de entender y probar su funcionalidad. También se contactó a los autores de dichos lenguajes para resolver inquietudes, se resumen las conversaciones a continuación:

- Roman Knöll, autor de Pegasus (comunicación personal, abril, 2020). Se estableció contacto con él para evaluar la posibilidad de conseguir un compilador del lenguaje,

ya que de acuerdo a las publicaciones encontradas ([Knöll & Mezini, 2006](#); [Knöll et al., 2011](#)) y a la información de su sitio web ([Knöll, 2019](#)), el lenguaje parecía prometedor, sin embargo, el autor mencionó que no se encuentra compilador estable disponible y tampoco habrá en un futuro cercano.

- Gerry Rzeppa, autor de Cal-4700 (comunicación personal, octubre, 2020). Cal-4700 fue encontrado en la literatura gris, su compilador y documentación correspondientes se encuentran disponibles en los sitios web relacionados, la documentación incluye varios ejemplos de la aplicación del lenguaje ([Rzeppa & Rzeppa, 2019b,a, 2020](#)). El autor tuvo una gran disposición en la comunicación de sus intenciones con la realización de este lenguaje, además de compartir ejemplos completamente corregidos. De igual manera compartió el código de Sal-1016, la versión en español de Cal-4700, a través de su colaborador Pablo Cayuela. Parte de las reflexiones de G. Rzeppa se reportan en el Capítulo 5.2.2, dedicado al Diseño de software.
- Oscar Pulido Prieto, autor de SN (comunicación personal constante desde 2019) ([Pulido-Prieto & Juárez-Martínez, 2019](#); [Pulido Prieto & Juárez Martínez, 2020](#)). SN es un prototipo producto del Instituto Tecnológico de Orizaba, lo que ha facilitado la comunicación con su autor, además de su disposición para proporcionar el mantenimiento necesario. El prototipo ha experimentado varios cambios identificados como resultado de los avances de este trabajo de investigación, tratando de cumplir cada vez más con los requerimientos del paradigma.

En el Capítulo 2 se presentó el estado del arte detallado de los tres lenguajes, además de una tabla comparativa de sus características de acuerdo al modelo naturalístico propuesto por [Pulido-Prieto & Juárez-Martínez \(2019\)](#).

3.3. Prueba de los lenguajes naturalísticos encontrados

Como complemento del paso anterior, se probó la funcionalidad de los lenguajes naturalísticos seleccionados (SN y Cal-4700, ya que Pegasus no cuenta con compilador disponible), mediante programas representativos de su gramática.

En la Sección 2.6 se pudieron observar algunos ejemplos de programas elaborados con los tres lenguajes naturalísticos. En el caso de SN y Cal-4700, los programas se probaron utilizando el editor y ambiente de desarrollo correspondientes.

La prueba de los lenguajes permitió conocer sus alcances y limitaciones, con la finalidad de determinar las condiciones que deberían considerarse en los casos de estudio. En el caso de SN, las pruebas permitieron la mejora de ciertas características del lenguaje, además de la conclusión del acceso a Bases de Datos y la inclusión de números aleatorios.

En el caso de Cal-4700, su robustez permitió considerarlo como opción para desarrollar la construcción de la solución de los casos de estudio propuestos. A manera de comprobación del nivel de expresividad del lenguaje, se realizó una prueba con estudiantes del séptimo semestre de la carrera de Ingeniería en Sistemas Computacionales, del ITO. Los resultados se reportan en el Apéndice C.

3.4. Análisis de las características de las metodologías actuales

La mayoría de los métodos actuales están vinculados a algún paradigma de programación, por lo que las notaciones y artefactos que utilizan se basan en los principios del paradigma utilizado. Al tratar de analizar los métodos actuales, específicamente las fases de Análisis y Diseño, la tendencia se estaba orientando a hacer un mapeo forzado de elementos, por lo que se decidió enfocarse en los principios que tienen en común, desde el punto de vista del proceso de desarrollo de software. Los resultados de esta actividad se reportan en las Secciones 5.1 (Análisis) y 5.2 (Diseño), se definen conceptos, principios y algunos artefactos a considerar. También se analizó la concordancia del método con algún estándar internacional, fácil de entender y de aplicar para cualquier empresa (ver reporte en Sección 4).

De acuerdo a G. Rzeppa (comunicación personal, 14 de noviembre, 2020), autor de Cal-4700 y Sal-1016, la programación en inglés plano se asemeja a la programación procedural, los lenguajes se construyeron para entender oraciones de la forma más normal y natural en la que la gente indica instrucciones a otros, en este caso, a la computadora, a través de instrucciones procedurales, utilizando tipos y variables. Estas reflexiones motivaron la consideración del uso de artefactos procedurales en la elaboración del método, siempre considerando su aporte naturalístico.

3.5. Realización de la propuesta de Análisis y Diseño naturalístico

En la Sección 6, se reporta la primera versión del método naturalístico, la propuesta se realizó considerando:

- Apego a un estándar internacional, en este caso, el estándar ISO/IEC 29110.
- Los principios de Análisis y Diseño de software analizados en el paso anterior.
- Las instrucciones naturalísticas como instrucciones procedurales.
- Artefactos con aporte naturalístico, es decir, que coincidan con el enfoque de utilizar elementos del lenguaje natural. Idealmente esto facilitará la trazabilidad de los requerimientos.
- La notación de “idea” propuesta por Roman Knöll, autor de Pegasus.

3.6. Determinación de las características que tendrán los problemas para probar el método, para una futura evaluación

La mayoría de las necesidades de los usuarios de sistemas incluyen la persistencia de datos, es decir, que permanecen aún después de terminada la ejecución del software, mediante el

uso de Sistemas Manejadores de Bases de Datos (SMBD) o de archivos. SN permite la persistencia a través del acceso a los SMBD PostgreSQL y MySQL, Cal-4700 permite la manipulación de archivos de texto plano, por lo que como prueba de concepto del método, se concluyeron todos los requerimientos del *Course Management System (CMS)*, incluyendo datos persistentes. Como se mencionó en la Sección 3.3, al ser Cal-4700 el lenguaje más estable, fue éste el considerado para la construcción de los programas.

Además del CMS, se plantearon dos propuestas: 1) una prueba con estudiantes de maestría, proporcionándoles una breve capacitación del método para el desarrollo de un Caso de Uso a partir del análisis de un problema. Se les dieron 24 horas para realizar los artefactos y el código correspondiente en Cal-4700. Los resultados de la práctica se pueden observar en la Sección 7.3. 2) El planteamiento de un problema factible de solucionar con Cal-4700, como parte de una tesis de la Maestría en Sistemas Computacionales del ITO. Esto proporcionó mayor formalidad y compromiso en la obtención de resultados.

Idealmente, se deberían comparar proyectos de software que fueron construidos con lenguajes tradicionales, creando otra versión que utilice lenguajes naturalísticos, sin embargo, no se cuenta con registros de sistemas previos para comparar; además de que, por el momento los lenguajes naturalísticos tienen muchas limitaciones en comparación de los lenguajes tradicionales actuales.

3.7. Evaluación del método de desarrollo

De acuerdo a [ISO/IEC/IEEE \(2017\)](#), *Validación* significa:

- confirmación, mediante la aportación de pruebas objetivas, de que se han cumplido los requisitos para un uso o aplicación específicos previstos.
- la garantía de que un producto, servicio o sistema satisface las necesidades del cliente y otras partes interesadas identificadas. A menudo implica aceptación e idoneidad con clientes externos.

La validación demuestra que los usuarios pueden utilizar el sistema para sus tareas específicas.

Hasta el momento, la validación del método ha consistido en comprobar que se puede construir software siguiendo el método propuesto y utilizando lenguajes naturalísticos. En el ciclo Agosto-Diciembre 2021 se reportó el desarrollo del CMS (*Course Management System*). En el año 2022 se reportó el desarrollo de una aplicación lúdica (un rompecabezas para niños) utilizando Cal-4700. El proyecto lo realizó el estudiante de la Maestría en Sistemas Computacionales del ITO, José Antonio Juárez Romero (cuya defensa de grado se llevó a cabo el 25 de noviembre del 2022), aplicando el método naturalístico propuesto en este trabajo de investigación.

En Diciembre del 2022 se realizó una prueba con estudiantes de la Maestría en Sistemas Computacionales, solicitándoles que elaboraran los artefactos de un CU que ellos seleccionaran de un conjunto de CU pertenecientes al análisis de un problema. Tuvieron 24 horas para desarrollar los artefactos y los programas correspondientes. Los resultados de la prueba están documentados en la Sección 7.3.

3.8. Realización de estancias

Como resultado de la estancia de investigación que se realizó en el Centro de Investigación en Matemáticas, Zacatecas (CIMAT), el método se documentó con base en el estándar internacional ISO/IEC 29110. La documentación se incluye en la sección [4](#).

3.9. Realización de ajustes y generación de la versión final

De las prácticas realizadas, no se identificó la necesidad de realizar ningún ajuste, por lo que la versión original permanece como la versión final. Se detectó la posibilidad de modularizar el código utilizando Cal-4700 para trabajos futuros, así como la necesidad de realizar más pruebas para explorar, actualizar o reemplazar alguno de los artefactos propuestos.

Capítulo 4

Estándar ISO/IEC TR 29110

Uno de los criterios más importantes a cumplir en el desarrollo de software es la trazabilidad, es decir, la capacidad de rastrear un camino desde los requerimientos hacia adelante o desde el código hacia atrás, sin que haya pérdida de información. La trazabilidad favorece la pronta identificación de defectos, la facilidad de mantenimiento, así como la facilidad en la administración de la verificación y validación del software.

Para que los artefactos propuestos en el método promuevan alcanzar este criterio, es necesario que el método se apegue a algún estándar que marque las pautas y normas a seguir para el desarrollo de software, de manera que garantice que los pasos están alineados a lo que dictan las buenas prácticas avaladas por comités de expertos. El estándar elegido fue el ISO/IEC 29110, debido a que es el que está orientado a Pequeñas Organizaciones, por lo tanto, es más sencillo y fácil de entender y de aplicar que otros estándares. El apego a un estándar permite evidenciar de manera práctica la relación entre las actividades de Análisis, Diseño y Construcción de software, como se explicará a continuación. Además permite verificar la trazabilidad de cada una de las actividades del método propuesto, otra de las razones por las cuales se decidió modelar el método en apego a un estándar como éste, como se mostrará más adelante.

El estándar **ISO/IEC TR 29110 - Systems and Software Engineering-Lifecycle profiles for Very Small Entities (VSEs)** (ISO/IEC, 2016), es un estándar que proporciona un marco de referencia para el desarrollo de software dentro de las Pequeñas Organizaciones.

Para los propósitos de la serie de Normas ISO/IEC 29110, una Pequeña Organización (PO) es (ISO/IEC, 2016):

“una empresa, una organización, un departamento o un proyecto conformada por no más de 25 personas”.

De acuerdo con el informe de la Perspectiva para las Pequeñas y Medianas Empresas y el Espíritu Emprendedor (2005) de la Organización para la Cooperación y Desarrollo Económicos (OCDE): “Las Pequeñas y Medianas Empresas (PYME) constituyen el sector dominante de las organizaciones empresariales en todos los países del mundo, llegando a cifras entre el 95 % y el 99 % del total de la población empresarial, dependiendo del país”.

El uso de las normas ISO/IEC está extendiéndose a organizaciones de todos los tamaños. La serie de Normas ISO/IEC 29110 facilita el uso de los procesos de las Normas:

- ISO/IEC/IEEE 12207:2017 *Systems and software engineering – Software life cycle processes*
- ISO/IEC/IEEE 15288:2015 *Systems and software engineering – System life cycle processes*
- ISO/IEC/IEEE 15289:2019 *Systems and software engineering – Content of life-cycle information items (documentation)*
- ISO 9001 *Quality management systems – Requirements*

El estándar provee conformidad a los subconjuntos de normas base seleccionadas, al brindar Perfiles de POs. Otorga los lineamientos para cada perfil de proceso junto con la hoja de ruta para la conformidad con las Normas mencionadas anteriormente.

De acuerdo a lo reportado en el estándar (ISO/IEC, 2016), se realizó una encuesta de mercado de las POs para indagar sobre la utilización de las normas ISO/IEC. La encuesta determinó que existen tres razones principales que impiden a las VSEs utilizar las normas ISO/IEC:

- la falta de recursos (28 %);
- las normas no son requeridas (24 %);
- 15 % de los encuestados considera que las normas son difíciles y burocráticas, y no proporcionan directrices adecuadas para su uso en entornos de negocio pequeños.

Para el 74 % de las VSEs encuestadas es muy importante ser reconocidas o certificadas respecto a una norma. La certificación ISO la solicitan el 40 % de ellas. Las VSEs manifiestan la necesidad de asistencia para adoptar e implementar normas. Más del 62 % declararon que les gustaría que hubiese más guías con ejemplos. Un 55 % pregunta por normas ligeras y fáciles de entender con plantilla.

La propuesta de la norma ISO/IEC 29110 es precisamente dar solución a lo problemas anteriormente mencionados. La norma pretende ser utilizada con cualquier ciclo de vida tales como cascada, iterativo, incremental, evolutivo o ágil.

La parte ISO/IEC 29110-4-1 provee la especificación para todos los perfiles del Grupo de Perfiles Genérico. El Grupo de Perfiles Genérico es aplicable a las POs que no desarrollan productos de software crítico. Los perfiles están basados en subconjuntos de elementos de estándares adecuados. Los perfiles de POs se aplican y están dirigidos a autores/proveedores de guías y de herramientas, y otro material de apoyo.

ISO/IEC 29110-5-1-2 Perfil Básico, es el primer documento publicado de esta parte que provee una guía de implementación sobre Administración e Ingeniería para el Grupo de Perfiles Genérico descrito en ISO/IEC IS 29110 Parte 4-1. El perfil Básico describe el desarrollo de software de una sola aplicación por un solo equipo de proyecto sin riesgos o factores situacionales especiales.

Las guías son publicadas como Informes Técnicos. Las guías de Gestión e Ingeniería proveen lineamientos para la implementación y uso de un perfil, por ejemplo: actividades recomendadas, medidas, técnicas, plantillas, modelos, métodos, entre otros.

En este caso se analizó la **Parte 5-1-2: Guía de gestión e ingeniería: Grupo de perfil genérico. Perfil básico** (ISO/IEC, 2011). Este perfil consta de dos procesos interrelacionados: Gestión de procesos e Implementación de Software (ver esquema en Figura 4.1).

Parte del estudio consistió en comprender las actividades de cada proceso, a su vez las tareas de cada actividad junto con su rol correspondiente (actor), además de los productos de entrada y/o salida relacionados.

De acuerdo al estándar, un diagrama es una representación gráfica de los procesos. Los rectángulos grandes con esquinas redondeadas indican procesos o actividades y los rectángulos más pequeños (con esquina recta) indican los productos. Las flechas gruesas direccionales o bidireccionales indican el mayor flujo de información entre los procesos o actividades. Las flechas delgadas direccionales o bidireccionales indican los productos de entrada o salida. La notación utilizada en los diagramas no implica el uso de algún proceso específico de ciclo de vida (ISO/IEC, 2011).

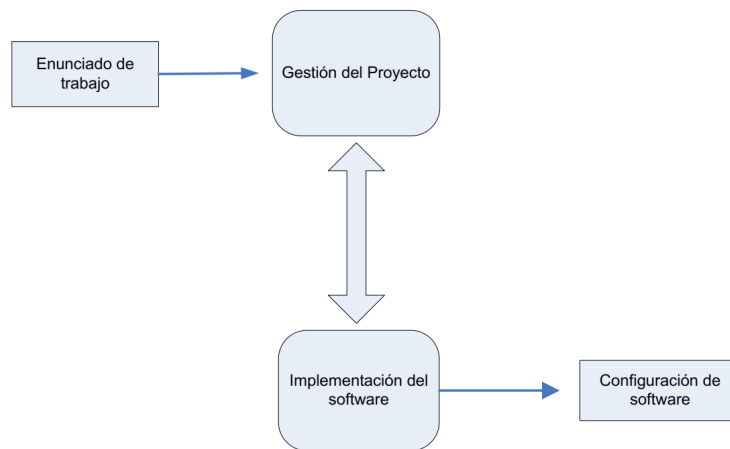


Figura 4.1: Procesos 29110 Perfil Básico (ISO/IEC, 2011).

Cada proceso tiene un conjunto de actividades, a su vez cada actividad tiene un conjunto de tareas identificadas, junto con sus roles y productos de trabajo de entrada/salida. Los roles o actores necesarios para llevar a cabo las tareas de cada actividad se indican a continuación, en orden alfabético (ISO/IEC, 2011):

- Analista (AN).
- Cliente (CL).
- Diseñador (DI).
- Equipo de Trabajo (ET).
- Gestor del Proyecto (GP).
- Líder Técnico (LT).

- Programador (PR).

En la Figura 4.2, se observa el diagrama de actividades del proceso de **Gestión del proyecto**, incluyendo los productos de trabajo más relevantes y la relación entre ellos.

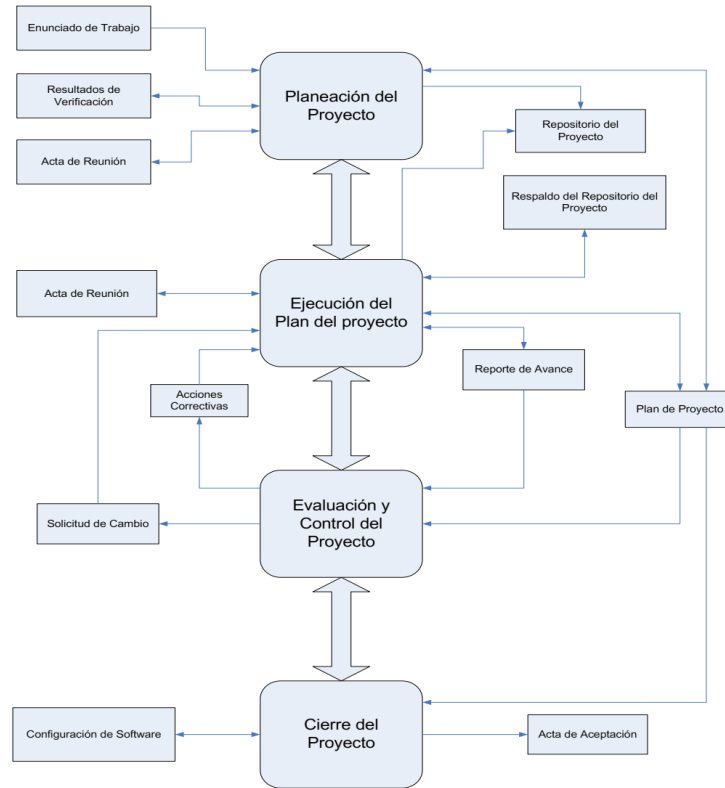


Figura 4.2: Diagrama del proceso Gestión del Proyecto (ISO/IEC, 2011).

En la Figura 4.3, se observa el diagrama de actividades del proceso de **Implementación del software**, de igual manera incluyendo los productos de trabajo más relevantes y la relación entre ellos.

La primera asignación fue realizar los diagramas detallados e integrados de cada actividad de cada proceso (lo cual no está incluido en la norma), con la finalidad de comprender a detalle cada una de sus tareas, los roles involucrados por tarea, además de los productos de trabajo correspondientes (de entrada/salida). El estándar incluye una serie de tablas descriptivas al respecto, sin embargo su interrelación se aprecia mejor de manera gráfica. Esto permitió analizar la incidencia y relación de las actividades propuestas en el método naturalístico dentro de lo que dicta el estándar. El diagrama de tareas de las actividades del proceso Gestión del Proyecto se observa en la Figura 4.4, el diagrama de tareas de las actividades del proceso Implementación del Software se observa en la Figura 4.5. Ambos siguen la notación gráfica utilizada en el estándar, de la cual se desconoce su nombre.

Para facilitar la comprensión de los diagramas se utilizaron abreviaturas para los productos de trabajo (verlos en la Tabla 4.1 ordenados alfabéticamente), además de colores que permitan identificar visualmente y con mayor facilidad, las actividades junto con sus tareas correspondientes.

Los productos de trabajo que se utilizan como entrada y/o salida de cada tarea, se distinguieron por colores, para identificar con mayor facilidad su uso dentro de cada actividad y sus tareas relacionadas (ver Figura 4.6). Es posible que algunos productos cambien de estado al terminar una tarea (p.ej. Revisado, Verificado, Validado, Aprobado, entre otros), fundamento que se considera en los diagramas escribiendo el texto del estado entre corchetes.

Tabla 4.1: Abreviaturas de los Productos de trabajo utilizados.

Producto de trabajo	Abreviatura
Casos de Prueba	CP
Configuración del Software	CS
Control de Versiones	CV
Diseño de Software	DSw
Especificación de Requisitos de Software	ERS
Manual de Operación	Manual Ope
Manual del usuario	Manual Usu
Plan del Proyecto	PPR
Procedimientos de Prueba	PP
Proyecto	PR
Registro de Trazabilidad	RegTraz
Repositorio del Proyecto	RP
Software	Sw

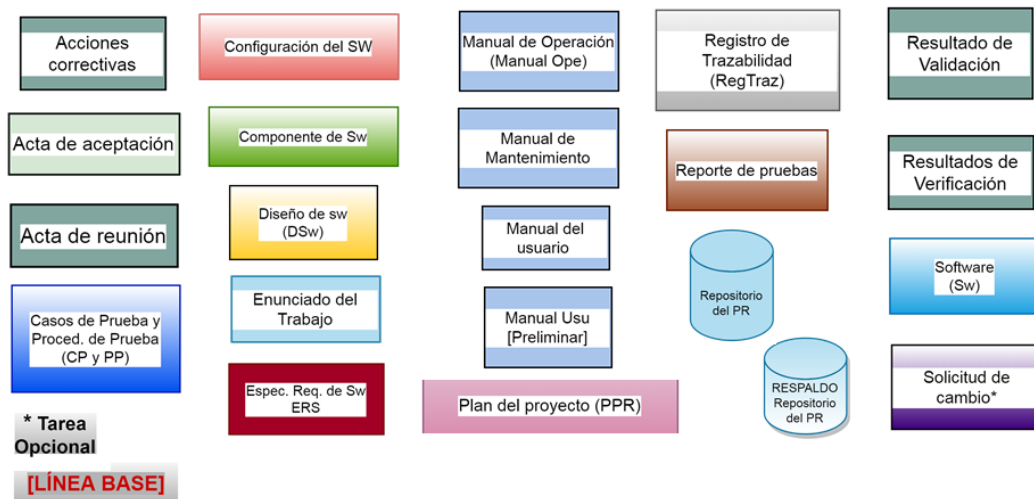


Figura 4.6: Productos de trabajo identificados por colores.

Consideraciones particulares:

- Cada actividad se representa en un contenedor con color de fondo, el cual coincide con las tareas y flechas que permiten identificar el conjunto de elementos dentro de todo el diagrama.
- Las tareas opcionales (indicadas en el estándar) se identifican con el símbolo asterisco “*”.
- Los datos de entrada de información o de salida se encuentran en letras cursivas.

- Los estados de cada producto se identifican entre corchetes con fondo blanco.
- Los actores se incluyen icónicamente en cada actividad para indicar quienes deben involucrarse. Para cada tarea se indica entre paréntesis, quien debe proveer información o participar en la misma, esto con la finalidad de no saturar con elementos gráficos cada diagrama, ya que se pierde el objetivo de clarificar la comprensión del estándar.

Se incluyen en los diagramas las siguientes consideraciones:

- Relaciones necesarias no indicadas en el estándar, con líneas punteadas, tanto en las flechas como en los recuadros, además de utilizar el color rojo para identificar estados propuestos.
- Se propone que cada tarea tenga un producto de salida, ya que debe haber evidencia del resultado, por ejemplo actas de reunión o documentos alusivos a la misma.
- En algunas tareas se indica implícitamente o explícitamente el estado del producto (ya sea de entrada o de salida). En este caso se incluye siempre explícitamente de acuerdo a la trazabilidad representada en el estándar, para lo cual se utiliza texto de letras rojas entre corchetes.

En las Figuras 4.7, 4.8 y 4.9, se muestran las tareas correspondientes a las actividades de Análisis, Diseño y Construcción, todas parte del Proceso de Implementación del Software, las cuales concuerdan con las que se propusieron en este trabajo de investigación. El estándar define tareas y productos de trabajo para cada actividad, sin embargo no indica de qué manera deben realizarse los diagramas o el código correspondiente del software, de hecho es una de sus ventajas, el ser flexible de aplicar a cualquier paradigma de programación. En este caso, el método propuesto define un conjunto de pasos específicos para el modelado y construcción de código naturalístico, dichos pasos son incorporables dentro de las tareas del estándar. Para una mejor navegación entre los diagramas, ver anexo electrónico [ComplementoApegoStd29110.pdf](#).

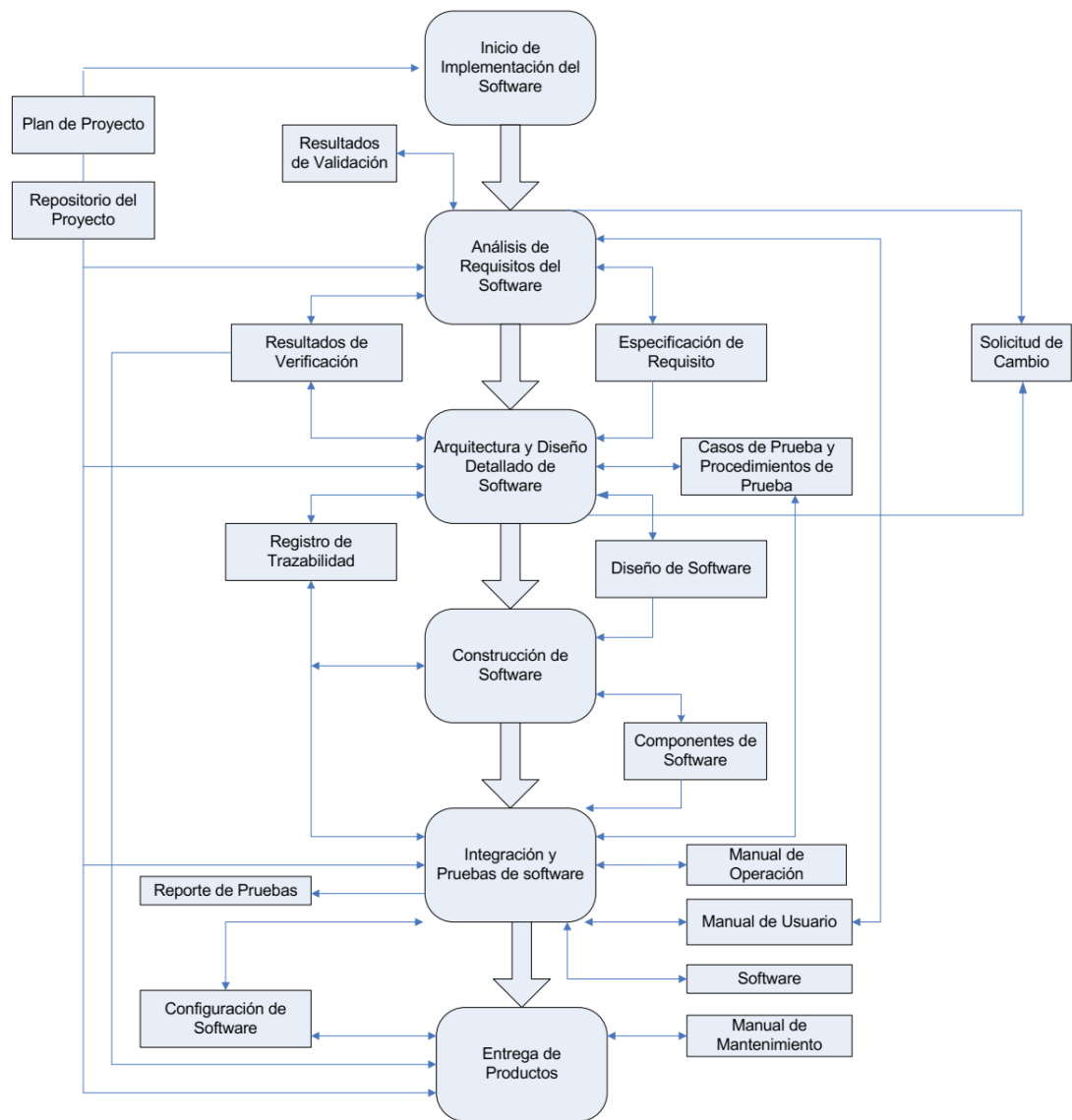


Figura 4.3: Diagrama del proceso Implementación del Software (ISO/IEC, 2011).

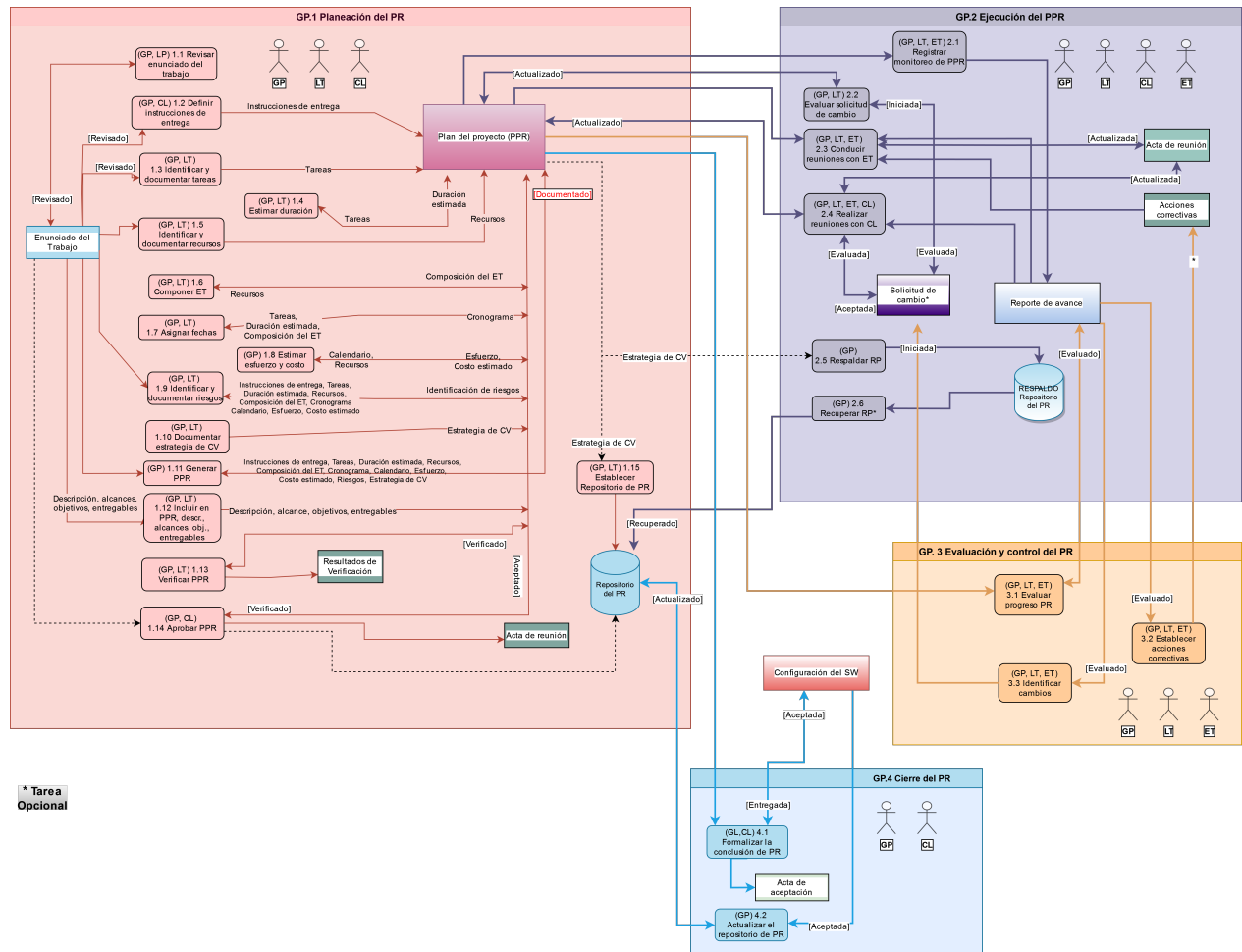


Figura 4.4: Tareas de las actividades - Proceso Gestión del Proyecto.

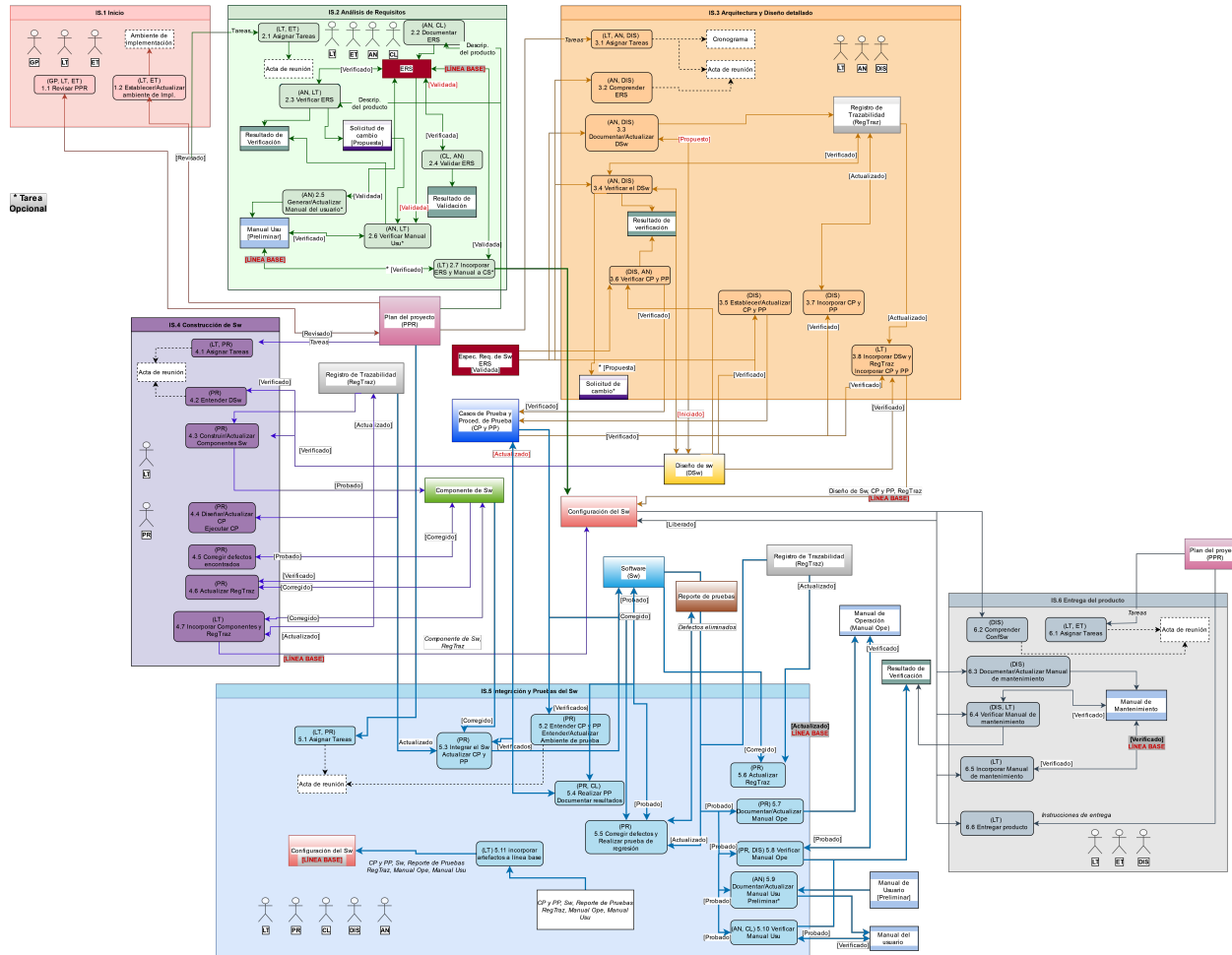


Figura 4.5: Tareas de las actividades - Proceso Implementación del Software.

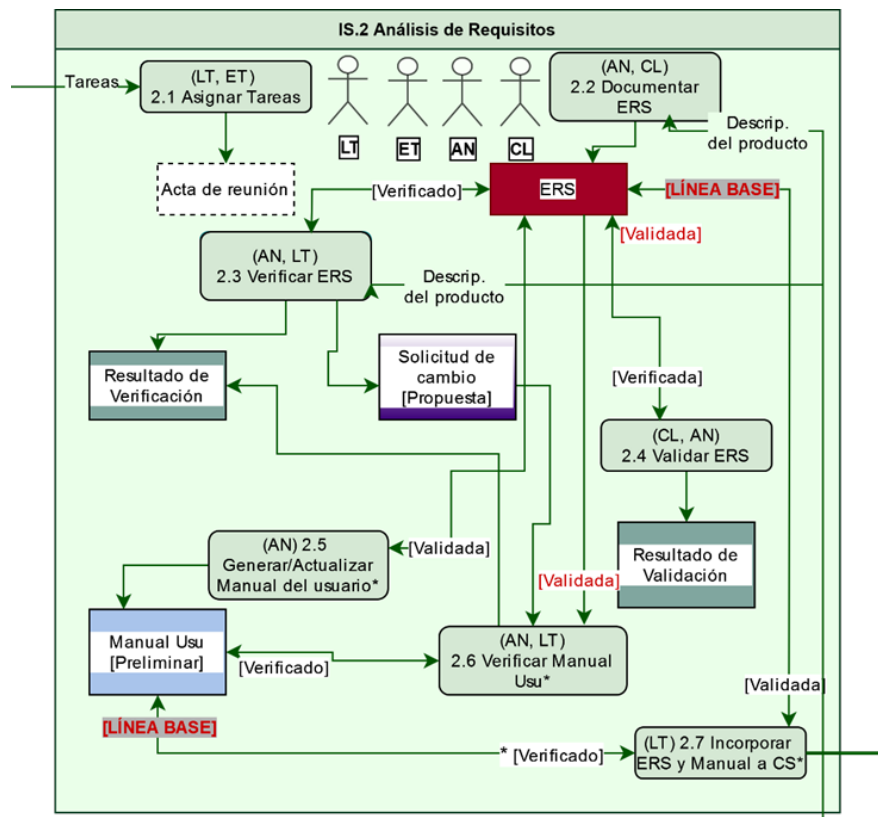


Figura 4.7: IS - Análisis de Requisitos.

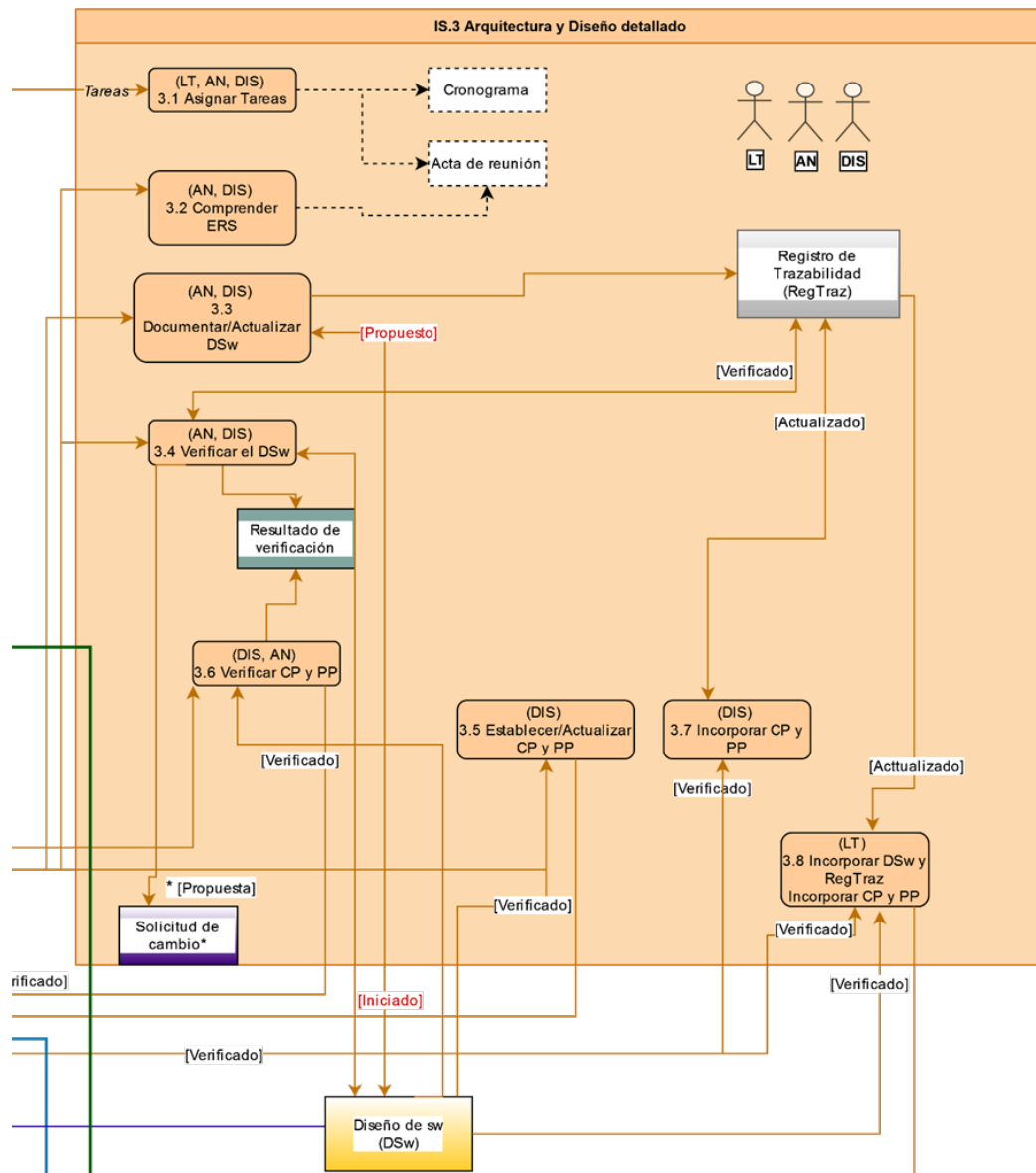


Figura 4.8: IS - Arquitectura y Diseño detallado.

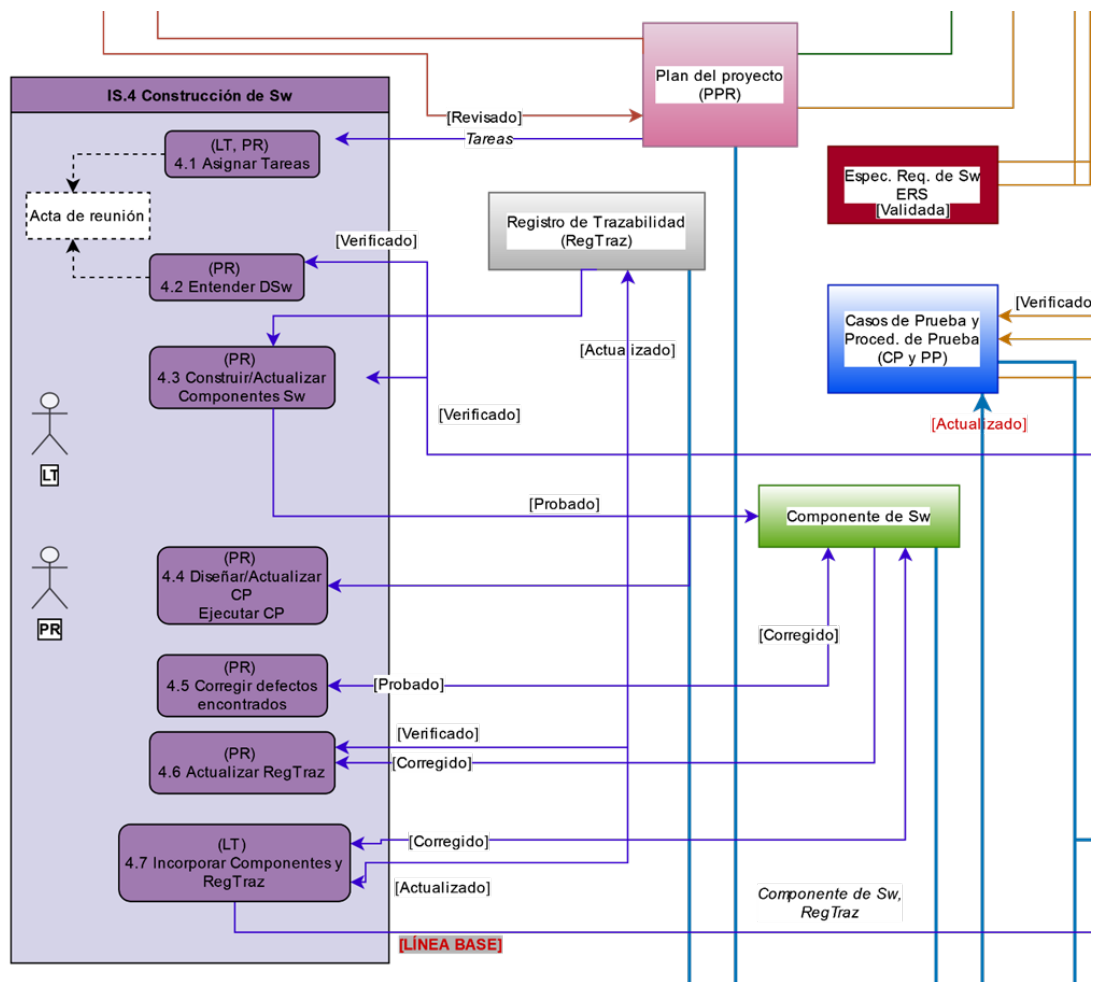


Figura 4.9: IS - Construcción de Software.

Capítulo 5

Principios de Análisis y Diseño de software

La mayoría de las metodologías y propuestas actuales están vinculadas a algún paradigma de programación, principalmente al Paradigma Orientado a Objetos (POO), por lo que la información encontrada no se asocia directamente con elementos naturalísticos, y la intención es evitar un mapeo forzado entre elementos. El Paradigma Orientado a Aspectos (POA), paradigma antecesor al Paradigma Naturalístico (PN), surgió de manera ortogonal al POO; tanto los métodos que surgieron como la notación empleada, debían respetar los fundamentos del POO. En el caso del PN, si bien las consideraciones lingüísticas del POA dieron pie a considerar al lenguaje natural como herramienta de programación de propósito general, al no ser ortogonal a ningún paradigma existente, la propuesta del método debe partir de cero.

De acuerdo a lo anterior, se realizó un análisis de los métodos y metodologías de desarrollo de software actuales, considerando los principios que tienen en común, desde el punto de vista del proceso de desarrollo de software¹, independientemente del paradigma que se esté modelando. El **proceso** implica traducir las necesidades del usuario en requisitos de software, transformar los requisitos de software en diseño, implementar el diseño en código, probar el código y, a veces, instalar y verificar el software para uso operativo. Estas actividades tienen la capacidad de superponerse o realizarse de forma iterativa(ISO/IEC/IEEE, 2017). Un ciclo de vida del desarrollo de software (CVDS) incluye los procesos de software utilizados para especificar y transformar los requerimientos de software en un producto software entregable (IEEE, 2014). De acuerdo al *Software Engineering Body of Knowledge* (IEEE, 2014), la naturaleza intangible y moldeable del software permite una amplia variedad de modelos de ciclo de vida² de desarrollo de software, es decir, maneras de realizar en cierta medida el mismo conjunto de actividades, pero que difieren en la forma en la que se ordenan y enlazan (Budgen, 2003), estos modelos son independientes del paradigma de programación utilizado, entre los más comunes se encuentran (IEEE, 2014):

- Modelo lineal o en cascada, en éste modelo se pueden apreciar claramente las fases

¹El proceso mediante el cual las necesidades del usuario se traducen en un producto de software(ISO/IEC/IEEE, 2017).

²Marco de procesos y actividades relacionados con el ciclo de vida que pueden organizarse en etapas, que también actúa como referencia común para la comunicación y el entendimiento(ISO/IEC/IEEE, 2017).

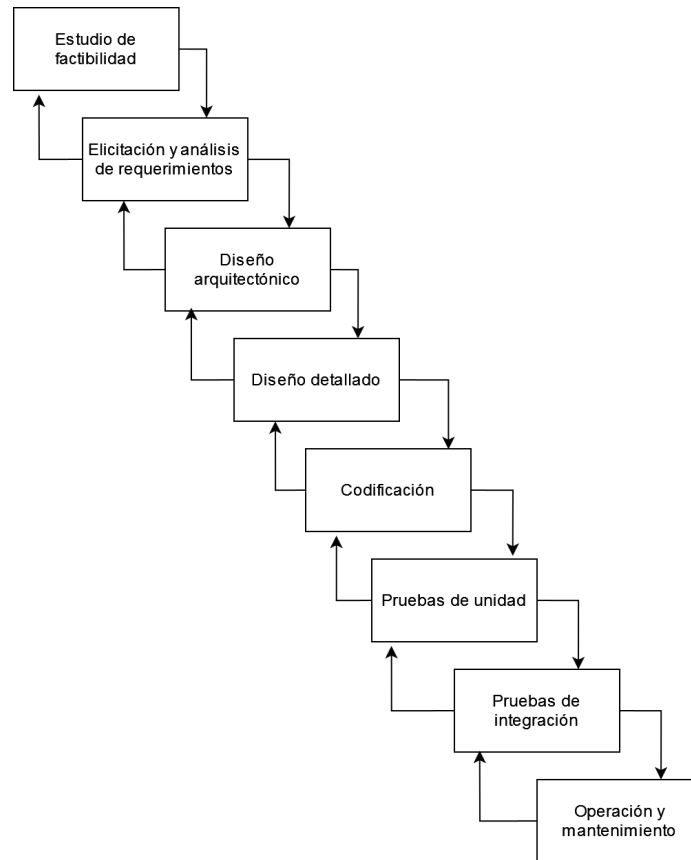


Figura 5.1: Modelo de ciclo de vida en cascada.

del proceso de desarrollo de software, las cuales se realizan de manera secuencial y predictiva, su representación se puede ver en la Figura 5.1.

- Modelo iterativo en el cual se desarrolla el software en incrementos de funcionalidad creciente en ciclos iterativos.
- Modelo ágil, en donde el desarrollo de software se dirige en ciclos iterativos cortos que producen software que funciona y se puede entregar.

Los métodos de desarrollo proporcionan un enfoque sistemático para la especificación, diseño, construcción, y verificación del software y los productos asociados (IEEE, 2014). El objetivo de este trabajo es proponer un método de desarrollo basado en el PN, con énfasis en las fases de Análisis y Diseño, por lo que, como primer paso se considera el papel que estas fases juegan en el desarrollo de software (Budgen, 2003):

- **Análisis.** Suele verse como un modelo más formal del problema, es decir, de las necesidades identificadas en los documentos producidos a partir de la etapa de obtención de requisitos.
- **Diseño arquitectónico.** Enfocado en la forma general de solución que se adoptará (por ejemplo, si se debe utilizar un sistema distribuido, cuáles serán los elementos principales y cómo interactuarán).

- **Diseño detallado.** Desarrollando descripciones de los elementos identificados en la fase anterior e involucrando interacción con ésta.

En las siguientes secciones se analizan con mayor detalle las fases de Diseño y Análisis, así como la primera versión del método propuesto.

5.1. Análisis de Requerimientos

Esta fase se ocupa de la obtención, análisis, especificación y validación de requerimientos, así como de la administración durante todo el ciclo de vida del producto de software (IEEE, 2014). Los requerimientos expresan las necesidades y limitaciones impuestas a un producto de software que contribuyen a la solución de algún problema del mundo real.

Por el momento, el enfoque es el desarrollo de los requerimientos, dejando la administración de los mismos para tratarla como trabajo a futuro. Dependiendo del ciclo de vida a seguir, se realizan estas actividades en diferentes momentos del proyecto y con distintos grados de profundidad o detalle (Wieggers & Beatty, 2013).

El desarrollo de requerimientos se subdivide en cuatro subdisciplinas: Elicitación, Análisis, Especificación y Validación. Las actividades esenciales de cada una de ellas se resumen a continuación (Wieggers & Beatty, 2013):

1. Elicitación

Comprende las actividades involucradas en el descubrimiento de los requerimientos, tales como entrevistas, análisis de documentos, prototipos, entre otros. Las acciones clave son:

- Identificar las clases de usuarios.
- Identificar *stakeholders*³.
- Entender las tareas y objetivos del usuario.
- Entender los objetivos del negocio (con los cuales se alinean las tareas).
- Entender el ambiente en el cual será usado el producto.
- Trabajar con individuos quienes representen las clases de usuarios.
- Entender sus necesidades y expectativas de calidad.

2. Análisis

Involucra alcanzar un entendimiento más preciso de cada requerimiento y representar conjuntos de requerimientos de múltiples maneras. Sus actividades esenciales son:

- Analizar la información recibida por parte de los usuarios para distinguir:
 - Metas, requerimientos funcionales, expectativas de calidad, reglas de negocio, soluciones sugeridas, etc.

³Individuo u organización que tiene un derecho, participación, reclamo o interés en un sistema o en su posesión de características que satisfagan sus necesidades y expectativas(ISO/IEC/IEEE, 2017).

- Descomponer los requerimientos de alto nivel en un nivel apropiado de detalle.
- Derivar requerimientos funcionales de otra información de requerimientos.
- Entender la importancia relativa de los atributos de calidad.
- Negociar prioridades de implementación.
- Identificar requerimientos no necesarios.
- Modelar requerimientos (de forma textual o gráfica), por ejemplo:
 - Historias de usuario.
 - Diagrama Entidad-Relación.
 - Diccionario de Datos.
 - Diagrama de Casos de Uso.
 - Diagrama de actividades.

3. Especificación

La especificación de requerimientos implica representar y documentar el conocimiento recopilado de una manera persistente y bien organizada. La actividad principal es:

- Traducir las necesidades recopiladas del usuario en requisitos escritos y diagramas adecuados para que sus destinatarios los comprendan, revisen y utilicen.

4. Validación

La validación confirma que se tiene el conjunto correcto de información que permitirá a los desarrolladores crear una solución que satisfaga los objetivos del negocio. Las actividades centrales son:

- Revisar los requisitos documentados para corregir cualquier problema.
- Desarrollar pruebas de aceptación y criterios para confirmar que el producto va a satisfacer las necesidades del cliente.

Por el momento el enfoque es el análisis, tratando de identificar los artefactos que podrían considerarse en la especificación de requisitos de software bajo el paradigma naturalístico.

5.1.1. Casos de Uso 2.0 (CU 2.0)

La identificación de CU es una técnica que ha demostrado ser compatible con cualquier paradigma, metodología, técnica o modelo de desarrollo. Su notación es entendible para comunicarse con los *stakeholders* (interesados), además de que se describen textualmente, en lenguaje natural, lo que facilitaría el paso del análisis al diseño y posteriormente a la construcción.

En (Medeiros et al., 2016), reportan los resultados de un estudio empírico para identificar los factores que comprometen la calidad de la Especificación de Requerimientos de Software (ERS) dentro del contexto de proyectos ágiles. Los desarrolladores entrevistados consideran que una ERS basada en historias de usuario es breve, vaga, ambigua e insuficiente para capturar la complejidad del sistema. Las historias de usuario son inadecuadas para describir

aspectos técnicos, además de que el nivel de detalle que proporcionan no es el apropiado. Para los Casos de Uso en su primera versión, encontraron que la percepción de los desarrolladores es la dificultad de incluir aspectos técnicos en su descripción, presentan mucha información y requieren de gran esfuerzo.

En este trabajo, se considera la opción de utilizar CU en su versión 2.0, dado que conservan la formalidad que los caracteriza, pero agregan flexibilidad para trabajar con características (*features*) y otras opciones como *Test-driven Development*, Kanban y Scrum. [Jacobson et al. \(2016\)](#), el autor de los Casos de Uso, argumenta que los casos de uso incluyen esencialmente las técnicas proporcionadas por las historias de usuario, pero ofrecen significativamente más para sistemas más grandes, equipos más grandes y desarrollos más complejos y exigentes. Son ligeros como las historias de usuario, pero también pueden escalar de una manera fluida y estructurada para incorporar tantos detalles como sea necesario.

En su versión 2.0, un CU se describe mediante un conjunto de historias estructuradas en forma de ([Jacobson et al., 2016](#)):

- Una narrativa de casos de uso que contiene flujos y requisitos especiales.
- Un conjunto de casos de prueba.

Se introduce el término porción (*slice*), la cual:

- Se crea seleccionando una o más historias para su implementación.
- Actúa como un marcador de posición para todo el trabajo requerido para completar la implementación de las historias.
- Evoluciona para incluir los cortes equivalentes a través del diseño, implementación y prueba.

En la Figura 5.2 se presenta el diagrama de CU de un sistema de biblioteca, en esta versión, las extensiones de cada Caso de Uso representan las historias de Caso de Uso, por lo que no es obligatorio que su nombre comience con verbo.

La Tabla 5.1 muestra el corte de CU en porciones, a partir de las historias de CU. También se incluye la complejidad determinada (Alta, Baja, Media), la cual permitirá planear las iteraciones de construcción correspondientes.

Tabla 5.1: Historias de CU y porciones del sistema de biblioteca.

Caso de Uso	Historia de CU	Porción de CU	Complejidad
Prestar libro	Prestar libro (básico)	Prestar libro éxito	Alta
	Registro de préstamo máximo	Prestar libro fracaso	Baja
	El prestatario debe multa	Prestar libro fracaso	Baja
Devolver libro	Devolver libro (básico)	Devolver libro éxito	Alta
	Regreso tardío	Devolver libro-problemas del usuario	Alta
	Libro dañado	Devolver libro-problemas del usuario	Baja
Reservar libro	Reservar libro (básico)	Reservar libro	Alta
	Libro ya reservado	Reservar libro	Baja
Buscar libro	Buscar libro (básico)	Buscar libro	Media

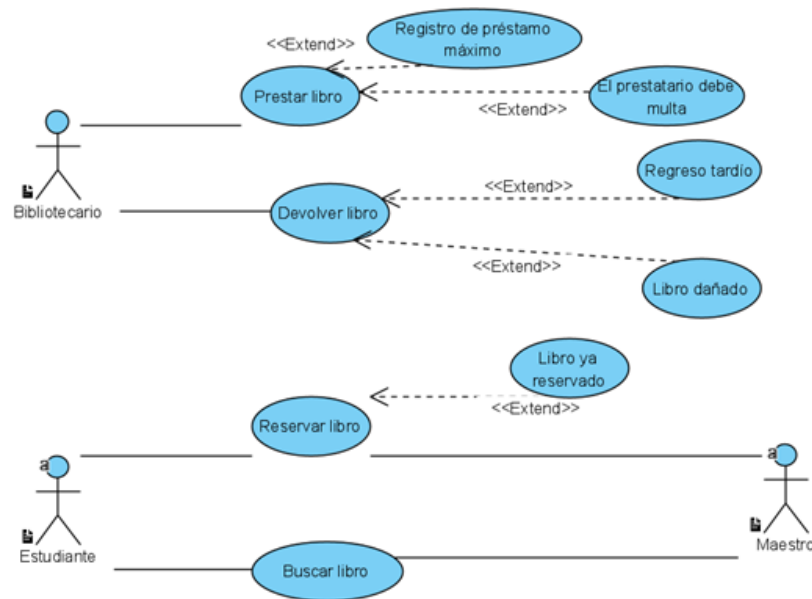


Figura 5.2: Diagrama de CU de un sistema de biblioteca.

Una vez que se tienen las porciones definidas, se realiza la planeación de las iteraciones utilizando el mapa de historias de la Figura 5.3, incluyendo los CU, las porciones y las historias de usuario especificadas anteriormente. Los diversos elementos de los CU 2.0 se transforman en diferentes elementos dentro del mapa de historias, de acuerdo a la correspondencia mostrada en la Tabla 5.2.

Los diagramas fueron elaborados en la herramienta CASE⁴ Visual Paradigm, siguiendo un tutorial de su sitio oficial⁵, esta herramienta facilita el modelado y administración de CU 2.0 con técnicas ágiles.

De acuerdo a Clark (2019), lo importante es comprender que un caso de uso es una colección estructurada de escenarios. Lo que hacen los CU 2.0 es proporcionar orientación sobre cómo estructurar los flujos de los Casos de Uso y garantizar que cada flujo (escenario) tenga al menos un caso de prueba.

Tabla 5.2: Correspondencia entre elementos de CU y elementos del mapa de historias.

Elemento CU 2.0	Elemento mapa de historias de usuario
Caso de Uso	Actividad de usuario
Porción de Caso de Uso	Tarea de usuario (<i>General Activity</i>)
Historia de Caso de Uso	Historia de usuario (<i>General User Task</i>)

Es posible agrupar uno o más flujos en un elemento llamado porción, la cual puede tratarse como una historia de usuario. Ésta se dimensiona y prioriza, además los casos de prueba se convierten en los criterios de aceptación. En el mismo trabajo, muestran la utilización de CU 2.0 con herramientas como Gherkin y Cucumber, en el desarrollo orientado al comportamiento, BDD por sus siglas en inglés (*Behavior-driven Development*). Gherkin presenta el comportamiento de la aplicación, utilizando lenguaje natural, entendiendo diferentes idiomas.

⁴Computer Aided Software Engineering

⁵<https://circle.visual-paradigm.com/steps-wizards/use-case-2.0>

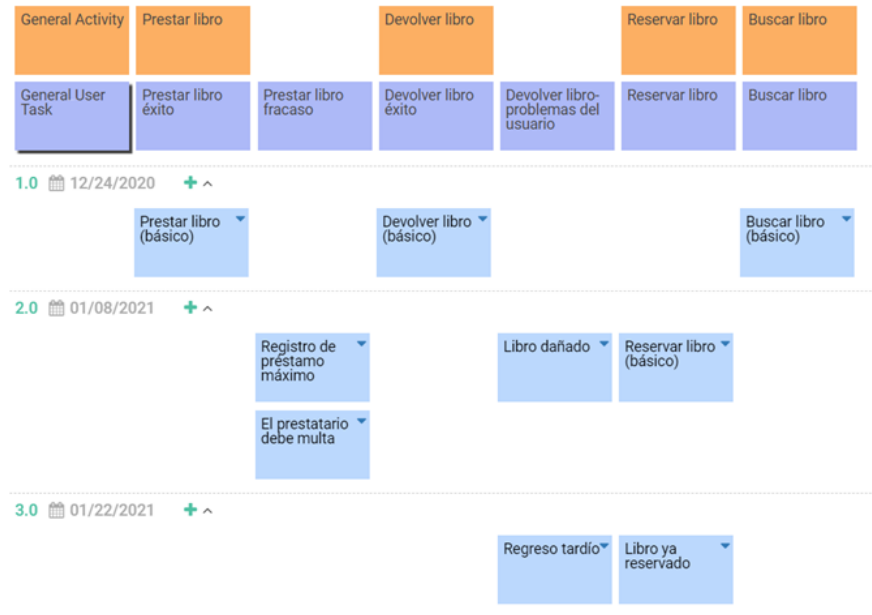


Figura 5.3: Planeación de iteraciones utilizando el mapa de historias.

A partir de la definición del comportamiento utilizando cláusulas como *Give*, *When* y *Then*, se utiliza un *framework* como Cucumber⁶ para generar las pruebas de la aplicación. La definición más simple de un escenario en Gherkin se observa en la Figura 5.4, la cual plasma una búsqueda en Google⁷.

```

Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

Scenario: Búsqueda simple en Google
  Given un navegador web en la página de Google
  When se introduce la palabra de búsqueda "pingüino"
  Then se muestra el resultado de "pingüino"

```

Figura 5.4: Definición simple BDD con Gherkin.

5.1.2. Otras herramientas y notaciones de modelado

En la actualidad se cuenta con varias notaciones para analizar y especificar los requerimientos de software, la mayoría de ellas presentan un nivel de abstracción alto, cercano al usuario, es decir, utilizan representaciones que los usuarios pueden entender con un sencillo entrenamiento, independientemente del paradigma de programación a utilizar en el diseño y la construcción del software.

En la literatura se mencionan los diagramas de actividades y la notación para modelar procesos de negocio, BPMN por sus siglas en inglés (*Business Process Model and Notation*).

⁶<https://cucumber.io/school/>

⁷<https://www.itdo.com/blog/ejemplos-bdd-behavior-driven-development-con-gherkin>

Aunque por el momento la propuesta se orienta a utilizar CU 2.0, no se limita a considerar otras notaciones ampliamente utilizadas como las mencionadas.

5.1.2.1. Diagrama de actividades

Se utilizan para mostrar el flujo de control de una actividad a otra. Es posible que se utilice para representar actividades concurrentes (IEEE, 2014). Este diagrama es útil para modelar procesos de negocio. En la Figura 5.5 se puede observar un ejemplo de un diagrama de actividades del proceso de préstamo de software de un área de sistemas.

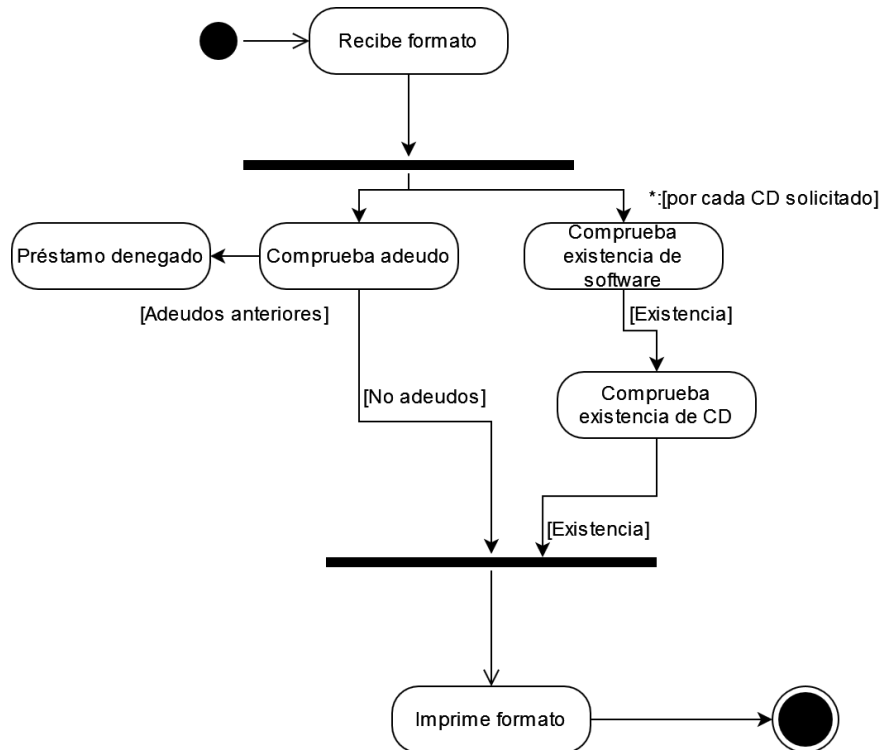


Figura 5.5: Diagrama de actividades préstamo de software.

5.1.2.2. BPMN (*Business Process Model and Notation*)

El estándar BPMN proporciona la capacidad de comprender los procedimientos comerciales internos de una empresa en una notación gráfica, y brinda a las organizaciones la capacidad de comunicar estos procedimientos de manera estándar. Además, la notación gráfica facilita la comprensión de las colaboraciones y las transacciones comerciales entre las organizaciones⁸.

En la Figura 5.6 se observa un ejemplo de diagrama de procesos utilizando BPMN para el proceso de mantenimiento de stock de productos⁹. En éste diagrama, se incluye el subproceso *Procurement*, el cual puede ser modelado en otro diagrama. Ésta notación es utilizada en

⁸<https://www.bpmn.org/>

⁹<https://www.omg.org/cgi-bin/doc?dtc/10-06-02 BPMN 2.0 by Example, v1>

la etapa de análisis y especificación de requerimientos. Dada su importancia y utilidad, se encuentran en la literatura reportes de su inclusión en propuestas como las siguientes:

- En (Ivanchikj & Pautasso, 2019), los autores diseñaron Sketch Miner, una herramienta que toma como entrada un proceso descrito en un DSL (*Domain Specific Language*), el cual permite a los participantes del proceso describir sus historias de usuario en un formato textual predefinido, y proporciona como resultado un borrador del modelo BPMN del proceso, el cual el analista de negocio podrá refinar y completar.
- En (Friedrich et al., 2011), los autores se basan en un enfoque automático para generar modelos BPMN a partir de texto en lenguaje natural. Combinan herramientas existentes del procesamiento del lenguaje natural mejoradas con un mecanismo de resolución de anáforas¹⁰ adecuado.

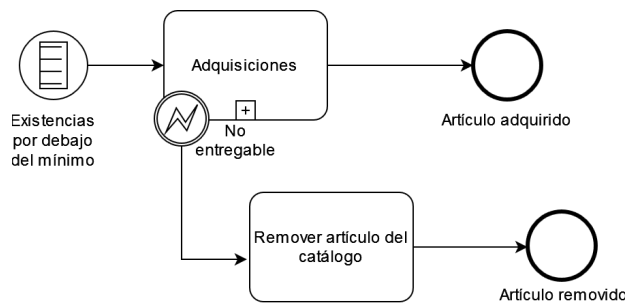


Figura 5.6: Proceso de mantenimiento de stock de productos.

Del análisis realizado, se decidió utilizar la técnica de Casos de Uso 2.0 para la descripción de requerimientos, ya que ha demostrado ser compatible con cualquier otra técnica de especificación de requerimientos o paradigma de desarrollo. Su descripción es en lenguaje natural, lo que concuerda con el paradigma naturalístico, pues se acerca al dominio del usuario. Su versión 2.0 permite adaptarse al uso de técnicas ágiles. De igual manera se utilizó la creación de prototipos que se menciona en la subdisciplina de Elicitación, pues además de auxiliar en el levantamiento de los requerimientos, se considera una actividad protectora de validación, al visualizar gráficamente los requerimientos desde etapas tempranas.

5.2. Diseño de software

De acuerdo a las definiciones del estándar 24765 ISO/IEC/IEEE (2017), diseño de software se puede considerar como:

El proceso de definir la arquitectura, componentes, módulos, interfaces, datos y otras características de un sistema o componente, para satisfacer los requerimientos especificados, y el resultado de ese proceso.

¹⁰Referencia indirecta cuyo referente ha sido previamente definido en el mismo texto, p.ej “Toma la llave, úsala en la cerradura”. *The Oxford English Dictionary*

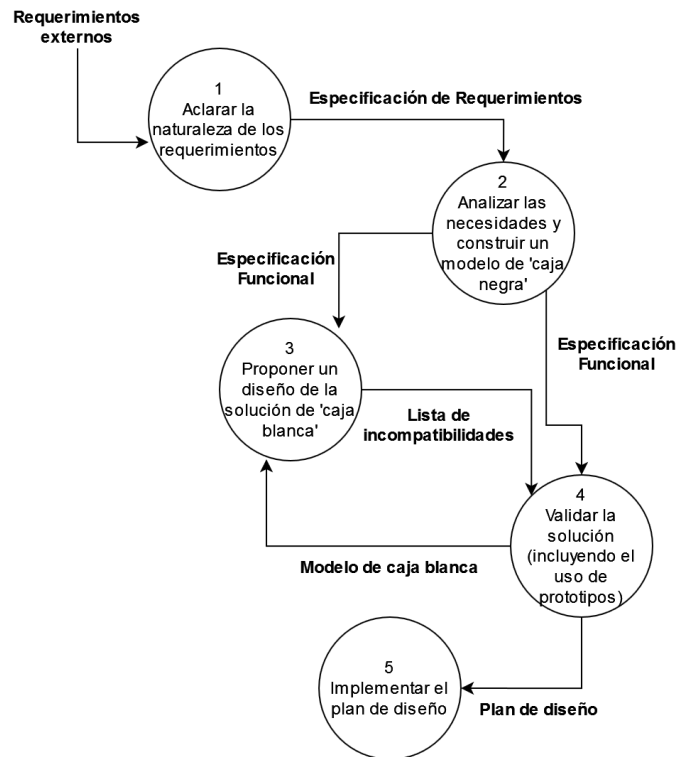


Figura 5.7: Modelo del proceso de Diseño (Budgen, 2003).

De acuerdo al estándar 12207 ISO/IEC/IEEE (2008), el diseño del software consta de “dos actividades que encajan entre el análisis de los requisitos del software y la construcción del software”:

- Diseño arquitectónico (o diseño de alto nivel): el cual desarrolla la estructura y organización de nivel superior del software e identifica los diversos componentes.
- Diseño detallado: especifica cada componente con suficiente detalle para facilitar su construcción. El modelo del proceso de diseño de acuerdo a Budgen (2003) se puede ver en la Figura 5.7, en donde a partir de la especificación de requisitos, se analizan las necesidades y se construye un modelo “de caja negra” (el diseño de alto nivel), se propone una solución “de caja blanca” (diseño detallado), se valida incluyendo el uso de prototipos, y finalmente se implementa la solución.

5.2.1. Diseño arquitectónico

De acuerdo a Taylor et al. (2009), la arquitectura de un sistema de software es el conjunto de decisiones de diseño principales sobre el sistema, a la vez que representa el modelo para la construcción y evolución de un sistema de software. Las decisiones de diseño abarcan todas las facetas del sistema en desarrollo:

- Estructura.

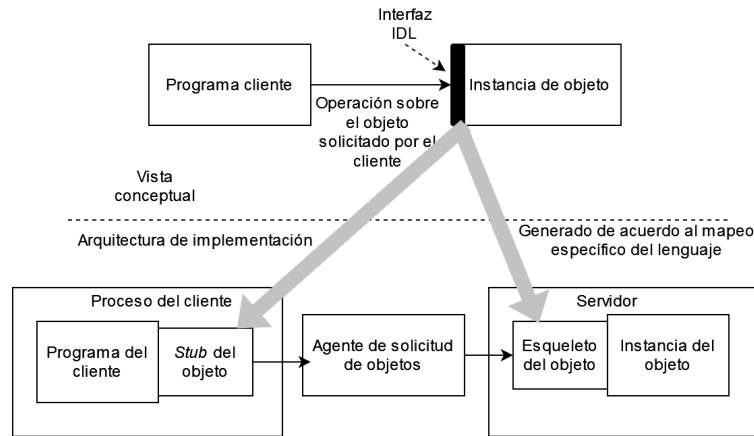


Figura 5.8: Diagrama de objetos distribuidos (Taylor et al., 2009).

- Comportamiento.
- Interacción.
- Requerimientos no funcionales.

La arquitectura de un sistema de software debe ser una composición e interacción de diferentes elementos: procesos, datos e interacción. Dada la evolución en el desarrollo de software, actualmente se identifican estilos arquitectónicos que cuentan con los siguientes elementos, en la Figura 5.8 se puede observar la arquitectura de un sistema de objetos distribuidos:

- Inclusión de elementos básicos específicos (por ejemplo, componentes, conectores, interfaces).
- Restricciones en las interacciones.
- Restricciones de comportamiento.
- Restricciones de simultaneidad.

Un aspecto importante a considerar en el modelo arquitectónico son los requerimientos no funcionales, los cuales representan restricciones en la manera en que el sistema implementa y entrega su funcionalidad, algunos ejemplos de estos requerimientos son:

- Eficiencia.
- Complejidad.
- Escalabilidad.
- Heterogeneidad.
- Adaptabilidad.

- Confiabilidad.

Estos requerimientos representan un gran desafío en el diseño, a considerar:

- Son entendidos parcialmente en varios dominios.
- Enfoque en lo cualitativo vs cuantitativo.
- Frecuentemente multidimensionales.
- Sujetos a presiones no técnicas (por ejemplo, tiempo de comercialización o características funcionales).

En la propuesta del método de la sección 6 no se reporta modelado arquitectónico, dado que no se cuenta con propuestas previas de diseño que proporcionen una pauta a seguir, además de que los lenguajes naturalísticos actuales tienen capacidades muy limitadas. No es claro aún cómo se podrían implementar los requisitos no funcionales, aunque la inclusión de circunstancias en SN es un elemento único que lo haría parcialmente posible. Tampoco es claro cómo aplicar un estilo arquitectónico o resolver la modularidad, por ejemplo, SN sólo permite trabajar con un archivo de código fuente al mismo tiempo, lo cual es una fuerte limitante.

Conforme se avance en la comprobación y validación del método propuesto, es probable que se cuente con elementos para proponer un modelo de arquitectura al menos superficial, ya que primero se tendrían que identificar los componentes naturalísticos y los mecanismos de comunicación entre ellos.

5.2.2. Diseño detallado

El diseño de software es una actividad creativa, en donde el propósito es especificar la mejor solución a un problema, y producir una descripción de cómo se debe organizar, es decir, crear un plan que se utilice para construir el sistema. Un plan de diseño debería incluir lo siguiente, considerando a la modularidad como una herramienta importante en la construcción del software:

- La estructura estática del sistema, incluyendo cualquier subprograma a ser usado en su jerarquía.
- Cualquier objeto de datos a ser utilizado.
- Los algoritmos a emplear.
- El empaquetado del sistema, en términos de cómo se agrupan los componentes.
- Interacciones entre componentes.

Existen elementos básicos que deben considerarse en el diseño de software, los cuales se explican a continuación.

5.2.2.1. Principios de diseño

Los principios de diseño de software son nociones clave que proporcionan la base para muchos enfoques y conceptos de diseño de software diferentes ([IEEE, 2014](#)):

- Acoplamiento y Cohesión. El acoplamiento se define como “una medida de la interdependencia entre módulos en un programa de computadora”, mientras que la cohesión se define como “una medida de la fuerza de asociación de los elementos dentro de un módulo” ([ISO/IEC/IEEE, 2017](#)).
- Descomposición y modularización.
- Encapsulación y ocultamiento de información, es decir, agrupar y empaquetar los detalles internos de una abstracción y hacer que esos detalles sean inaccesibles para entidades externas.
- Separación de interfaz e implementación.
- Suficiencia e integridad. Asegurarse de que un componente de software captura todas las características importantes de una abstracción y nada más.
- Separación de asuntos. Un asunto de diseño es un área de diseño que es relevante para uno o más de sus interesados. Cada vista de la arquitectura enmarca uno o más asuntos.

Además de los principios de diseño, existen una serie de cuestiones clave al diseñar software. Algunas son asuntos de calidad que todo software debe abordar, por ejemplo: rendimiento, seguridad, fiabilidad, usabilidad, etc. Otras cuestiones “tratan con algún aspecto del comportamiento del software que no está en el dominio de la aplicación, pero que se ocupa de algunos de los dominios de apoyo” ([Bosch, 2000](#)). Tales problemas, que a menudo cruzan la funcionalidad del sistema, se han referido como **aspectos**, los cuales “tienden a no ser unidades de la descomposición funcional del software, sino más bien propiedades que afectan el rendimiento o la semántica de los componentes de maneras sistémicas” ([Kiczales et al., 1997](#)). Algunos de estos aspectos son:

- Concurrencia.
- Control y manejo de eventos.
- Persistencia de datos.
- Distribución de componentes.
- Manejo de errores y excepciones, así como tolerancia a fallos.
- Seguridad en el acceso a la información y los datos.

5.2.2.2. Atributos de calidad (requisitos no funcionales)

Varios atributos contribuyen a la calidad de un diseño de software. Existe una distinción interesante entre los atributos de calidad discernibles en tiempo de ejecución (por ejemplo, rendimiento, seguridad, disponibilidad, funcionalidad, usabilidad), aquellos que no se pueden discernir en tiempo de ejecución (por ejemplo, modificabilidad, portabilidad, reutilización, capacidad de prueba) y aquellos relacionados con las cualidades intrínsecas de la arquitectura (por ejemplo, integridad conceptual, corrección, integridad) ([IEEE, 2014](#)).

Todos estos principios, aspectos y atributos de calidad, además de las restricciones¹¹ (relativas a la organización, recursos, experiencia, reutilización de software existente) y cierto conocimiento del dominio del problema, deben considerarse en el plan de diseño, en algunos métodos se consideran también los riesgos asociados así como su plan de contingencia.

5.2.2.3. Abstracción

La abstracción es esencial en el proceso de diseño, ya que permite al diseñador concentrarse en las características más importantes de un problema. En el contexto del diseño de software, una representación es utilizada para proporcionar una abstracción particular de las características de un sistema, entre sus propósitos se encuentran ([Budgen, 2003](#)):

- Capturar las ideas del diseñador.
- Explicar las ideas del diseñador a otros.
- Verificar la consistencia y completitud de la solución.

Cada método de diseño hace uso de un conjunto particular de representaciones para formar las entradas y salidas de las diversas transformaciones que utiliza. Dentro de este contexto, un punto de vista de un modelo de diseño, es un medio para capturar un conjunto particular de propiedades de diseño, el cual se proyecta mediante el uso de una representación. Las representaciones de diseño pueden ser clasificadas de la siguiente manera ([Budgen, 2003](#)):

- Formas constructivas (o estructurales), en las cuales el punto de vista se ocupa de aspectos estáticos del sistema.
- Formas de comportamiento, un conjunto de puntos de vista que buscan describir los vínculos causales entre eventos y respuestas del sistema durante su ejecución.
- Formas funcionales, puntos de vista que buscan describir lo que el sistema hace en términos de sus tareas.
- Formas de modelado de datos, relacionados con los objetos de datos utilizados dentro del sistema, y las relaciones entre ellos.

El uso de texto, diagramas y expresiones matemáticas suelen ser las tres formas básicas en la construcción de representaciones de diseño.

¹¹Restricción de diseño. Un requisito de software que afecta o limita el diseño de un sistema o un componente del sistema de software ([ISO/IEC/IEEE, 2017](#)).

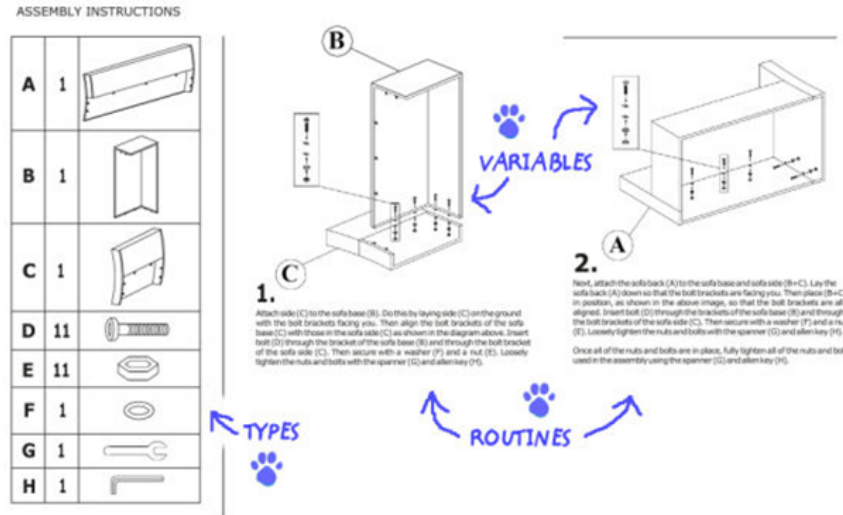


Figura 5.9: Instrucciones de ensamblaje en el mundo real (proporcionada por Gerry Rzeppa).

5.2.2.4. Formas de representación del diseño

De acuerdo a [Budgen \(2003\)](#), las representaciones del diseño pueden ser de caja negra (para el diseño arquitectónico) o de caja blanca (para el diseño detallado), las de caja negra indican vagamente el qué se quiere y las de caja blanca indican el cómo. Para propósitos de este trabajo, todas las formas de representación del diseño en UML se descartan por corresponder al POO.

De acuerdo a G. Rzeppa (comunicación personal, 14 de noviembre, 2020), autor de Cal-4700 y Sal-1016 (versión en español de Cal), la programación en inglés plano se asemeja a la programación procedural, ambos lenguajes se construyeron para entender oraciones de la forma más normal y natural en la que la gente indica instrucciones a otros, en este caso, a la computadora, a través de instrucciones procedurales, utilizando tipos y variables, como se ve en el folleto de instrucciones de ensamblaje de un mueble, en la Figura 5.9.

Por esta razón, se consideran como base las siguientes representaciones de caja negra, las cuales se utilizan desde hace décadas, en el modelado de software procedural:

- **Diagramas de flujo de datos (DFD):** desde el punto de vista funcional, se utilizan para mostrar el flujo de datos entre elementos. Un diagrama de flujo de datos proporciona “una descripción basada en modelar el flujo de información alrededor de una red de elementos operativos, y cada elemento hace uso o modifica la información que fluye hacia ese elemento” ([Budgen, 2003](#)). Un ejemplo se puede observar en la Figura 5.10.

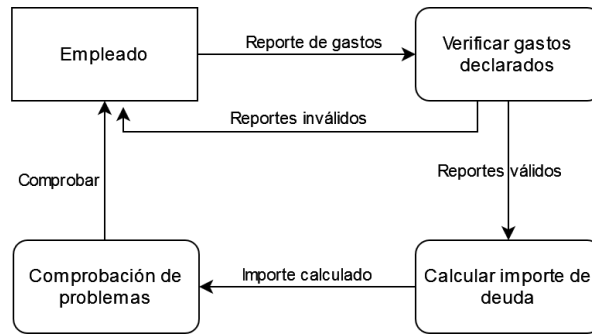


Figura 5.10: DFD Verificar reporte de gastos (ISO/IEC/IEEE, 2017).

- **Diagramas de transición de estado y diagrama de estado:** desde el punto de vista del comportamiento, se utilizan para mostrar el flujo de control de un estado a otro y cómo cambia el comportamiento de un componente en función de su estado actual en una máquina de estado (IEEE, 2014). Un ejemplo del diagrama de transición de estados se observa en la Figura 5.11.

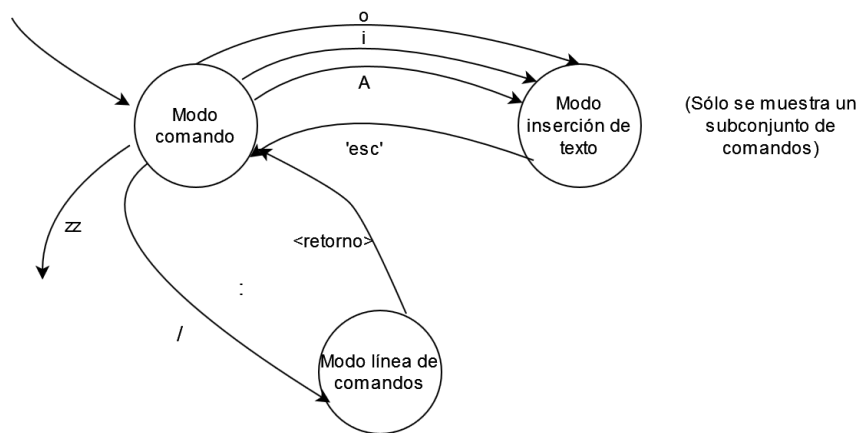


Figura 5.11: Descripción de transición de estados del editor vi de Unix (Budgen, 2003).

- **Diagrama entidad-relación (DER):** permite modelar las relaciones estáticas entre las entidades de datos. Los DER se utilizan ampliamente para desarrollar el esquema de bases de datos relacionales (Budgen, 2003). Un ejemplo se puede observar en la Figura 5.12.
- **Diagrama de estructura de Jackson (DEJ):** desde el punto de vista funcional, de comportamiento y de modelado de datos, un diagrama de estructura de Jackson trata básicamente de describir la estructura secuencial de la solución, en términos de las tres formas de estructuración clásicas: secuencia, selección e iteración. Estas características lo hacen único, ya que la mayoría de notaciones se enfocan a representar sólo un punto de vista. En la Figura 5.13 se aprecia un ejemplo.

Las figuras se incluyeron tal y como están en su fuente original, a manera de ejemplo¹².

¹²Los diagramas de la versión definitiva de la tesis se traducirán al español

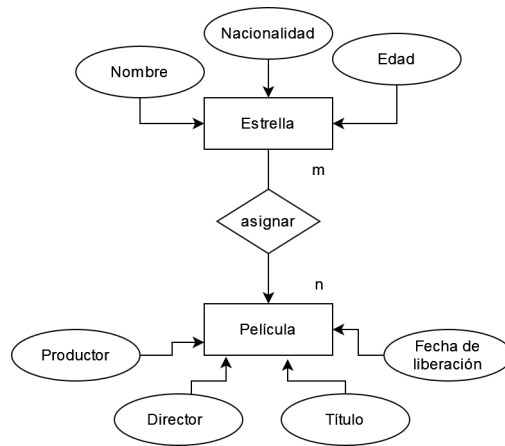


Figura 5.12: DER de dos clases de entidad reconocibles (Budgen, 2003).

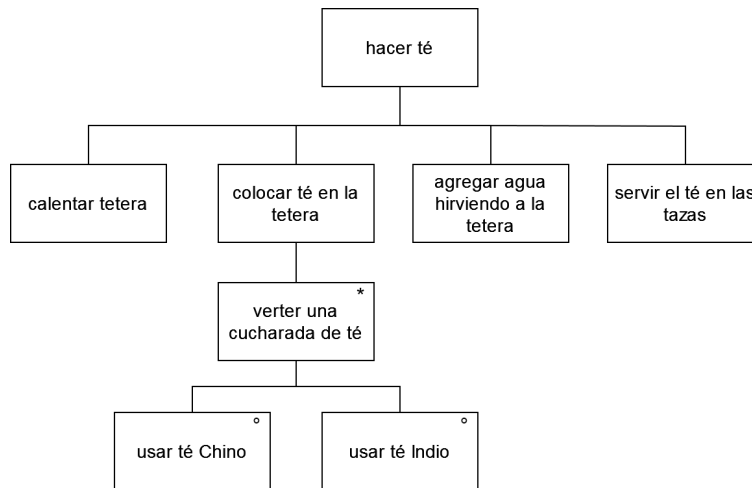


Figura 5.13: DEJ que describe la operación de una máquina de té (Budgen, 2003).

Para la representación de diseño de caja blanca, que en POO podría ser un diagrama de clases y objetos o un diagrama de secuencia, se puede considerar que dadas las ventajas de la programación naturalística, se limitaría a la creación de un pseudocódigo, el cual podría ser prácticamente el programa naturalístico, con algunos ajustes.

El estado actual de los lenguajes naturalísticos de propósito general, dificulta considerar por el momento aspectos como atributos de calidad, restricciones y riesgos, así como hacer énfasis en el diseño de interfaces de usuario, ya que los lenguajes existentes se encuentran en un modo elemental, primario.

Para documentar el diseño detallado, por el momento se considera como mínimo cumplir la recomendación del estándar 29110 ISO/IEC (2016), perfil básico, el cual describe una sola aplicación para un solo equipo de proyecto:

- Desarrollar el diseño detallado de cada componente de software, describiendo las unidades de software que se van a construir.
- Definir las interfaces externas de cada unidad de software.

- Establecer la consistencia y trazabilidad entre el diseño detallado, los requisitos y el diseño arquitectónico.

En el caso del Diseño de software, los lenguajes de programación naturalística actuales, presentan muchas limitaciones que impiden considerar la mayoría de los elementos mencionados en este capítulo, por el momento se consideraron:

- Diagrama de estructura de Jackson, ya que permite reflejar la estructura secuencial de la solución en términos de las tres formas de estructuración clásicas: secuencia, selección e iteración. Coincide con la forma de pensamiento procedural del paradigma naturalístico.
- Diagrama de flujo de datos (DFD). En este caso el DFD se trasladará al análisis para permitir un paso más natural entre las descripciones de Casos de Uso y los diagramas de estructura.
- Diagrama de datos. De igual manera, aunque puede ser parte del Diseño de software de acuerdo a lo presentado en este capítulo, se trasladará al Análisis para considerar desde la etapa más temprana, los sustantivos necesarios que serán referenciados como anáforas dentro del código naturalístico.

Capítulo 6

Método de desarrollo de software naturalístico

De acuerdo a [Pressman \(2010\)](#), los principios del modelado de software sugieren que los modelos tengan un propósito, que sean simples, sin crear más de lo que se necesita, que sean modificables, y que no se espere crear modelos perfectos. Es importante evitar limitarse a utilizar sólo lo que se conoce o domina, permaneciendo en una zona de confort, también es importante evitar considerar propuestas sólo por cambiar la forma de programar. En este caso fue necesario analizar las propuestas que coinciden con el pensamiento naturalístico, teniendo en cuenta los principios de Análisis y Diseño de Software presentados en [5.1](#) y [5.2.2](#), respectivamente. Existen varias limitaciones técnicas impuestas por los lenguajes naturalísticos actuales, por lo que la consideración de algunos aspectos de modelado que incluyen la arquitectura y requerimientos no funcionales se abordarán como trabajo a futuro.

El método propuesto, presentado en la Figura [6.1](#), parte de la concepción de *ideas* que representan el pensamiento humano, término propuesto y utilizado en la construcción de Pegasus ([Knöll & Mezini, 2006](#)) para la traducción de lenguaje natural a código fuente. La notación de idea representa una fortaleza que se consideró por el momento para promover la trazabilidad y validación de los requerimientos, como se explicará más adelante. El método considera el uso de artefactos que propician el pensamiento naturalístico, es decir, que el programador conciba sus ideas y las transcriba en lenguaje natural, utilizando una gramática reducida. A continuación se explican los pasos del método.

6.1. Análisis de Requerimientos

Dentro de la actividad del Análisis de Requerimientos existen dos etapas: Identificación de elementos y Realización. Dentro de la etapa de **Identificación** se encuentran los siguientes pasos, los cuales es posible que se realicen en diferente orden, dependiendo de la información con la que se cuente:

- **I.1.** e **I.2.** Identificar ideas atómicas, compuestas y sus connotaciones. El término *idea* se propuso en ([Knöll & Mezini, 2006](#)), cuyos autores lo conceptualizaron para que fuera interpretado por la computadora en el proceso de traducción. De acuerdo a su interpretación, las percepciones humanas constituyen un conjunto de ideas, y

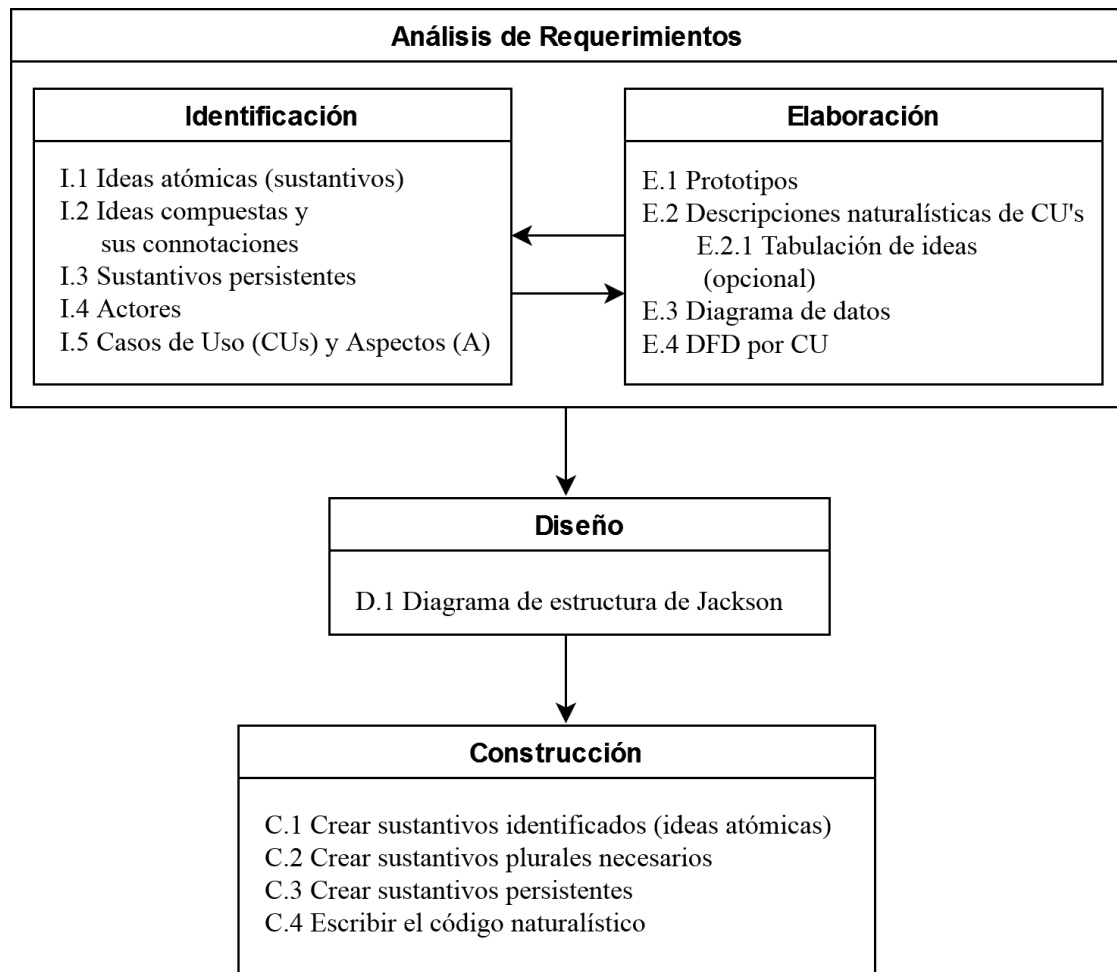


Figura 6.1: Método de Desarrollo Naturalístico.

pensar significa componer estas ideas. Por lo general, todos los pensamientos se pueden expresar en lenguaje natural. Las ideas pueden ser atómicas (por ejemplo un sustantivo), complejas (caracterizadas por asociaciones entre ideas atómicas y complejas) o compuestas, las cuales representan pensamientos, combinando varias ideas complejas por un breve momento. La notación de idea se presenta en la Figura 6.2.

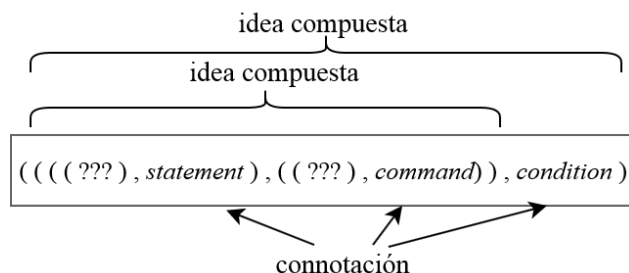


Figura 6.2: Notación de *idea*.

Esta notación indica que se ha encontrado una idea compuesta y que la idea atómica *condition* es una de sus connotaciones. La otra connotación es en sí misma una idea compuesta, constituida a su vez por otras dos ideas, una con la connotación *statement* y la otra con la connotación *command*. Estas connotaciones las agrega el lector del lenguaje al analizar la oración. Los autores llaman *patrones de idea* a las ideas que contienen el marcador de posición “???”. Para la idea:

If the first element of the second row of the matrix is smaller than 3 then write “I can understand you!”

su representación de acuerdo a la notación anterior se observa en la Figura 6.3.

```
((
  ((
    (be, predicate),
    ((element, first, (row, second, (matrix))), subject),
    (smallness, comparative),
    ((three, than), object)
  ), statement),
  ((
    (writing, predicate),
    (you, subject),
    (string, object)
  ), command)
), condition)
```

Figura 6.3: Ejemplo de una idea en Pegasus.

Las ideas compuestas, en lenguaje natural, formarán parte de la descripción de cada CU, (flujos de los caminos normal, alternos y excepciones). Aunque la notación de la Figura 6.2 es usada específicamente para el proceso de traducción a código fuente por Knöll & Mezini (2006) y Mefteh et al. (2018), es una concepción naturalística

importante a considerar, por lo que se propone retomarla con propósitos de validación de la trazabilidad de las ideas.

En los lenguajes naturalísticos como SN y Cal-4700, la unidad mínima es *Thing*, por lo que las ideas atómicas identificadas pasarán a ser sustantivos o cosas dentro del lenguaje naturalístico seleccionado.

- **I.3.** e **I.4.** Identificar cuáles de estos sustantivos serán actores del sistema y cuáles de ellos serán persistentes (que permanezcan aún después de concluido el programa). Es importante identificar cuáles de los sustantivos sólo son actores del sistema, sin tener ningún otro papel como dato o cosa (*Thing*) dentro del programa.
- **I.5.** Identificar los Casos de Uso (CU) y cuáles de ellos serán aspectos (*concerns*) que hacen un corte transversal en la funcionalidad del sistema). Además de su amplio uso en el desarrollo de software, la técnica de CU muestra el uso de descripciones en lenguaje natural para una mejor comprensión de los requerimientos por parte de los *stakeholders*, característica que concuerda con la filosofía del Paradigma Naturalístico (PN). La manera en la que se describen los CU (actor-acción-objeto), facilitará su comprensión y el paso de una fase a otra.

Para la etapa de **Elaboración** se propone:

- **E.1.** La creación de prototipos no funcionales por cada Caso de Uso, esto como actividad protectora de validación de requerimientos desde etapas tempranas.
- **E.2.** La redacción naturalística de los flujos normal, alternos y excepciones para cada CU. Esta redacción naturalística puede incluir anáforas, uno de los principales elementos del paradigma.
- **E.2.1** De manera opcional, se propone utilizar una representación tabular de las ideas, de manera que facilite el seguimiento y validación hasta su construcción.
- **E.3.** Se incluye la elaboración de un Diagrama de Datos, ya que se necesita un esquema de persistencia. Dadas las limitaciones tecnológicas actuales, aún no es factible considerar pasos específicos para su implementación y por lo tanto modelado naturalístico. Es posible que el Diagrama de Datos corresponda a un diagrama Entidad-Relación o diagrama relacional de la Base de Datos correspondiente, en el caso de SN. Para el caso de Cal-4700, el cual sólo permite la persistencia a través del uso de archivos de texto plano, se podrá incluir un esquema que refleje la estructura y/o relación de estos archivos.
- **E.4.** Se propone realizar un Diagrama de Flujo de Datos (DFD) por cada CU, para mostrar de manera gráfica las acciones necesarias, los actores y datos involucrados. Por el momento el principal referente es el DFD de [Yourdon \(1993\)](#), ya que su notación es sencilla e incluye los elementos antes mencionados. Las descripciones de cada burbuja de un DFD se escriben en lenguaje natural abreviado, facilitando la comprensión y la trazabilidad.

6.2. Diseño y Construcción

Para la actividad de **Diseño** se propone

- **D.1.** Realizar el Diagrama de Estructura de [Jackson \(2002\)](#), el cual trata de describir la estructura secuencial de la solución, en términos de las tres formas de estructuración clásicas: secuencia, selección e iteración. De acuerdo a [Budgen \(2003\)](#), estas características lo hacen único, ya que la mayoría de notaciones se enfocan a representar sólo un punto de vista. De igual manera, su representación utiliza frases en lenguaje natural, acorde con el enfoque naturalístico.

Finalmente, para la fase de **Construcción**, por el momento sólo se presentan algunas pautas como:

- **C.1.** Crear los sustantivos identificados como ideas atómicas. Aunque cada lenguaje naturalístico provee su propia manera de crear los sustantivos, una vez que fueron identificados en el paso I.1, estos permitirán ser referenciados como anáforas dentro del código naturalístico, de acuerdo a la descripción de cada CU y a la tabulación de ideas realizadas en los pasos E.2 y E.2.1.
- **C.2.** Crear los plurales necesarios. Si dentro de las tablas generadas en los pasos E.2 y E.2.1 se identificaron sustantivos que serán utilizados como colecciones de datos, estos deberán ser definidos como sustantivos plurales dentro del código naturalístico, de igual manera para ser referenciados como anáforas dentro del código naturalístico.
- **C.3.** Crear los sustantivos persistentes, en caso de ser necesario, con base en la propuesta de cada lenguaje. En el caso de SN, podrá utilizarse una Base de Datos relacional, con la sintaxis de acceso correspondiente. En el caso de Cal-4700, podrán utilizarse archivos de texto plano, así como las rutas de acceso y búfers correspondientes dentro del código naturalístico para manejar la persistencia de los datos.
- **C.4.** Escribir el resto del código naturalístico. Dependiendo del problema a resolver, se escribirá el resto de procedimientos necesarios.

Capítulo 7

Pruebas de concepto del método

7.1. Sistema de Administración de Cursos (CMS)

Para ejemplificar el método propuesto, se consideró el modelado de un sistema de administración de cursos (*Course Management System (CMS)*). En (Baniassad & Clarke, 2004), los autores presentan el modelado Orientado a Aspectos para este sistema, lo cual permite tener un punto de referencia de los elementos considerados (recordar que el POA es ortogonal al POO). Además de contar con el modelo de Análisis y Diseño Orientado a Aspectos, y por ende, Orientado a Objetos, se cuenta con un Caso de Uso codificado en SN (sin acceso a BD), lo que sirvió como punto de partida (o regreso) hacia el diseño y el análisis de los requerimientos. El sistema completo se codificó en Cal-4700, ya que es el lenguaje más estable según el análisis realizado en el Capítulo 2. La herramienta de modelado utilizada fue draw.io¹, la cual cuenta con el conjunto de notaciones necesarias para realizar este ejercicio.

Los requerimientos del CMS se listan a continuación:

- R1. Los estudiantes pueden inscribirse a cursos.
- R2. Los estudiantes pueden darse de baja de cursos.
- R3. Cuando un estudiante se inscribe, esto debe ser registrado en un archivo de seguimiento.
- R4. Cuando un estudiante borra su inscripción, esto también debe ser registrado en un archivo de seguimiento.
- R5. Los profesores pueden dar de baja estudiantes.
- R6. Cuando un profesor da de baja un estudiante, también debe ser registrado.
- R7. Cuando un profesor da de baja a un estudiante, éste debe ser marcado como especial.
- R8. Los profesores pueden calificar a los estudiantes.

¹<https://app.diagrams.net/>

- R9. Cuando un profesor asienta una calificación, esto también debe ser registrado.

El diagrama de Casos de Uso (CU) para el CMS se puede observar en la Figura 7.1. Se identificaron dos actores: **Estudiante** y **Profesor**, además de los CU **Inscribir estudiante**, **Borrar inscripción de estudiante** y **Calificar estudiante**. Los requerimientos R3, R4, R6 y R9 son transversales al resto de los requerimientos, puesto que todos implican el registro de las actividades resultantes en un archivo de seguimiento. Éstos requerimientos se encapsulan en un aspecto identificado como el CU **Registrar seguimiento**, incluido en el resto de los CU con el estereotipo <<trigger>>. Jacobson & Ng (2004) proponen representar los aspectos por medio de la extensión de CU, sin embargo, ésta propuesta se contrapone con el modelado de CU 2.0, también propuesto por Jacobson et al. (2016), en donde un CU se divide en porciones y éstas se manejan como extensiones, razón por la cual se decidió incluir el estereotipo <<trigger>>, para evitar confusiones. En este ejercicio no hubo necesidad de dividir los CU en porciones.

Los registros de tiempos y defectos para este ejercicio se incluyen en el Apéndice ??, esto permitirá a corto plazo analizar y comparar los tiempos de desarrollo y de retrabajo con respecto a otras propuestas.

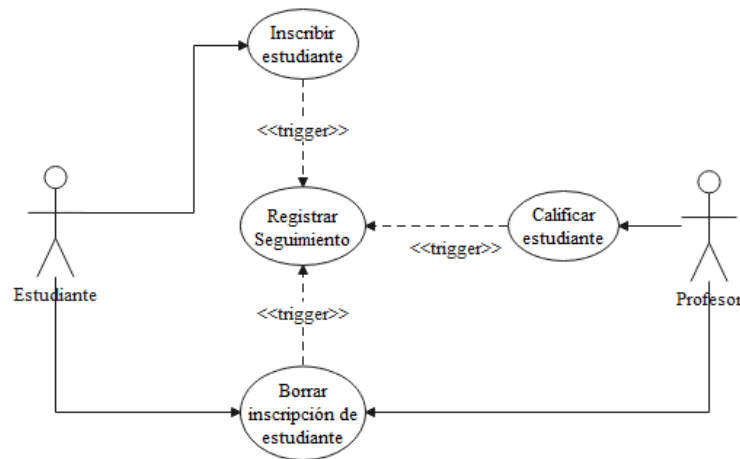


Figura 7.1: Diagrama de CU Registro de cursos.

7.1.1. Caso de Uso Inscribir estudiante

La especificación del CU se puede observar en la Tabla 7.1, la cual incluye los flujos básico y alternos, elementos mínimos necesarios que se usaron para el modelado. La descripción está basada en la plantilla propuesta por Wiegers & Beatty (2013). En la descripción del flujo normal se aprecia el uso de referencias anafóricas (pasos 2, 3 y 5), elemento distintivo de la programación naturalística y por lo tanto del lenguaje natural. La validación de las excepciones de cada caso de uso (verificar si existe el estudiante y curso introducidos) quedaron pendientes para una segunda versión. El prototipo correspondiente para este CU se observa en la Figura 7.2.

Tabla 7.1: Especificación del CU Inscribir.

ID:	CU01
Nombre:	Inscribir estudiante
Autor(es):	Lizbeth A. Hernández González
Fecha de creación:	15-Abr-2021
Fecha de actualización:	30-Abr-2021
Actor(es):	Estudiante
Descripción:	El Estudiante podrá seleccionar los cursos deseados, de la lista proporcionada.
Flujo Normal:	1. Sistema despliega lista de cursos. 2. Éste solicita número de control. 3. Estudiante lo introduce. 4. Sistema solicita número de curso. 5. Estudiante lo introduce. 6. Sistema guarda número de control y número de curso en inscripciones. 7. Sistema pregunta ¿Deseas registrar otro curso? 7.1 Si: regresar a paso 4. 7.2 No: termina CU. 8. Sistema imprime los cursos inscritos.
Flujos Alternos:	
Excepciones:	E.1 Cupo lleno en curso 1. Sistema despliega “Curso sin lugares disponibles” 2. Regresar a paso 6
Poscondiciones:	Registrar al Estudiante en el archivo de seguimiento, junto con los cursos inscritos. Indicar la transacción realizada.
Entradas:	El número de curso y el número de control del estudiante.
Salidas:	Mensaje de confirmación.
Incluye: (relación Include)	
Extiende: (relación extend)	
Lanzar: (relación trigger)	Registrar en archivo Seg
Prioridad:	Alta

Registro de cursos
3. NodeJS 2. JavaScript 1. MongoDB 0. Salir Introduce número de control: 13011074 Selecciona un curso: 1 Registro exitoso! ¿Deseas inscribir otro curso? 1-Si 0-No 0 [Estudiante con número: 13011074 inscrito a curso:1]

Figura 7.2: Prototipo CU Inscribir.

La representación de las ideas compuestas, es decir, los requerimientos del usuario incluidos en la descripción de los caminos del CU, se muestra en la Tabla 7.2. Para cada idea, se debe identificar si su connotación es de enunciado, comando o condición; el agente quien realiza la acción, la relación anafórica en caso de haberla, así como el objeto directo sobre el cual recae la acción; el destino es opcional. Por ejemplo, el paso 5 “Estudiante lo introduce”, hace referencia al número de curso mencionado en el paso 4. En la tabla se representa como una idea del tipo **Enunciado**, cuya acción implica **Capturar**, el agente es el **Estudiante** y la referencia anafórica es **numero_curso**. Los pasos 7.1 y 7.2 responden al

deseo de registrar o no otro curso para el estudiante, ambos son identificados como Comandos **Ejecutar** y **Salir** respectivamente, cuyo agente es el sistema. El objeto directo para la acción 7.1 es el paso 4, esto significa que el usuario decidió agregar otro curso y se le debe solicitar el número de curso correspondiente. La representación de las ideas de esta manera permitirá la trazabilidad de los requerimientos, ya que facilitará la creación de casos de prueba para la verificación y validación correspondientes.

Tabla 7.2: Tabulación de ideas CU Inscribir.

#	Enunciado/ Statement (acción)	Comando/ Command (frase)	Condición / Condition	Agente	Anáfora	Objeto directo/ direct object *para plurales	Destino (opcional)
1	Desplegar			Sistema		Curso*	
2	Solicitar			Sistema		numero_control	
3	Capturar			Estudiante	numero_control		
4	Solicitar			Sistema		numero_curso	
5	Capturar			Estudiante	numero_curso		
6	Guardar			Sistema		numero_control numero_curso	Inscripciones
7			Preguntar	Sistema		Curso	
61		Si: Ejecutar		Sistema		Paso 4	
62		No: Salir		Sistema			
8	Imprimir			Sistema	Cursos inscritos		

Como parte del análisis del sistema en su totalidad, y una vez que se tabularon las ideas compuestas, se identificaron las ideas atómicas de la Figura 7.3. Se aprecia que coinciden con las entidades identificadas en el Diagrama Entidad-Relación de la Figura 7.4. El diagrama proporcionó una referencia para identificar la estructura de los datos a almacenarse dentro de archivos de texto. Por el momento, se parte del supuesto de que se cuenta con una lista de estudiantes y cursos.

Cada idea atómica será parte del código naturalístico en Cal-4700 como un registro o como un identificador, además de que serán parte de la solución como archivos de texto para almacenar de manera persistente los datos de los estudiantes, cursos, inscripciones y el seguimiento. Las entidades de datos se ven reflejadas en los diagramas de análisis correspondientes.

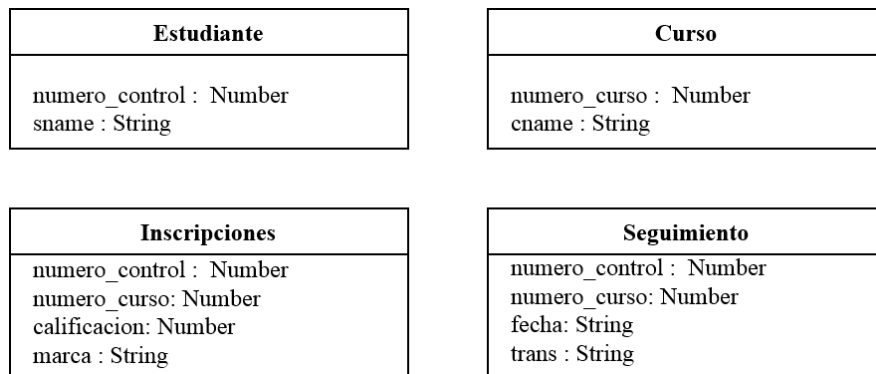


Figura 7.3: Ideas atómicas identificadas.

El Diagrama de Flujo de Datos (DFD) se representa en la figura 7.5(a), en él se observan: el actor involucrado, las acciones y almacenes correspondientes, además del aspecto **Registrar**

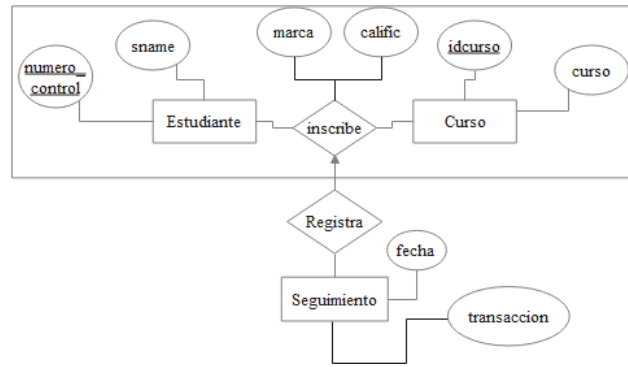


Figura 7.4: Diagrama Entidad-Relación.

seguimiento, representado con una línea punteada, manteniendo la trazabilidad del mismo. El diagrama de estructura de Jackson se muestra en la Figura 7.5(b), en él se observa repetidamente el aspecto de **Seguimiento**, nuevamente identificado con una relación del tipo <<trigger>>.

Finalmente, el código fuente original (en SN) se observa en el Listado 7.1, se aprecia el uso del sustantivo plural **Estudiante** (línea 1), el uso de referencias anafóricas (líneas 8, 9, 24, 28 y 34) así como el uso de una circunstancia para llevar el control del registro de las actividades (línea 10). El programa utilizando Cal-4700 con persistencia de datos a través de archivos de texto, se observa en el Listado 7.3. De acuerdo a las intenciones del PN, el código naturalístico debería autodocumentarse, es decir, el lector debería ser capaz de comprenderlo sin necesidad de una explicación exhaustiva. Queda a su juicio evaluar el nivel de expresividad que cada lenguaje proporciona.

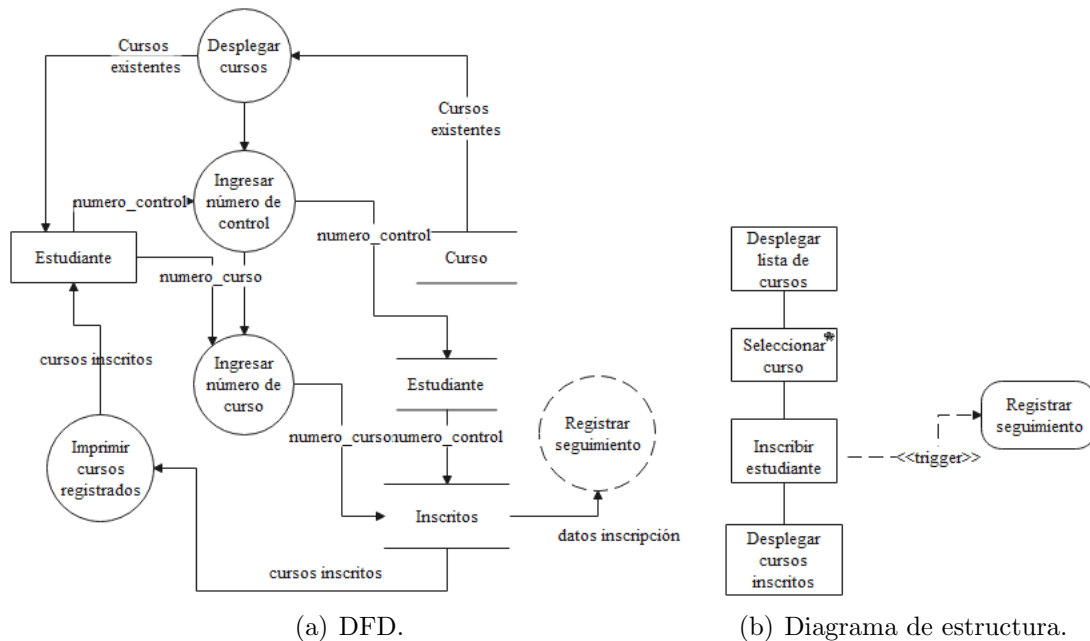


Figura 7.5: Diagramas de análisis(a) y diseño(b) del CU Inscribir.

Listado de código 7.1: CU Inscribir usando SN.

```
1 noun Estudiante with plural as Estudiantes:
2   attribute numero_control as a Number.
3   attribute idcurso as a Number.
4 verb itself registrar:
5   System prints "Registro exitoso" and newline.
6 overridden derived attribute string as a String:
7   a String with "" as value.
8   add "Numero control: " to it; and add numero_control to it.
9   add "Curso: " to it; add idcurso to it; and return the String.
10 circumstance: it registrar when this is created.
11
12 main MainCourse:
13   opcion is -1.
14   otro is 0.
15   numero is 0.
16   a Estudiante.
17   some Estudiantes.
18   System prints "3.-NodeJS" and newline.
19   System prints "2.-JavaScript" and newline.
20   System prints "1.-MongoDB" and newline.
21   System prints "0.-Salir" and newline.
22   System prints "Ingresa numero de control" and newline.
23   System reads.
24   numero is the integer of it.
25   repeat the next 9 instructions until option is equal to 0.
26     System prints "Selecciona un curso".
27     System reads.
28     otro is the integer of it.
29     execute the next 5 instructions when otro is distinct to 0.
30     a Estudiante with numero as numero_control and otro as curso.
31     add it to the Estudiantes.
32     System prints "Deseas registrar otro curso? 1.-Si 0.-No" and newline.
33     System reads.
34     opcion is the integer of it.
35     System prints the string of the Estudiantes and newline.
```

Los proyectos de Cal-4700 se trabajan en una sola carpeta, en ella es necesario que se encuentre el archivo cerebro *the noodle*, el cual contiene todas las definiciones y rutinas del lenguaje. Los archivos de código fuente que existan en la carpeta deben carecer de extensión, y al compilarse como un todo, permiten separar la solución en varios archivos como una opción de modularización. Al compilar cualquiera de los programas de la carpeta, se generará un archivo ejecutable cuyo nombre será igual al nombre de la misma carpeta del proyecto. En este caso la carpeta del proyecto se llama CMS y contiene:

- Archivo **inscribir**, contiene las rutinas que permiten registrar la inscripción de un estudiante en el archivo de texto **Inscripciones.txt**.
- Archivo **borrar**, contiene las rutinas que permiten dar de baja de un curso a un estudiante. De acuerdo con los requerimientos, el estudiante se marca como baja, en este caso se antepone la letra B dentro de las inscripciones. Como se mencionó anteriormente, el diagrama de datos proporcionó un punto de referencia para definir la estructura

de los datos persistentes, sin embargo, al trabajar con archivos de texto, se decidió manejar cada curso de inscritos de manera independiente, facilitando la calificación de estudiantes, el acceso y lectura desde el exterior. Se parte entonces del supuesto de que existe un proceso que divide las inscripciones en diferentes archivos de texto (omitiendo los estudiantes marcados con B), de acuerdo al curso.

- Archivo `calificar`, contiene las rutinas que permiten registrar la calificación de un estudiante.
- Archivo `principal`, contiene las rutinas necesarias de inicio y fin del programa (To run: start up. shut down.), las cuales son necesarias para crear el programa ejecutable y llamar al resto de las rutinas de cada CU.
- archivo `aspectoSeg` contiene las rutinas correspondientes al seguimiento del sistema.

Los archivos de texto de los datos persistentes también se encuentran en esta carpeta (`Estudiante.txt`, `Curso.txt`, `Inscripciones.txt`, `Seguimiento.txt`, así como los archivos correspondientes a cada curso de inscritos finales, `1.txt`, `2.txt`, `3.txt`).

El nombre del archivo ejecutable es `cms.exe`.

El programa principal contiene la rutina `To run` (ver Listado 7.2), se aprecia la lectura del programa como si fuera un pseudocódigo, facilitando la comprensión del mismo.

En el programa del Listado 7.3, el cual permite inscribir estudiantes, se observa el registro `Student`, el cual contiene los campos `control_number` y `course_number`, los cuales se pueden acceder mediante el uso de posesivos (líneas 14 y 17), como se hace en lenguaje natural (en Inglés). La persistencia se logra a través del uso de un búfer auxiliar que almacena los registros durante la ejecución del programa (rutina de la línea 33). Antes de terminar, el búfer se guarda en un archivo de texto llamado `inscripciones.txt` (línea 24, rutina definida en las líneas 26-31). La salida del programa se observa en la Figura 7.6.

Listado de código 7.2: Programa principal.

```
1  To run:
2    Start up.
3  Loop.
4    write "BIENVENIDO AL CMS" to the console.
5    write "Selecciona la opcion a realizar" to the console.
6    skip one line on the console.
7    write "3. Calificar" to the console.
8    write "2. Borrar inscripcion" to the console.
9    write "1. Inscribir" to the console.
10   write "0. Salir" to the console.
11   skip one line to the console.
12   write "Opcion: " to the console without advancing.
13   read a string from the console.
14   if the string is "0", break.
15   if the string is "1", inscribir.
16   if the string is "2", borrar.
17   if the string is "3", calificar.
18 Repeat.
19 Shut down.
```

Listado de código 7.3: CU Inscribir.

```
1  A student has
2    a control_number string,
3    a course_number string.
4
5  To inscribir:
6    Loop.
7      write "REGISTRO DE CURSOS" to the console.
8      write "3. NodeJS" to the console.
9      write "2. JavaScript" to the console.
10     write "1. MongoDB" to the console.
11     skip one line to the console.
12     write "Ingresa numero de control:" to the console.
13     read a string from the console.
14     put the string into a student's control_number.
15     write "Selecciona un curso:" to the console.
16     read another string from the console.
17     put the other string into the student's course_number.
18     enroll the student into a buffer.
19     write "Deseas registrar otro curso? Si[1] No[0]: " to the console without advancing.
20     read a string called condition from the console.
21     if the condition is "0", break.
22     skip one line to the console.
23   Repeat.
24   save enrollments from the buffer.
25
26  To save enrollments from a buffer:
27    put "D:\cal-4700 initiative\CMS\inscripciones.txt" into a path.
28    if the path is not in the file system, create the path in the file system.
29    read the path into a buffer called original.
30    write the original then the buffer to the path.
31    If the i/o error is not blank, write the i/o error on the console.
32
33  To enroll a student into a buffer:
34    Append the student's control_number then " - " then the student's course_number
35      then the crlf string to the buffer.
36    write "Estudiante con numero " then the student's control_number then " "
37      then "inscrito en curso " then the student's course_number to the console.
38    [seguimiento]
39    put "I" into an activity's action. put the system's date/time into the activity's date.
40    put the student's control_number into the activity's control.
41    put the student's course_number into the activity's course.
42    register the activity.
```

En las líneas 39 a 42 se observa el llamado a la rutina que dará seguimiento a la actividad de inscripción, es decir, la llamada del Aspecto **Registro de seguimiento** (ver Listado 7.6). Cada CU llena una actividad con los datos correspondientes y la envía a través del llamado a la rutina **register the activity**.

7.1.2. Caso de Uso Calificar estudiante

La especificación del CU Calificar estudiante se aprecia en la Tabla 7.3. Las pantallas del prototipo correspondiente se observan en la Figura 7.7. En la tabla descriptiva se aprecia

```

BIENVENIDO AL CMS
Selecciona la opción a realizar

3. Calificar
2. Borrar inscripción
1. Inscribir
0. Salir

Opción: 1
REGISTRO DE CURSOS
3. NodeJS
2. JavaScript
1. MongoDB

Ingresa número de control:
19052
Selecciona un curso:
2
Estudiante con número 19052 inscrito en curso 2
¿Deseas registrar otro curso? Si[1] No[0]:

```

Figura 7.6: Salida CU Inscribir.

el lanzamiento del aspecto **Registrar seguimiento**, el cual debe mantener su trazabilidad hasta la implementación.

Nuevamente, cada uno de los requerimientos del flujo normal se incluye en la Tabla de ideas 7.4, esto permite identificar los registros, cosas o variables que se integrarán en el código naturalístico, junto con los datos que serán persistentes a través de archivos, por medio del objeto directo y del destino. Además se identifican los agentes involucrados, acciones y condicionales correspondientes. Como se comentó anteriormente, ésta herramienta facilita la trazabilidad de los requerimientos para el resto de las actividades.

Tabla 7.3: Especificación del CU Calificar.

ID:	CU02
Nombre:	Calificar estudiante
Autor(es):	Lizbeth A. Hernández González
Fecha de creación:	16-Nov-2021
Fecha de actualización:	
Actor(es):	Profesor
Descripción:	El Profesor selecciona un curso para registrar las calificaciones de cada uno de los estudiantes.

Tabla 7.3 continuación de la página anterior.

ID:	CU02
Flujo Normal:	1. Sistema despliega lista de cursos. 2. Sistema solicita número de curso. 3. Profesor lo introduce. 4. Sistema pregunta ¿Desea calificar al grupo entero [1] o a un estudiante [2]? 5. Si Opción 1: 5.1 Sistema despliega uno a uno número de control (de estudiantes). 5.2 Profesor registra calificación de cada estudiante. 6. Si Opción 2: 6.1 Sistema solicita número de control del estudiante. 6.2 Profesor registra calificación. 7. Sistema guarda calificaciones.
Flujos Alternos:	
Excepciones:	E.1 Estudiante no existe 1. Sistema despliega “Estudiante inexistente”. 2. Regresa a paso 1. E2. Número de curso no existe 1. Sistema despliega “Número de curso no existe”. 2. Regresa a paso 5.
Poscondiciones:	Registrar la calificación del o los estudiantes en el archivo de seguimiento, mediante su número de control, el número de curso y el tipo de transacción realizada.
Entradas:	El número de curso, si el Profesor desea calificar sólo a un estudiante, deberá ingresar su número de control.
Salidas:	Mensaje de confirmación.
Incluye: (relación Include)	
Extiende: (relación extend)	
Lanzar: (relación trigger)	Registrar en archivo Seg
Prioridad:	Alta

Tabla 7.4: Tabulación de ideas CU Calificar.

#	Enunciado/ Statement (acción)	Comando/ Command (frase)	Condición / Condition	Agente	Anáfora	Objeto directo/ direct object *para plurales	Destino (opcional)
1	Desplegar			Sistema		Curso*	
2	Solicitar			Sistema		numero_curso	
3	Capturar			Profesor	numero_curso		
4			Preguntar	Sistema			
5			Opción 1				
5.1	Desplegar			Sistema		numero_control* por estudiante	
5.2	Capturar			Profesor		calificación* por estudiante	
6			Opción 2				
6.1	Solicitar			Sistema		numero_control	
6.2	Capturar			Profesor		calificación	
7	Guardar			Sistema		calificaciones	Inscripciones

Calificar estudiantes	Calificar estudiantes
3. NodeJS 2. JavaScript 1. MongoDB 0. Salir Selecciona un curso: 1 Calificar todos [1], Seleccionar un estudiante [2]: 1 Introducir calificación para 43257: 8.3 Introducir calificación para 89542: 9.4 Introducir calificación para 45629: 8.3 Introducir calificación para 34261: 5.6 Calificaciones registradas!	3. NodeJS 2. JavaScript 1. MongoDB 0. Salir Selecciona un curso: 1 Calificar todos [1], Seleccionar un estudiante [2]: 2 Introduce número de control: 45629 Introduce calificación: 8.3 Calificación registrada!

(a) Opción Todos.

(b) Opción Un estudiante.

Figura 7.7: Prototipos CU Calificar.

En la Figura 7.8 se incluye el Diagrama de Flujo de Datos (DFD) 7.8(a), parte del análisis del sistema, además del Diagrama de estructura de Jackson 7.8(b), parte del diseño. En el DFD se observan las actividades, almacenes, actores y datos correspondientes al CU. En el Diagrama de estructura se observan las tres formas de estructuración clásicas: secuencia, selección (°) e iteración (*). En ambos se puede observar la trazabilidad de los requerimientos, incluyendo el aspecto de seguimiento.

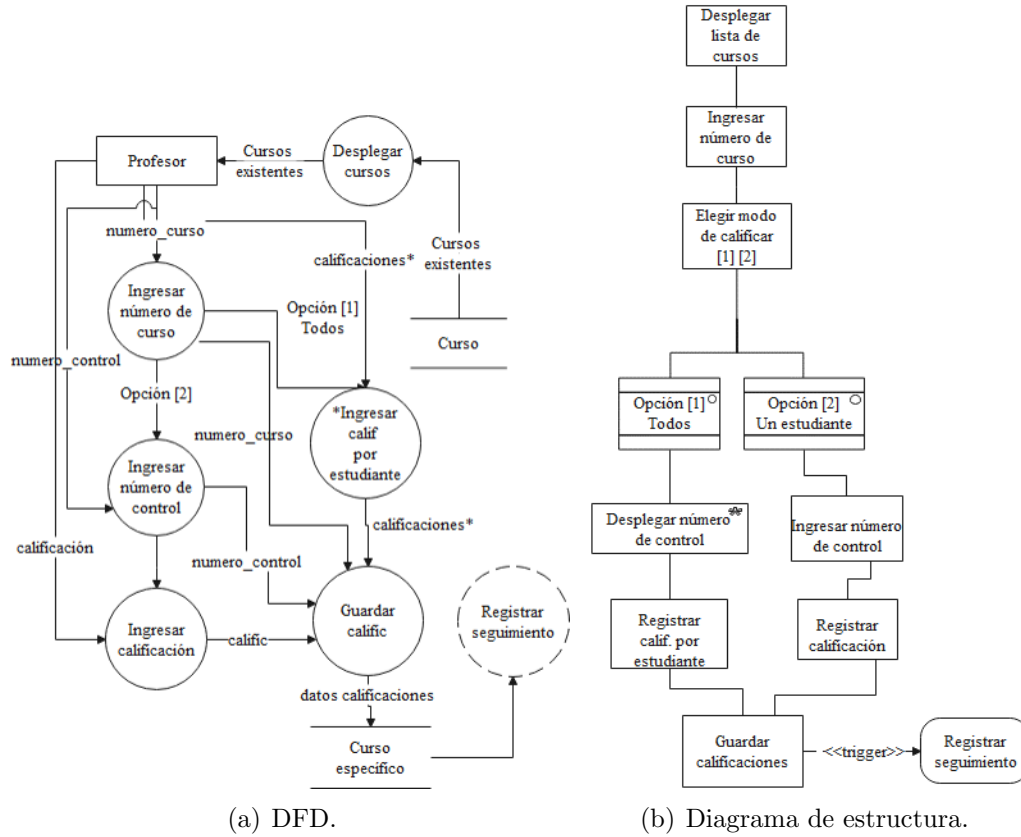


Figura 7.8: Diagramas de análisis(a) y diseño(b) del CU Calificar.

En el Listado 7.4 se incluyen fragmentos de código relevantes del CU. En la línea 34, se observa nuevamente la llamada al registro de seguimiento.

Listado de código 7.4: CU Calificar.

```

1  To calificar :
2      write "CALIFICAR ESTUDIANTES" to the console .
3      write "3. NodeJS" to the console .
4      write "2. JavaScript" to the console .
5      write "1. MongoDB" to the console .
6      skip one line to the console .
7      write "Selecciona un curso:" to the console .
8      read a string from the console .
9      put the string into a course .
10     write "Calificar Todos [1], Un estudiante [2]:" to the console .
11     read a string called option from the console .
12     if the option is "1", calificar the course .
13     if the option is "2", calificar estudiante giving the course .
14
15  To calificar a course :
16     if the course is "1", put "D:\cal-4700 initiative\CMS\1.txt" into a path .
17     if the course is "2", put "D:\cal-4700 initiative\CMS\2.txt" into the path .
18     if the course is "3", put "D:\cal-4700 initiative\CMS\3.txt" into the path .
19     read the path into a buffer called original. If the i/o error is not blank ,
20         write the i/o error on the console .

```

```

21 slap a rider on the original.
22 get a string called encabezado from the original given the rider.
23 append the encabezado to a buffer.
24 Loop.
25     get a string from the original given the rider.
26     if the string is blank, break.
27     get a token from the string.
28     write "Califica a: " then the token then " - " to the console without advancing.
29     read another string from the console.
30     append the token then " - " then the other string then the crlf string to the buffer.
31     [seguimiento]
32     put "C" into an activity's action. put the system's date/time into the activity's date.
33     put the token into the activity's control. put the course into the activity's course.
34     register the activity.
35 Repeat.
36 write the buffer to the path.
37 If the i/o error is not blank, write the i/o error on the console.
38 write "Calificaciones registradas" to the console.

```

7.1.3. Caso de Uso Borrar inscripción de estudiante

De igual manera que los CU anteriores, para el CU Borrar inscripción de estudiante, se muestra la especificación del CU en la Tabla 7.5, la tabulación de ideas en la Tabla 7.6. El Requerimiento 7 (marcar como especial a un estudiante una vez que se da de baja), se implementa como un paso más del flujo normal.

El prototipo se observa en la Figura 7.9, el DFD en la Figura 7.10(a) y el Diagrama de estructura en la Figura 7.10(b). Finalmente, el código representativo del programa se presenta en el listado 7.5.

Tabla 7.5: Especificación del CU Borrar inscripción de estudiante.

ID:	CU03
Nombre:	Borrar inscripción de estudiante
Autor(es):	Lizbeth A. Hernández González
Fecha de creación:	10-nov-21
Fecha de actualización:	
Actor(es):	Estudiante, Profesor
Descripción:	El Estudiante o el Profesor podrán eliminar la inscripción de un estudiante, mediante su número de control y el número de curso correspondiente.
Flujo Normal:	1. Sistema despliega la lista de cursos. 2. Sistema solicita número de control del estudiante. 3. Estudiante o Profesor lo introduce. 4. Sistema solicita número de curso a dar de baja. 5. Estudiante o Profesor lo introduce. 6. Sistema marca al estudiante como baja del curso especificado. 7. Sistema pregunta ¿Deseas dar de baja otro curso? 7.1 Si: regresa a paso 1. 7.2 No: termina CU. 8. Sistema despliega número de cursos dados de baja.
Flujos Alternos:	
Excepciones:	- E.1 Estudiante no existe 1. Sistema despliega "Estudiante inexistente". 2. Regresa a paso 1. - E2. Número de curso no existe 1. Sistema despliega "Número de curso no existe". 2. Regresa a paso 5.
Poscondiciones:	Registrar la baja del Estudiante en el archivo de seguimiento, mediante su número de control y el número de cada curso dado de baja. Indicar la transacción realizada.

Tabla 7.5 continuación de la página anterior.

ID:	CU03
Entradas:	El número de control del estudiante así como el número de cada curso a dar de baja.
Salidas:	Lista de cursos dados de baja.
Incluye: (relación Include)	
Extiende: (relación extend)	
Lanzar: (relación trigger)	Registrar en archivo Seg
Prioridad:	Alta

Tabla 7.6: Tabulación de ideas CU Borrar inscripción de estudiante.

#	Enunciado/ Statement (acción)	Comando/ Command (frase)	Condición / Condition	Agente	Anáfora	Objeto directo/ direct object *para plurales	Destino (opcional)
1	Desplegar			Sistema		Curso*	
2	Solicitar			Sistema		numero_control	
3	Capturar			Estudiante/ Profesor	numero_control		
4	Solicitar			Sistema		numero_curso	
5	Capturar			Estudiante/ Profesor	numero_curso		
6	Marcar (guardar)			Sistema		Estudiante	Inscripciones
7			Preguntar	Sistema		Curso	
7.1		Si: Ejecutar		Sistema		Paso 1	
7.2		No: Salir		Sistema			
8	Imprimir			Sistema	Cursos* dados de baja		

Baja de estudiantes
3. NodeJS 2. JavaScript 1. MongoDB 0. Salir Introduce número de control: 13011074 Selecciona un curso: 1 Estudiante ha sido dado de baja! ¿Deseas dar de baja otro estudiante? 1-Si 0-No 0

Figura 7.9: Prototipo CU Borrar inscripción de estudiante.

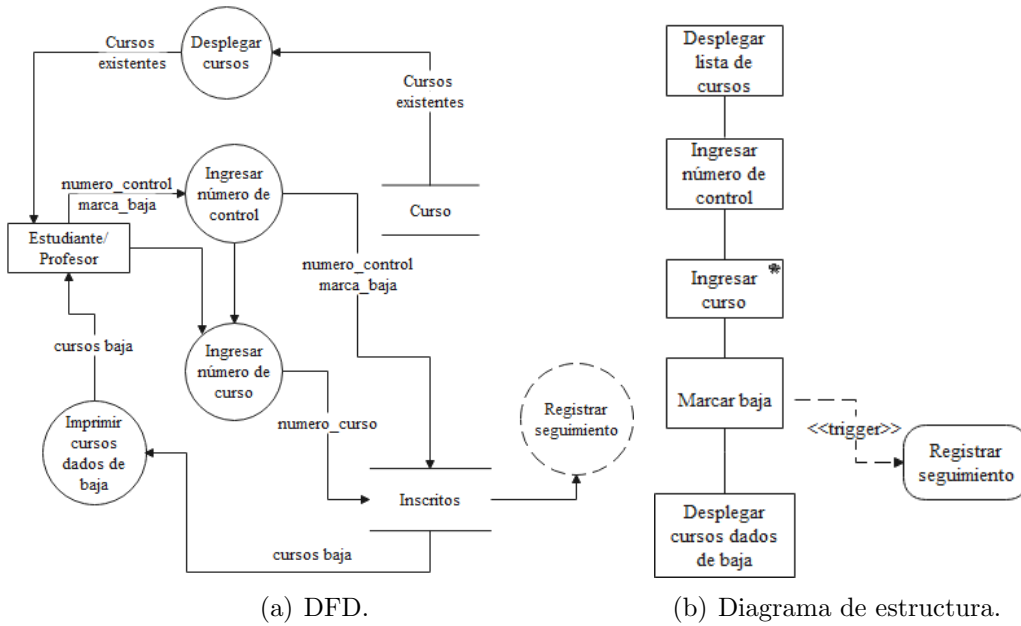


Figura 7.10: Diagramas de análisis(a) y diseño(b) del CU Borrar inscripción de estudiante.

Listado de código 7.5: CU Borrar inscripción.

```

1  To borrar:
2      Loop.
3          write "BAJA DE ESTUDIANTES" to the console.
4          write "3. NodeJS" to the console.
5          write "2. JavaScript" to the console.
6          write "1. MongoDB" to the console.
7          skip one line to the console.
8          write "Ingresa numero de control:" to the console.
9          read a string from the console.
10         put the string into a student's control_number.
11         write "Selecciona un curso:" to the console.
12         read another string from the console.
13         put the other string into the student's course_number.
14         mark the student.
15         write "Deseas dar de baja otro curso? Si[1] No[0]: " to the console without advancing.
16         read a string called condition from the console.
17         if the condition is "0", break.
18         skip one line to the console.
19     Repeat.
20
21 To mark a student:
22     put "D:\cal-4700 initiative\CMS\inscripciones.txt" into a path.
23     read the path into a buffer called original. If the i/o error is not blank,
24         write the i/o error on the console.
25     slap a rider on the original.
26     Loop.
27         get a string from the original given the rider.
28         if the string is blank, break.
29         if the string is the student, mark the student on a buffer.

```

```

30     if the string is not the student , append the string then the crlf string to the buffer .
31 Repeat .
32 write the buffer to the path .
33 If the i/o error is not blank , write the i/o error on the console .
34 [seguimiento]
35 put "B" into an activity's action . put the system's date/time into the activity's date .
36 put the student's control_number into the activity's control .
37 put the student's course_number into the activity's course .
38 register the activity .

```

7.1.4. Aspecto Registrar seguimiento

Este CU es transversal a otros requerimientos, por lo que se considera un aspecto, carece de prototipo ya que se lanza para completar la funcionalidad del CU que lo llama, sin que el usuario lo note. El CU debe registrar cada una de las actividades de **Inscripción, Baja y Calificación del estudiante**. Deberá guardar el tipo de transacción (I,B,C), el número de control del estudiante, el número de curso y la fecha de la transacción. En la Tabla 7.7 se muestra la especificación del CU, incluye los elementos mínimos a documentar en un aspecto (la descripción, el flujo normal y las entradas). En la Tabla 7.8 se muestran las ideas compuestas correspondientes.

Tabla 7.7: Especificación del aspecto Registrar seguimiento.

ID:	AS01
Nombre:	Registrar seguimiento
Autor(es):	Lizbeth A. Hernández González
Fecha de creación:	17-nov-21
Fecha de actualización:	
Descripción:	El aspecto registra en un archivo de seguimiento las transacciones realizadas: Inscribir, Baja, Calificar.
Flujo Normal:	1. Transacción Inscribir: 1.1. Sistema guarda "I", fecha, número de control, número de curso. 2. Transacción Borrar inscripción: 2.1. Sistema guarda "B", fecha, número de control, número de curso. 3. Transacción Calificar: 3.1 Sistema guarda "C", fecha, número de control, número de curso.
Entradas:	Tipo de transacción, fecha, número de curso y número de control del estudiante.
Prioridad:	Alta

Tabla 7.8: Tabulación de ideas aspecto Registrar seguimiento.

#	Enunciado/ Statement (acción)	Comando/ Command (frase)	Condición / Condition	Agente	Anáfora	Objeto directo/ direct object *para plurales	Destino (opcional)
1			Opción I: inscribir				
1.1	Guardar			Sistema		"I", fecha, número de control, número de curso.	Seguimiento
2			Opción B: borrar				
2.1	Guardar			Sistema		"B", fecha, número de control, número de curso.	Seguimiento
3			Opción C: calificar				

Tabla 7.8 continuación de la página anterior.

#	Enunciado/ Statement (acción)	Comando/ Command (frase)	Condición / Condition	Agente	Anáfora	Objeto directo/ direct object *para plurales	Destino (opcional)
3.1	Guardar			Sistema		"C", fecha, número de control, número de curso.	Seguimiento

En la Figura 7.11 se muestra el DFD 7.11(a) y el diagrama de estructura 7.11(b) correspondientes.

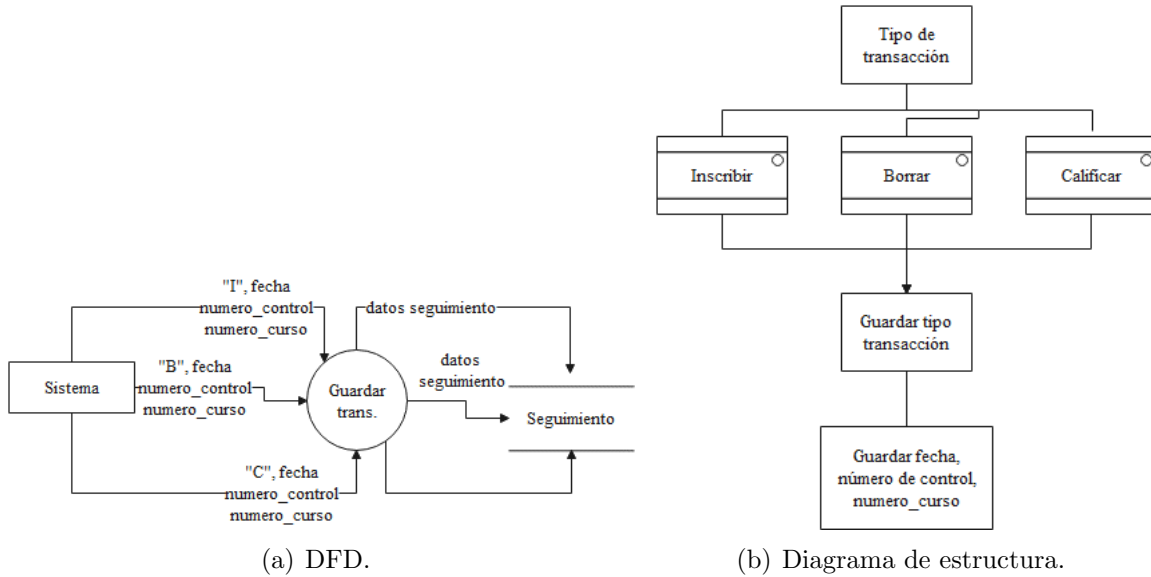


Figura 7.11: Diagramas de análisis(a) y diseño(b) del Aspecto Seguimiento.

En el Listado 7.6 se muestra el programa correspondiente en Cal-4700, se observa la creación de un registro (*record*) *activity*, el cual contiene los datos necesarios para guardar el seguimiento. La rutina *register activity* recibe una actividad como parámetro para almacenar lo correspondiente en el archivo *seguimiento.txt*.

Listado de código 7.6: AS Registrar seguimiento.

```

1 An activity has
2   an action string,
3   a date date/time,
4   a control string,
5   a course string.
6
7 To register an activity:
8   put "D:\cal-4700 initiative\CMS\seguimiento.txt" into a path.
9   if the path is not in the file system, create the path in the file system.
10  read the path into a buffer called original.
11  put the activity's action then " - " then the activity's control then " - "
12    then the activity's course into a string called auxiliar.
13  append " - " then the activity's date's string to the auxiliar.
14  write the original then the auxiliar then the crlf string to the path.
15  If the i/o error is not blank, write the i/o error on the console.

```

7.2. Aplicación del método en un contexto lúdico

En esta sección se presenta la aplicación del método en la construcción de un rompecabezas (*puzzle*), utilizando el lenguaje Cal-4700. Tanto los artefactos de modelado como la construcción de los programas correspondientes, los elaboró el estudiante José Antonio Juárez Romero, como parte de su tesis de la Maestría en Sistemas Computacionales “Desarrollo de software utilizando el lenguaje naturalístico Cal-4700” (Juárez-Romero, 2022). El estudiante se tituló en Octubre del 2022. Siguiendo el método propuesto en este trabajo, el estudiante construyó satisfactoriamente la aplicación lúdica. El seguimiento que se le dio incluyó la capacitación y revisión de avances, y fue de aproximadamente un año y medio. A continuación se describe la aplicación y se incluyen los artefactos y descripciones más relevantes que el estudiante elaboró.

El sistema consta de una aplicación lúdica que tiene como objetivo que niños pequeños de entre 5 y 7 años (los jugadores), aprendan a usar los elementos de la computadora jugando. El juego es el clásico *puzzle* o rompecabezas, en donde a través del uso del ratón, al hacer clic y arrastrar, el jugador podrá ordenar las piezas desordenadas de una imagen dividida en secciones.

Los requerimientos de la aplicación lúdica se explican a continuación:

- R1. El jugador podrá elegir entre tres opciones diferentes de dificultad. La dificultad está en función del número de partes en las que se dividirá el rompecabezas.
- R2: Para cada nivel de dificultad el jugador podrá elegir entre cuatro opciones distintas de imagen para jugar.
- R3. En cualquier momento los jugadores podrán salir del juego actual para seleccionar otra imagen o cancelar el juego.
- R4. El sistema lleva registro de la cantidad de movimientos y tiempo que cada intento de resolver el rompecabezas llevó a los jugadores. Para cada segundo que pasa desde el inicio del juego hasta su finalización, el sistema sumará 1 punto a un contador de resultado. Para cada movimiento que realiza el jugador en el intento de resolver el rompecabezas, el sistema sumará 1 punto a un contador de resultado. Estos cálculos serán la puntuación del jugador.
- R5. El sistema de puntuaciones se registrará de manera independiente para cada nivel de dificultad. La puntuación se mostrará en dos secciones diferentes, una para la puntuación según el criterio de tiempo y otra para la puntuación según el número de movimientos.
- R6. El control de puntuaciones se ordenará de manera ascendente, donde el primer lugar tendrá el puntaje mínimo mientras que el último lugar tendrá el puntaje máximo.
- R7. El jugador deberá registrar su nombre o *nickname* (apodo) al iniciar el juego. El nombre del jugador se agregará al registro de puntuación de la partida.
- R8. El sistema hará el registro de puntuaciones solo cuando el juego esté resuelto.

- R9. Cuando el juego termina el jugador podrá ver el puntaje obtenido en la partida actual.
- R10. El jugador podrá consultar la tabla de puntuaciones para cada uno de los niveles antes de iniciar o al finalizar cualquier partida.

A continuación se presentan los artefactos elaborados por el estudiante, con su explicación correspondiente. Para mayor detalle, consultar la tesis ([Juárez-Romero, 2022](#)).

7.2.1. Ideas atómicas (sustantivos)

A partir de los requerimientos se identificaron las siguientes ideas atómicas:

- **Jugador (*Player*)**. Sustantivo que identifica al usuario que va a interactuar con el sistema. Está es el único actor del sistema, contiene como atributo el dato de tipo cadena: nombre (*name*) que representa al nombre del jugador o su *nickname*. En la Figura 7.12 se observa su representación gráfica.

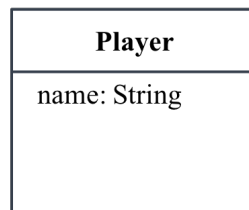


Figura 7.12: Idea atómica jugador.

- **Nivel (*Level*)**. Sustantivo que identifica el nivel de dificultad. Contiene el atributo dificultad (*difficulty*) de tipo cadena, el cual representa el nivel de dificultad: fácil, medio o difícil; y las referencias a las cuatro imágenes del nivel (imagen 1 (*image1*), imagen 2 (*image2*), imagen 3 (*image3*) e imagen 4 (*image4*)) de tipo cadena. En la Figura 7.13 se aprecia la representación gráfica de la idea.

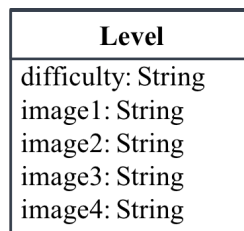


Figura 7.13: Idea atómica nivel.

- **Puntuación (*Score*)**. Esta idea atómica representa la puntuación del jugador. Contiene un dato tipo numérico llamado puntuación (*score*), el cual representa el registro numérico de la puntuación del jugador. Además, contiene el dato criterio (*criterion*) de tipo cadena, en donde se especifica el tipo de puntuación: por tiempo

o por número de movimientos. Esta idea se relaciona fuertemente con las ideas Nivel (*Level*) y Jugador (*Player*) como se observa en la Figura 7.14, en donde se aprecian las referencias a estas ideas atómicas.

Score
level: Level player: Player score: Number criterion: String

Figura 7.14: Idea atómica puntuación.

7.2.2. Casos de uso y aspectos

El diagrama de CU para la aplicación lúdica se observa en la Figura 7.15. El requerimiento R1 especifica que el actor llamado **Jugador** tiene la facultad de elegir entre tres niveles de dificultad (CU Seleccionar nivel). Una vez elegida la dificultad, en el requerimiento R2 se precisa que al jugador se le permite seleccionar entre cuatro imágenes disponibles (CU Seleccionar Imagen). Al iniciar el juego se solicita el *nickname* del jugador (CU Iniciar juego), éste es capaz de cancelar el juego iniciado en cualquier momento. Una vez que las piezas del rompecabezas están ordenadas el juego se considera terminado (CU Terminar juego) y se registra la puntuación del juego (aspecto **Registrar datos de puntuaciones**). El jugador tiene la facultad de consultar en cualquier momento las puntuaciones registradas (CU Consultar puntuaciones).

El aspecto **Registrar datos de puntuaciones** se considera como un aspecto con el estereotipo <<trigger>> que lo diferencia del resto de casos de uso porque su responsabilidad es compartida y transversal a los requerimientos R4, R5, R6 y R8.

7.2.3. Persistencia de datos

Cal-4700 maneja un sistema de persistencia basado en archivos de texto plano. Para la implementación se organizó la información en tres archivos de texto (**easyScores**, **mediumScores** y **hardScores**), en donde cada archivo corresponde a las puntuaciones de cada nivel de dificultad.

En el interior de los archivos la información de cada registro se almacena por línea con el formato: **Criterio - Nombre del jugador - Puntuación** usando el caracter guion (-) como delimitador de propiedad. La propiedad **Criterio** tiene dos posibles valores “H” para el registro de puntuación por tiempo y “M” para el registro por número de movimientos.

En la Figura 7.16 se esquematiza a manera de ejemplo la estructura del sistema de persistencia desarrollado para la implementación. Los archivos de persistencia se almacenan en el directorio principal del proyecto.

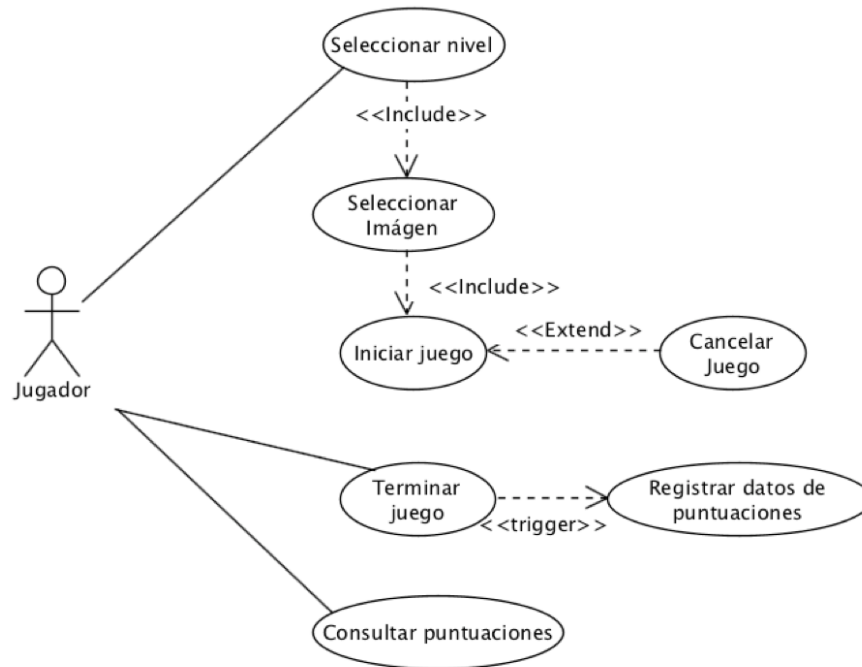


Figura 7.15: Diagrama de casos de uso de la aplicación lúdica.



Figura 7.16: Esquema de persistencia de datos del sistema.

7.2.4. Estructura del proyecto en Cal-4700

Los proyectos de Cal-4700 se administran mediante un directorio único dentro del sistema operativo. Para el presente ejemplo se creó un directorio principal con el nombre **AplicaciónLúdica** con la dirección local **C:\AplicacionLúdica**. El nombre del directorio es importante porque Cal-4700 nombra así al archivo ejecutable que resulta después de compilar el proyecto (como se mencionó anteriormente). En el directorio principal, el proyecto se divide en 10 partes que atienden a las diferentes tareas realizadas dentro del objetivo en común del proyecto. Las partes que lo componen están escritas con Cal-4700 y son las siguientes:

- **Inicio.** Un archivo con el código que pone en marcha al sistema. Dentro de él se encuentra la rutina principal **To run** y las rutinas de inicialización y finalización de los elementos de la interfaz en memoria.
- **Interfaz.** Un archivo que contiene toda la codificación para la interfaz gráfica de usuario. Se considera el fondo de pantalla, botones, imágenes para los rompecabezas, entre otros.

- **Puzzle.** Contiene las definiciones necesarias para generar los diferentes rompecabezas. Considera las tareas de crear y mostrar los rompecabezas, cortar la imagen en piezas, revolver las piezas y controlar los movimientos de las piezas, entre otros.
- **CU Seleccionar Nivel.** Archivo que contiene las definiciones para el CU Seleccionar nivel.
- **CU Seleccionar Imagen.** Archivo que considera las definiciones del CU Seleccionar imagen.
- **CU Iniciar Juego.** Contiene las rutinas necesarias para solicitar el nombre al jugador e iniciar el juego.
- **CU Terminar juego.** Considera la rutina que controla lo que ocurre cuando el Jugador termina el rompecabezas.
- **CU Consultar puntuaciones.** Un archivo que contiene el código que permite al Jugador consultar las listas de puntuaciones.
- **A Registrar puntuaciones.** Un archivo de texto con las definiciones que permiten al sistema almacenar las puntuaciones de la partida a partir de la transacción Terminar juego.
- **The noodle.** Es una biblioteca de Cal-4700 que contiene definiciones de tipos y rutinas, necesario para la correcta compilación de los programas.

7.2.5. Caso de Uso Terminar juego

En esta sección se presenta el desarrollo de un CU, el desarrollo del resto de los CU se puede observar en el Apéndice D. El CU Terminar juego tiene lugar cuando el rompecabezas está completo. Cuando termina el CU, se lanza el aspecto Registrar datos de puntuaciones. Al concluir el rompecabezas se muestra la puntuación de la partida actual.

En la Tabla 7.9 se muestra la descripción del CU, la Figura 7.17 presenta el prototipo. En la Tabla 7.10 se observa la tabulación de ideas, la Figura 7.18 corresponde al DFD, así como en la Figura 7.19 se observa el diagrama de estructura de Jackson.

Tabla 7.9: Descripción del CU Terminar juego.

ID	CU04
Nombre:	Terminar juego
Autor(es):	José Antonio Juárez Romero
Fecha de creación:	01-dic-21
Fecha de actualización:	04-mayo-22
Actor:	Jugador
Descripción:	El juego se termina cuando el rompecabezas está completo.
Flujo Normal:	1. El sistema registra las puntuaciones (AS01-Registra puntuaciones) 2. El sistema muestra el puntaje obtenido en la partida actual. 3. Al confirmar puntuación. Ir al CU01. 4. Termina el CU.
Flujos Alternos:	
Excepciones:	E1. El sistema no recibe respuesta válida. 1. Termina el CU.
Precondiciones:	El rompecabezas debe estar completo.
Poscondiciones:	Agrega los datos de puntuaciones al registro de puntuaciones correspondiente.
Entradas:	1. Selección del jugador.
Salidas:	1. Puntaje de la partida actual.
Incluye: (relación include)	
Extiende (relación extend)	
Lanzar: (relación trigger)	AS01-Registrar datos de puntuaciones
Prioridad:	Alta

El prototipo de la interfaz de usuario para el CU Terminar juego presenta una ventana con tres paneles principales: 'Información', 'Área de juego' y 'Partida actual'. Un modal centralizado muestra el mensaje 'Felicidades. Juego terminado.' y cuatro campos de entrada para 'Nivel:', 'Jugador:', 'Tiempo:' y 'Num. Mov:'. Debajo de estos campos hay un botón 'Terminar'.

Figura 7.17: Prototipo del CU Terminar juego.

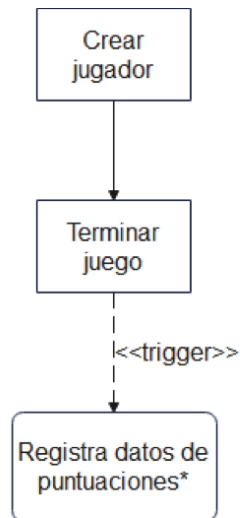
Tabla 7.10: Tabulación de ideas del CU Terminar juego.

No.	Enunciado Statement (acción)	Comando Comand (frase)	Condición Condition	Agente	Anáfora	Objeto directo Direct object	Destino (opcional)
1	Registra		Juego completo	Sistema		Puntuaciones	AS01-Registra puntuaciones
2	Muestra			Sistema		Puntuación	
3	Ir		Puntuación confirmada	Sistema		CU01	
4	Termina			Sistema			



***Datos del juego:** nombre_jugador, segundos_transcurridos, num_movimientos

Figura 7.18: DFD del CU Terminar juego.



***Datos de puntuaciones:**
nombre_jugador, segundos_transcurridos,
num_movimientos, nivel.

Figura 7.19: Diagrama de estructura del CU Terminar juego.

En el Listado 7.7 se presenta el programa correspondiente al CU Terminar juego. Por medio de la instrucción **Register the score** se realiza el registro de puntuaciones, esto corresponde al lanzamiento (llamada) del aspecto **Registrar datos de puntuaciones**. En la Figura 7.20 se aprecia la interfaz que muestra el puntaje del juego terminado.

Listado de código 7.7: CU Terminar juego.

```
1 To finish the game:
2   Register the score.
3 Loop.
4   Deque an event.
5   If the event is nil, break.
6   If the event's kind is not "left click", repeat.
7   If the event's spot is in the confirm button, show levels; exit.
8 repeat
```



Figura 7.20: Interfaz de usuario del CU Terminar juego.

Como se mencionó al inicio de este apartado, el desarrollo del resto de los CU se presenta en el Apéndice D.

7.3. Prueba de aplicación del método con alumnos de maestría

Esta actividad se realizó el 13 de diciembre de 2022, con un grupo de seis estudiantes de primer semestre de la Maestría en Sistemas Computacionales del ITO. El objetivo fue evaluar la factibilidad de la aplicación del método.

El perfil de los participantes era ser egresado de alguna carrera de computación, lo único que se sabía de antemano, era que no tenían mucha experiencia en el modelado de software.

La guía de la práctica fue la siguiente:

1. Se les dio la bienvenida y explicó el objetivo de la práctica.

Se les pidió que mantuvieran las cámaras encendidas para observar sus reacciones durante la práctica.

2. Se les proporcionó el vínculo para descargar el material necesario para trabajar con Cal-4700, además de la presentación que incluía la explicación y ejemplo de la aplicación del método.
3. Utilizando la presentación se introdujo a los estudiantes al paradigma naturalístico, al lenguaje y al método.
4. Se les mostraron ejemplos de aplicación de Cal-4700 que ellos mismos podían ejecutar en su computadora.
5. Se les explicó el método naturalístico mediante el ejemplo del CMS (*Course Management System*).
6. Se les proporcionó el diagrama de CU de un Sistema de Biblioteca, además de la descripción de dos CU: Reservar libro y Prestar libro, junto con sus prototipos correspondientes.
7. A partir de la selección de uno de los CU, la práctica consistió en:
 - a) Tabular las ideas.
 - b) Elaborar el diagrama de datos.
 - c) Realizar el Diagrama de Flujo de Datos (DFD).
 - d) Realizar el Diagrama de estructura de Jackson.
 - e) Codificar el CU utilizando Cal-4700
 - f) Enviar los resultados dentro de las próximas 24 horas.
8. Después de pasadas las 24 horas, se les pidió contestar la siguiente encuesta anónima, para recabar su opinión acerca del uso del método y de los lenguajes naturalísticos. Para ver la encuesta hacer [clic aquí](#).

Observaciones y resultados

En resumen, ningún estudiante fue capaz de realizar correctamente los diagramas, los confundieron sin intentar corregirlos a pesar de contar con la documentación y datos de contacto para resolver dudas. Algunos lograron codificar el CU en Cal-4700, sin embargo no había trazabilidad con los artefactos generados a partir del método. Ésta trazabilidad se revisó manualmente. Se puede concluir que tanto la explicación como la aplicación de un método de desarrollo de software no es trivial, se necesita tiempo para la comprensión, práctica y revisión de los artefactos para lograr su correcta aplicación y sobre todo, la trazabilidad correspondiente con su código de salida.

En la Tabla 7.11 se pueden observar los resultados de la encuesta, para cada una de las preguntas se contabilizaron las respuestas, las cuales se encuentran en la escala de Likert con los siguientes valores:

- Totalmente de acuerdo
- De acuerdo

- Indiferente
- En desacuerdo
- Totalmente en desacuerdo

Los comentarios y sugerencias que se recabaron en la encuesta son los siguientes:

- “Considero que el método es fácil de entender ya que en pocas horas y sin mucha investigación se logra un programa básico, a diferencia de cualquier otro lenguaje, la curva creo yo es más difícil de completarla. Me hubiera gustado un poco más de repaso sobre los diagramas y poder probarlo junto con la doctora para refinar detalles”.
- “Es una propuesta muy interesante, el proceso es intuitivo y fácil de implementar”.
- “Se podría considerar que el código escrito en lenguaje natural es más fácil de leer. Sin embargo, en algunos casos una sintaxis más reducida puede llegar a ser más fácil de leer. Aunque esto no aplica para lambdas o la sintaxis de Go. Ayudaría tener el resaltado en palabras reservadas. La definición de variables y funciones me resultó más fácil de comprender”.
- “El método naturalístico (con lo poco que he llegado a conocer) aminora la barrera de entrada para el aprendizaje de la programación, sin embargo, la ventaja que puede dar al principio, se aminora en medida que se tiene conocimiento de otros lenguajes, por ejemplo, la terminal de un sistema operativo requiere un mayor nivel de conocimiento y tiene una mayor barrera de entrada que el uso de la interfaz gráfica, pero una vez dominada, resulta mucho mas práctica y precisa. Además de las ambigüedades propias de un idioma natural. Entre ms lejana sea la lengua materna de un individuo al inglés (incluso si es alguien con conocimiento del mismo como fue mi caso) para el caso que estamos tratando, menor será la ventaja que le de este método, sino es que lo vuelve más confuso que el uso de un lenguaje tradicional basado es una lógica más dura. Se usa un ingles simple, tratando de evitar lo anteriormente mencionado (al menos supongo) lo cual hace que el lenguaje pueda parecer mas intuitivo de lo que en realidad es. Debo familiarizarme más con el lenguaje para dar una declaración más completa y fundamentada, pero podríamos decir que este método, que es para el usuario súper final es también de ”súper alto nivel”pero no migraría de un lenguaje mas ”tradicionalz también de alto nivel como python por ejemplo, hasta que mejoren las facilidades que otorga y que exista una comunidad y documentación mas abundante”.

7.4. Discusión

De las pruebas de concepto guiadas se deduce que para evaluar un método de desarrollo se necesita tiempo. Tiempo para capacitar a los desarrolladores en la comprensión, seguimiento y aplicación del método. En este caso, también se debía capacitar o proporcionar lo necesario para el aprendizaje del lenguaje de programación naturalístico. Todo esto conlleva una curva

Tabla 7.11: Encuesta de evaluación del método con resultados

	Totalmente de acuerdo	De acuerdo	Indiferente	En desacuerdo	Totalmente en desacuerdo
En referencia al Método Naturalístico propuesto considera que:					
1. El desarrollo de software es más transparente.	3	3			
2. Es fácil de aplicar.	3	2	1		
3. Es comprensible o entendible.	2	4			
4. Facilita el paso de una fase a otra.	2	3	1		
5. La tabulación de ideas permite analizar con detalle los requerimientos.	3	1	2		
6. La tabulación de ideas permite dar seguimiento a los requerimientos.	3	2	1		
7. La tabulación de ideas facilitará crear casos de prueba.	3	2	1		
Considera que el uso de lenguajes naturalísticos como Cal-4700:					
8. Generan código autodocumentado.	2	3		1	
9. Generan programas más claros y entendibles.	3		2	1	
10. Facilitan el mantenimiento de software.					
	Si		Tal vez	No	
11. ¿Recomendaría el uso de lenguajes naturalísticos?	3		2	1	
12. ¿Recomendaría el uso del método naturalístico?	3		3		
13. ¿Considera que el uso del Método Naturalístico es más sencillo respecto a otros enfoques?	3			3	

de aprendizaje. Para muchos de los egresados de una carrera de computación, es un reto modelar, no están acostumbrados a seguir un proceso formal o a realizar artefactos. La prueba de concepto del caso de una aplicación lúdica fue exitosa, sin embargo fue una tesis de maestría. Esto significa que el estudiante dedicó meses al estudio del contexto de la aplicación, al estudio del método, a la elaboración de los artefactos y en caso de la autora, a la revisión de los mismos. Esto se repitió varias veces. De manera paralela, el estudiante aprendió el lenguaje de programación Cal-4700. Es comprensible entonces, que los resultados de la prueba realizada con los estudiantes de maestría, no fueran los esperados. Lo anterior hace difícil de evaluar el método en corto tiempo, sin embargo, es posible a mediano plazo, involucrando a colegas o estudiantes de maestría en otros trabajos de investigación.

Conclusiones

Uno de los objetivos de este documento es dar a conocer los lenguajes de programación del paradigma naturalístico, con la intención de promover su uso en la construcción de software de propósito general, ya que las ventajas que ofrece el paradigma son atractivas. Se expusieron los lenguajes naturalísticos de propósito general Pegasus, SN y Cal-4700, debido a que cumplen con la filosofía del paradigma naturalístico: contar con lenguajes que incluyan elementos del lenguaje natural, que sean reflexivos y generen programas más expresivos y ejecutables. El hecho de analizar las características que cada uno ofrece, permitió contar con una visión más concreta de sus capacidades, a la vez que identificar la necesidad de avanzar hacia el uso intensivo de estas propuestas.

La historia indica que una vez que se tienen lenguajes de programación que soportan un paradigma, lo consiguiente es determinar las actividades de diseño y posteriormente las de análisis de software. Para promover el uso del paradigma, se propuso un método que proporciona las pautas necesarias para realizar estas fases del proceso de desarrollo de software hasta llegar hasta la construcción.

Después de revisar los elementos que debe contener el Análisis y Diseño de software, se identificaron algunas notaciones que podrían servir de base en el modelado de un software naturalístico, como es el caso de los Diagramas de Flujo de Datos (DFD), los Diagramas de Bases de Datos (DE-R) y los Diagramas de Estructura; los cuales se usan desde tiempo atrás en el modelado del software procedural. Si bien la sintaxis de un programa naturalístico es distinta a la de un programa procedural, se asemejan en la manera en la que el ser humano piensa un conjunto de instrucciones, con tipos y variables involucradas. En cuanto al Análisis de Requerimientos, se decidió utilizar la técnica de Casos de Uso 2.0, ya que, además de su amplio uso en la academia y en la industria de software, utiliza descripciones en lenguaje natural para una mejor comprensión por parte de los involucrados, esto concuerda con el enfoque del paradigma y facilita la trazabilidad de los requerimientos hacia el resto de las fases del proceso de desarrollo.

El método se basa en la concepción de *idea*, la cual refleja los pensamientos del usuario que se plasman en los requerimientos funcionales. La representación tabular se propuso como herramienta de trazabilidad, estableciendo un camino desde la idea (el requerimiento) hasta la construcción, verificación y validación, pasando por las fases de Análisis y Diseño. La técnica de CU es flexible puesto que se adapta a marcos ágiles, por ejemplo, con su versión 2.0. Aunque no se ha comprobado, se considera que la inclusión de diagramas de flujo de datos también aporta a la flexibilidad del método, ya que representan los requerimientos con el nivel de detalle que se necesite, es decir, permiten agregar actividades, almacenes o datos que mejoren el entendimiento del problema. Para el diagrama de estructura es conveniente

analizar si es necesario agregar mayor detalle o quizá ampliar la notación para incluir mayor expresividad en los mismos.

El hecho de documentar el método en apego a un estándar internacional, en este caso el ISO/IEC 29110, facilitará su comprensión y aceptación.

Como resultado de las pruebas de concepto del método, parte de la validación e identificación de contextos a aplicar, se comprobó que es posible construir software con él utilizando lenguajes naturalísticos de propósito general, apreciando un paso natural entre una fase y otra. Por el momento se probó el método en un sistema transaccional como el CMS (*Course Management System*) y en un sistema de interfaz gráfica de contexto lúdico, sin embargo se debe seguir probando en más contextos y de mayor complejidad.

Aunque probar un método no es trivial, pues consume varios meses de trabajo, ya que se debe capacitar al desarrollador y dar seguimiento a la elaboración/corrección de artefactos, se concluye que fue posible proponer un método utilizando el paradigma naturalístico, que permite el desarrollo de software de una forma más clara y comprensible para los interesados.

Con esta primera versión, se aprecia que el paradigma naturalístico, además de aprovechar la concepción de *idea*, es posible modelarse con un subconjunto de enfoques efectivamente probados, que en un orden particular, ofrecen una propuesta inicial. Lo anterior es una ventaja para su uso y aceptación.

Productos Académicos

- Artículo JCR requisito de titulación

Título: “A Proposal of Naturalistic Software Development Method”.

Autores: Hernández-González Lizbeth A., Juárez-Martínez Ulises, Mejía Jezreel, Aguilar-Laserre Alberto

Idioma: Inglés.

Revista: *The Journal of Universal Computer Science*².

Estado: Publicado (ver Figura 7.21).

Cuartil: Q2

Factor de impacto: 1.056

DOI = <https://doi.org/10.3897/jucs.105637>

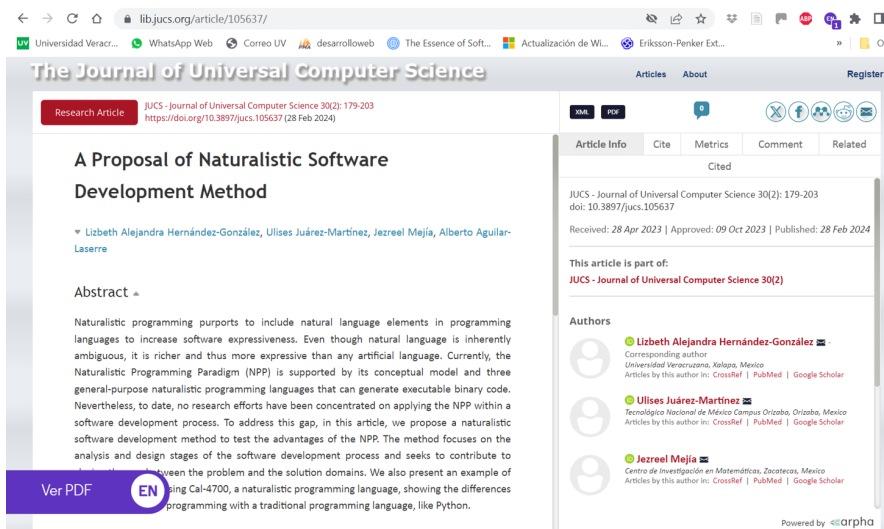


Figura 7.21: Artículo JCR publicado

- Escritura de un Capítulo para el libro “Theory and practice in software engineering: A decade of evolution”, dentro del marco de la celebración de los 10 años del Congreso Internacional de Mejora de Procesos de Software (CIMPS).

²<https://lib.jucs.org/>

Título: “**A walk-through of the evolution of the naturalistic programming languages**”.

Autores: Hernández-González Lizbeth A., Juárez-Martínez Ulises, Mejía Jezreel

Idioma: Inglés.

Estado: Aceptado.

- Presentación del artículo “**Evolution of Naturalistic Programming: A Need**” ([Hernández-González et al., 2020](#)), dentro del marco del Congreso Internacional de Mejora de Procesos de Software (CIMPS 2020), el 21 de Octubre del 2020 (ver primera página en la Figura [7.22](#)).

Autores: Hernández-González Lizbeth A., Juárez-Martínez Ulises, Alducin-Francisco Luisa

Idioma: Inglés.

DOI = 10.1007/978-3-030-63329-5_13

- Participación como co-autora en el Artículo “**Implementación práctica del paradigma naturalístico mediante un caso de estudio lúdico**”.

Autores: José Antonio Juárez-Romero, Ulises Juárez-Martínez, Lizbeth Alejandra Hernández-González, Lisbeth Rodríguez-Mazahua, María Antonieta Abud-Figueroa

Idioma: Español.

UABC - Facultad de Ciencias Químicas en Ingeniería, Revista Aristas: Investigación Básica y Aplicada

ISSN 2007-9478

Ver primera página en la Figura [7.23](#).

En este artículo se presenta la construcción de un rompecabezas para niños de entre 5 y 10 años, únicamente se reporta lo correspondiente al lenguaje Cal-4700.



Evolution of Naturalistic Programming: A Need

Lizbeth A. Hernández-González¹ (✉), Ulises Juárez-Martínez²,
and Luisa M. Alducin-Francisco²

¹ Faculty of Statistics and Informatics, University of Veracruz, Xalapa, Mexico
lizhernandez@uv.mx

² Technological Institute of Orizaba, National Technological of Mexico, Orizaba, Mexico
ujuarez@orizaba.tecnm.mx, luisa.alducinf@gmail.com

Abstract. For years, researchers and programmers have studied, proposed, and reflected on how to enhance the expressive power of programming languages. One of the most popular thoughts on this matter is that natural language can be used as a programming tool. In this work, we review the state-of-the-art of naturalistic programming, whose goal is to achieve greater expressiveness in programming languages and better approach the user domain. We review the only three naturalistic general-purpose languages that can generate executable code by comparing them concerning a conceptual model – Pegasus, Cal-4700, and SN. We discuss the similarities and differences between each language. Then, we introduce and discuss some examples of computer programs developed with these languages, showing their level of expressiveness. Finally, we reflect on the findings that show the need for the paradigm to evolve having a software development method, to be considered as a formal proposal in Software Engineering.

Keywords: Naturalistic programming · Naturalistic paradigm · Natural language programming · Expressiveness · Software engineering methods

1 Introduction

For decades, programmers have been in the search for greater expressiveness when programming. In this sense, programming languages have evolved chiefly in terms of a closer approach to the user domain to achieve greater expressiveness. However, there is still a gap between the problem domain and the solution domain that has not been properly addressed. This gap significantly contributes to customer dissatisfaction and high costs because of a lack of clarity in abstractions and information loss across the phases of the software development process.

The naturalistic programming paradigm (NPP) seeks to incorporate elements of natural language into the software development process to reduce ambiguity and thus reach greater expressiveness – i.e. greater clarity when representing ideas. Naturalistic programming was first proposed in 2003 [1]. Nowadays, 17 years later, a conceptual model for naturalistic programming exists [2], along with several general-purpose programming languages supporting the NPP. Nevertheless, not a case study reported in the literature

© The Editor(s) (if applicable) and The Author(s), under exclusive license
to Springer Nature Switzerland AG 2021
J. Mejia et al. (Eds.): CIMPS 2020, AISC 1297, pp. 185–198, 2021.
https://doi.org/10.1007/978-3-030-63329-5_13

Figura 7.22: Artículo Evolution of NP: a Need

Implementación práctica del paradigma naturalístico mediante un caso de estudio lúdico

José Antonio Juárez-Romero^{1,a}, Ulises Juárez-Martínez^{1,b},
Lizbeth Alejandra Hernández-González^{2,c}, Lisbeth Rodríguez-
Mazahua^{1,d}, María Antonieta Abud-Figueroa^{1,e}.

¹Tecnológico Nacional de México, Campus Orizaba, División de Estudios de Posgrado e Investigación, Ote. 9, Emiliano Zapata, C.P. 94320, Orizaba, Veracruz, México.

²Universidad Veracruzana, Zona Xalapa, Facultad de Estadística e Informática, Av. Xalapa SN, Colonia Centro, C.P. 91000, Xalapa, Veracruz, México.

^aaj.romero@me.com, ^bujuares@ito-dept.edu.mx, ^clizhernandez@uv.mx, ^dlrodriguezcm@ito-dept.edu.mx, ^emabud@ito-dept.edu.mx.

Resumen

Durante los últimos años, se ha observado un creciente interés en incorporar elementos del lenguaje natural en las técnicas de programación, con la finalidad de volver más expresivo el código fuente. Aunque las técnicas de programación actuales consideran abstracciones de alto nivel que permiten organizar de manera más sencilla el código, el nivel de expresividad de este aún es escaso.

A este problema, se suma la pobre documentación del software, debida entre otras cosas, a la falta de disciplina del programador o a los cortos tiempos de entrega, que generan como resultado un producto de software hermético que dificulta las tareas de mantenimiento y evolución.

Como alternativa de solución, la programación naturalística propone crear lenguajes de programación a partir de la incorporación de elementos del lenguaje natural, que ofrezcan mayor expresividad y generen código autodocumentado.

Es en este contexto que se presenta el lenguaje de programación naturalístico de propósito general Cal-4700, el cual permite escribir código ejecutable a partir de un subconjunto de frases escritas en el idioma inglés.

En este artículo se presenta una implementación naturalística usando Cal-4700 en una aplicación lúdica. La finalidad es demostrar su capacidad expresiva y características avanzadas, para considerarlo en la construcción de aplicaciones.

Este trabajo muestra que emplear un subconjunto controlado del idioma inglés como lenguaje de programación permite alcanzar los objetivos de ofrecer mayor expresividad y código autodocumentado, además de demostrar que Cal-4700 es suficientemente robusto para crear aplicaciones pequeñas y medianas de propósito general.

Palabras clave— Cal-4700, desarrollo de software, expresividad, lenguajes de programación, programación naturalística.

Abstract

Recently, there has been growing interest in incorporating natural language elements into programming techniques to make source code more expressive. Although current programming techniques consider high-level abstractions that allow a simpler organization of the code, the level of expressiveness of the code is still low.

In addition, the software project is usually poorly documented. This is due, among other things to a lack of

discipline on the part of programmers or short delivery times, resulting in a software product that makes maintenance and evolution tasks difficult.

As an alternative solution, naturalistic programming proposes the creation of programming languages based on the incorporation of natural language elements, which offer greater expressiveness and generate self-documenting code.

In this context, the general-purpose naturalistic programming language Cal-4700 is presented, which allows writing executable code from a subset of sentences written in English.

This paper presents a naturalistic implementation using Cal-4700 in a playful application, to demonstrate its expressive capacity and advanced features, to be considered in the construction of applications.

This paper shows that using a controlled subset of English language as a programming language, achieves the goals of providing greater expressiveness and self-documenting code, and demonstrates that Cal-4700 is robust enough to create small and medium-sized general-purpose applications.

Keywords— Cal-4700, expressiveness, naturalistic programming, programming languages, software development.

1. INTRODUCCIÓN

La programación naturalística se define en [1] como una técnica que busca crear software haciendo uso de elementos del lenguaje natural en la sintaxis del lenguaje de programación, con el propósito de reducir la complejidad y dar más expresividad y claridad al código fuente.

El lenguaje natural ha sido usado con éxito para escribir sistemas complejos. Se han escrito todo tipo de documentos que contienen información sobre estructuras y procesos. Un ejemplo claro, es este mismo documento, que además de estar compuesto de estructuras gramaticales correctas, precisa de dividir el contenido en unidades cada vez más limitadas, como son capítulos, secciones, subsecciones, párrafos y oraciones, que finalmente transmiten un significado claro y expresivo al lector.

En este sentido, investigadores han buscado la evolución de las técnicas de programación que promuevan la creación de lenguajes más expresivos desde el enfoque de los lenguajes naturales, reportado en [1], [2], [3] y [4].

Desde el origen de la Ingeniería de Software, los programadores se han visto en la necesidad de usar los lenguajes de programación como una forma de escribir la información procedimental de los sistemas, sin embargo, dichos lenguajes son inadecuados cuando se trata de transmitir información relevante entre personas. La dificultad proviene de varios factores, por una parte, los sistemas en los campos de la ingeniería, la ciencia y la economía, por mencionar algunos, son complejos, debido a las intrincadas relaciones que existen entre las estructuras y procesos que considera su dominio. Por otro lado, el programador se ve en la necesidad de transformar los requerimientos desde un lenguaje natural (las ideas) a un lenguaje formal (el lenguaje de programación) [3]. De esta manera, las ideas deben adaptarse a un lenguaje y paradigma específico haciendo uso de sus abstracciones propias (objetos, funciones,

Figura 7.23: Artículo Caso de estudio lúdico utilizando el PN

Apéndice A

Revisión Sistemática de la Literatura

Una revisión sistemática es la revisión de una pregunta claramente formulada que utiliza métodos sistemáticos y explícitos para identificar, seleccionar y evaluar críticamente la investigación relevante, y para recopilar y analizar datos de los estudios que se incluyen en la revisión. Los métodos estadísticos (metaanálisis) se usan para analizar y resumir los resultados de los estudios incluidos ([Cochrane-Collaboration, 2005](#)).

El paradigma naturalístico propone utilizar elementos del lenguaje natural para lograr una mayor expresividad al programar, no se trata de cubrir al 100 % un idioma en particular, sino de construir lenguajes de propósito general que se asemejen al lenguaje natural, bajo una gramática restringida, de lo contrario se tendrían incontables variantes.

Dado lo anterior, el objetivo de la revisión fue encontrar todo lo que permitiera conocer el estado del arte del paradigma naturalístico en cuanto a su aplicación en el desarrollo de software de propósito general: propuestas de aplicación, lenguajes de programación actuales, métodos de Ingeniería de Software relacionados y casos de estudio. De manera complementaria, ya que el objetivo principal del paradigma naturalístico es lograr una mayor expresividad al programar, interesaba conocer las diferentes propuestas que han intentado lograr este objetivo. Esto permitió identificar logros y retos comunes.

Las preguntas de investigación que guiaron la revisión son las siguientes:

1. ¿Cuáles son los lenguajes naturalísticos de propósito general actuales?
2. ¿Qué propuestas se reportan asociadas a la expresividad de los lenguajes de programación?
3. ¿Se encuentran casos de estudio o de aplicación reportados en la literatura?
4. ¿Qué métodos o procedimientos de Ingeniería de software son aplicables al paradigma?
5. ¿Qué términos son similares en la literatura con el término “programación naturalística”?

Previamente se contó con información de los estudios más representativos. Para comprobar y ampliar esta información, se realizó un estudio piloto que permitió identificar términos similares y los criterios a aplicar en las cadenas de búsqueda, con la finalidad de obtener los resultados más relevantes.

A.1. Método de revisión

En esta sección se especifica la estrategia que se siguió para realizar la búsqueda, las fuentes de búsqueda, los criterios de inclusión y exclusión que se aplicaron para filtrar los resultados, así como la manera de extraer y sintetizar los datos (Turner, 2010; Kitchenham & Charters, 2007; Moher et al., 2009; Galster et al., 2018; Siddaway, 2014).

A.1.1. Estrategia de búsqueda

Con base en las preguntas de investigación se identificaron las siguientes **palabras clave**:

- *General purpose naturalistic languages.*
- *Programming languages expressiveness.*
- *Naturalistic programming.*
- *Naturalistic paradigm.*
- *Naturalistic software development.*
- *Naturalistic software engineering methods.*

De acuerdo a *The Cambridge English Dictionary*, la palabra “*naturalistic*” significa “*similar to what exists in nature*”, por lo que en este contexto, un lenguaje naturalístico es aquel que se asemeja al lenguaje natural. Sin embargo, aunque el término está bien definido, es común encontrar trabajos relacionados con otros términos: procesamiento de lenguaje natural, programación en lenguaje natural, programación en inglés plano, entre otros. El término “*naturalistic*” arrojó más de 24,000 resultados, el 90 % de ellos enfocados a ciencias de la salud, por lo que se descarta de manera individual. Las consultas relacionadas con el Procesamiento de Lenguaje Natural de los últimos 10 años arrojaron 18,794 resultados, más de 11,000 enfocados a áreas de la salud y/o relacionados con la Inteligencia Artificial en aplicaciones específicas para cierto dominio, el resto enfocados en áreas como la lingüística y computación en general. Dada la diversidad de resultados, este término se descartó para cualquier tipo de análisis.

En el estudio piloto, también se observó que los términos relacionados con la búsqueda de métodos de desarrollo de software, métodos de Ingeniería de Software y en general sobre desarrollo de software bajo el enfoque naturalístico, arrojaron cero resultados. Sobre lenguajes de programación naturalísticos de propósito general, se encontró 1 resultado, cuando de antemano se conocían dos. Sobre programación naturalística de propósito general se encontraron 0 resultados.

Tomando lo anterior en consideración, el refinamiento de los términos de búsqueda resultó en los siguientes:

- *Naturalistic programming.*
- *Naturalistic syntax.*
- *Naturalistic paradigm.*

- *General purpose natural language processing.*
- *General purpose natural language programming.*
- *Programming languages expressiveness.*
- *Natural language processing.*
- *Natural language programming.*
- *Plain english programming.*

Estos términos se utilizaron en las cadenas de búsqueda de *Title* y *Abstract*, además de consultas a Texto completo (*Full data*) con la intención de aumentar la probabilidad de obtener resultados relevantes.

A.1.2. Fuentes de búsqueda

La búsqueda se realizó en las principales bibliotecas digitales del área de las ciencias de la computación, entre ellas:

- IEEE Xplore Digital Library.
- ACM Digital Library.
- Elsevier Science Direct.
- SpringerLink.

Además de considerar la literatura gris seleccionada directamente desde Internet.

A.1.2.1. Herramientas

El uso de herramientas de software permitió facilitar la búsqueda, manipulación y análisis de los datos, además de minimizar la incidencia de errores.

Las herramientas de software utilizadas fueron:

- Dimensions¹: herramienta para acceder y analizar datos de investigaciones desde diferentes fuentes de datos: publicaciones, patentes, documentos de política, entre otros. Tiene acceso a las principales bases de datos digitales de todas las áreas de la ciencia.
- VOSviewer²: herramienta que permite elaborar mapas bibliométricos a partir de los resultados de las consultas.
- Kopernio³: complemento para buscadores como Google Chrome o Mozilla Firefox, el cual permite localizar los archivos PDF de las publicaciones, si es que se encuentran disponibles en Internet.

¹<https://www.dimensions.ai/>

²<https://www.vosviewer.com/>

³<https://kopernio.com/>

- JabRef⁴: herramienta para administrar referencias bibliográficas en formato BibTex.

Dimensions permite:

- La exportación limitada a 2500 registros, suficiente para este estudio.
- Filtrar los resultados por campos de investigación, años de publicación, investigadores (autores), tipo de publicación: *Article*, *Proceeding*, *Chapter*, *Preprint*, *Edited Book*, *Monograph*, tipo de fuente, revista o tipo de acceso.
- Analizar la información desde diferentes perspectivas: número de publicaciones por año, por campo de investigación, por fuente o por autor, número de citas y otras métricas alternas como las citas en redes sociales, entre otras.

Las consultas se realizaron en Dimensions y se exportaron a archivos con formato CSV para utilizarse en VOSViewer, la cual permite seleccionar más de un archivo para la elaboración de mapas. Los tipos de mapas que VOSviewer permite realizar son para determinar la coocurrencia de términos basados en datos textuales o los mapas creados a partir de datos bibliográficos⁵, los cuales permiten analizar:

- *Co-authorship*: la afinidad de elementos se determina con base en el número de elementos de coautoría.
- *Citation*: la afinidad de elementos se determina con base en el número de veces que los autores se citan.
- *Bibliographic coupling*: la afinidad de elementos se determina con base en el número de referencias que los autores comparten.
- *Co-citation*: la afinidad de elementos se determina con base en el número de veces que los autores son citados juntos.

Para reportar el flujo de información a través de las diferentes fases de la revisión sistemática, se utilizó el Flujo de trabajo de PRISMA (Moher et al., 2009), el cual determina el conjunto de elementos mínimos basados en evidencia para reportar revisiones sistemáticas y metaanálisis. Las fases incluidas en su flujo de trabajo son: Identificación, Escaneo, Elegibilidad e Inclusión.

A.1.3. Criterios

A continuación se describen los criterios aplicados en la selección de fuentes relevantes: años de selección y otros criterios que determinaron si el material encontrado debía ser incluido o no.

⁴<https://www.jabref.org/>

⁵Información tomada directamente del soporte de la herramienta

A.1.3.1. Años de selección

El artículo que propone el surgimiento del paradigma se publicó en el año 2003, a partir del cual se reporta interés en el enfoque, por lo que el criterio de búsqueda para este término se limitó en un inicio a los años de publicación 2003 en adelante, con la intención de analizar la cronología de las diferentes propuestas. En las diferentes fases se aplicó el filtro a cinco años atrás, es decir de 2015-2020, para identificar las propuestas más recientes.

En cuanto a la búsqueda de expresividad de lenguajes de programación, la publicación que reporta un modelo matemático para determinar la expresividad de los lenguajes de programación es de 1991, por lo que el criterio de años de publicación para este término se definió a partir de 1990 en adelante. La intención era determinar algunos trabajos pioneros en este tema, para que a partir de estos, se localizaran trabajos actuales (de cinco años atrás) basados en los aportes de los 90.

A.1.3.2. Criterios de inclusión

Los filtros aplicados fueron los siguientes:

- Los correspondientes a los años de publicación (2003-2015 para obtener los fundamentos, 2015-2020 para obtener los trabajos más recientes).
- Las categorías de investigación que incluyeran los campos de estudio relacionados con las ciencias de la computación y matemáticas, ya que ahí están reportados los trabajos relevantes.
- Se aceptó cualquier tipo de publicación a excepción de Monografías.
- Se aceptaron publicaciones con cualquier tipo de acceso, ya que con ayuda de Kopernio es posible ubicar los archivos PDF, en caso de que no los encontrara, se buscaron en ResearchGate o en Internet en general.
- Para la literatura gris, se aceptaron reportes técnicos y sitios web, siempre y cuando los resultados fueran relevantes, es decir, directamente relacionados a contestar alguna de las preguntas de investigación.

Los criterios de inclusión por fase son los siguientes:

- Identificación: número de registros obtenidos de las consultas consideradas representativas, es decir, aquellas que arrojen resultados relacionados con el paradigma naturalístico. Se eliminaron los registros duplicados.
- Escaneo: a través de la revisión del título y el *abstract*, se seleccionaron aquellas publicaciones que aportaban evidencia directa a alguna de las preguntas de investigación. Si era necesario, también se revisaban las conclusiones.
- Elegibilidad e Inclusión: posterior al escaneo, se revisaron las publicaciones restantes a texto completo, para decidir por sus aportes, cuales serían incluidas finalmente como base de la investigación.

A.1.3.3. Criterios de exclusión

Los siguientes criterios de exclusión se aplicaron a las publicaciones resultantes después de utilizar todos los filtros correspondientes a año de publicación, campo de investigación y tipo de publicación. Específicos para las fases de Escaneo y Elegibilidad se determinó lo siguiente:

- Para la fase de Escaneo, dado que el término Paradigma Naturalístico se suele relacionar con *Natural Language Processing*, *End-user Programming*, *Programming by demonstration*, entre otros; se descartaron todas aquellas publicaciones que consideraran alguno de estos términos, ya que no se enfocan al desarrollo de software naturalístico de propósito general.
- Para la fase de Elegibilidad, se excluyeron las publicaciones que no aportaron directamente a dar respuesta a alguna de las preguntas de investigación.

A.1.4. Análisis de los datos

El análisis de los datos se determinó por la extracción y la síntesis de los mismos.

A.1.4.1. Extracción de datos

Como ya se mencionó, los resultados de la búsqueda se exportaron a archivos de texto en formato CSV, para revisar la información, eliminar los duplicados y generar los mapas bibliométricos. Las publicaciones escaneadas, elegidas e incluidas se almacenaron en JabRef.

Las publicaciones elegidas e incluidas se representaron tabularmente con los siguientes campos:

- Publicación.
- Autores y año de publicación.
- Aportes/Propuesta.
- Limitaciones.
- Número de pregunta de investigación a la que da respuesta.

A.1.4.2. Síntesis de datos

En la Sección A.2, se presenta el resumen de los resultados de las consultas. El número de publicaciones encontradas, escaneadas, elegidas e incluidas se reporta utilizando el flujo de trabajo de PRISMA (Moher et al., 2009). El número de publicaciones y citas por fuente (revista), el número de publicaciones por año y el número de publicaciones por campo de investigación, se presentan mediante gráficas generadas con las herramientas de análisis de Dimensions⁶.

⁶<https://www.dimensions.ai/>

A.2. Resultados

Dimensions presenta los resultados de cada consulta, junto con la lista de años de publicación y campos de investigación asociados a los resultados de esa búsqueda, para que el usuario aplique los filtros convenientes. También permite realizar análisis del número de publicaciones y citas, ya sea por autor, por año, por revista o por campo de investigación.

En la Tabla A.1 se muestra el número de resultados obtenidos por cada término de búsqueda, se muestran sólo los términos y combinaciones que arrojaron resultados relevantes de acuerdo a los criterios de inclusión y exclusión determinados previamente. Para cada término se muestran los filtros aplicados por año y campos de investigación. Inicialmente, las consultas se realizaron buscando las coincidencias en el título y resumen de cada publicación, sin embargo dado el número reducido de publicaciones encontradas, algunas consultas se expandieron a buscar en todos los datos de la publicación (indicadas como *Full data*). De igual manera, las consultas iniciales se limitaron a considerar los campos de investigación relacionados con las ciencias de la computación, excluyendo el campo de Inteligencia Artificial (IA), ya que no se considera directamente relacionado con el paradigma naturalístico. La intención del paradigma naturalístico está lejos de considerar la inferencia en la construcción de lenguajes de propósito general, tampoco se pretende que los lenguajes aprendan, etiqueten, clasifiquen o se entrenen. Sin embargo, el considerar campos de investigación relacionados con Matemáticas, Lingüística e IA arrojaron algunos resultados relacionados relevantes.

Tabla A.1: Número de resultados por término de búsqueda.

Término(s) de búsqueda	Años de publicación	Campos de investigación	Número de Resultados
“Naturalistic programming”	>2003		8
“Naturalistic programming” (Full data)	>2003	Information and Computing Sciences OR Computer Software OR Artificial Intelligence and Image Processing Information Systems	33
“Natural language programming”	>2003	Information and Computing Sciences OR Artificial Intelligence and Image Processing OR Computer Software OR Information Systems	32
“Natural language programming” (Full data)	>2003	Information and Computing Sciences OR Information Systems	342
“Naturalistic programming” OR “Natural language programming” (Full data)	>2003	Information and Computing Sciences OR Artificial Intelligence and Image Processing OR Information Systems OR Language, Communication and Culture OR Linguistics OR Computer Software OR Engineering OR Computation Theory and Mathematics	410

Tabla A.1 continuación de la página anterior.

Término(s) de búsqueda	Años de publicación	Campo de investigación	Número de Resultados
“Naturalistic programming” OR “Natural language programming” (Full data)	>2015	Information and Computing Sciences OR Artificial Intelligence and Image Processing OR Information Systems OR Language, Communication and Culture OR Linguistics OR Computer Software OR Engineering OR Computation Theory and Mathematics	204
“Programming languages” AND “Expressiveness”	>1990	Information and Computing Sciences OR Computer Software OR Information Systems OR Artificial Intelligence and Image Processing	356

Como se observa, la consulta del término *Naturalistic programming* en el título y resumen de las publicaciones arrojó 8 resultados, que de hecho son los más relevantes del paradigma, las editoriales de estas publicaciones son ACM, IEEE, Applied Science y Elsevier.

El término *Natural language programming* arrojó resultados relevantes relacionados con el paradigma y las preguntas de investigación, motivo por el cual ambos términos se utilizaron en combinación para reportar el flujo de información a través de las diferentes fases de la revisión (número de publicaciones encontradas, escaneadas, elegidas e incluidas). Este cálculo se realizó de acuerdo al Flujo de trabajo de PRISMA (Moher et al., 2009), el cual se presenta en la Figura A.1). El número de publicaciones incluidas se obtuvo de elegir a las 8 publicaciones exclusivamente relevantes del paradigma naturalístico además de otros hallazgos relacionados. Se incluye un registro identificado en la literatura gris, el lenguaje de escritura en inglés plano llamado Cal-4700⁷, el cual cumple con la filosofía del paradigma naturalístico y permite contestar una de las preguntas de investigación.

La Tabla A.2 muestra la cantidad de publicaciones relacionadas de los últimos 5 años por fuente, se presentan las principales fuentes de mayor a menor número de publicaciones. Se observa que la fuente con mayor número de publicaciones relacionadas es *Lecture Notes in Computer Science* de la editorial Springer.

Tabla A.2: Número de Publicaciones por fuente. Datos generados con la herramienta Dimensions.

Fuente	Publicaciones	Citas
Lecture Notes in Computer Science	15	30
Advances in Intelligent Systems and Computing	6	19
IFIP Advances in Information and Communication Technology	4	4
Communications in Computer and Information Science	4	8
arXiv	4	0
ACM Computing Surveys	3	73
Engineering Applications of Artificial Intelligence	3	0
IEEE Access	2	13
Journal of Systems and Software	2	21
IEEE Latin America Transactions	2	0
IEEE Transactions on Automation Science and Engineering	1	18

⁷<http://www.osmosian.com/>

Tabla A.2 continuación de la página anterior.

Fuente	Publicaciones	Citas
Science of Computer Programming	1	1
Communications of the ACM	1	0
IEEE Transactions on Software Engineering	1	7
Software Engineering	1	19
Software and Systems Modeling	1	5
Empirical Software Engineering	1	0
ACM SIGSOFT Software Engineering Notes	1	2
Proceedings of the ACM on Programming Languages	1	31
Automated Software Engineering	1	15
Journal of Computer Languages	1	0

La Figura A.2 muestra el número de publicaciones por año, de los últimos cinco años. Como se observa, el número de publicaciones relacionadas ha aumentado año con año, lo que demuestra el interés en el desarrollo de software utilizando como base el lenguaje natural.

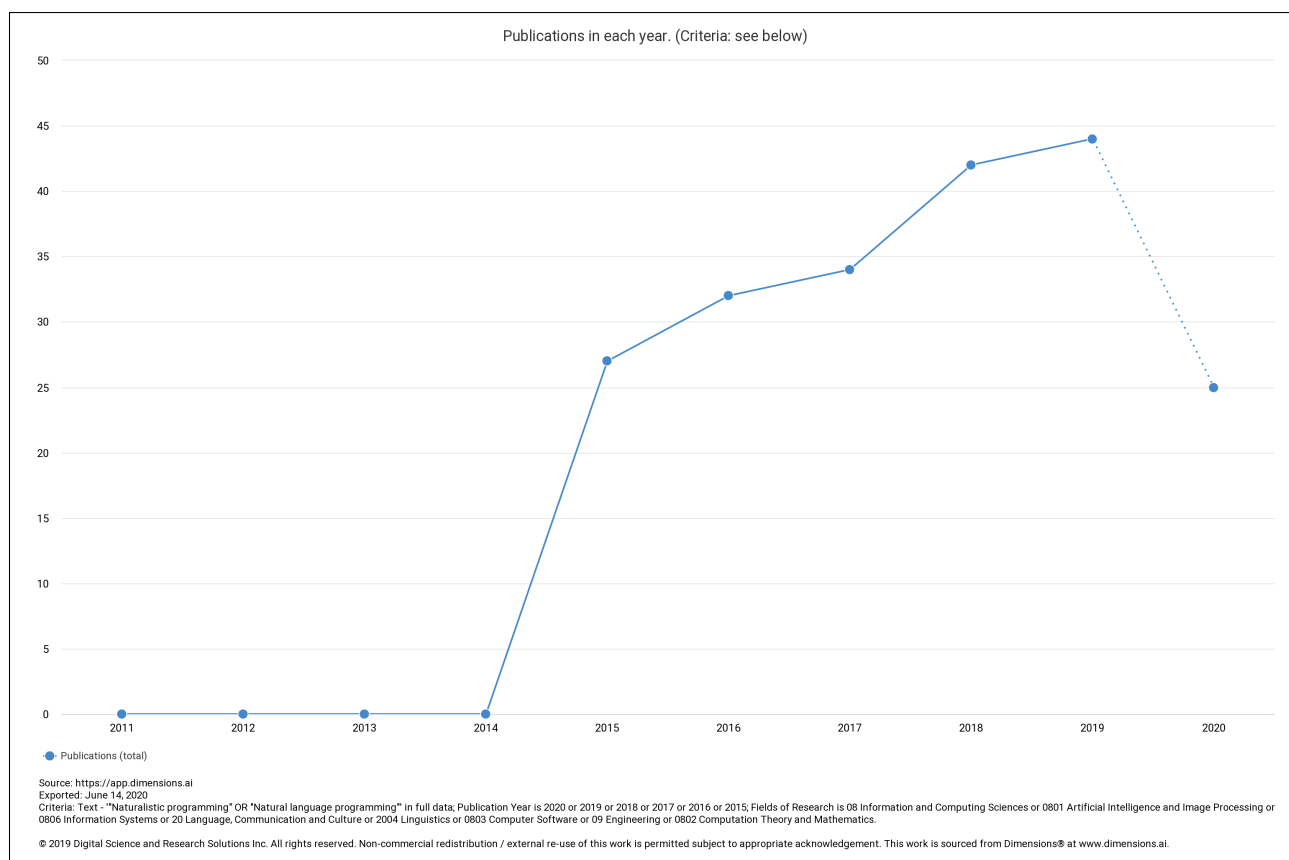


Figura A.2: Número de publicaciones por año (generada con Dimensions).

En la Figura A.3 se observa el número de publicaciones por campo de investigación (de los últimos cinco años), la mayoría publicadas en las Ciencias de la Computación e Información. Se aprecian campos relacionados con las ciencias médicas, ya que aunque se aplicaron los filtros correspondientes, es posible que las publicaciones pertenezcan a más de un campo de investigación.

Diagrama de Flujo PRISMA

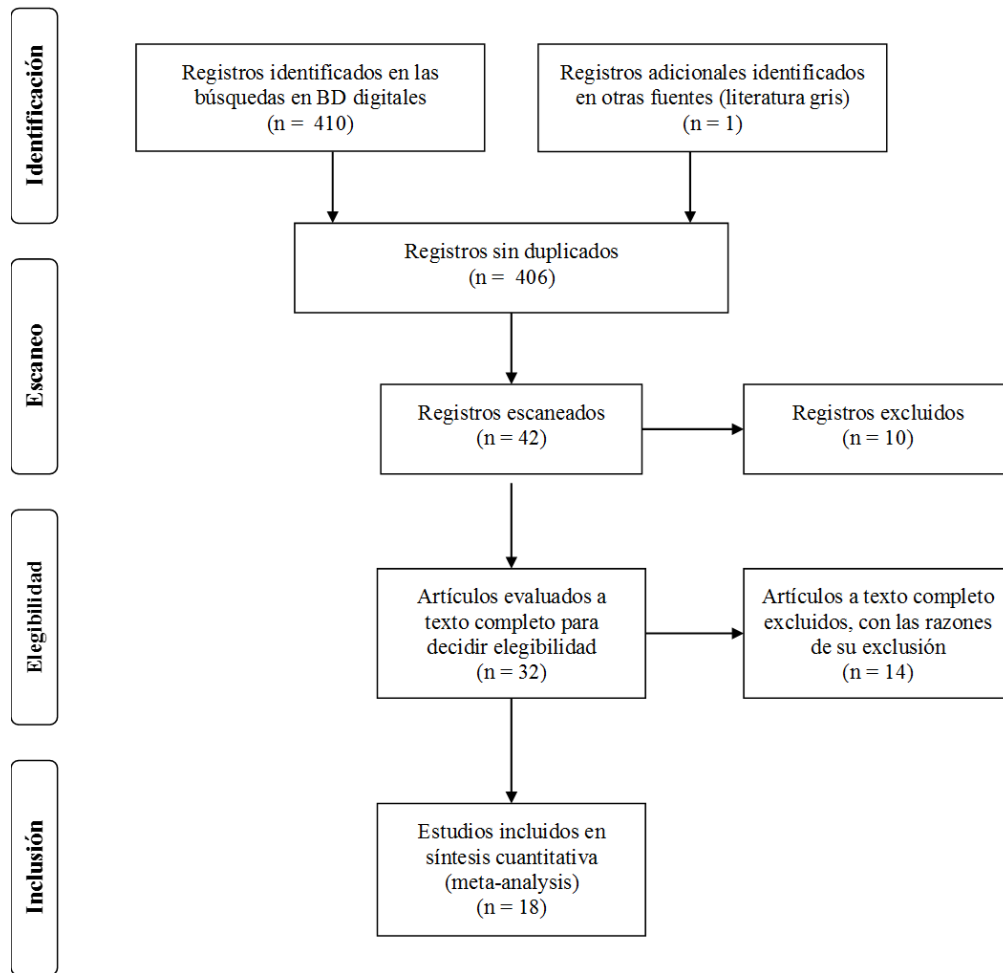


Figura A.1: Flujo de información a través de las diferentes fases de la revisión sistemática (Moher et al., 2009).

La última fila de la Tabla A.1, muestra los resultados de la combinación de términos “*Programming languages*”, “*AND*”, “*Expressiveness*”; con apoyo de la herramienta VOSviewer, se analizaron los resultados de las diferentes consultas, identificando al autor Felleisen como uno de los más citados en la búsqueda de expresividad mediante el lenguaje natural, su publicación (Felleisen, 1991) presenta una notación formal para comparar la expresividad de diferentes lenguajes de programación. Este hallazgo permitió buscar trabajos recientes que basaron sus propuestas en los aportes de Felleisen. Otro dato importante obtenido de la elaboración de mapas bibliométricos es que los países con mayor número de publicaciones citadas son Estados Unidos, Reino Unido y Alemania, éste último es el origen del primer lenguaje naturalístico de propósito general reportado, (Knöll & Mezini, 2006).

Se observa un ejemplo de mapa bibliométrico en la Figura A.4. Este mapa muestra la ocurrencia y relación de términos resultantes de las consultas correspondientes a: “*Naturalistic*

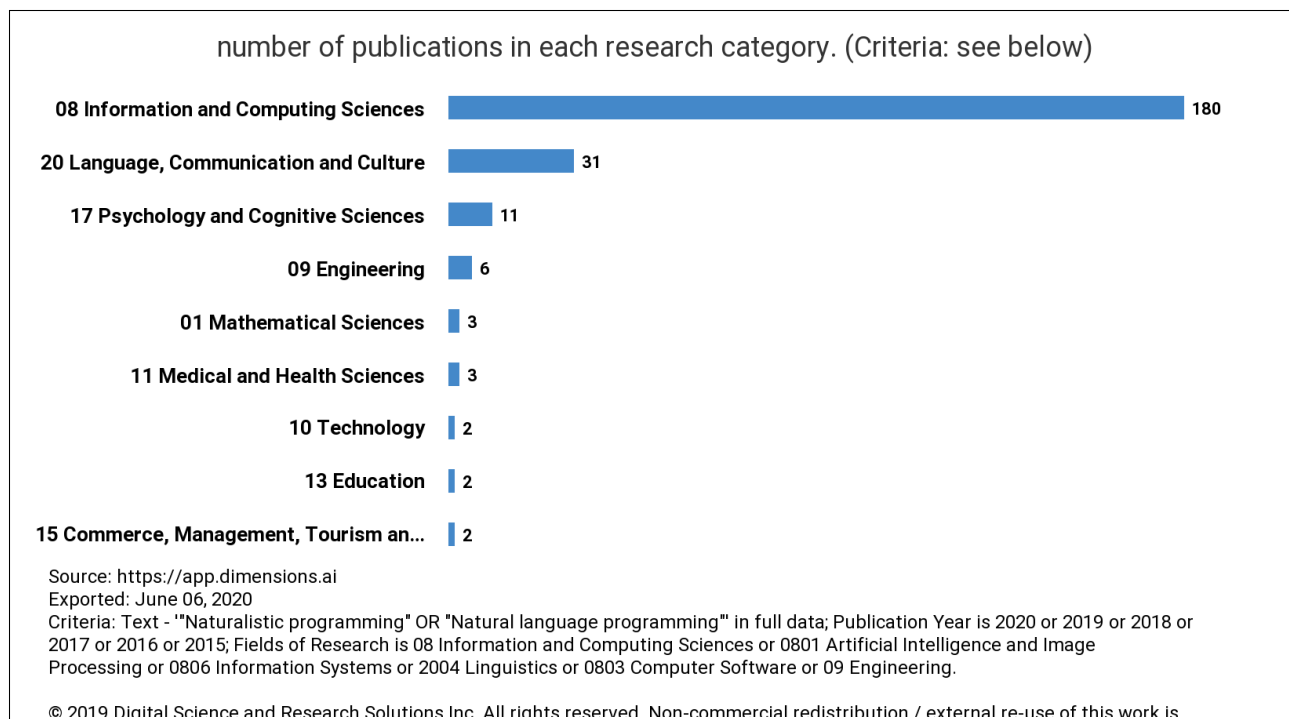


Figura A.3: Número de publicaciones por campo de investigación. Generada con Dimensions.

programming", "*Expressiveness*" y "*General Purpose Natural Language Programming*". La Figura A.5 muestra un acercamiento al conjunto de clústers relacionados más representativo, en la figura se observan términos como:

- *domain specific language*
- *natural language processing.*
- *programming language.*
- *object oriented programming.*
- *logic programming.*
- *model.*
- *method.*
- *expressive power.*

Como resultado de éste y otros análisis se llegó a la conclusión de que en la búsqueda y comprobación de expresividad en lenguajes de programación, se han considerado propuestas relacionadas con lenguajes específicos de dominio, procesamiento de lenguaje natural, programación orientada a objetos, programación lógica, entre otros.



Figura A.4: Mapa bibliométrico de términos relacionados.

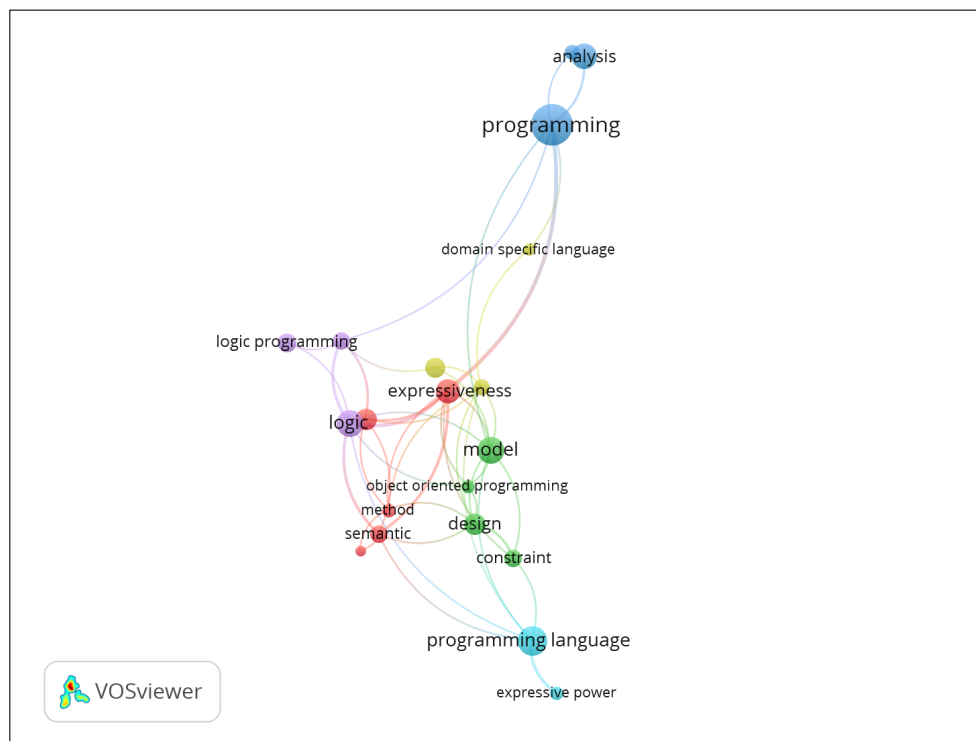


Figura A.5: Conjunto de clústers de términos relacionados

En la Tabla B.1 (Apéndice B) se muestran los resultados exclusivamente relacionados

con el paradigma naturalístico, se incluye el lenguaje Cal-4700 encontrado en la literatura gris. Otros resultados relevantes muestran el interés por lograr una mayor expresividad al programar mediante el uso del lenguaje natural, algunos hallazgos se discuten en la siguiente sección.

A.3. Discusión

Del análisis de los resultados obtenidos se pudo dar respuesta a cuatro de las cinco preguntas de investigación:

1. ¿Cuáles son los lenguajes naturalísticos de propósito general actuales?

Los lenguajes naturalísticos de propósito general actuales son Pegasus (Knöll & Mezini, 2006), SN (Pulido-Prieto & Juárez-Martínez, 2019) y Cal-4700 (Rzeppa & Rzeppa, 2019a).

2. ¿Qué propuestas se reportan asociadas a la expresividad de los lenguajes de programación?

Las propuestas se han enfocado a la construcción de lenguajes específicos de dominio (Mernik et al., 2005), generadores de pseudódigo (Din & Adnan, 2019), creación de intérpretes de lenguaje natural (Landhäußer et al., 2017), creación de cuasi lenguajes (Yin, 2010), procesamiento de lenguaje natural, específicamente enfocadas al área de robótica y a la programación por parte de usuarios finales (*End-user Programming*) (Schlegel et al., 2019), intérpretes de lenguaje natural para consultas de bases de datos (More et al., 2020), entre otros.

Muchas de las propuestas citadas se limitan a generar código fuente a partir de oraciones de lenguaje natural, en lenguajes de otros paradigmas de programación, cuando la intención del paradigma naturalístico es generar programas expresivos, pero también directamente ejecutables, independientes de otros paradigmas.

3. ¿Se encuentran casos de estudio o de aplicación reportados en la literatura?

En la literatura se reporta un caso de aplicación del paradigma naturalístico en el desarrollo de software, en (Mefteh et al., 2018) los autores generan código fuente orientado a objetos a partir de descripciones de Casos de Uso, utilizando una gramática restringida. El código fuente generado debe completarse de acuerdo a abstracciones de otro paradigma, lo cual representa una limitación importante.

4. ¿Qué métodos o procedimientos de Ingeniería de software son aplicables al paradigma?

No se encontró reporte alguno en la literatura.

5. ¿Qué términos son similares en la literatura con el término “programación naturalística”?

Programación en inglés plano, programación en lenguaje natural.

Otros hallazgos interesantes son el estudio de áreas como *Programming by demonstration* y *End-user Programming*, lo cual permite relacionar el paradigma con otras áreas que en un principio no estaban consideradas.

Los resultados de esta revisión, permitieron actualizar las publicaciones relevantes directamente relacionadas con el paradigma (ver Tabla B.1 del Apéndice B), encontrando un lenguaje naturalístico de propósito general en la literatura gris, llamado Cal-4700 (Rzeppa & Rzeppa, 2019a). Además se comprobó que el interés por lograr una mayor expresividad al programar, data de muchos años atrás. Aunque se ha tratado de dar respuesta a esta inquietud desde diferentes enfoques, la intención de utilizar el lenguaje natural como herramienta de programación persiste como motivación común.

Como se observó al inicio de esta sección, la pregunta cuatro no fue respondida en la revisión, ninguno de los resultados mostró evidencia de que se utilice algún lenguaje naturalístico de propósito general dentro del proceso de desarrollo de software, lo cual motiva esta investigación, ya que es necesario contar con métodos de desarrollo que favorezcan la aplicación y evolución del paradigma, dadas las ventajas que propone.

Apéndice B

Tabla de datos extraídos

Tabla B.1: Publicaciones exclusivamente relacionadas con el paradigma naturalístico.

Publicación	Aportes/Propuesta	Limitaciones	#Pregunta
Beyond AOP: Toward Naturalistic Programming (Lopes et al., 2003)	Contar con tecnología reflexiva, capaz de representar relaciones anafóricas. Unidad mínima: “thing”	No incluye propuesta concreta, sólo motiva la existencia del paradigma.	2
Pegasus: First Steps Toward a Naturalistic Programming Language (Knöll & Mezini, 2006)	Lenguaje de propósito general. Características: referencia implícita, comprensión y dependencia del contexto. Traducción entre idiomas: Alemán, Inglés, Chino, Español, Hindi, Ruso y otros. Traducción a: Java, C, C++, Ruby, Python, Haskell y Prolog. Unidad mínima: “idea”	Necesita de una BD. No implementa aspectos.	1
Cal-4700 ¹	Lenguaje de programación en inglés plano, el cual cuenta con un editor, compilador y enlazador, biblioteca de subrutinas de propósito general y documentación.	Requiere de su archivo cerebro (“the noodle”) para compilar cualquier programa.	1
Naturalistic Types (Knöll et al., 2011)	Definición de tipos naturalísticos, los cuales se podrían considerar en el modelado.		1
A Survey of Naturalistic Programming Technologies (Pulido-Prieto & Juárez-Martínez, 2017)	Presenta los lenguajes naturalísticos específicos de dominio y de propósito general actuales. Útil para guiar esta investigación.		1 y 2

¹<http://www.osmosian.com/>

Tabla B.1 continuación de la página anterior.

Publicación	Aportes/Propuesta	Limitaciones	#Pregunta
A Model for Naturalistic Programming with Implementation (Pulido-Prieto & Juárez-Martínez, 2019)	Modelo para la construcción de lenguajes naturalísticos. Prototipo de lenguaje SN que incluye referencias anafóricas y catafóricas. Implementación de aspectos.	Limitado al idioma inglés y al bytecode de la Máquina Virtual de Java	1
Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications. (Mefteh et al., 2018)	Traducción de descripciones de Casos de Uso a código Java, pasando por una traducción intermedia en XML (eXtensible Markup Language).	Gramática limitada. Utiliza el patrón de diseño Model-View-Presenter. El código debe completarse.	3
Perspectives for software development using the naturalistic language SN (Alducin-Francisco et al., 2019)	Presenta la prueba de funcionalidad del lenguaje naturalístico SN.	La prueba de funcionalidad es a través de 13 programas ejecutables, no de un estudio de caso particular.	1
Naturalistic Programming: Model and Implementation (Pulido Prieto & Juárez Martínez, 2020)	Presenta y evalúa el modelo naturalístico para la construcción de lenguajes de programación de propósito general.	La evaluación no presenta un estudio de caso de aplicación, sólo una serie de encuestas realizadas a estudiantes.	1

Apéndice C

Pruebas guiadas de Cal-4700

Esta actividad se realizó el 13 de diciembre de 2021, con un grupo de cinco estudiantes del Séptimo semestre de la carrera de Ingeniería en Sistemas Computacionales, del ITO. El objetivo fue evaluar la capacidad expresiva del lenguaje naturalístico Cal-4700 con programadores que tuvieran experiencia previa con otros lenguajes de programación, para que tuvieran punto de comparación.

La guía de la práctica fue la siguiente:

1. Se les dio la bienvenida y explicó el objetivo de la práctica.
Se les pidió que mantuvieran las cámaras encendidas para observar sus reacciones durante la práctica.
2. Se les proporcionó el vínculo para descargar el material necesario ([clic aquí](#)), explicando su contenido, ejemplos y el ambiente de desarrollo de Cal-4700.
También se les proporcionó el vínculo a la página que contiene ejemplos de Cal-4700 creados por sus autores¹.
3. Utilizando la presentación se introdujo a los estudiantes al paradigma naturalístico y al lenguaje.
4. Se les mostraron ejemplos de aplicación que ellos mismos podían ejecutar en su computadora.
5. Después de crear un ejemplo base, se les pidió realizar un programa que escribiera la serie de Fibonacci², resaltando la importancia de este cálculo en la vida real.
6. Se observaron sus reacciones al programar.
7. Después tres horas se analizaron los programas resultantes.
8. Se les pidió contestar la siguiente encuesta anónima, para recabar su opinión acerca del paradigma y del lenguaje Cal-4700. Para ver la encuesta hacer [clic aquí](#).

¹<https://osmosianplainenglishprogramming.blog/>

²<https://www.educ.ar/recursos/132013/la-matematica-incrustada-en-la-inmensa-variedad-de-formas-de-vida>

Observaciones y resultados

Sólo dos estudiantes terminaron completamente el programa, el resto comentó que pudo programar y ejecutar algunas funciones pero no podían permanecer más tiempo en la sesión. No se observó alguna expresión de molestia o frustración, más bien parecían muy concentrados revisando la documentación y realizando pruebas. En la Tabla C.1 se observan los resultados de la encuesta, para cada una de las preguntas se contabilizaron las respuestas, las cuales se encuentran en la escala de Likert con los siguientes valores:

- Totalmente en desacuerdo.
- En desacuerdo.
- Neutral.
- De acuerdo.
- Totalmente de acuerdo.

Como se puede observar, cuatro de los cinco estudiantes respondieron favorablemente a cada una de las preguntas. Uno de los estudiantes respondió desfavorablemente, sin embargo contestó que sí utilizaría Cal-4700 para resolver problemas, además recomendaría su uso y el de lenguajes naturalísticos en general.

Los comentarios y sugerencias que se recabaron en la encuesta son los siguientes:

- “Me pareció muy interesante en aprender a utiliza Cal-4700 mediante el lenguaje naturalista, es algo muy nuevo en lo personal para mi, pero me pareció muy practico, considero que si se requiere algo de practica ya que estamos muy acostumbrados a lenguajes con una sintaxis y estructura muy estricta, con sentencias ya asignadas, y en este lenguaje naturalista podemos implementar nuevas sentencias, me gusto mucho esta platica y por su puesto que me gustaría seguir practicando. No logre terminar la practica realizada durante la platica, pero la terminare ya que me quede con ganas de aprender mas”.
- “Es una interfaz fácil de utilizar, en mi caso, la sintaxis es un poco complejo, ya que viene en inglés y me es un poco difícil recordar las sentencias”.
- “El modo de explicación, aprendizaje es muy eficiente al ser algo sencillo y tener conocimientos previos”.
- “Me pareció muy interesante la forma de programar, y sin duda me gustó y aprendería mas sobre Cal - 4700”.

Como se puede observar, sus comentarios en general fueron positivos, hacia la comprensión y objetivo del lenguaje. La única limitante mencionada se refiere a la dificultad de comprensión del idioma Inglés.

Tabla C.1: Encuesta de evaluación Cal-4700 con resultados.

	Totalmente en desacuerdo	En desacuerdo	Neutral	De acuerdo	Totalmente de acuerdo
1. Considero que Cal-4700:					
a. Es fácil de utilizar (aprender).	1		1		3
b. Es fácil de recordar.	1			2	2
c. Es eficiente.	1			2	2
d. Es flexible.	1			1	3
e. Su interfaz tiene manejo adecuado de errores.	1			2	2
f. Su interfaz minimalista es práctica.	1			3	1
g. Su interfaz es fácil de usar.	1			3	1
h. Su interfaz es fácil de recordar.	1			1	3
i. Es consistente con el paradigma (¿es naturalístico?)	1			1	3
j. Proporciona ayuda y documentación suficiente.	1			1	3
	SI			NO	
2. Utilizaría Cal-4700 para resolver problemas.	5				
3. Recomendaría utilizar Cal-4700.	5				
4. Recomendaría utilizar lenguajes naturalísticos.	5				

Apéndice D

Artefactos de modelado del caso de aplicación lúdico

D.1. Caso de Uso Seleccionar nivel

Este caso de uso da inicio al camino principal de la aplicación, considera que el jugador tiene la facultad de seleccionar entre tres niveles disponibles de dificultad: fácil, medio y difícil. En la Tabla D.1 se muestra la descripción del CU, la Figura D.1 considera el prototipo. En la Tabla D.2 se observa la tabulación de ideas. La Figura D.2 corresponde al DFD y en la Figura D.3 se observa el diagrama de estructura de Jackson.

Tabla D.1: Descripción del CU Seleccionar nivel.

ID	CU01
Nombre:	Seleccionar nivel
Autor(es):	José Antonio Juárez Romero
Fecha de creación:	30-abril-22
Fecha de actualización:	2-mayo-22
Actor:	Jugador
Descripción:	El jugador podrá seleccionar entre tres niveles de dificultad: fácil, medio y difícil.
Flujo Normal:	1. El sistema muestra los tres niveles de dificultad. 2. A. Ir al CU Seleccionar imagen (nivel fácil). 2. B. Ir al CU Seleccionar imagen (nivel medio). 2. C. Ir al CU Seleccionar imagen (nivel difícil). 3. Termina CU.
Flujos Alternos:	
Excepciones:	E.1 El sistema recibe una respuesta no válida. 1. Regresar al paso 1 del flujo normal.
Poscondiciones:	
Entradas:	1. Selección de nivel.
Salidas:	1. Lista de niveles (fácil, medio y difícil).
Incluye: (relación include)	CU Seleccionar imagen.
Extiende (relación extend)	
Lanzar: (relación trigger)	
Prioridad:	Alta

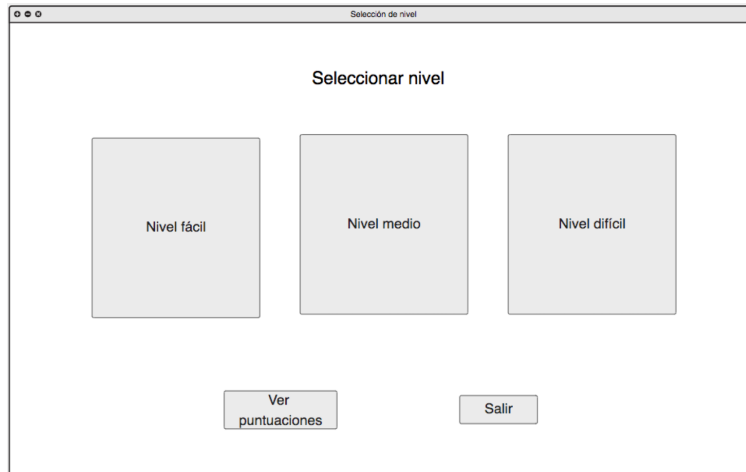


Figura D.1: Prototipo del CU Seleccionar nivel.

Tabla D.2: Tabulación de ideas del CU Seleccionar nivel.

No.	Enunciado Statement (acción)	Comando Comand (frase)	Condición Condition	Agente	Anáfora	Objeto directo Direct object	Destino (opcional)
1	Mostrar			Sistema		Nivel	
2			Pregunta	Sistema		Nivel	
2a		Nivel fácil: Carga		Sistema		CU Seleccionar imagen(Nivel fácil)	
2b		Nivel medio: Carga		Sistema		CU Seleccionar imagen(Nivel medio)	
2c		Nivel difícil: Carga		Sistema		CU Seleccionar imagen(Nivel difícil)	
6	Termina			Sistema			

En el Listado D.1 se observa el programa del CU Seleccionar Nivel. El código considera la llamada de la rutina `show main interface`, la cual realiza la función de dibujar la interfaz de usuario en la pantalla, su definición se encuentra en el programa del Listado D.6. En la Figura D.4 se aprecia el resultado de la implementación del presente CU.

Listado de código D.1: CU Seleccionar Nivel.

```

1 To show levels:
2   show main interface.
3 Loop.
4   Deque an event.
5   If the event is nil, break.
6   If the event's kind is not "left click", repeat.
7   If the event's spot is in the easy level button, show images of the easy level.
8   If the event's spot is in the medium level button, show images of the medium level.

```

9 **If the** event's spot **is** in **the** hard level button, show images of **the** hard level.
 10 **If the** event's spot **is** in **the** quit button, close **the** program; **exit**.
 11 **repeat**.

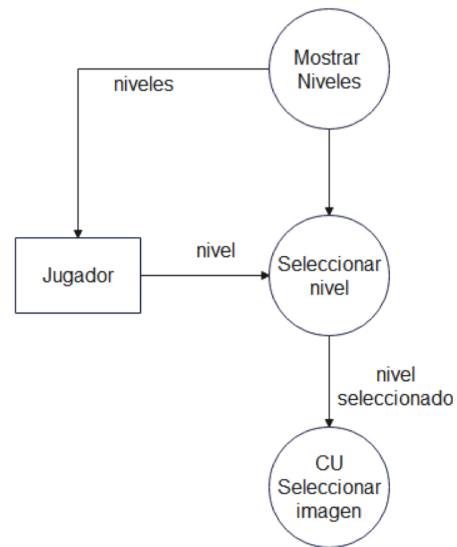


Figura D.2: DFD del CU Seleccionar nivel.

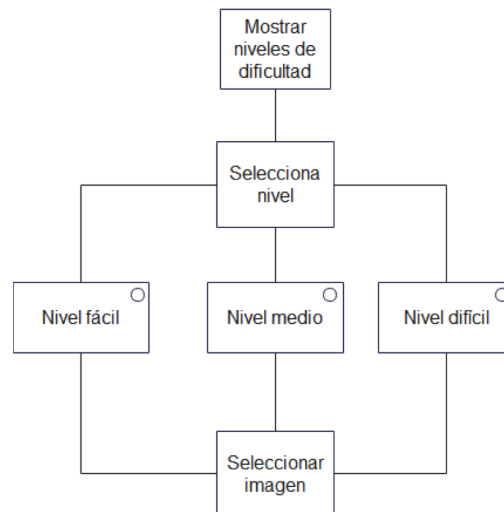


Figura D.3: Diagrama de estructura del CU Seleccionar nivel.



Figura D.4: Interfaz de usuario del CU Seleccionar nivel.

D.2. Caso de Uso Seleccionar imagen

El CU Seleccionar imagen considera que al jugador se le permite hacer una selección entre cuatro opciones de imagen del nivel previamente seleccionado. Las imágenes se muestran en miniatura. Después de seleccionar la imagen, el jugador es capaz de volver al CU Seleccionar nivel mediante el uso de un botón. En la Tabla D.3 se muestra la descripción de CU, la Figura D.5 muestra el prototipo, en la Tabla D.4 se observa la tabulación de ideas. La Figura D.6 corresponde al DFD, y finalmente, en la Figura D.7 se observa el diagrama de estructura de Jackson.

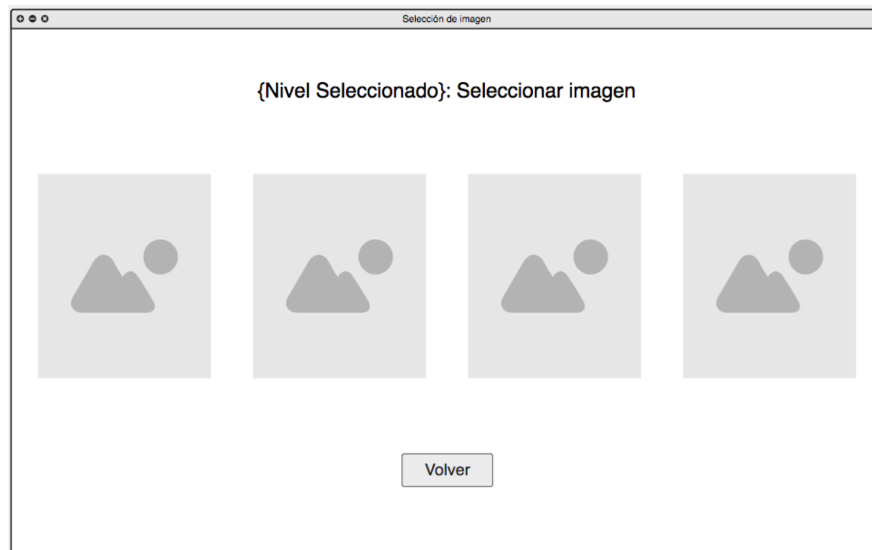


Figura D.5: Prototipo del CU Seleccionar imagen.

Tabla D.3: Descripción de CU Seleccionar imagen.

ID	CU02
Nombre:	Seleccionar imagen
Autor(es):	José Antonio Juárez Romero
Fecha de creación:	30-abril-22
Fecha de actualización:	08-mayo-22
Actor:	Jugador
Descripción:	El jugador podrá seleccionar una de las cuatro imágenes disponibles del nivel seleccionado para jugar con ella.
Flujo Normal:	1.El sistema muestra las cuatro imágenes disponibles del nivel seleccionado. 2. a. Imagen 1: Cargar CU Iniciar Juego. 2. b. Imagen 2: Cargar CU Iniciar Juego. 2. c. Imagen 3: Cargar CU Iniciar Juego. 2.D. Imagen 4: Cargar CU Iniciar Juego. 3. Termina CU.
Flujos Alternos:	1. Selecciona volver. 1.a Volver al CU Seleccionar nivel.
Excepciones:	E1. El sistema no recibe respuesta válida. 1. Regresar al paso 1 del flujo normal.
Precondiciones:	Nivel seleccionado (CU Seleccionar Nivel).
Poscondiciones:	
Entradas:	1. Imagen seleccionada.
Salidas:	1. Lista de las imágenes disponibles así como la imagen miniatura de cada una.
Incluye: (relación include)	CU Iniciar juego
Extiende (relación extend)	
Lanzar: (relación trigger)	
Prioridad:	Alta

Tabla D.4: Tabulación de ideas del CU Seleccionar imagen.

No.	Enunciado Statement (acción)	Comando Comand (frase)	Condición Condition	Agente	Anáfora	Objeto directo Direct object	Destino (opcional)
1	Mostrar			Sistema		Imagen[]	
2			Pregunta	Sistema		Imagen	
2a		Imagen 1: Carga		Sistema		CUIniciar juego	
2b		Imagen 2: Carga		Sistema		CUIniciar juego	
2c		Imagen 3: Carga		Sistema		CUIniciar juego	
2d		Imagen 4: Carga		Sistema		CUIniciar juego	
3	Termina			Sistema			

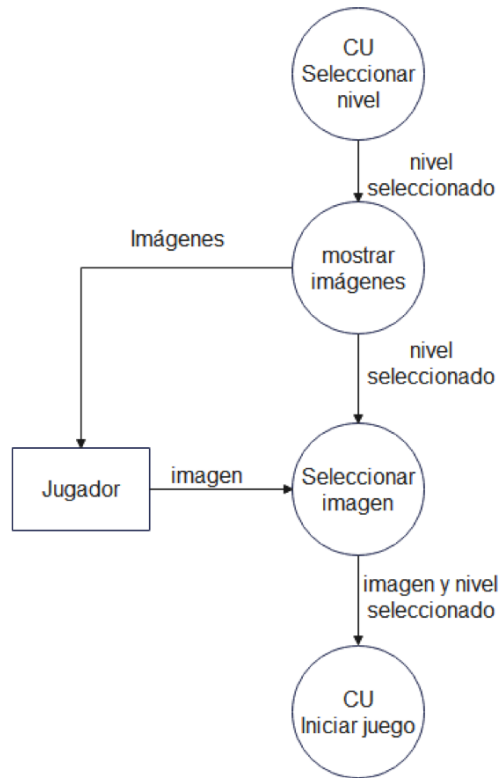


Figura D.6: DFD del CU Seleccionar imagen.

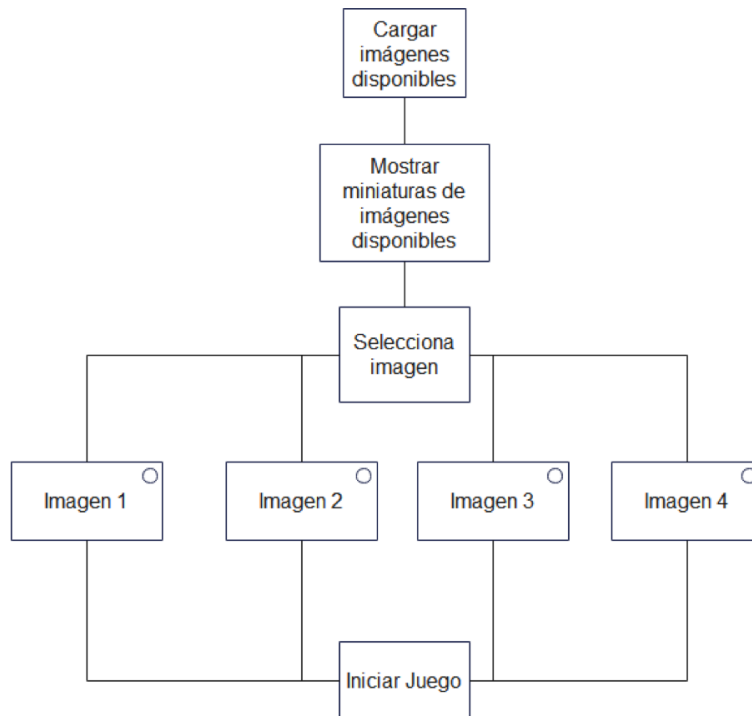


Figura D.7: Diagrama de estructura del CU Seleccionar imagen.

En el Listado D.2 se muestra a modo de ejemplo la definición de la rutina `Show images of the easy level` encargada de mostrar las imágenes disponibles para el nivel fácil, además de administrar lo que ocurre cuando el usuario selecciona alguna de éstas.

Listado de código D.2: Procedimiento que muestra las imágenes del nivel fácil en el CU Seleccionar imagen.

```
1 To show images of the easy level:
2   show level thumbnails given the easy level' difficulty .
3   Loop.
4     Deque an event.
5     If the event is nil, break.
6     If the event's kind is not "left click", repeat.
7     If the event's spot is in the picture button one,
8     go to easy difficulty puzzle given the image list' location one.
9     If the event's spot is in the picture button two,
10    go to easy difficulty puzzle given the image list' location two.
11    If the event's spot is in the picture button three,
12    go to easy difficulty puzzle given the image list' location three.
13    If the event's spot is in the picture button four,
14    go to easy difficulty puzzle given the image list' location four.
15    If the event's spot is in the return button, show levels; exit.
16  repeat.
```

En la Figura D.8 se observa la interfaz de usuario de la implementación del CU Seleccionar imagen en el nivel fácil.

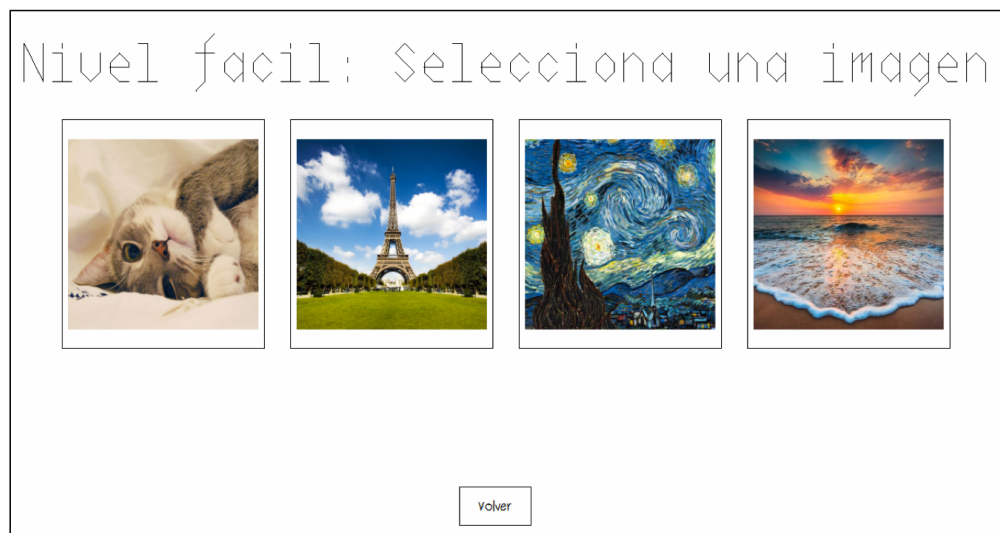


Figura D.8: Interfaz de usuario del CU Seleccionar imagen - nivel fácil.

D.3. Caso de Uso Iniciar juego

Este CU ocurre después de que el jugador ha seleccionado el nivel de dificultad y la imagen. Al iniciar el CU, el sistema solicita al Jugador su nombre, posterior a su captura

inicia el juego. Al comenzar el juego, los contadores de número de movimientos y tiempo transcurrido (en segundos) se reinician a cero, se carga la imagen seleccionada y se divide en partes de acuerdo al nivel de dificultad. Una vez que el juego ha iniciado, el Jugador puede cancelar el juego en cualquier momento.

En la Tabla D.5 se muestra la descripción del CU y en la Tabla D.6 se observa la tabulación de ideas.

Tabla D.5: Descripción del CU Iniciar juego.

ID	CU03
Nombre:	Iniciar juego
Autor(es):	José Antonio Juárez Romero
Fecha de creación:	30-abril-22
Fecha de actualización:	04-mayo-22
Actor:	Jugador
Descripción:	El jugador podrá iniciar un nuevo juego en el sistema. Previamente el sistema pedirá al usuario su nombre o <i>nickname</i> .
Flujo Normal:	1. Se reinician los contadores de número de movimientos y segundos transcurridos. 2. El sistema solicita el nombre o <i>nickname</i> del jugador. 3. El jugador lo escribe. 4. Se inicia el juego con la imagen y nivel seleccionados. 5. Termina CU.
Flujos Alternos:	1. El jugador selecciona cancelar el juego. 2. El sistema pregunta ¿Está seguro de cancelar el juego? 2.a. Sí: Ir a CU02 Seleccionar imagen. 2.b. Salir.
Excepciones:	- E1. El nombre del jugador está vacío. 1. Regresar al paso 2 del flujo normal. - E2. El sistema recibe una respuesta no válida. 1. Termina CU.
Precondiciones:	Los CU Seleccionar nivel y seleccionar imagen están concluidos.
Poscondiciones:	
Entradas:	1. Nombre o <i>nickname</i> del jugador.
Salidas:	1. La imagen seleccionada dividida en secciones.
Incluye: (relación include)	
Extiende (relación extend)	CU Terminar juego
Lanza: (relación trigger)	
Prioridad:	Alta

Tabla D.6: Tabulación de ideas del CU Iniciar juego.

No.	Enunciado Statement (acción)	Comando Comand (frase)	Condición Condition	Agente	Anáfora	Objeto directo Direct object	Destino (opcional)
1	Reiniciar			Sistema		Contador de movimientos Contador de segundos.	
2	Solicita			Sistema		Nombre o nickname	
3	Escribe			Jugador	Nombre o nickname		
4	Inicia			Sistema		Juego	
5	Termina			Sistema			

Para éste CU se consideran dos prototipos, puesto que el CU toma en cuenta dos actividades importantes: solicitar el nombre del jugador (ver Figura D.9) y comenzar el juego con la imagen dividida en secciones (ver Figura D.10). La Figura D.11 corresponde al DFD y la Figura D.12 presenta el diagrama de estructura de Jackson.

Figura D.9: Prototipo solicitar nombre de jugador.

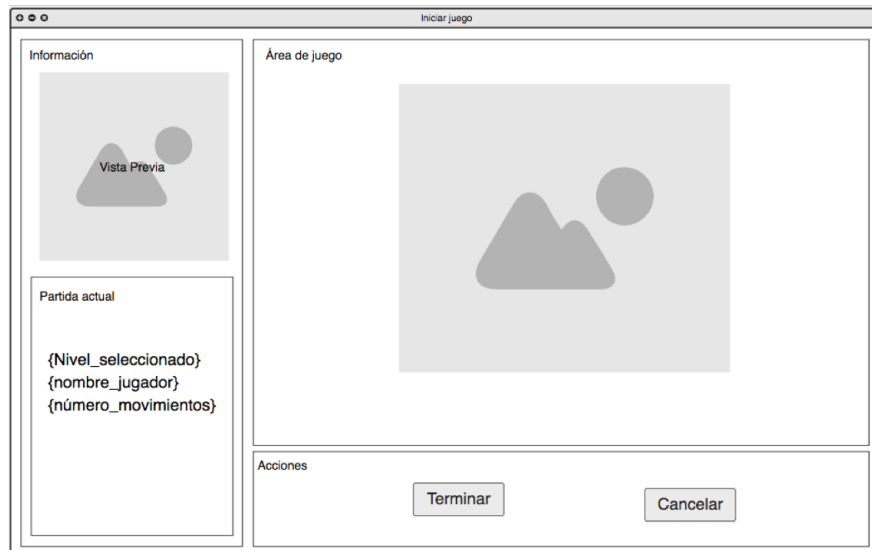


Figura D.10: Prototipo inicia juego.

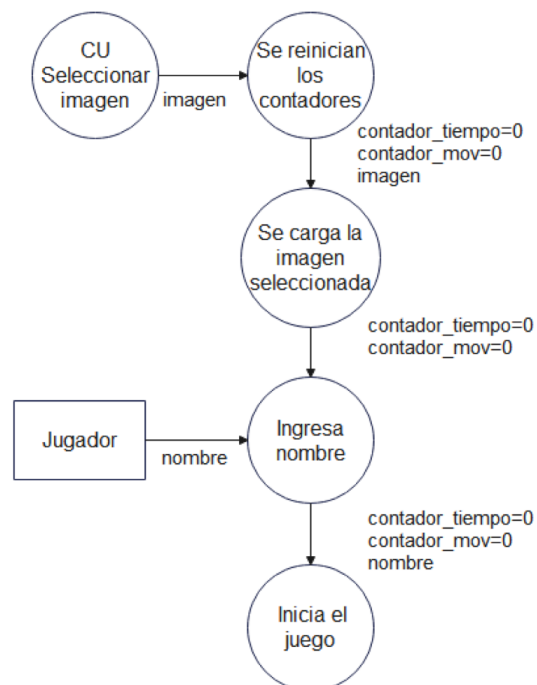


Figura D.11: DFD del CU Iniciar juego.

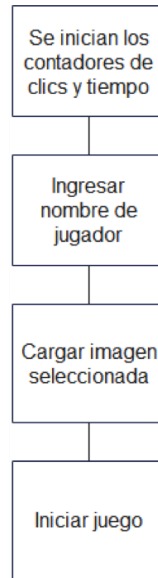


Figura D.12: Diagrama de estructura del CU Iniciar juego.

En el Listado D.3 se observa la definición del procedimiento encargado de iniciar el rompecabezas, para este ejemplo se presenta el nivel fácil. En el Listado D.4 se muestra la rutina que solicita el nombre al jugador. El procedimiento del Listado D.5 se encarga de crear los rompecabezas al recibir una dirección de imagen y un tamaño de pieza.

En la Figura D.13 se muestra la interfaz de usuario que solicita el nombre del jugador, la Figura D.14 presenta la interfaz de usuario con el juego iniciado en el nivel fácil.

Listado de código D.3: Pocedimiento que inicia el rompecabezas con la dificultad fácil.

```

1 To go to easy difficulty puzzle given a location:
2   Restart counters.
3   Request player name.
4   Create a puzzle from the location with the easy level' piece size.
5   Display the puzzle.
6   Loop.
7     Deque an event.
8     If the event is nil, break.
9     If the event's kind is not "left click", repeat.
10    If the event's spot is in the check button, check results.
11    If the event's spot is in the back button, quit; go to level easy; exit.
12    Track the mouse on the puzzle.
13  Repeat.
14  Destroy the puzzle.
  
```

Listado de código D.4: Procedimiento que solicita el nombre del jugador.

```

1 To request player name:
2   Show the arrow cursor.
3   Fill the request box with the white color.
4   Draw the request box.
5   Use small letters.
6   Write "Escribe tu nombre: " with the black pen at the top of the request box.
7   Draw the accept button.
  
```

```

8 Draw the text's box.
9 Refresh the screen.
10
11 The text has a box and a string.

```

Listado de código D.5: Procedimiento que crea los rompecabezas.

```

1 To create a puzzle from a path given a piece size:
2   Allocate memory for the puzzle.
3   Make the puzzle's box 6 inches by 6 inches.
4   Center the puzzle's box in the screen's box.
5   Fetch the puzzle's picture from the path.
6   Resize the puzzle's picture to 6 inches by 6 inches.
7   Center the puzzle's picture in the puzzle's box.
8   Cut the puzzle into pieces with the piece size.
9   Splatter the puzzle's pieces.

```



Figura D.13: Interfaz de usuario para solicitar nombre del jugador.

D.4. Caso de Uso Consultar puntuaciones

Para este CU se considera que el jugador es capaz de consultar el registro de puntuaciones en cualquier momento. El sistema permite consultar las listas de puntuaciones agrupadas por nivel de dificultad. Una vez que se ha seleccionado un nivel se muestran dos listas de puntuaciones: una para la puntuación por número de movimientos y otra para la puntuación por tiempo transcurrido. En la Tabla D.7 se muestra la descripción del CU, en la Tabla D.8 se presenta la tabulación de ideas. La Figura D.15 muestra el prototipo, la Figura D.16 corresponde al DFD, mientras que la Figura D.17 presenta el diagrama de estructura de Jackson.



Figura D.14: Interfaz de usuario del CU Iniciar juego en el nivel fácil.

Tabla D.8: Tabulación de ideas del CU Consultar puntuaciones.

No.	Enunciado Statement (acción)	Comando Comand (frase)	Condición Condition	Agente	Anáfora	Objeto directo Direct object	Destino (opcional)
1			Pregunta	Sistema		Nivel	
2	Mostrar			Sistema		Nivel	
3	Selecciona			Jugador		Nivel	
3a		Nivel 1: Carga listas		Sistema		Puntuaciones del nivel 1	
3b		Nivel 2: Carga listas		Sistema		Puntuaciones del nivel 2	
3c		Nivel 3: Carga listas		Sistema		Puntuaciones del nivel 3	
4	Mostrar	Volver		Sistema			
4a	Selecciona	Volver		Jugador			CU01
5	Termina			Sistema			

Selección de nivel

{nivel_seleccionado}: Consulta de puntuaciones

Por número de movimientos

nombre jugador	num movimientos
Ada Lovelace	09
Grace Hopper	11
Margaret Hamilton	12
Joan Clarke	14

Por tiempo

nombre jugador	segundos transcurridos
Ada Lovelace	14
Grace Hopper	23
Margaret Hamilton	24
Joan Clarke	29

Salir

Figura D.15: Prototipo para el CU Consultar puntuaciones.

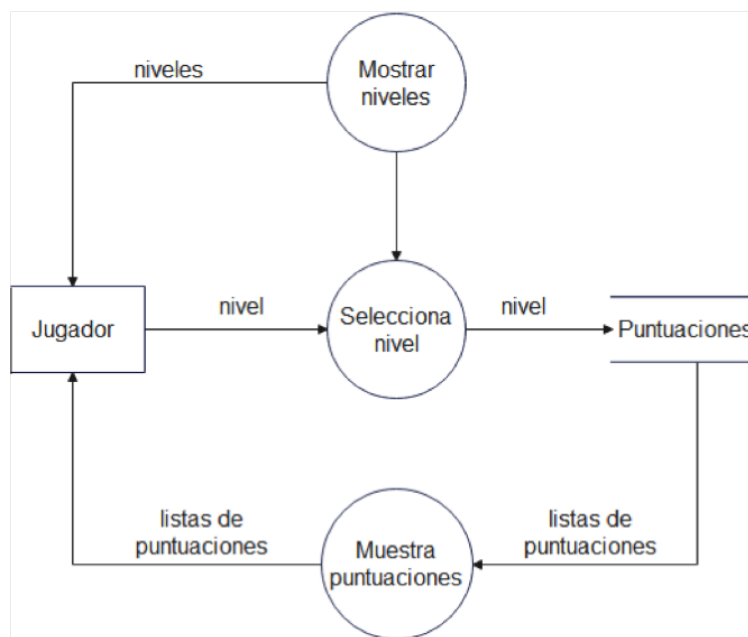


Figura D.16: DFD del CU Consultar puntuaciones.

Tabla D.7: Descripción del CU Consultar puntuaciones.

ID	CU05
Nombre:	Consultar puntuaciones
Autor(es):	José Antonio Juárez Romero
Fecha de creación:	01-dic-21
Fecha de actualización:	16-may-22
Actor:	Jugador
Descripción:	El jugador puede consultar las puntuaciones registradas.
Flujo Normal:	1. El sistema pregunta al usuario el nivel que desea consultar. 2. El sistema muestra los niveles. 3. El jugador selecciona el nivel. 3. a. Nivel 1: se cargan las dos listas del nivel 1; a la derecha se muestra la lista por tiempo y a la izquierda la lista por movimientos. 3. b. Nivel 2: se cargan las dos listas del nivel 2 3. c. Nivel 3: se cargan las dos listas del nivel 3. 4. Se muestra la opción de volver. 4a. Si: Ir al CU01. 5. Termina C.U.
Flujos Alternos:	
Excepciones:	- E1. Registro de puntuaciones vacío. 1. El sistema muestra mensaje “No hay puntuaciones que mostrar”. 2. Termina el CU.
Poscondiciones:	
Entradas:	Selección del nivel a consultar por parte del jugador.
Salidas:	El registro de puntuaciones en la forma de una tabla ordenada de manera ascendente según el puntaje.
Incluye: (relación include)	
Extiende (relación extend)	
Lanzar: (relación trigger)	
Prioridad:	Alta

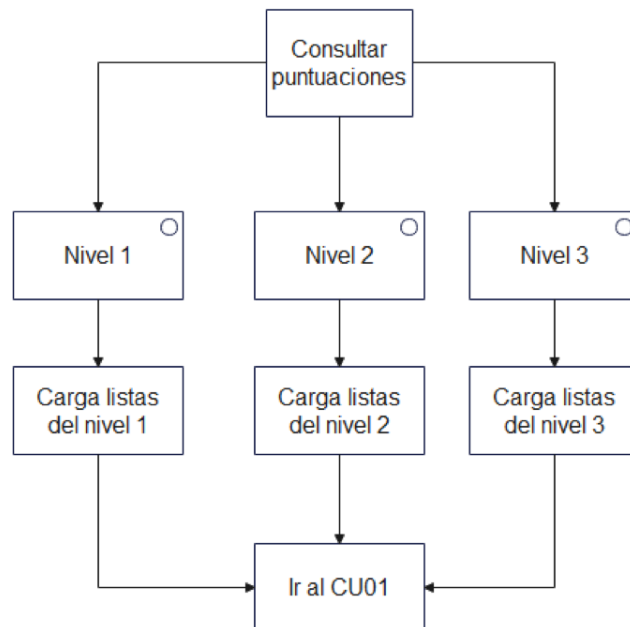


Figura D.17: Diagrama de estructura del CU Consultar puntuaciones.

En el Listado D.6 se muestra la implementación del CU Consultar puntuaciones. Primero se muestran al jugador los tres niveles de dificultad (ver Figura D.18), al elegir uno, el sistema carga las dos listas de puntuaciones: por número de movimientos y por tiempo (ver Figura D.19).



Figura D.18: Interfaz para la consulta de puntuaciones a través del nivel de dificultad.



Figura D.19: Interfaz de usuario que muestra las listas de puntuaciones del nivel fácil.

Listado de código D.6: Procedimiento que solicita el nivel de dificultad al usuario.

- 1 **To** show levels **to** query scores :
- 2 Show levels interface .

```

3  Loop.
4      Deque an event.
5      If the event is nil, break.
6      If the event's kind is not "left click", repeat.
7      If the event's spot is in the easy level button,
8      put "easy" into the score's level's difficulty; check score.
9      If the event's spot is in the medium level button,
10     put "medium" into the score's level's difficulty; check score.
11     If the event's spot is in the hard level button,
12     put "hard" into the score's level's difficulty; check score.
13     If the event's spot is in the back button, exit.
14 repeat

```

Listado de código D.7: Procedimiento para consultar el registro de puntuaciones.

```

1  The scoreM is a string.
2  The scoreT is a string.
3
4  To check a score:
5      if the score's level's difficulty is "easy",
6      put "C:\AplicacionLudica\easyScores.txt" into a path.
7      if the score's level's difficulty is "medium",
8      put "C:\AplicacionLudica\mediumScores.txt.txt" into the path.
9      if the score's level's difficulty is "hard",
10     put "C:\AplicacionLudica\hardScores.txt" into the path.
11     read the path into a buffer called original.
12     If the i/o error is not blank, debug the i/o error.
13     slap a rider on the original.
14 Loop.
15     get a string from the original given the rider.
16     if the string is blank, break.
17     get a token from the string.
18     if the token is "M", get the token from the string;
19     put the scoreM then the string into the scoreM.
20     if the token is "T", get the token from the string;
21     put the scoreT then the string into the scoreT.
22 repeat.
23 show score list.

```

D.5. Aspecto Registrar datos de puntuaciones

El CU Registrar datos de puntuaciones al ser transversal a otros requerimientos se considera un aspecto. El aspecto **Registrar datos de puntuaciones** se encarga de la tarea de escribir los resultados del juego en los archivos de texto plano que funcionan como almacenamiento de datos. El aspecto almacena el nombre del jugador, el puntaje y el criterio en el archivo del nivel correspondiente, los criterios considerados son: puntaje por número de movimientos y por tiempo transcurrido. El aspecto **Registrar datos de puntuaciones** se lanza por el CU Terminar juego en el momento en que se terminó el rompecabezas. En la Tabla D.9 se muestra la descripción del CU, utilizando sólo los elementos necesarios para un aspecto.

Tabla D.9: Descripción del aspecto Registrar datos de puntuaciones.

ID	AS01
Nombre:	Registrar datos de puntuaciones
Autor(es):	José Antonio Juárez Romero
Fecha de creación:	01-dic-21
Fecha de actualización:	12-mayo-22
Actor:	Jugador
Descripción:	Aspecto que registra en los archivos de puntuaciones para la transacción Terminar juego .
Flujo Normal:	1 El sistema almacena el nombre, numero de movimientos, y tiempo (en segundos) en el registro puntuaciones de nivel correspondiente. 2. Termina C.U.
Entradas:	Nombre, numero de movimientos, tiempo y nivel.
Prioridad:	Alta

En la Tabla D.10 se observa la tabulación de ideas. La Figura D.20 muestra el DFD correspondiente. Finalmente, la Figura D.21 muestra el diagrama de estructura de Jackson. Es importante mencionar que un aspecto no cuenta con un diseño de prototipo dado que el usuario no interactúa con él. En este sentido el aspecto funciona complementando la funcionalidad del CU que lo lanza.

Tabla D.10: Tabulación de ideas del aspecto Registrar datos de puntuaciones.

No.	Enunciado Statement (acción)	Comando Comand (frase)	Condición Condition	Agente	Anáfora	Objeto directo Direct object	Destino (opcional)
1	Almacena			Sistema		Registro de puntuaciones (nombre, tiempo, num. de movimientos y nivel).	
2	Termina			Sistema			

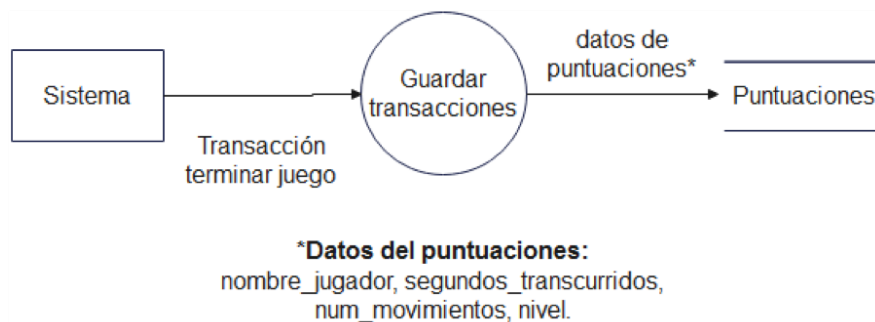


Figura D.20: DFD del aspecto Registrar datos de puntuaciones.

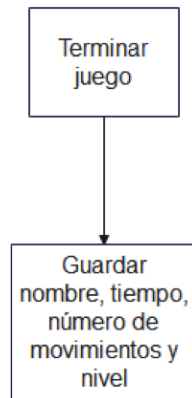


Figura D.21: Diagrama de estructura del aspecto Registrar datos de puntuaciones.

En el Listado D.8 se muestra código correspondiente al aspecto **Registrar datos de puntuaciones**, el cual tiene la función de proveer la persistencia de la información. Esto se logra mediante archivos de texto plano en formato **txt**. El aspecto hace uso de la idea atómica **Score** y su relación con las ideas **Level** y **Player** para obtener y almacenar la información necesaria.

Listado de código D.8: Aspecto Registrar datos de puntuaciones.

```

1  A score has a level Level, a player Player, a score number and a criterion string.
2
3  To register a score:
4    if the score's level's difficulty is "easy",
5    put "C:\AplicacionLudica\easyScores.txt" into a path.
6    if the score's level's difficulty is "medium",
7    put "C:\AplicacionLudica\mediumScores.txt.txt" into the path.
8    if the score's level's difficulty is "hard",
9    put "C:\AplicacionLudica\hardScores.txt" into the path.
10   if the path is not in the file system,
11   create the path in the file system.
12   read the path into a buffer called original.
13   put the score's criterion then " - " then the score's player's name then " - "
14   then the score's score into a string called auxiliar.
15   write the original then the auxiliar then the crlf string to the path.
16   If the i/o error is not blank, debug the i/o error.
  
```

D.6. El archivo interfaz

En el Listado D.9 se muestran fragmentos de código del archivo **interfaz**, el cual contiene las instrucciones para construir la interfaz gráfica.

Listado de código D.9: Contenido del archivo **interfaz**.

```

1  \fondo de pantalla
2  The background is a picture.
3
4  To draw a button:
5    Fill the button's box with the white color.
  
```

```

6   if the button's name is "Nivel facil.", create picture level
7   with the button and "facil.jpg" .
8   if the button's name is "Nivel medio.", create picture level
9   with the button and "medio.jpg" .
10  if the button's name is "Nivel dificil.", create picture level
11  with the button and "dificil.jpg" .
12  if the button's name is "Imagen1", create picture level
13  with the button and the image list' location one.
14  if the button's name is "Imagen2", create picture level
15  with the button and the image list' location two .
16  if the button's name is "Imagen3", create picture level
17  with the button and the image list' location three.
18  if the button's name is "Imagen4", create picture level
19  with the button and the image list' location four .
20  Draw the button's name in the button's box with "center".
21  Draw the button's box.
22
23  To create picture level given a button and a string:
24      allocate memory for the button's picture.
25      fetch the button's picture from the string.
26      Resize the button's picture to 2-1/2 inches by 2-1/2 inches.
27      Center the button's picture in the button's box.
28      draw the button's picture.
29
30  \inicializar los botones del CU seleccionar niveles
31  To initialize the buttons:
32      Put the screen's top plus 5-1/2 inch into a spot's y.
33      Put the screen's left plus 4 inch into the spot's x.
34      make the easy level button given the spot and "Nivel facil.".
35      Put the medium level button's right plus 7-1/2 inch into the spot's x.
36      make the medium level button given the spot and "Nivel medio.".
37      Put the hard level button's right plus 11 inch into the spot's x.
38      make the hard level button given the spot and "Nivel dificil.".
39      Put the screen's center-bottom into the spot.
40      Put the screen's bottom minus 1/2 inch into the spot's y.
41      Put the screen's left plus 8 inch into the spot's x.
42      make the quit button given the spot and "Salir".
43      Put the screen's left plus 5 inch into the spot's x.
44      make the score button given the spot and "Ver puntuaciones".
45
46  \Miniaturas de imagen.
47  To show level thumbnails given a string:
48      show the arrow cursor.
49      Draw the background.
50      Write the image message given the string.
51      Draw the picture button one.
52      Draw the picture button two.
53      Draw the picture button three.
54      Draw the picture button four.
55      Draw the return button.
56      refresh the screen.

```

D.7. El archivo puzzle

En el Listado D.10 se muestran fragmentos de código del archivo `puzzle`, el cual contiene los procedimientos para la creación y manipulación de los rompecabezas.

Listado de código D.10: Contenido del archivo `puzzle`.

```
1 \declaraciones del rompecabeza
2 A puzzle is a thing with a box, a picture and some pieces.
3 A piece is a thing with a box, a number and a picture.
4
5 To create a puzzle from a path given a piece size:
6   Allocate memory for the puzzle.
7   Make the puzzle's box 6 inches by 6 inches.
8   Center the puzzle's box in the screen's box.
9   Fetch the puzzle's picture from the path.
10  Resize the puzzle's picture to 6 inches by 6 inches.
11  Center the puzzle's picture in the puzzle's box.
12  Cut the puzzle into pieces with the piece size.
13  Splatter the puzzle's pieces.
14
15 To cut a puzzle into pieces given a piece size:
16   Draw the puzzle's picture.
17   Put the puzzle's left-top into a spot.
18   Loop.
19     Allocate memory for a piece.
20     Append the piece to the puzzle's pieces.
21     Put the spot and the spot plus the piece size into the piece's box.
22     Extract the piece's picture using the piece's box.
23     Add the piece size to the spot's left.
24     If the spot's left is less than the puzzle's right, repeat.
25     Put the puzzle's left into the spot's left.
26     Add the piece size to the spot's top.
27     If the spot's top is the puzzle's bottom, break.
28   Repeat.
29
30 To display a puzzle:
31   Draw the interface.
32   Show everything in the puzzle.
33   Draw guide image with the puzzle.
34   Loop.
35     Get a piece from the puzzle's pieces.
36     If the piece is nil, break.
37     Draw the piece's picture.
38   Repeat.
39   Refresh the screen.
40
41 To track the mouse on a puzzle:
42   Imagine a big box 10-1/5 inch by 7 inch.
43   Move the big box to the right side of the screen's box.
44   Find a piece of the puzzle given the mouse's spot.
45   If the piece is nil, exit.
46   Loop.
47     If the mouse's left button is up, exit.
48     Put the mouse's spot into a spot.
```

49 Round **the** spot **to the** nearest multiple of 1/4 inch.
50 Center **the** piece's box on **the** spot.
51 Center **the** piece's picture on **the** spot.
52 Display **the** puzzle.
53 if **the** piece's box **is** not in **the** big box, **exit**.
54 **Repeat**.

Referencias bibliográficas

- Alducin-Francisco, L.-M., Juarez-Martinez, U., Pelaez-Camarena, S. G., Rodriguez-Mazahua, L., Abud-Figueroa, M.-A., & Pulido-Prieto, O. (2019). Perspectives for software development using the naturalistic language SN. In *2019 8th International Conference On Software Process Improvement (CIMPS)* (pp. 1–11).
- Awasthi, I., Gupta, K., Bhogal, P. S., Anand, S. S., & Soni, P. K. (2021). Natural Language Processing (NLP) based Text Summarization - A Survey. In *2021 6th International Conference on Inventive Computation Technologies (ICICT)* (pp. 1310–1317).
- Baniassad, E. & Clarke, S. (2004). Theme: an approach for aspect-oriented analysis and design. In *Proceedings. 26th International Conference on Software Engineering* (pp. 158–167). ISSN: 0270-5257.
- Baudinet, M. (1989). Temporal logic programming is complete and expressive. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '89* (pp. 267–280). New York, NY, USA: Association for Computing Machinery.
- Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B., Conallen, J., & Houston, K. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, third edition.
- Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press.
- Budgen, D. (2003). *Software Design*. International Computer Science Series. Pearson Addison Wesley, second edition.
- Chandra, A. & Manna, Z. (1975). On the power of programming features. *Computer Languages*, 1(3), 219 – 232.
- Clark, B. F. (2019). Case Study at a Major US Bank. USE CASE site.
- Cochrane-Collaboration, T. (2005). Glossary terms in the Cochrane Collaboration. <https://community.cochrane.org/glossary>.
- Din, A. U. & Adnan, A. (2019). Text to Code: Pseudo Code Generation. In P. C. Vinh & A. Rakib (Eds.), *Context-Aware Systems and Applications, and Nature of Computation and Communication*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (pp. 20–37). Cham: Springer International Publishing.

- Felleisen, M. (1991). On the expressive power of programming languages. *Science of Computer Programming*, 17(1), 35–75.
- Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., & Tobin-Hochstadt, S. (2018). A programmable programming language. *Communications of the ACM*, 61(3), 62–71.
- Friedrich, F., Mendling, J., & Puhlmann, F. (2011). Process Model Generation from Natural Language Text. In H. Mouratidis & C. Rolland (Eds.), *Advanced Information Systems Engineering*, Lecture Notes in Computer Science (pp. 482–496). Berlin, Heidelberg: Springer.
- Fu, Z. (2019). An introduction of deep learning based word representation applied to natural language processing. In *2019 International Conference on Machine Learning, Big Data and Business Intelligence (MLBDI)* (pp. 92–104).
- Galster, M., Weyns, D., Tang, A., Kazman, R., & Mirakhorli, M. (2018). From Craft to Science: The Road Ahead for Empirical Software Engineering Research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)* (pp. 77–80).
- Guetterman, T. C., Chang, T., DeJonckheere, M., Basu, T., Scruggs, E., & Vydiswaran, V. G. V. (2018). Augmenting Qualitative Text Analysis with Natural Language Processing: Methodological Study. *Journal of Medical Internet Research*, 20(6), 231.
- Hernández-González, L. A., Juárez-Martínez, U., & Alducin-Francisco, L. M. (2020). Evolution of Naturalistic Programming: A Need. In J. Mejia, M. Muñoz, I. Rocha, & Y. Quiñonez (Eds.), *New Perspectives in Software Engineering*, Advances in Intelligent Systems and Computing (pp. 185–198). Cham: Springer International Publishing.
- Hoare, C. (1989). The varieties of programming languages. In *International Joint Conference on Theory and Practice of Software Development, Lecture Notes in Computer Science* (pp. 1–18).: Springer. Berlín.
- IEEE, C. S. (2014). *Software Engineering Body of Knowledge*. IEEE, v3 edition.
- ISO/IEC (2011). TR 29110-5-1-2:2011(E) -Software engineering-Lifecycle profiles for Very Small Entities (VSEs) Part 5-1-2: Management and engineering guide: Generic profile group: Basic profile.
- ISO/IEC (2016). TR 29110-1:2016 - Systems and Software Engineering-Lifecycle profiles for Very Small Entities (VSEs)-Part 1: Overview.
- ISO/IEC/IEEE (2008). Std 12207- Standard for Systems and Software Engineering - Software Life cycle Processes.
- ISO/IEC/IEEE (2017). Std 24765- International Standard - Systems and Software Engineering–Vocabulary.

- Ivanchikj, A. & Pautasso, C. (2019). Sketching Process Models by Mining Participant Stories. In T. Hildebrandt, B. F. van Dongen, M. Röglinger, & J. Mendling (Eds.), *Business Process Management Forum*, Lecture Notes in Business Information Processing (pp. 3–19). Cham: Springer International Publishing.
- Jackson, M. (2002). *Jackson Development Methods*. American Cancer Society.
- Jacobson, I. & Ng, P.-W. (2004). *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.
- Jacobson, I., Spence, I., & Kerr, B. (2016). Use-case 2.0. *Communications of the ACM*, 59(5), 61–69.
- Juárez-Romero, J. A. (2022). Desarrollo de software utilizando el lenguaje naturalísticoCal-4700. Master’s thesis, Instituto Tecnológico de Orizaba.
- Kazman, R. (2017). Software Engineering. *Computer*, 50(7), 10–11.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-m., & Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (pp. 220–242).: Springer-Verlag.
- Kitchenham, B. & Charters, S. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report 2.3, School of Computer Science and Mathematics, Keele University and Department of Computer Science, University of Durham, UK.
- Knöll, R. (2019). Pegasus Natural Programming. TU Darmstadt. <http://www.pegasus-project.org/en/Welcome.html>.
- Knöll, R., Gasiunas, V., & Mezini, M. (2011). Naturalistic Types. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011 (pp. 33–48). New York, NY, USA: ACM. event-place: Portland, Oregon, USA.
- Knöll, R. & Mezini, M. (2006). Pegasus: First Steps Toward a Naturalistic Programming Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA ’06 (pp. 542–559). New York, NY, USA: ACM. event-place: Portland, Oregon, USA.
- Lample, G., Ott, M., Conneau, A., Denoyer, L., & Ranzato, M. (2018). Phrase-Based & Neural Unsupervised Machine Translation. *arXiv:1804.07755 [cs]*. arXiv: 1804.07755.
- Landhäuser, M., Weigelt, S., & Tichy, W. F. (2017). NLCI: a natural language command interpreter. *Automated Software Engineering*, 24(4), 839–861.
- Lieberman, H. & Liu, H. (2006). Feasibility Studies for Programming in Natural Language. Human-Computer Interaction Series (pp. 459–473). Dordrecht: Springer Netherlands.

- Liu, X. & Wu, D. (2018). *From Natural Language to Programming Language*, chapter 4, (pp. 110–130). IGI Global.
- Lopes, C. V., Dourish, P., Lorenz, D. H., & Lieberherr, K. (2003). Beyond AOP: Toward Naturalistic Programming. *SIGPLAN Notices*, 38(12), 34–43.
- Mai, P. X., Pastore, F., Goknil, A., & Briand, L. C. (2018). A Natural Language Programming Approach for Requirements-Based Security Testing. In *Proc. IEEE 29th Int. Symp. Software Reliability Engineering (ISSRE)* (pp. 58–69).
- Medeiros, J., Goulão, M., Vasconcelos, A., & Silva, C. (2016). Towards a Model about Quality of Software Requirements Specification in Agile Projects. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 236–241).
- Mefteh, M., Bouassida, N., & Ben-Abdallah, H. (2018). Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications. *Science of Computer Programming*, 166, 89–119.
- Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 316–344.
- Moher, D., Liberati, A., Tetzlaff, J., Altman, D. G., & Group, T. P. (2009). Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *PLOS Medicine*, 6(7), e1000097.
- More, P., Kudale, B., & Deshmukh, P. (2020). Natural Language Interface for Relational Database. In A. Pandian, T. Senjyu, S. M. S. Islam, & H. Wang (Eds.), *Proceeding of the International Conference on Computer Networks, Big Data and IoT (ICCBI - 2018)*, Lecture Notes on Data Engineering and Communications Technologies (pp. 499–505). Cham: Springer International Publishing.
- Ni, Y., Zhang, T., Xu, L., & Han, P. (2021). : (pp. 150–157).: IEEE.
- Parihar, A. S., Thapa, S., & Mishra, S. (2021). Hate Speech Detection Using Natural Language Processing: Applications and Challenges. In *2021 5th International Conference on Trends in Electronics and Informatics (ICOEI)* (pp. 1302–1308).
- Paterson, M. S. & Hewitt, C. E. (1970). Comparative schematology. In *Comparative schematology* (pp. 119–127). New York, NY, USA: Association for Computing Machinery.
- Pressman, R. S. (2010). *Software Engineering, A Practitioner's Approach*. McGraw-Hill, seventh edition edition.
- Pulido-Prieto, O. (2019). *Modelo conceptual para la implementación de Lenguajes de Programación Naturalísticos de propósito general*. PhD thesis, Tecnológico Nacional de México. Instituto Tecnológico de Orizaba.
- Pulido-Prieto, O. & Juárez-Martínez, U. (2017). A Survey of Naturalistic Programming Technologies. *ACM Comput. Surv.*, 50(5), 70:1–70:35.

- Pulido-Prieto, O. & Juárez-Martínez, U. (2019). A Model for Naturalistic Programming with Implementation. *Applied Sciences*, 9(18), 3936.
- Pulido Prieto, O. & Juárez Martínez, U. (2020). Naturalistic Programming: Model and Implementation. *IEEE Latin America Transactions*, 18(07), 1230–1237.
- Rajput, A. (2020). Chapter 3 - Natural Language Processing, Sentiment Analysis, and Clinical Analytics. *Next Gen Tech Driven Personalized Med&Smart Healthcare* (pp. 79–97). Academic Press.
- Rzeppa, G. & Rzeppa, D. (2019a). The Osmosian Order of Plain English Programmers. <http://www.osmosian.com/>.
- Rzeppa, G. & Rzeppa, D. (2019b). The Osmosian Order of Plain English Programmers Blog. <https://osmosianplainenglishprogramming.blog>.
- Rzeppa, G. & Rzeppa, D. (2020). Natural Language Programming. A computer science portal for geeks. <https://www.geeksforgeeks.org/natural-language-programming/>.
- Schlegel, V., Lang, B., Handschuh, S., & Freitas, A. (2019). Vajra: step-by-step programming with natural language. In *Proceedings of the 24th International Conference on Intelligent User Interfaces, IUI '19* (pp. 30–39). Marina del Ray, California: Association for Computing Machinery.
- Siddaway, A. (2014). What is a Systematic Literature Review and How Do I Do One. In *Semantic Scholar*. <https://www.semanticscholar.org/>.
- Taylor, R. N., Medvidovic, N., & Dashofy, E. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley.
- Turner, M. (2010). Digital Libraries and Search Engines for Software Engineering Research: An Overview. Evidence-Based Software Engineering. <http://community.dur.ac.uk/ebse/tools.php>.
- Vasilakes, J., Zhou, S., & Zhang, R. (2021). Chapter 6 Natural language processing. In *Machine Learning in Cardiovascular Medicine* (pp. 123–148).
- Wang, C., Pastore, F., Goknil, A., & Briand, L. C. (2019). Automatic Generation of System Test Cases from Use Case Specifications: an NLP-based Approach. *arXiv:1907.08490 [cs]*. arXiv: 1907.08490.
- Wieggers, K. & Beatty, J. (2013). *Software Requirements*. Microsoft Press, third edition.
- Yin, P. (2010). Natural Language Programming Based on Knowledge. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, volume 2 (pp. 69–73).
- Yourdon, E. (1993). *Análisis Estructurado Moderno*. Prentice-Hall Hispanoamericana S.A., primera edición edition.