EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Orizaba

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCIÓN I.- TESIS

TRABAJO PROFESIONAL

"DESARROLLO DE SOFTWARE UTILIZANDO EL LENGUAJE NATURALÍSTICO CAL-4700"

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:

I.S.C. José Antonio Juárez Romero

DIRECTOR DE TESIS:

Dr. Ulises Juárez Martínez

CODIRECTOR DE TESIS:

M.I.S. Lisbeth Alejandra Hernández González

ORIZABA, VERACRUZ, MÉXICO. NOVIEMBRE 2022

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Orizaba
División de Estudios de Posgrado e Investigación

Orizaba, Veracruz, 11/noviembre/2022
Dependencia: **División de Estudios de Posgrado e Investigación**
Asunto: **Autorización de Impresión**
OPCION: **I**

C. JOSÉ ANTONIO JUÁREZ ROMERO
CANDIDATO A GRADO DE MAESTRO EN:
SISTEMAS COMPUTACIONALES
P R E S E N T E.-

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

" Desarrollo de software utilizando el lenguaje naturalístico Cal-4700"

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

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

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

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

OG-13-F06

Instituto Tecnológico de Orizaba
División de Estudios de Posgrado e Investigación

Orizaba, Veracruz, 20/septiembre/2022
Asunto: **Revisión de trabajo escrito**

C. MARIO LEONCIO ARRIOJA RODRÍGUEZ
JEFE DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN
P R E S E N T E.-

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

JOSÉ ANTONIO JUÁREZ ROMERO

La cual lleva el título de:

Desarrollo de software utilizando el lenguaje naturalístico Cal-4700

Y concluyen que se acepta.

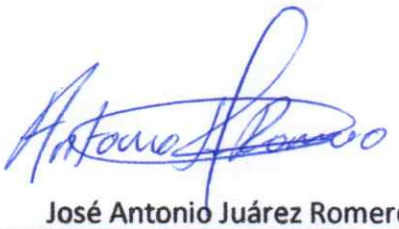
A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

PRESIDENTE:	DR. ULISES JUÁREZ MARTÍNEZ	FIRMA
SECRETARIO:	DRA. LISBETH RODRÍGUEZ MAZAHUA	FIRMA
VOCAL:	M.C. MARÍA ANTONIETA ABUD FIGUEROA	FIRMA
VOCAL SUP.:	M.I.S. LIZBETH ALEJANDRA HERNÁNDEZ GONZÁLEZ	FIRMA

TA-09-18

CARTA DE ORIGINALIDAD

En la ciudad de Orizaba, Veracruz, el día 22 del mes de noviembre del año 2022, el (la) que suscribe José Antonio Juárez Romero, alumno (a) del programa de Maestría en Sistemas Computacionales con número de control M09011274, manifiesta que es autor (a) del trabajo de tesis titulado "Desarrollo de software utilizando el lenguaje naturalístico Cal-4700" y declaro que el trabajo es original, ya que sus contenidos son producto de mi directa contribución intelectual. Todos los datos y las referencias a materiales ya publicados están debidamente identificados con su respectivo crédito e incluidos en las notas bibliográficas y en las citas que se destacan como tal y, en los casos que así lo requieran, cuento con las debidas autorizaciones de quienes poseen los derechos patrimoniales. Por lo tanto, me hago responsable de cualquier litigio o reclamación relacionada con derechos de propiedad intelectual, exonerando de toda responsabilidad al Tecnológico Nacional de México / Instituto Tecnológico de Orizaba.



José Antonio Juárez Romero

Nombre y firma

CARTA DE CESIÓN DE DERECHOS

En la ciudad de Orizaba, Veracruz, el día 22 del mes de noviembre del año 2022, el (la) que suscribe José Antonio Juárez Romero, alumno (a) del programa de Maestría en Sistemas Computacionales con número de control M09011274, manifiesta que es autor (a) del trabajo de tesis bajo la dirección de Dr. Ulises Juárez Martínez y cede los derechos del trabajo de tesis titulado "Desarrollo de software utilizando el lenguaje naturalístico Cal-4700" al TecNM/Instituto Tecnológico de Orizaba para su difusión y divulgación, con fines académicos y de investigación.

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

José Antonio Juárez Romero

Nombre y firma

Agradecimientos

Al tecnológico nacional de México por la oportunidad de ingresar a la maestría en sistemas computacionales.

Al Consejo Nacional de Ciencia y Tecnología, CONACyT, por el apoyo económico otorgado para la realización del presente proyecto.

A mi director de tesis, Dr. Ulises Juárez Martínez, por sus conocimientos invaluables brindados para llevar a cabo esta investigación y sobretodo su gran paciencia y apoyo.

A los miembros del jurado, M.I.S. Lizbeth Alejandra Hernández González, M.I.S. Lizbeth Alejandra Hernández González y M.I.S. Lizbeth Alejandra Hernández González por las valiosas contribuciones que hicieron al trabajo final y por el tiempo que dedicaron para revisarlo.

A mi familia, por todo el apoyo brindado.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

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 proyecto 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. Se muestra que es posible emplear un subconjunto controlado del idioma inglés como lenguaje de programación y 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.

Abstract

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

This problem is compounded by the poor documentation of the software, due, among other things, to the lack of discipline of the programmer or to the short delivery times, resulting in an airtight software product that makes maintenance and evolution tasks difficult.

As an alternative solution, naturalistic programming proposes the creation of programming languages based on incorporating natural language elements, which offer greater expressiveness and generate self-documented code. In this context, the general-purpose naturalistic programming language Cal-4700 is presented, which allows for writing executable code from a subset of sentences written in the English language.

This project presents a naturalistic implementation using Cal-4700 in a playful application. The purpose is to demonstrate its expressive capability and advanced features to be considered in the construction of applications.

This work shows that employing a controlled subset of the English language as a programming language allows achieving the objectives of offering greater expressiveness and self-documenting code, in addition to self-documenting code and demonstrating that Cal-4700 is robust enough to create small and medium-sized general-purpose applications.

Introducción

Es de consenso general que el proceso de desarrollo de software cuenta con una complejidad inherente. En primer lugar se considera el aspecto mental, en la que el ingeniero de software modela la solución realizando una abstracción del problema a resolver y a partir de ello escribe el código fuente que representa la forma en que se satisfacen todas las necesidades. Durante este proceso es habitual que las ideas originales se vean deformadas a causa de la necesidad de adaptarse a la forma en que funciona la tecnología y a la brecha de comunicación entre usuarios y desarrolladores donde los usuarios no encuentran la forma precisa de transmitir sus necesidades a los desarrolladores. Lo que lleva a que algunas veces el resultado se desvíe de los objetivos iniciales o que el producto final difiera del esperado.

Como alternativa de solución se propone hacer uso de la programación naturalística. La programación naturalística es una nueva forma de crear lenguajes de programación que usen los elementos del lenguaje natural. El paradigma naturalístico tiene la capacidad de analizar, modelar e implementar software, reduciendo las dificultades propias del lenguaje de programación y preservando las ideas originales con mayor fidelidad, ya que se utilizan los elementos del lenguaje natural para escribir las instrucciones del código fuente. Además, el uso de los elementos del lenguaje natural en el código fuente aumenta la expresividad y la claridad del mismo, haciendo incluso que el código fuente se auto-documente. De esta manera es posible que cualquier persona ajena al proceso de Ingeniería de Software sea capaz de leer y comprender las instrucciones escritas en el lenguaje naturalístico.

En el presente proyecto de tesis se propuso desarrollar una aplicación utilizando el

lenguaje naturalístico Cal-4700 y con ello demostrar los beneficios del paradigma. Adicionalmente, señalar el estilo de los lenguajes naturalísticos y aplicar los elementos de modelación adecuados al contexto del paradigma naturalístico.

Para ofrecer una mejor visión de esta investigación, el presente documento está constituido por cinco capítulos que se describen a continuación.

El capítulo 1 se enfoca en dar a conocer los conceptos relevantes para la comprensión del proyecto, el planteamiento del problema, el objetivo general, los objetivos específicos y justificación del proyecto.

En el capítulo 2 se describen los trabajos relacionados enfocados en el paradigma naturalístico y sus elementos de modelación. Se analizan los lenguajes de programación naturalísticos de propósito general reportados. Se presenta además la propuesta de solución y las herramientas a utilizar para alcanzar los objetivos de la tesis.

En el capítulo 3 se muestra el desarrollo de la metodología de tesis, se exploran las características y capacidades de Cal-4700 y el método de desarrollo naturalístico.

En el capítulo 4 se presentan los resultados obtenidos tomando como base un caso de estudio lúdico. Se presentan las ventajas y desventajas de la solución programada.

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

Capítulo 1

Antecedentes

Este capítulo se divide en tres partes. Primero se presentan las ideas más importantes relacionadas al presente trabajo de investigación. Después, el objetivo general y los objetivos específicos. Finalmente se presenta la justificación del proyecto.

1.1. Marco teórico

En esta sección se presentan los conceptos más relevantes relacionados con el tema de investigación.

1.1.1. Programación naturalística

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

En [2] se define la programación naturalística como el uso de los elementos del lenguaje natural en el proceso de desarrollo de software para reducir la ambigüedad y aumentar la expresividad, es decir, una mayor claridad al momento de expresar ideas. El uso de elementos lingüísticos como anáforas, temporalidad y abstracciones, por mencionar algunas, son un medio para definir e implementar un lenguaje de programación naturalístico

de propósito general. La programación naturalística apunta hacia la dirección del pensamiento humano, para expresar claramente las ideas sin necesidad de realizar otro tipo de abstracciones conceptuales de lo que se quiere comunicar.

La programación naturalística se propuso por primera vez en 2003 [3]. Al día de hoy se han realizado varios esfuerzos en el área, desde contar con un modelo conceptual de programación naturalística [2] hasta varios lenguajes de programación que admiten elementos del lenguaje natural [4] [5] que generan código directamente ejecutable.

1.1.2. Lenguaje natural

El término lenguaje es muy ambiguo. Se usa para describir la comunicación entre dos individuos, así como para describir un sistema de símbolos y signos usados en un contexto determinado. En [6] se afirma que el lenguaje está compuesto de dos entidades complementarias: el habla y la lengua. El habla se define como el uso de la lengua por un individuo en un acto individual. En cambio, la lengua trasciende lo individual hacia una propiedad social que adquiere una forma de institución social; es decir, la lengua es el conjunto de los sistemas lingüísticos que los integrantes de una sociedad poseen. En ese marco la lengua es un sistema de signos, convenciones y reglas, listos para usarse por las comunidades con fines comunicativos . La lengua, entonces, es un modelo lingüístico que determina el habla.

Deixis

El diccionario inglés de Cambridge define a deixis como: “la función o uso de palabras o expresiones cuyo significado depende de dónde, cuándo y quién las use” [1].

La deixis indica que una oración, frase o enunciado en lenguaje natural depende del contexto en que se usa. La deixis está presente en casi todos los lenguajes naturales y tiene la propiedad de expresar ideas con el mínimo número de palabras.

Contexto

El contexto se define en [5] como las circunstancias que ocurren alrededor de un evento determinado. A partir del contexto se puede interpretar o entender un evento. El contexto está compuesto por una serie de circunstancias (como el espacio y el tiempo) que facilitan el entendimiento de un mensaje.

Por otra parte, el contexto lingüístico se define como los factores relacionados con la etapa de generación de un enunciado y que inciden en su significado e interpretación. Esto implica que un enunciado depende de la sintaxis, de la gramática y del léxico, pero también del contexto.

Sintaxis

En lingüística, la sintaxis estudia los elementos de una lengua y sus combinaciones, establece la estructura que deben tener las oraciones. La sintaxis ofrece pautas para saber cómo unir y relacionar palabras, con el propósito de elaborar oraciones y definir conceptos de forma coherente [7]. En los lenguajes de programación no es diferente, en [8] se define la sintaxis como las reglas que deben cumplir los diferentes elementos de un lenguaje de programación para considerarse como sentencias válidas.

Semántica

Mientras la sintaxis especifica cómo deben disponerse los diferentes elementos del lenguaje para formar sentencias válidas, en [8] se define a la semántica como la encargada de comprobar el sentido correcto de las sentencias válidas. La semántica establece cómo se comportan los diferentes elementos del lenguaje y determina qué construcciones sintácticas son válidas y cuáles no.

Gramática

Se define la gramática [7] como el estudio de los componentes de la lengua y sus combinaciones. La gramática especifica un grupo de principios, reglas y preceptos que

guían el empleo de un lenguaje en particular. Una parte importante de la gramática es la sintaxis del idioma que consiste en estudiar cómo se combinan las palabras y las relaciones que existen entre ellas.

Ambigüedad

La ambigüedad [7] es una característica propia del lenguaje, se define como el significado de una frase o una declaración que no se expresa claramente y que se presta a múltiples interpretaciones que varían según el contexto específico. La ambigüedad es un concepto reconocido y resuelto fácilmente por las personas, pero que las computadoras no pueden manejar correctamente todavía debido a la naturaleza formal del cómputo.

Expresividad

En [9] se define la expresividad como la capacidad de lenguaje natural para describir las ideas con la claridad y precisión deseada. Su importancia radica en la facilidad que otorga a los lenguajes de programación para desarrollar ideas. En la actualidad, los lenguajes de programación carecen de la expresividad necesaria para describir abstracciones genéricas que abarquen todas las características de su dominio.

Llevar la expresividad de los lenguajes naturales a la programación haría que programar fuese como hablar con un colega acerca de una idea, lo que es uno de los objetivos de la programación naturalística.

Relaciones anafóricas

El diccionario de la lengua española define la anáfora [7] como el vínculo de identidad establecido entre un elemento de la gramática y uno o más términos que ya fueron nombrados en el texto precedente.

Para ampliar el término se considera el siguiente ejemplo *"El cartero trajo una carta para María. Ella no lo esperaba"*, donde el pronombre *"ella"* remite a la persona llamada María, por lo tanto, ya es conocida cuando se nombra en la secuencia subsiguiente el

anafórico *"ella"*.

1.1.3. Modelo semántico

La semántica se define como el significado lingüístico de las palabras y el significado resultante de su combinación en forma de enunciados [10]. El "significado" hace referencia al contenido semántico de las expresiones; el contenido se identifica con la referencia de una expresión, con uno o varios conceptos, o con una proposición de estructura conceptual.

El objetivo del modelo semántico es explicar la construcción de representaciones mentales similares, comunes a distintos objetos lingüísticos, usando una estructura lingüística determinada mediante un proceso de abstracción semántico.

1.1.4. Expresiones y enunciados

Una expresión en un lenguaje de programación se define en [8] como un bloque básico de construcción que tiene la capacidad de acceder a los datos del programa y devuelve un resultado. A partir de las expresiones se construyen bloques de código complejos llamados enunciados. Los enunciados, también llamados sentencias son el bloque de construcción en que se basan los lenguajes imperativos como los lenguajes orientados a objetos. Una expresión es una constante, una variable, una invocación a un subprograma o un operador. Las expresiones representan valores, es decir, su evaluación da como resultado un valor. Finalmente, un enunciado se define como una colección de expresiones dispuestas en secuencias que forman un subprograma o un programa entero.

1.1.5. Gramáticas libres de contexto

Una gramática libre de contexto G se define en [8] como una 4-tupla G (T, N, S, P). Donde T es un alfabeto del lenguaje, es decir, un conjunto de símbolos que forma las cadenas del lenguaje que se está definiendo. N es un conjunto de variables también llamados símbolos no terminales. S es una variable definida en el alfabeto que funciona como símbolo inicial. Por último, P es el conjunto finito de producciones de la gramática. El

lenguaje definido por una gramática libre de contexto se denota como L(G) y se denomina como lenguaje libre de contexto. Las gramáticas pueden utilizarse para decidir si una cadena pertenece al lenguaje y para comprobarlo se parte del símbolo inicial y se realizan sustituciones utilizando las producciones de la gramática.

Las gramáticas libres de contexto permiten describir la mayoría de los lenguajes de programación, la sintaxis de estos lenguajes está definida mediante gramáticas libres de contexto.

1.1.6. Lenguaje formal

Un lenguaje formal se define en [11] como un lenguaje artificial, que se compone de símbolos primitivos y que sigue un conjunto de reglas estrictas para la formación de sus enunciados. Al conjunto de símbolos primitivos se le llama alfabeto o vocabulario, y al conjunto de reglas se le llama gramática formal o sintaxis. Una cadena de símbolos formada a partir la gramática formal se le llama palabra.

El término "formal" hace referencia a que es abstraído de cualquier tipo de contenido, es decir, un lenguaje sin significados. La palabra formal también hace referencia al procedimiento de seguir de manera mecánica las reglas o instrucciones de formación o transformación de enunciados. Un lenguaje formal apunta a un lenguaje de símbolos siempre que estos no tengan ningún tipo de contenido semántico. Los lenguajes formales están relacionados con las matemáticas y la informática.

1.1.7. Programación orientada a aspectos

La programación orientada a aspectos (AOP por sus siglas en inglés; *Aspect-Oriented Programming*) se propone en [12] como una técnica para mejorar la separación de *asuntos* de un sistema para trabajarlos por separado y luego usando los mecanismos del entorno de la AOP componerlos juntos en una pieza de software coherente. En este contexto un *asunto* es algo de interés para las personas involucradas en un proyecto de software y abarca cualquier fragmento de código, artefacto o requerimiento relacionado a un objetivo,

característica o tipo de funcionalidad.

La AOP considera que un problema dado involucra diferentes tipos de *asuntos*, que deben identificarse y separarse para reducir la complejidad y lograr factores de calidad de ingeniería requeridos. Este concepto conocido como el *Principio de separación de asuntos*, fundamento clave en la Ingeniería de Software, se atiende ya en métodos orientados a objetos, donde se separan los asuntos en clases y objetos, de igual forma, los métodos estructurados que los representan como procedimientos. Sin embargo, la AOP extiende el término de *asunto* con propiedades transversales al software como lo son la sincronización, la persistencia y la gestión de memoria.

1.2. Planteamiento del problema

Las técnicas actuales de programación y de resolución de problemas, consideran que factores como la parte mental, la comunicación humana, los lenguajes de programación y en general los aspectos técnicos, son los más influyentes en la tradicional brecha que existe entre el dominio del problema y el dominio de la solución [3]. Para atender esta problemática, diversos autores consideran que el estado actual de la programación ha alcanzado un nivel suficiente, en donde un cambio de paradigma permitiría desarrollar los lenguajes de programación aún más al otorgarles claridad y expresividad [4] [9].

La programación naturalística es una nueva manera de implementar software que utiliza elementos del lenguaje natural y organiza los programas mediante el concepto de deixis. Al igual que los lenguajes de programación convencionales, utiliza abstracciones, pero éstas se basan en elementos del lenguaje natural. En un sentido más amplio, diversas investigaciones han contribuido con lenguajes de programación naturalística de propósito general [4] [13], lenguajes de programación complementados con lenguaje natural (KlarDeutsch, AppleScript, NaturalJava) [4], un modelo conceptual naturalístico [2], tipos naturalísticos (incluyendo polimorfismo) y tipos naturalísticos [14].

Para impulsar el desarrollo de aplicaciones bajo el paradigma naturalístico, es necesario ponerlo en práctica, considerando la implementación de al menos una aplicación,

utilizando un lenguaje naturalístico. Actualmente se reportan tres esfuerzos relevantes: una plataforma de programación naturalística llamada Pegasus [4], un prototipo naturalístico de propósito general llamado SN (Sicut Naturali) [13], y una plataforma para programar en inglés plano llamada Cal-4700 [20]. Éste último incluye un ambiente de desarrollo accesible a cualquier programador, además de que cuenta con características avanzadas que permiten considerarlo en la construcción de aplicaciones.

1.3. Objetivo general y específicos

En las siguientes dos sub-secciones se detalla el objetivo general para el trabajo de tesis, así como sus objetivos específicos.

1.3.1. Objetivo general

Desarrollar una aplicación utilizando el lenguaje Cal-4700 y elementos de modelación adecuados, de tal manera que se demuestren los beneficios del paradigma naturalístico.

1.3.2. Objetivos específicos

- Revisar los fundamentos de la programación naturalística para comprender el estilo de los lenguajes naturalísticos.
- Utilizar el lenguaje Cal-4700 para implementar una solución naturalística.
- Identificar y aplicar los elementos adecuados de modelación útiles en el contexto del paradigma naturalístico.
- Identificar y desarrollar una aplicación que muestre las ventajas y limitaciones de trabajar con Cal-4700.

1.4. Justificación

La comunicación eficiente entre los clientes y analistas requiere de habilidades y esfuerzo para modelar adecuadamente el software, así como plasmar de forma correcta las decisiones de diseño. Una limitación específica al respecto es la frecuente deformación de las ideas del cliente para dar forma a los diversos artefactos de modelación. Adicionalmente, la posible ausencia de documentación que los acompañe y la imposición de reglas del lenguaje de programación seleccionado para la solución, complican que las ideas originales del cliente se mantengan.

El paradigma naturalístico ofrece la posibilidad de analizar, modelar e implementar software, preservando las ideas de los clientes con un menor grado de deformación, mejorando la documentación, ya que el código fuente se autodocumenta. Considerando las herramientas existentes, la programación naturalística con Cal-4700 apertura la posibilidad de programar también en español plano, hecho significativo para el desarrollo de software dentro de un mercado de habla hispana. Lo anterior potencializa la consideración del paradigma como una opción seria dentro del desarrollo de software.

Capítulo 2

Estado de la práctica

El uso del lenguaje natural para el desarrollo de software es un tema que se ha propuesto desde hace décadas, pero no ha sido sino hasta los últimos años que se han visto realizados algunos de los objetivos mediante el paradigma de programación naturalístico. Durante este tiempo, diversos autores han presentado estudios que intentan utilizar un subconjunto del idioma inglés como un lenguaje de programación para resolver problemas de software. A continuación, se recopila, analiza y se presenta una breve descripción de estos trabajos de investigación como parte del estado del arte del presente estudio.

2.1. Trabajos relacionados

Los sistemas de software son naturalmente complejos. Por una parte, se encuentra la complejidad del dominio, y por otro, y más agravado las tecnologías de desarrollo de software existentes. Uno de los problemas es que los lenguajes de programación son inadecuados cuando se trata de transmitir información relevante a las personas sobre los sistemas de software. En el año 2003 [3] se analizó el papel de los lenguajes naturales en el diseño de lenguajes de programación como una forma de aportar expresividad y restar complejidad. En primer lugar, la Programación Orientada a Aspectos (AOP por sus siglas en inglés; *Aspect-Oriented Programming*) que usa algunos mecanismos de referencia que ocurren normalmente en los lenguajes naturales, hizo notar que es posible manifestar algunas ideas

mediante código que no se expresan fácilmente en los lenguajes de programación tradicionales. La AOP admite un conjunto enriquecido de referencias estructurales y temporales que se asemejan a los lenguajes naturales. Estas lecciones aprendidas con la AOP llevan a considerar que es valioso volver a examinar la importancia de los lenguajes naturales en la programación el día de hoy.

La principal bondad que busca el uso de los lenguajes naturales en la programación es el poder expresivo con que están respaldados, en gran medida, por sus sofisticados mecanismos de referencia. Como son, las anáforas de pronombres: *esto, aquello, eso, estos*, por mencionar algunos. Los referentes temporales como: *último, primero, después de, antes de*, entre otros. Los referentes de grupo como: *todos, cualquiera*. Los referentes reflexivos: *iteración, bucle, operación*, entre otros.

La importancia de los lenguajes naturales en la programación es la forma en que permiten organizar ideas de forma natural. Tanto así que los lenguajes naturales son, de hecho, el soporte para todos los demás lenguajes formales, como los formalismos matemáticos o las instrucciones de microprocesador. Todo lo que se logra expresar en lenguajes formales se puede expresar en cualquier idioma del lenguaje natural como inglés o español.

En el año 2006 [4] se realizaron los primeros esfuerzos en la Universidad Tecnológica de Darmstadt de desarrollar un lenguaje de programación naturalístico mediante la construcción de un prototipo científico llamado Pegasus. El desarrollo del prototipo surge de la idea de los autores de que las técnicas de programación contemporáneas necesitan un cambio fundamental de paradigma para poder desarrollarse más, debido a que existe una brecha entre las técnicas de programación actuales y cómo se espera que se comporten las computadoras.

Los autores consideran que los principales problemas de los lenguajes de programación actuales son:

- **El problema mental.** El programador se ve forzado a ajustar sus ideas a las condiciones de un lenguaje de programación específico. Por ejemplo, para un lenguaje orientado a objetos significaría estructurar las ideas del programa a la forma de clases,

métodos y atributos.

- **El problema del lenguaje de programación.** El mismo programa debe implementarse una y otra vez para estar disponible en los sistemas de computadora contemporáneos y los nuevos lenguajes de programación.

- **El problema del lenguaje natural.** Es ineficaz escribir, comentar y documentar software en inglés, el idioma más comúnmente utilizado, para los hablantes no nativos del idioma.

- **El problema técnico.** Los desarrolladores dedican la mayor parte del tiempo en depurar el programa, además de lidiar con problemas menores como elegir el conjunto de caracteres correcto, hacer conversiones numéricas, administrar el acceso a la base de datos, entre otros, en lugar de enfrentar la tarea de describir y mejorar la idea del programa.

Los autores identifican que el uso de los elementos del lenguaje natural en la programación genera los siguientes beneficios:

- Al escribir los programas en lenguaje natural no es necesaria una transformación de ideas en otras estructuras de lenguajes.

- La programación en lenguaje natural libera a los desarrolladores de la carga de aprender una y otra vez nuevos lenguajes de programación.

- Una vez que la idea se expresó en un lenguaje natural será válida durante mucho tiempo.

- La programación se volverá más fácil, escribir un programa de computadora será como explicar los pensamientos a un colega.

Con lo anterior en consideración se desarrolla el prototipo Pegasus. La arquitectura de Pegasus tiene tres características básicas: leer el lenguaje natural, generar una salida de

código de programación (Java) y expresar los resultados en otros idiomas del lenguaje natural.

Para la lectura del lenguaje natural Pegasus analiza la estructura gramatical de cada oración por separado; distingue entre la entidad, acción y propiedad mediante el uso de un diccionario. Posteriormente Pegasus comprende el sentido de la oración mediante una biblioteca de acciones, en donde se almacenan las respuestas para cada una de las ideas concretas interpretadas. Finalmente, Pegasus agrupa los diferentes significados y genera una sola idea. Una vez que Pegasus ha resuelto toda la idea, ejecuta los resultados a través de una biblioteca de significados y genera la salida en un lenguaje de programación como Java.

Al finalizar la ejecución Pegasus es capaz de expresar las ideas leídas en todos los lenguajes naturales compatibles. Al momento de la publicación del artículo, Pegasus soporta los lenguajes naturales inglés y alemán y se espera agregar más a futuro. Los autores están seguros que la programación en lenguaje natural será una tendencia futura en el desarrollo de técnicas de programación, enfocadas hacia el pensamiento humano.

En [14] los autores conceptualizan la idea de tipos naturalísticos y posibles aplicaciones para su uso en programación. El lenguaje natural tiene el mismo objetivo que los lenguajes de programación; comunicar información, incluido el conocimiento y comportamiento. El uso de los lenguajes naturales en la programación prevé sacar provecho de las características esenciales de los primeros aplicando sus propiedades básicas, listadas a continuación:

- **Evitar redundancia.** Los lenguajes naturales reducen la cantidad de datos comunicados por unidad al evitar palabras redundantes; se tiende a expresar información con la menor cantidad de palabras posibles. De esta manera, se omiten detalles innecesarios y se evita tener en la memoria conceptos temporales.

- **Localidad.** El lenguaje natural se estructura en declaraciones de tamaño limitado, oraciones y párrafos. Por lo general, existen referencias solo a elementos de la estructura actual o la anterior sin abordar elementos más distantes, a diferencia de

los lenguajes de programación que pueden hacer referencia a variables definidas al comienzo de un fragmento largo de código.

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

"La ventaja de un lenguaje de programación naturalístico se trata del potencial para expresar y organizar ideas de una manera natural". [14]

Apoyándose de los conceptos previos se desarrolla un sistema de tipos naturalísticos con los conceptos de referenciación natural, generalización y descripción de instancia, con las siguientes características:

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

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

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

El tipo naturalístico es un conjunto de cualidades, representado por predicados lógicos que todas las instancias deben completar para pertenecer a ese tipo. Su estructura es:

[una cantidad] {una propiedad} un concepto {una condición}
[una] {hermosa} casa moderna {hecha de piedra}

- **Una cantidad.** Es un número escrito o representado. "1,2,3"; "uno, dos, tres".

- **Una propiedad.** Es una cualidad atribuida, un adjetivo que representa una propiedad para los lenguajes naturales.
- **Un concepto.** Es un "sustantivo" en los lenguajes naturales, cosas como "una casa", "un tigre".
- **Una condición.** La condición se considera como una cualidad de tipo que permite codificar predicados arbitrarios, condiciones.
- **Subtipo.** Polimorfismo de subtipo, en el sistema de tipos naturalístico se puede definir que un tipo A es subtipo de B sí y solo sí A es compatible con B. "Un coche rojo es un vehículo rojo."

A través de esta declaración de tipos naturales se demuestra que varias propiedades de los lenguajes naturales pueden ser interesantes para los lenguajes de programación. Estudiar el lenguaje natural no solo es importante para mejorar los lenguajes de programación, sino también para mejorar el estilo de programación dentro de los lenguajes existentes.

Posteriormente en [15] se presentó un nuevo enfoque hacia la programación naturalística. El enfoque adoptado toma requisitos en forma de casos de uso multilingües a partir de los cuales un generador de código produce el código de destino (*target*). La riqueza de este nuevo enfoque consiste en la implementación de código XML que representa la información de los escenarios de casos de uso.

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

sentador. El enfoque desarrollado, a diferencia de otros es muy simple porque no requiere ningún estudio previo del idioma para utilizarse gracias al modelo semántico. Además, el enfoque tiene el potencial de extraer la información adecuada de los escenarios de caso de uso ambiguos y generar entradas de ReDSeeDs bastante completas y correctas para en consecuencia obtener un buen código Java.

Finalmente, se evaluó cuantitativamente la precisión del enfoque y se determinó que existe una precisión media del 87.43 % al 89 % en términos de modelos. Además, se obtuvo una precisión de entre el 64.4 % y 93.43 % en términos de eficiencia de código Java.

El enfoque desarrollado es útil para el proceso inicial de desarrollo de cualquier proyecto recopilando información relacionada con el dominio de la aplicación. Al emplear esta herramienta los desarrolladores podrían ahorrar tiempo al tener acceso a elementos de código bastante completos y correctos a partir de los escenarios de caso de uso involucrados en la fase del análisis de la aplicación. Sin embargo, el enfoque requiere la construcción de una API (del inglés API: *Application Programming Interface*) que comunique los modelos generados por el enfoque y los del generador de código ReDseeDS. ReDseeDS usa formatos de modelo específicos no públicos, lo que ha limitado el desarrollo del potencial del enfoque.

Con las bondades del lenguaje natural ya conocidas y con herramientas desarrolladas hacia el mismo esfuerzo se plantea la pregunta: ¿Por qué la programación de todos los días no se realiza utilizando lenguajes naturales? En [9] se desarrolló una serie de planteamientos que exploran esta cuestión y encaminan a definir una propuesta conceptual. En [3] y [4] los investigadores han intentado utilizar lenguajes naturales para construir herramientas de desarrollo de software como las encontradas en [14] y [15]. Sin embargo, los investigadores se han enfrentado al hecho de que las computadoras no pueden lidiar con la ambigüedad de los lenguajes naturales que los lenguajes de programación han resuelto con especificaciones formales. En el razonamiento humano es el proceso cognitivo que ayuda a los programadores a resolver esta ambigüedad, ayuda a las personas a comprender y distinguir no solo los verbos y sustantivos, sino también el significado en un contexto en particular. Como solución, se propone obtener un lenguaje de programación de tipo

natural pero también restringido y formalizado para ser procesado por las computadoras.

En [9] se analizaron varias herramientas que utilizan una gramática controlada del idioma inglés con un medio para traducir, esta versión controlada, a un lenguaje de programación o a generadores de código de programación que controlan la ambigüedad. Sin embargo, los autores observaron que muchas de las tecnologías dependen de diccionarios de datos o bibliotecas pre-compiladas, lo cual reduce su uso a dominios particulares y aleja el objetivo de definir un modelo naturalístico de propósito general. También se encontraron lenguajes con alto nivel de expresividad, con un proceso cognitivo similar al humano para lidiar con la ambigüedad, no obstante, estos están limitados a dominios particulares y utilizan un lenguaje natural controlado. Con las manifestaciones anteriores se consideró que el objetivo del paradigma naturalístico va más allá de programar en lenguaje natural, un modelo naturalístico debe ser capaz de abordar todos los dominios sin problemas de ambigüedad.

En el año 2019, en el Tecnológico Nacional de México Campus Orizaba, se diseñó un lenguaje de programación naturalístico de propósito general basado en el idioma inglés llamado SN (*Sicut Naturali*) [5]. El lenguaje permite trabajar con referencias indirectas, además permite complementar una entidad a partir de la composición de un sustantivo y varios adjetivos para darle un comportamiento especializado. SN soporta ciclos y condiciones y permite definir el contexto de una oración a partir de la asociación de una oración con otra de forma indirecta. La sintaxis de SN permite trabajar con diversas abstracciones que lo asemejan a un lenguaje natural.

SN es un lenguaje emergente que permite expresar código con una sintaxis simple y similar a la del idioma inglés, que incluye elementos naturales y permite tener un nivel alto de expresividad. SN cuenta con todas las propiedades ya documentadas de los lenguajes de programación naturalístico como lo es la ventaja de generar un código autodocumentado dadas las construcciones similares a las de un lenguaje natural. Las pruebas realizadas en [5] dieron como resultado pequeños programas con instrucciones similares a oraciones, donde los resultados mejoraron su expresividad según se avanzaron las pruebas.

SN es un lenguaje nuevo, con necesidad de refinarse, ya que cuenta con algunas li-

mitantes como lo son la ejecución lenta: debido a la gran cantidad de evaluaciones que realiza; la dependencia de Scala y AspectJ, además de la falta de una API más robusta. Los autores se proponen seguir con pruebas que permitan explorar otras capacidades del lenguaje. Además, se plantea realizar una evaluación subjetiva del lenguaje por medio de la escala Likert con personas sin conocimiento de SN, encontrar deficiencias en su sintaxis y semántica y capacitar a usuarios que usen SN para retroalimentar el proceso de evolución del lenguaje.

En [2] se presentó un modelo conceptual que describe los elementos necesarios para diseñar lenguajes de programación naturalísticos de propósito general, que integra abstracción, elementos temporales y referencias indirectas en su gramática. Dicho modelo se implementó en el lenguaje de programación naturalístico SN.

El modelo presentado consta de cuatro elementos principales:

- **El sustantivo.** Es la palabra que se usa para identificar "una cosa" o un conjunto de "cosas".
- **El adjetivo.** Se consideró como una abstracción que se usa para proporcionar las propiedades del sustantivo.
- **El verbo.** El verbo representa acciones del sustantivo; es equivalente a las funciones en otros paradigmas, como en programación funcional o los métodos en la programación orientada a objetos.
- **Escritura basada en propiedades.** Permite usar propiedades para identificar una entidad particular.

Se consideró que estos elementos brindan la información más relevante en una oración escrita en inglés. Los elementos seleccionados permitirán además el diseño de otros lenguajes de programación naturalísticos de propósito general sin requerir diccionarios de datos u otras técnicas. Se observó que SN a partir del modelo implementado contribuye a reducir la brecha entre los dominios del problema y la solución. SN genera como resultado un código auto-documentado.

Como resultado, SN requiere un proceso de aprendizaje centrado no sólo en la gramática sino también en adaptar el proceso cognitivo, porque, aunque permite la programación de abstracciones como las encontradas en Java, cuando éstas se implementan el lenguaje pierde la naturalidad, lo cual va en contra de los objetivos de la programación naturalística. SN carece todavía de optimización suficiente para su uso en entornos reales, sin embargo, su trabajo está encaminado a lograr ese potencial; SN permite desacoplar las abstracciones y con ello aumentar la modularidad. Además, SN permite el uso de identificadores para el manejo de gran número de instancias. En el futuro SN se refinará para que el modelo conceptual sea más completo y variado, se brindará soporte para que los programadores puedan definir sus propias gramáticas. Se analizarán los iteradores y condicionales para comprobar que tengan el nivel adecuado de naturalidad. SN se modificará para generar código sin lenguajes intermedios, ya que actualmente utiliza Scala y AspectJ para el análisis y compilación.

En [13] se continúa el trabajo con el lenguaje prototipo SN. Los investigadores sabían que el principal obstáculo del lenguaje natural es que es ambiguo y depende del contexto. Hasta ahora las computadoras procesan sin problema lenguajes de programación porque estos se basan en formalismos que no presentan ambigüedad. Para controlar la ambigüedad del lenguaje natural se ha propuesto usar un modelo formalizado y restringido del lenguaje natural. En esta nueva revisión del lenguaje SN los investigadores agregan además del uso de sustantivos, verbos, adjetivos y referencias indirectas reportados en [2], el uso de plurales para colecciones de datos y, asimismo, eventos denominados circunstancias que describen el contexto de una oración. En SN las instrucciones se definen de tres formas: sujeto-verbo-objeto, verbo-objeto y verbo-objeto-preposición-objeto. La implementación del lenguaje SN dio como resultado un lenguaje verboso que permite expresar instrucciones con un nivel de expresividad más cercano al idioma inglés gracias a la gran cantidad de reglas que utilizan para su creación. El lenguaje SN permite reducir la ambigüedad de los lenguajes naturales gracias al modelo. Como trabajo a futuro los investigadores propusieron realizar casos de pruebas más complejos que permitan dar validez de forma más completa al modelo. Además, los investigadores se propusieron trabajar con la integración

de mecanismos que permitan el uso de gramáticas embebidas dada la complejidad para describir elementos de dominios particulares.

Para el año 2020 se realiza una revisión de los únicos tres lenguajes naturalísticos de propósito general que pueden generar código ejecutable [16]: Pegasus, Cal-4700 y SN. Estos, a consideración de los autores, son los lenguajes de programación naturalísticos más representativos en la actualidad que trazan una línea de tendencia en la evolución del paradigma de programación naturalístico de acuerdo a una comparación de sus características y ejemplos realizados en los tres lenguajes. En general, los tres lenguajes crean declaraciones que utilizan elementos comunes del lenguaje natural como los sustantivos. Los tres lenguajes pueden identificar frases, hacer referencias anafóricas, ejecutar instrucciones de software (como responder a eventos y definir declaraciones condicionales) y usar iteradores naturalísticos.

Pegasus no permite definir tipos, sustantivos y adjetivos de manera explícita, de acuerdo a lo reportado en la literatura, se infiere que están definidos en un diccionario, sin embargo, estos son factibles de usarse en el lenguaje, de acuerdo al sitio web oficial que muestra su forma de uso. SN y Cal-4700 pueden usar sustantivos ("cosas") usando declaraciones en singular y plural y permiten al programador administrar el conjunto de "cosas". Cal-4700 a diferencia de SN y Pegasus, no usa estructuras gramaticales estrictas, funciona mediante instrucciones imperativas guiadas de una estructura flexible que no realiza validaciones gramaticales sino de similitud. Para su ejecución SN no necesita archivos externos, simplemente se instala como cualquier otro lenguaje de programación, Pegasus necesita una base de datos con significados y Cal-4700 cuenta con su propio entorno que incluye un archivo de compilación y una biblioteca de acciones. Pegasus al usar un generador de código permite a los usuarios elegir el lenguaje de programación resultante, mientras que SN y Cal-4700 compilan directamente a código ejecutable. En Cal-4700 resalta la ventaja de que consigue incrementar el dominio de su lenguaje al permitirle a los programadores agregar definiciones no existentes a su biblioteca, además, es posible nombrar de diferentes maneras a las definiciones existentes para hacer que el lenguaje sea más adecuado al estilo de sintaxis del programador. En la práctica, SN y Pegasus no se encuentran disponibles al

público ya que están en fase de desarrollo todavía. Cal-4700 cuenta con una versión estable disponible al público a través de su sitio web [17]. Se concluyó que Pegasus, Cal-4700 y SN ofrecen más de lo necesario para denominarlos como lenguajes de uso general, siguen la filosofía de programación naturalística al integrar elementos del lenguaje natural, ser reflexivos y generar programas expresivos y ejecutables. Después de identificar los logros de cada uno de estos lenguajes se concluye que se necesitan utilizar de manera intensiva para mejorar el estado del paradigma de programación naturalístico como un paradigma de desarrollo de software serio.

2.2. Análisis comparativo

La Tabla 2.1 presenta un análisis comparativo de los trabajos relacionados descritos anteriormente, donde se observan las diferencias y similitudes entre ellos. Cuenta con las columnas descripción, problema, contribución, resultado y estado final.

Tabla 2.1: Análisis comparativo de los trabajos relacionados.

Artículo	Problema	Contribución	Tecnologías	Resultado
Beyond AOP: toward naturalistic programming — Lopes et al. [3]	Los lenguajes de programación son complejos y carecen de expresividad	Generar un debate sobre el papel de los lenguajes naturales en el diseño de lenguajes de programación.	ApectJ, D, Demeter para AOP.	Los autores invitan a la creación de un lenguaje de programación de lenguaje natural.
Pegasus: first steps toward a naturalistic programming language — Knöll y Mezini [4]	Las técnicas de programación contemporáneas necesitan un cambio fundamental de paradigma para poder desarrollarse más.	Se desarrolló un prototipo científico de un lenguaje naturalístico llamado Pegasus.	Java	Los autores prueban que la programación en lenguaje natural es posible y esperan que sea tendencia en el futuro.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo – continuación

Artículo	Problema	Contribución	Tecnologías	Resultado
Naturalistic types — Knöll et al. [14]	El uso de los lenguajes naturales en la programación prevé sacar provecho de las características esenciales de los primeros aplicando sus propiedades de referencias naturales, generalización y descripción de instancias.	Se conceptualiza la idea de tipos de datos de lenguaje natural.	No se menciona.	Se demuestra que las propiedades características de los lenguajes naturales son importantes para los lenguajes de programación.
Towards naturalistic programming: mapping language-independent requirements to constrained language specifications — Mefteh et al. [15]	Se presentó un nuevo enfoque hacia la programación naturalística utilizando requisitos independientes del lenguaje.	Se desarrolló un enfoque que mediante la escritura de requisitos independiente del idioma, genera automáticamente el código destino.	El generador de código ReDSeeDS y Java.	La herramienta desarrollada podría usarse por desarrolladores para ahorrar tiempo al tener acceso a elementos de código completos y correctos.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo – continuación

Artículo	Problema	Contribución	Tecnologías	Resultado
A survey of naturalistic programming technologies — Pulido y Juárez [9]	Se trata de dar respuesta a la pregunta ¿Por qué no logran establecerse las tecnologías de programación naturalísticas en la Ingeniería de Software?	Se prueban y analizan diferentes herramientas desarrolladas por otros investigadores en el campo de la programación naturalística.	ABAP/4, COBOL, CPL/CPL-Lite, FLOW-MATIC, HyperTalk, Informix-4GL, LiveCode, MOOSE Crossing, Nomad Software, OpernEdge ABL, Pegasus, SAS, Spoken C, Spoken Java, SQL y Visual FoxPro	Se concluyó que el paradigma naturalístico aún no estaba preparado para abordar todos los dominios sin problemas de ambigüedad.
Naturalistic Programming: Model and Implementation — Pulido y Juárez [2]	Se requiere un lenguaje de programación naturalística de propósito general; ya que las investigaciones previas abordan dominios particulares.	Se presentó un modelo conceptual para diseñar lenguajes de programación de propósito general, además, el modelo se implementó con el lenguaje de programación SN.	Scala y AspectJ.	El lenguaje carece de optimización para su uso en entornos reales, pero se espera pueda usarse en el futuro para la programación de propósito general.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo – continuación

Artículo	Problema	Contribución	Tecnologías	Resultado
Perspectives for software development using the naturalistic language SN — Alducin et al. [5]	La brecha entre el dominio del problema y la solución es amplia y con bastas complejidades. Se necesitan crear lenguajes de programación más expresivos.	Se pone a prueba el lenguaje de programación naturalístico SN, se corrobora que cuenta con elementos del lenguaje natural que le permiten tener un alto nivel de expresividad.	Scala y AspectJ.	El lenguaje necesita refinarse. Se proponen pruebas que aumenten las capacidades del lenguaje. Se planea evaluar de manera subjetiva y retroalimentar el proceso de evolución de SN.
Naturalistic Programming: Model and Implementation — Pulido y Juárez [13]	La ambigüedad del lenguaje natural sigue siendo una limitante para el desarrollo de un lenguaje de programación naturalístico de uso general.	Se amplía el modelo presentado en [2] agregando plurales y circunstancias.	AspectJ	Se logra un lenguaje de programación naturalístico verboso con un nivel de expresividad cercano al inglés con mecanismos que permiten reducir la ambigüedad. Se pretende refinar el lenguaje.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo – continuación

Artículo	Problema	Contribución	Tecnologías	Resultado
Evolution of Naturalistic Programming: A Need — Hernández et al. [16]	Considerando que el pensamiento popular es que se puede utilizar el lenguaje natural como herramienta de programación, se revisa el estado del arte de la programación naturalística.	Se revisan los lenguajes de programación naturalísticos: Pegasus, Cal-4700 y SN. Se comparan y discuten sus propiedades.	Los lenguajes de programación Pegasus, Cal-4700 y SN	Es necesario implementar de manera intensiva el uso de estos lenguajes para mejorar el estado del paradigma de programación naturalística.

Continúa en la siguiente página

Se concluye, después de analizar los trabajos de investigación previos, que el paradigma naturalístico está creciendo y mejorando continuamente según pasa el tiempo. Los estudios realizados por estos investigadores contribuyen a que la idea de tener un lenguaje de programación que use el lenguaje natural esté cada vez más cerca de verse realizada. Con esto se confirma que el trabajo realizado se encuentra todavía inconcluso y es necesario del trabajo de la comunidad de desarrolladores de software entusiastas para completarlo. Esta investigación contribuye a dicha idea al generar discusión sobre el papel de los lenguajes naturales en el diseño de los lenguajes de programación.

2.3. Propuesta de solución

A continuación, se presenta la propuesta de solución, el conjunto de las tecnologías seleccionadas que permitió resolver la investigación y la metodología de desarrollo que permitió alcanzar los objetivos de la investigación.

2.3.1. Planteamiento de la solución

La programación naturalística busca sacar provecho de los lenguajes naturales en el proceso de desarrollo de software. Implementar los elementos del lenguaje natural en el código fuente logra otorgarle más claridad y expresividad a éste.

La solución que se propone consiste en utilizar el lenguaje de programación naturalístico Cal-4700, para demostrar los beneficios del paradigma naturalístico aprovechando las características con que cuenta y que lo distinguen de las demás alternativas. Cal-4700 dispone de documentación detallada, disponibilidad de descarga inmediata, fácil instalación y un entorno de desarrollo completo que incluye un editor de textos, un manejador de archivos y un compilador. Estas ventajas hacen conveniente y ventajoso usar Cal-4700 para escribir software naturalístico. Para el modelado de software se utilizó la herramienta EdrawMax [18] ya que permite realizar diagramas de diseño estructurado y cuenta con un conjunto de notaciones personalizable.

2.3.2. Herramientas tecnológicas utilizadas

Para el desarrollo del presente proyecto de investigación se seleccionaron las siguientes herramientas:

- El lenguaje de programación naturalístico con entorno de desarrollo integrado de Cal-4700 [17].
- La herramienta de modelado *EdrawMax* [18] para los diagramas del caso de estudio.
- La herramienta *Balsamiq Wireframes* [19] para los prototipos de interfaz de usuario del caso de estudio.

2.3.3. Factibilidad de implementación de la solución propuesta

El desarrollo del presente proyecto de tesis se considera fiable de acuerdo a las siguientes cuestiones:

- Las ventajas de Cal-4700 son extensas. Su documentación detallada, la biblioteca de implementaciones y vocabulario, su disponibilidad de descarga y fácil instalación hacen factible la implementación del software naturalístico.

- Se cuenta con experiencia previa del manejo de la herramienta de modelado *Edraw-Max* así como la herramienta de maquetación de interfaz *Balsamiq Wireframes*.

Capítulo 3

Aplicación de la metodología

En este capítulo se presenta de forma detallada el desarrollo de la metodología de la tesis, y el desarrollo del caso de estudio seleccionado que permite verificar las características del lenguaje y su aplicación en el caso de estudio.

3.1. Capacidades de Cal-4700

Cal-4700 es un entorno de desarrollo y lenguaje de programación en la forma de un programa de *Windows*, incluye una interfaz gráfica dedicada, un administrador de archivos, un editor de texto, un editor de documentos, y un compilador/analizador que genera código nativo compatible con el sistema operativo Microsoft Windows y la arquitectura Intel. Todo el proyecto se encuentra escrito en *Plain English* (inglés simple) con un código fuente de alrededor de 25 mil frases. *Plain English* es el estilo y modo de escribir las instrucciones en Cal-4700.

3.1.1. Componentes de Cal-4700

Cal-4700 se compone de los siguientes elementos:

- *The desktop* (el escritorio). Es una interfaz de usuario ordenada con menús y pestañas.

- *The finder* (el administrador de archivos). Otorga el acceso directo al sistema de archivos.
- *The editor* (el editor). Una herramienta simple que permite la manipulación de archivos de texto.
- *The writer* (el escritor). Comprende un procesador y editor de textos que permite escribir un documento mostrando directamente el resultado final.
- *The compiler* (el compilador). Un traductor de *Plain English* a lenguaje máquina.
- *The noodle* (el espagueti). Una amplia biblioteca que contiene definiciones de tipos, encabezados de rutinas, además de la gramática y el vocabulario del lenguaje.

The desktop

El escritorio, es la interfaz de usuario de Cal-4700. En la Figura 3.1 se observa que el escritorio cuenta con menús alfabéticos en la parte superior, así como la barra de estado en la parte superior a la derecha. El área de trabajo se encuentra en medio y cuenta con pestañas en la zona inferior para elegir entre una u otra área de trabajo. Es destacable manifestar que el escritorio no cuenta con barras de desplazamiento, para este fin se utiliza el ratón, y los atajos de teclas con CTRL y ALT.

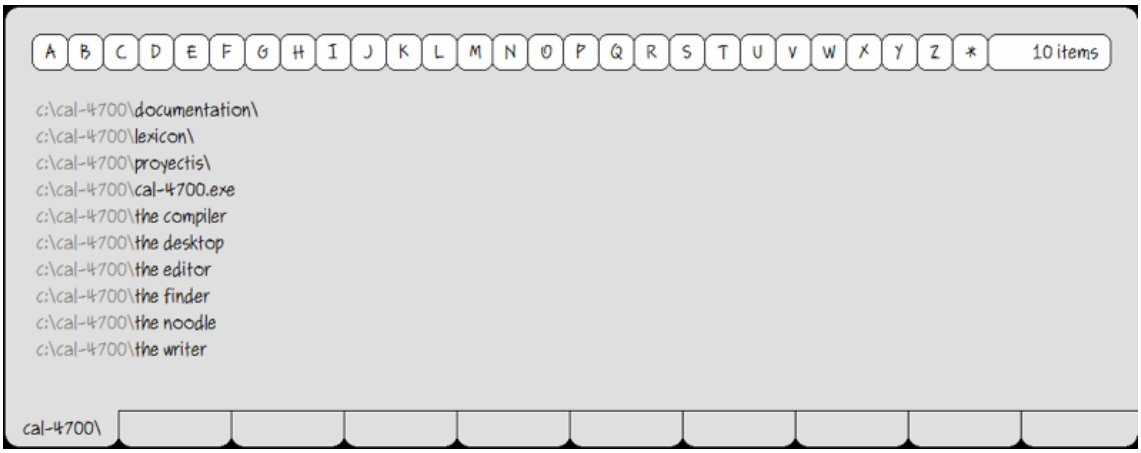


Figura 3.1: Interfaz de Cal-4700.

The finder

El administrador de archivos muestra el sistema de archivos en forma de texto, con directorios y lo que hay en ellos. Al abrir el escritorio se inicia al mismo tiempo el administrador de archivos, inicialmente a nivel de raíz del disco duro. En la Figura 3.1 se observa el administrador de archivos con el directorio principal de Cal-4700 y sus componentes en él. Para interactuar con el administrador de archivos se hace uso del cursor y de los menús alfabéticos, también es posible usar los atajos del teclado. Ejemplificando lo anterior, al crear un documento nuevo existen dos opciones, hacer uso del menú alfabético N, y enseguida seleccionar la opción *"new document"* o usar atajo de teclado Alt + N.

The editor

El editor permite visualizar y manipular archivos de texto en el área de trabajo haciendo uso del teclado y el ratón de manera habitual. En la Figura 3.2 se observa el editor en el área trabajo y en él una rutina del *noodle* que funciona para crear un texto.

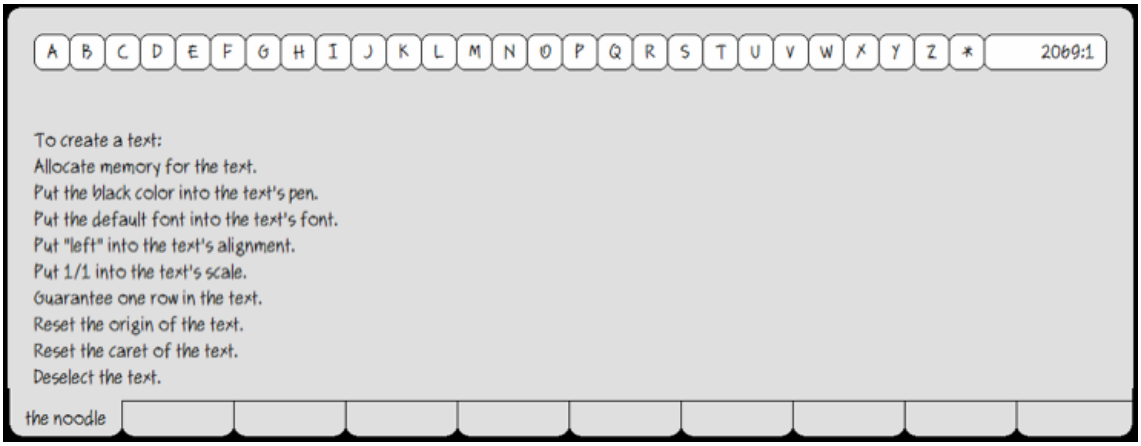


Figura 3.2: Editor de Cal-4700 con una parte del archivo *noodle* en visualización.

The writer

El escritor es un programa de diseño de documentos de la forma WYSIWYG (*What You See Is What You Get*, lo que ves es lo que obtienes) lo que se refiere a que es un editor

que trabaja en modo de diseño. Es posible que las páginas contengan gráficos vectoriales, imágenes del mapa de bits y texto, permite verificar la ortografía, imprimir, agrandar y reducir, entre otras opciones. La Figura 3.3 muestra al escritor en el área de trabajo, se observa una parte del archivo de documentación de Cal-4700.

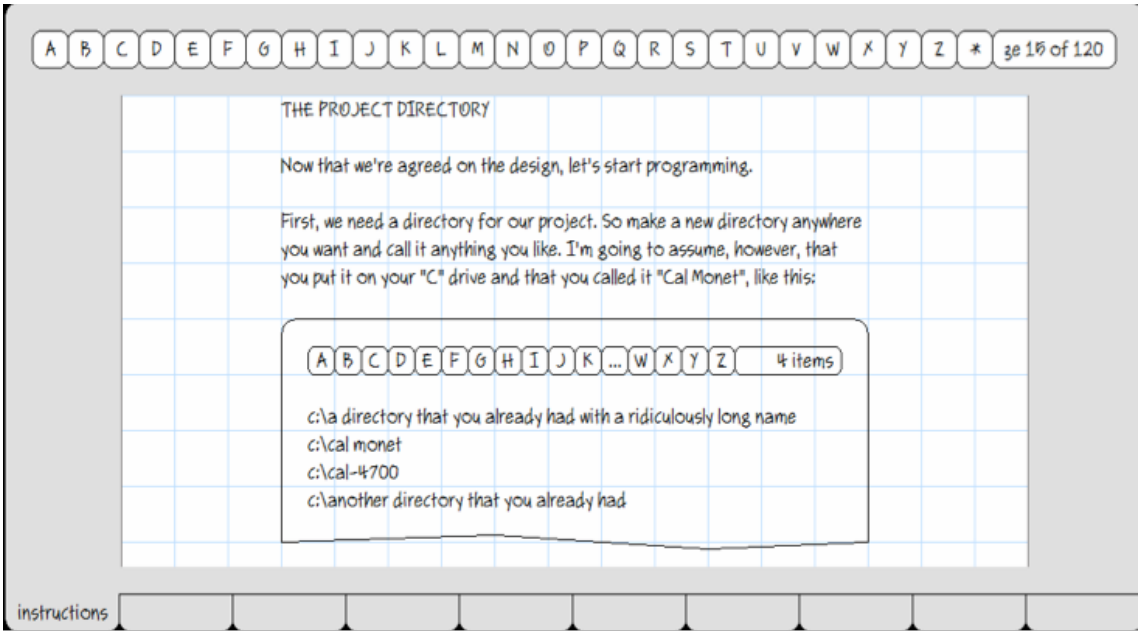


Figura 3.3: El escritor de Cal-4700 con el archivo de documentación en visualización.

3.1.2. Gramática de *Plain English*

Cal-4700 cuenta con un sistema de estructuración única para formar sus instrucciones llamado *Plain English*. *Plain English* usa elementos del lenguaje natural del idioma inglés para definir la estructura y la manera en que se forman las oraciones válidas para el lenguaje. En seguida se presenta la estructura y reglas de formación de las oraciones de *Plain English* en Cal-4700.

Tipos de oraciones en Cal-4700

Cal-4700 solo admite cinco tipos de oraciones:

- Definiciones de datos, que siempre comienzan con *A, AN* o *SOME*;

- Definiciones de variables globales, que siempre comienzan con *THE*;
- Encabezados de rutina, que comienzan con *TO*;
- Declaraciones condicionales que comienzan con *IF*; y, finalmente
- Declaraciones imperativas que comienzan con cualquier otro elemento.

La Figura 3.4 muestra las reglas de formación de los tipos de oraciones en Cal-4700.

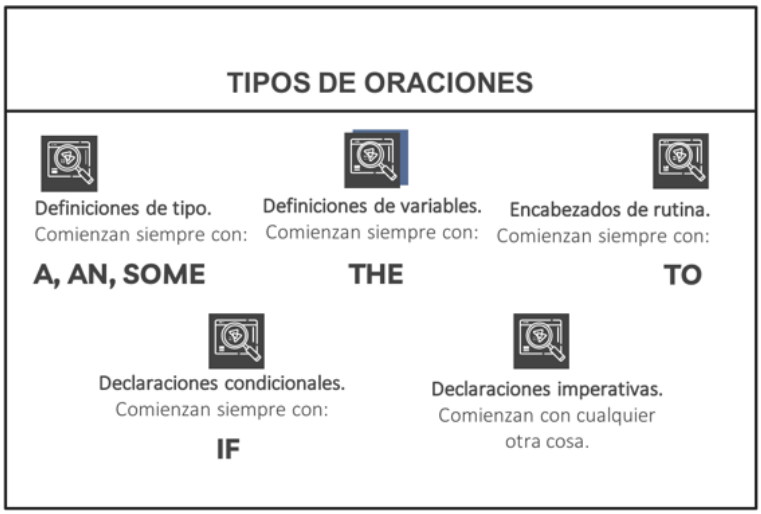


Figura 3.4: Tipos de oraciones válidas en Cal-4700.

Nombres de variables en Cal-4700

Cal-4700 trata como nombre de variable a todo lo escrito después de las palabras *A, AN, ANOTHER, SOME* o *THE* y hasta encontrar:

- Algún verbo simple, como *IS, ARE, CAN, DO,* o
- Alguna conjunción, como *AND* u *OR*, o
- Alguna preposición, como *OVER, UNDER, AROUND o THRU*, o
- Alguna literal, como 123 u "¡hola mundo!",
- O algún signo de puntuación.

La Figura 3.5 muestra la forma en que se nombran a las variables en Cal-4700.

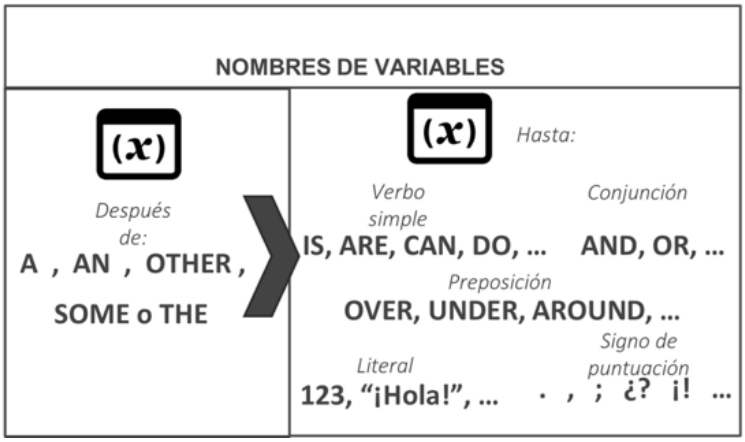


Figura 3.5: Nombres de variables en Cal-4700.

Consideraciones de la sintaxis de *Plain English*

Cal-4700 no les da consideración especial a las demás palabras, que considera solamente como "palabras" sin significado especial, a excepción de los siguientes casos:

- Operadores infijos (notación que describe a los operadores entre los operandos, ej. 5+3): *PLUS, MINUS, TIMES, DIVIDED BY* y *THEN*;
- Palabras de definición especial: *CALLED y EQUAL*; y
- Verbos en imperativos reservados: *LOOP, BREAK, EXIT, REPEAT* y *SAY*.

En la Figura 3.6 se muestran las consideraciones del lenguaje *Plain English* para la construcción de oraciones.

3.1.3. Ortografía de Cal-4700

En el presente apartado se presentan las normas consideradas en la escritura del lenguaje *Plain English*, la forma correcta de escribir las palabras válidas y de utilizar los signos.

- Es posible escribir el código usando mayúsculas o minúsculas o mezclas.
- Es indistinto el orden de las definiciones. Es posible colocar las definiciones al inicio del texto, o al final sin que represente un problema para el compilador.

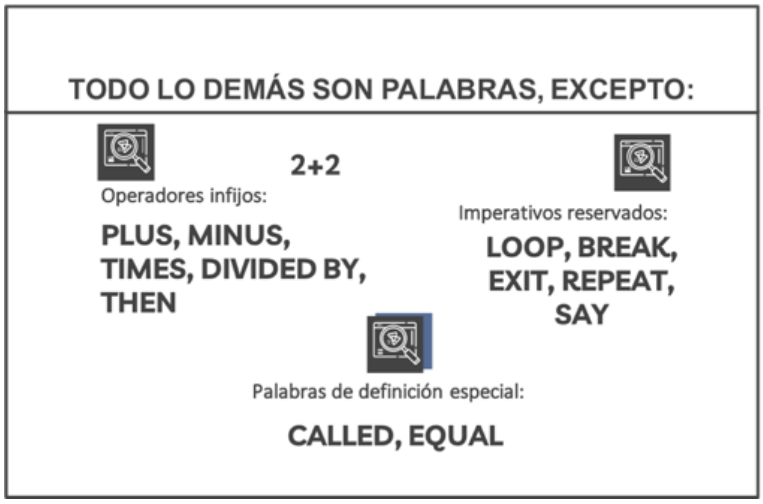


Figura 3.6: Otras consideraciones del lenguaje.

- No es posible realizar *IF* anidados. Esto se debe a la filosofía de sus creadores quienes consideran que los *IF* anidados son una señal de un razonamiento poco claro por parte de los programadores.

- No es posible realizar *LOOP* anidados. Al igual que el punto anterior, esto es a causa de la filosofía de los creadores que animan a los programadores a dividir adecuadamente el código en fragmentos manejables.

- No se permite el uso de números reales. El sistema usa fracciones en lugar de números reales. El uso de fracciones permite reducir, ampliar y redimensionar proporcionalmente formas creadas dentro y fuera de agrupamientos de una manera más naturalística.

- No se permite el uso de ecuaciones matemáticas. El lenguaje de *Plain English* no admite el uso de incógnitas matemáticas. Es decir, espacios en memoria a los que se les permita asignar directamente un valor con el signo igual (=) como usualmente funcionan las variables en los lenguajes tradicionales. El sistema realiza cálculos matemáticos con operadores infijos pero el resultado de dicho cálculo se almacena en el segundo elemento de la expresión. En la instrucción "`add the first number to the second number` (sumar el primer número al segundo número)" el resultado

es alcanzable a través de la definición de *the second number* (el segundo número) cuyo valor original cambió permanentemente.

- *Plain English* no usa objetos: Es importante considerar que el lenguaje no está orientado a objetos. Sin embargo, el lenguaje permite realizar abstracciones, otorga formas de reducir tipos de datos de unos a otros (subtipos) y el manejo de atributos y funciones (llamados rutinas).

3.1.4. *Plain English*

Plain English es una delimitación del idioma inglés diseñada para Cal-4700 con una gramática y sintaxis fundamentada por dicho idioma. Usa los elementos del lenguaje natural para definir la gramática y la ortografía de la sintaxis válida para el compilador. *Plain English* trabaja igual que lo hacen los lenguajes de programación procedimentales como Pascal o C pero con una sintaxis menos matemática y más natural.

Plain English surge del siguiente planteamiento: Un programador experimentado, que ha trabajado con diferentes lenguajes de programación a menudo piensa sus algoritmos en pseudocódigo y posteriormente realiza una traducción de estos pensamientos a la sintaxis del lenguaje de programación que esté usando en ese momento. Sin embargo, surge la idea: ¿Por qué no codificar en lenguaje natural y omitir el proceso de traducción? Así es como los creadores se dieron el trabajo de desarrollar el compilador de *Plain English*.

La idea de *Plain English* es que un programa debe escribirse principalmente en lenguaje natural, con fragmentos de código en la sintaxis más apropiada, según sea necesario. Se compara con un libro de matemáticas: escrito principalmente en lenguaje natural con fragmentos de fórmulas intercalados, en *Plain English* se vería así:

```
To add a number to another number:
Intel $8B85080000008B008B9D0C0000000103.
```

Donde se observa el lenguaje natural y el lenguaje máquina coexistiendo en una misma instrucción.

Volviendo a lo simple, una definición típica en Cal-4700 con *Plain English* se ve así:

```
A polygon is a thing with some vertices.
```

Internamente, el nombre *"polygon"* está asociado a una estructura asignada dinámicamente que contiene una lista de vértices doblemente enlazados. Es importante aclarar que para que esta oración tenga sentido, *"vertex"* debe estar definido en otra parte (antes o después de esta definición) y el plural *"vertices"* se entiende automáticamente. Una rutina en *Plain English* se ve de la siguiente manera:

```
To append an x coord and a y coord to a polygon:
    Create a vertex given the x and the y.
    Append the vertex to the polygon's vertices.
```

Una rutina es una parte del algoritmo principal que permite resolver una tarea específica, en este caso agregar una coordenada x y una coordenada y a un polígono.

Plain English usa oraciones imperativas para definir la secuencia de instrucciones, las instrucciones de Cal-4700 van encadenadas una detrás de la otra en cada momento para lograr el objetivo deseado. Cuando el compilador de Cal-4700 encuentra una oración imperativa, éste busca un encabezado de rutina coincidente en el código fuente actual o en la biblioteca *noodle*, al encontrarlo la considera como entendida, incluso si la coincidencia es parcial o descuidada, a diferencia de los lenguajes de programación tradicionales que exigen una coincidencia total. La forma de coincidencia parcial se explica a continuación.

Para lograr la coincidencia parcial o "descuidada" el compilador de Cal-4700 realiza un análisis inspirado en cómo operan los centros de análisis del cerebro humano; por un lado, en el hemisferio derecho, de la creatividad, se tiene una imagen ligada a una palabra, por ejemplo, "un cuadrado"; por otro lado, en el hemisferio izquierdo, del razonamiento, se tiene una actividad preexistente como "dibujar". Entonces, el cerebro al igual que el compilador en Cal-4700 hace coincidir las imágenes (Tipos) con las habilidades (Rutinas) ignorando el resto. Esta forma permite que al identificar ambos elementos se relacionen entre sí sin ser demasiado exigente en el resto de la sentencia. De esta manera se permite

al usuario definir sus propios tipos y rutinas haciendo que el lenguaje aprenda "el dialecto local", así el vocabulario y la gramática se definen además por el usuario y no solo por el compilador.

Algunas veces al realizar el análisis del código el compilador responde un mensaje con *"I don't know how to..."* (No sé cómo...) esto es a causa de que no se encontró un encabezado de rutina para la oración imperativa mostrada, en este caso el programador de *Plain English* tiene tres alternativas de solución (como se haría en una conversación diaria):

- Buscar la expresión como otros la han codificado;
- Reformular su oración con la esperanza de ser mejor entendida; o
- "Enseñar" al compilador a comprender el dialecto local (codificando una nueva rutina o un encabezado de rutina alternativo para una rutina existente).

Esto hace que la programación sea una tarea menos frustrante para programadores principiantes, sin obstaculizar su creciente comprensión de los conceptos fundamentales de la programación como: tipos, variables, rutinas, secuencias, condicionales y ciclos, entre otros.

En este sentido, los lenguajes de programación tradicionales son diferentes dado que tienden a enfatizar la sintaxis en lugar de la semántica, preocupados más por el "cómo" se dice algo que lo "qué" se dice, mientras que los lenguajes de programación naturalísticos como *Plain English* se enfocan en el "qué" se dice, incluso si diferentes personas lo dicen de distintas maneras. Los lenguajes de programación tradicionales animan al programador a definir una y sólo una forma de hacer cada una de las cosas que se tienen que hacer. Limpiar la pantalla, por ejemplo, se puede lograr diciendo algo preciso, pero poco naturalístico como:

```
graphics.lib (clrScr ());
```

El programador necesita conocer el comando correcto o debe buscarlo. Sin embargo, en *Plain English* se permite expresar la misma idea de diferentes formas como:

```
Erase the screen.
Blank out the screen.
Wipe off the screen.
Clear the screen.
```

Con esto se anima a los programadores de *Plain English* a añadir formas adicionales de decir lo mismo y que consideren oportuno, para crecer el lenguaje y hacerlo más fácil de usar cuando más se usa.

Como se abordó anteriormente, el alcance de *Plain English* se observa en el archivo *noodle*, una biblioteca que contiene vocabulario, tipos de rutinas y tipos de datos, entre otros, definidos con anticipación por sus creadores, pero además el programador puede agregar nuevas definiciones a medida que usa el lenguaje y escribe su aplicación. Es conveniente para el programador novato leer el archivo *noodle* para enterarse de las rutinas ya definidas y además mejorar el entendimiento de la gramática y sintaxis de *Plain English*.

3.1.5. El compilador de *Plain English*

El compilador de Cal-4700 escrito en *Plain English* contiene más de 3 mil oraciones imperativas, de las cuales alrededor del 42 % son declaraciones condicionales. En el Código 3.1, las primeras dos líneas muestran declaraciones condicionales triviales mientras que las líneas 3 y 4 muestran declaraciones condicionales más complejas pero que aún así se explican en una línea.

Código 3.1: Declaraciones condicionales en *Plain English*.

```
1        If the item is not found, break.
2        If the compiler's abort flag is set, exit.
3        If the length is 4, attach $FF32 to the fragment's code;
            exit.
4        If the rider's token is any numeric literal, compile the
            literal given the rider; exit.
```

De las restantes oraciones, aproximadamente el 9 % son declaraciones de asignación simple como la observada en el Código 3.2. Alrededor del 7 % corresponden a la estructura de varios ciclos, ver el Código 3.3. El 6 %, cumple la función de agregar algo al final de alguna lista, se muestra en el Código 3.4. El 5 % corresponde a declaraciones que se usan para devolver resultados booleanos, iniciar y detener temporizadores, mostrar el estado actual del programa o escribir cosas en la salida del compilador, mostrado en el Código 3.5. Alrededor del 4 %, observado en el Código 3.6, permite mover la zona de enfoque del compilador en el código fuente. Aproximadamente el 3 % del código se usa para mantener actualizados los índices internos en frases como las observadas en el Código 3.7. El 2 % se usa para localizar cosas en las listas como se observa en el Código 3.8. El 1 % hace llamadas a rutinas de conversión, mostrado en el Código 3.9. Otro 1 % se usa para generar código de máquina, se observa en el Código 3.10.

Código 3.2: Declaraciones de asignación simple en *Plain English*.

```
1        Put the type name into the field's type name.
```

Código 3.3: Estructura de ciclos en *Plain English*.

```
1        Loop.
2        Get a field from the type's fields.
3        [other stuff here]
4        Repeat.
```

Código 3.4: Manejo de listas en *Plain English*.

```
1        Add the field to the type's fields.
```

Código 3.5: Variedad de declaraciones en *Plain English*.

```
1        Say no.
2        Say yes.
```

```
3         Set the variable's compiled flag.
4         Start the compiler's timer.
5         Stop the compiler's timer.
6         Show status "Compiling".
7         List the globals in the compiler's listing.
```

Código 3.6: Manejo del enfoque en el código de *Plain English*.

```
1         Bump the rider.
2         Move the rider (code rules).
```

Código 3.7: Manejo de índices internos de *Plain English*.

```
1         Create the type index using 7919 for the bucket count.
2         Index the type given the type's name.
3         Destroy the type index.
```

Código 3.8: Buscar "cosas" en varias listas de *Plain English*.

```
1         Find a variable given the name.
```

Código 3.9: Rutina de conversión en *Plain English*.

```
1         Loop.
2         Get a field from the type's fields.
3         [other stuff here]
4         Repeat.
```

Código 3.10: Generación de código máquina en *Plain English*.

```
1         Attach $E8 and the address to the fragment.
```

Todo lo anterior representa el 80 % del código del compilador. Del restante, solo el 2 % representan oraciones de naturaleza matemática como las mostradas en el Código 3.11. El resto representa cosas fáciles, rutinas que hacen la mayoría de los programas y están expresadas en lenguaje natural con *Plain English*, el Código 3.12 muestra un ejemplo de estás instrucciones.

Código 3.11: Oraciones de naturaleza matemática en *Plain English*.

```
1  Add 4 to the routine's parameter size.
2  Subtract the length from the local's offset.
3  Multiply the type's scale by the base type's scale.
4  Calculate the length of the field's type.
5  Round the address up to the nearest multiple of 4096.
```

Código 3.12: Resto de instrucciones en el compilador.

```
1  Copy the field into another field.
2  Append the fragment to the current routine's fragments.
3  Abort with "I was hoping for a definition but all I found
      was" then the token.
4  Initialize the compiler.
5  Remove any trailing backslashes from the path name.
6  Reduce the monikette's type to a type for utility use.
7  Eliminate duplicate nicknames from the type's fields.
8  Prepend "original" to the term's name.
```

3.1.6. Trazabilidad de *Plain English*

Plain English se diseñó para ser, no solo el lenguaje de Cal-4700, sino un sistema "todo en uno": un entorno donde las conexiones entre el código fuente en inglés de alto nivel y las llamadas al sistema operativo y el código de máquina requerido para programas ejecutables independientes se logran rastrear fácilmente, ver Figura 3.7

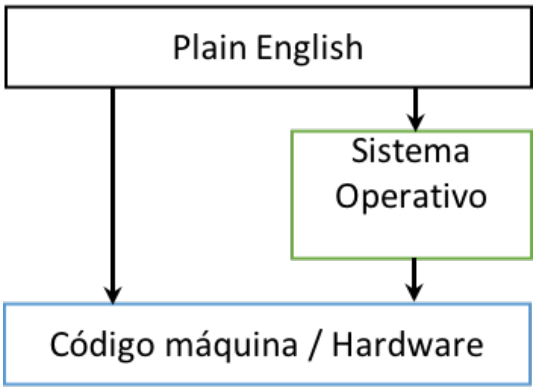


Figura 3.7: Diagrama que muestra las llamadas que realiza *Plain English*.

Enseguida se desarrolla un ejemplo del entorno de *Plain English*. Se muestra un programa simple en el editor de Cal-4700, ver Figura 3.8. Aunque el código es pequeño, incluye una secuencia, una declaración condicional, un bucle, llamadas internas, una llamada externa al sistema operativo y código de máquina escrito a mano.

En seguida, el programa hace uso de la biblioteca de Cal-4700 de tipos, variables y rutinas de propósito general (*the noodle*). Específicamente la línea con la instrucción *"Buzz"* hace una llamada al sistema operativo, la especificación de la llamada se encuentra en el *noodle* y se muestra en la Figura 3.9.

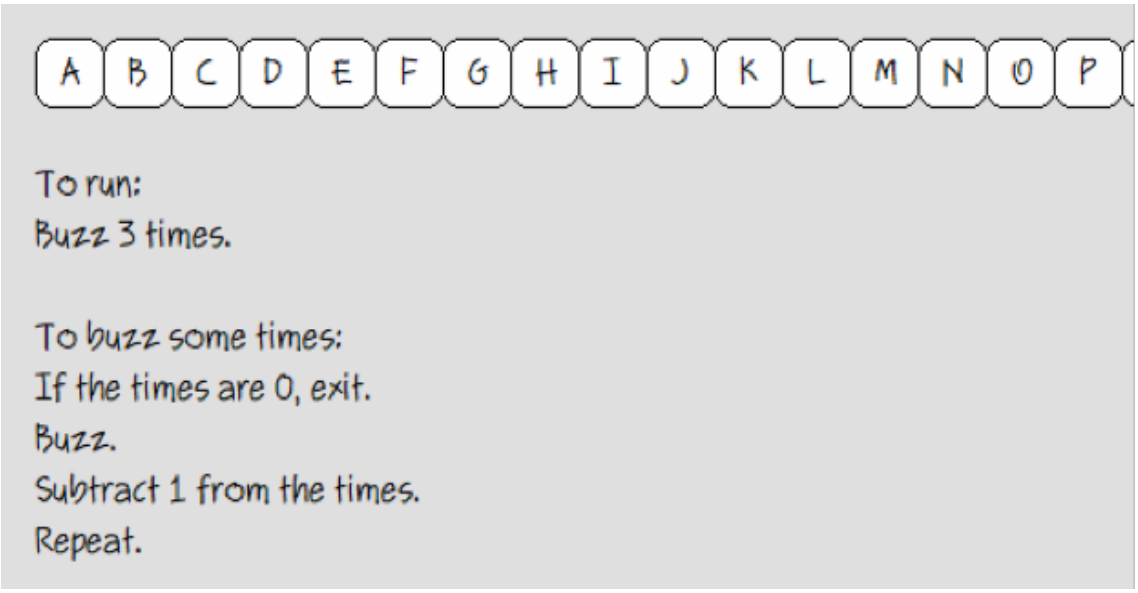


Figura 3.8: Programa simple en editor de Cal-4700.

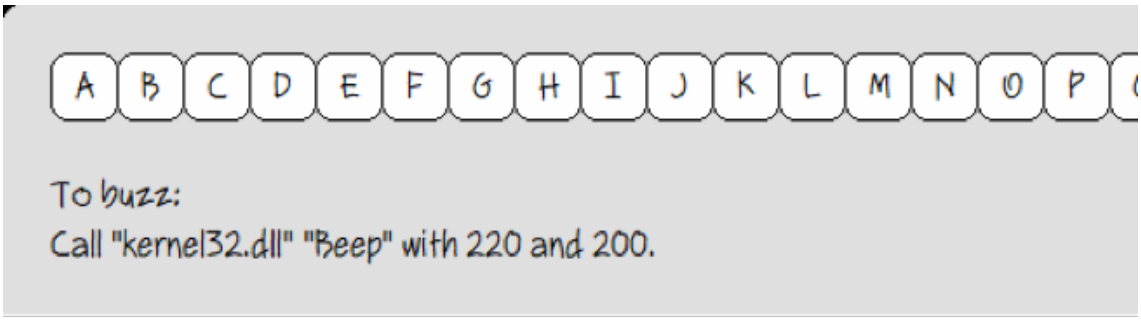


Figura 3.9: Llamada al sistema operativo desde el *noodle* en el editor de Cal-4700.

Luego, en la misma biblioteca, se encuentra localizada la rutina para comparar un número con otro, vista en la Figura 3.10, esto se encuentra a nivel de código máquina.

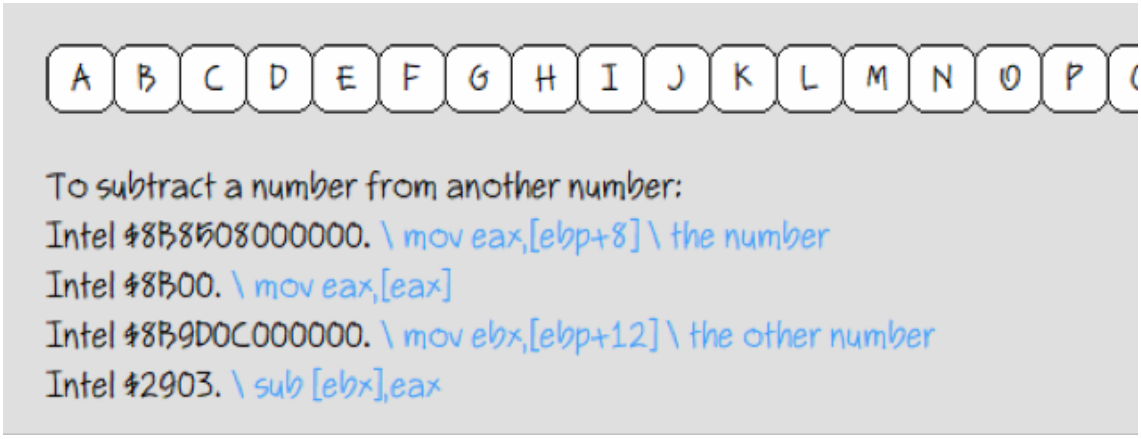


Figura 3.10: Código máquina en el *noodle* de Cal-4700 mostrada en el editor.

Finalmente, al usar el comando "*List*" se aprecia el proceso que realiza el compilador durante la ejecución del presente ejemplo. Al abrir el archivo en el editor de Cal-4700, Figura 3.11, se observa cómo se manejan los parámetros, se hacen las llamadas, se manejan la declaración condicional y el ciclo. El código máquina se muestra a la derecha de cada línea de código. Desde el lenguaje sencillo de *Plain English* se realizan las llamadas al sistema operativo y al código de máquina, todo dentro de los límites de *Plain English*.

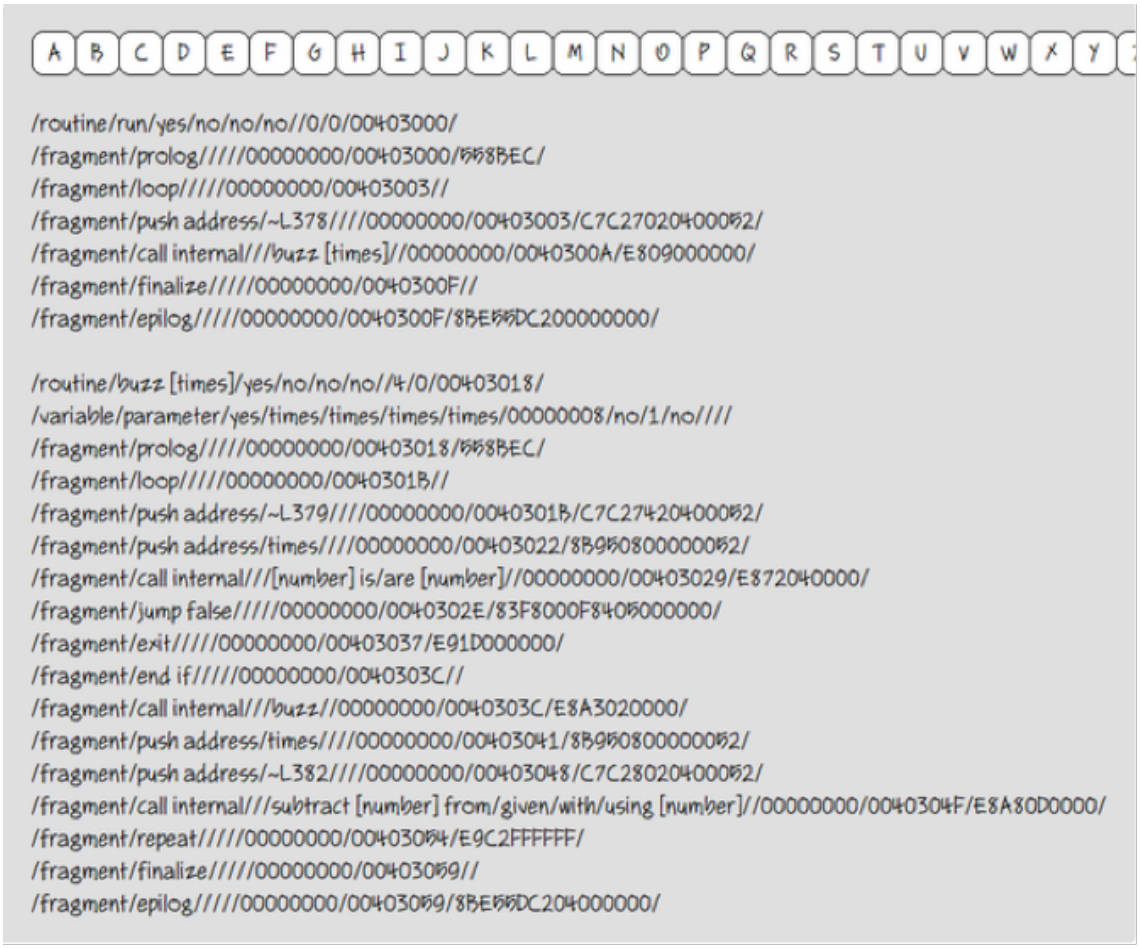


Figura 3.11: Detalle de los procesos que realiza el compilador mostrado en el editor de Cal-4700.

3.1.7. *Plain English*, un enfoque naturalístico

En *Plain English* no son necesarios nombres propios para parámetros y variables. Esta es una idea importante porque en el mundo real cuando se hace referencia a objetos como una silla o una mesa, usando lenguaje natural, nunca se les nombra "*m*" o "*myTable*" sino que simplemente se les llama como "la silla" o "la mesa", nombres naturalísticos para estas variables. En *Plain English* se permiten espacios en los nombres de las variables, cosas como "`x coord`" además de apodos para estos nombres, como "`x`" para "`x coord`". En *Plain English* los posesivos se utilizan de manera naturalística para hacer referencia a campos dentro de registros, cosas como "*polygon's vertices*" (los vértices del polígono).

Algunas personas podrían pensar que programar en lenguaje natural es una idea absurda, pero es importante considerar que el código en la mayoría de los programas hace cosas simples como "mostrar eso en la pantalla" o "mover esto para allá" cosas que pueden expresarse de manera conveniente y naturalística en el lenguaje natural.

Otra objeción al paradigma naturalístico considera que el lenguaje natural es demasiado detallado para programar, sin embargo, de ser cierto no tiene porque ser tan malo. Se consideran los siguientes ejemplos.

En programación tradicional se dibuja un cuadro con texto usando una declaración como la siguiente:

```
substring.draw ( box, color, source.text.font, source.text.alignment );
```

Lo que comprende 10 palabras y 11 signos de puntuación: 21 elementos en total.

La instrucción equivalente en *Plain English* es la siguiente:

```
Draw the substring in the box with the color and the source's text's font
                                and alignment.
```

Que son 16 palabras y 3 signos de puntuación: 19 elementos en total.

Otro ejemplo es el siguiente, una declaración condicional para evaluar el color de una fuente. En lenguaje de programación tradicional se usa una declaración como la siguiente:

```
substring.draw (if ( ! source.colorized ( ) ) color = black ;
```

Que son 5 palabras y 8 signos de puntuación: 13 elementos en total.

Mientras que en *Plain English* se usa la siguiente expresión:

```
If the source is not colorized, put black into the color.
```

Que son 11 palabras y 2 signos de puntuación: 13 elementos en total.

Es notable que el lenguaje natural no es problema para programar. En este sentido se concluye que la cuestión principal es considerar si es mejor escribir palabras o código especializado, si es mejor pensar en dos formas sintácticas y gramaticales diferentes simultáneamente o en una sola, si se desea que el código sea autodocumentado, si se desea

que el código sea amigable para principiantes, y además considerar que un idioma natural como el inglés es probable que siga siendo de uso común por muchos años.

Se considera que el programador del futuro programará las computadoras de la misma manera que hoy "se programan" los asistentes virtuales tales como *Amazon Echo*, *Apple Siri* o *Google Home*. Cuando las personas decimos a los asistentes frases como:

"`Alexa, pon un temporizador de 4 minutos`".

Se está escribiendo y ejecutando un pequeño programa en idioma natural. Lo anterior prueba la viabilidad del enfoque naturalístico, asegurando que está en el camino correcto.

3.2. Método de desarrollo naturalístico

El método naturalístico ha sido propuesto en [20] y parte del modelado del lenguaje natural realizado en [4], donde se considera que dado que el lenguaje natural es complejo es necesario realizar un modelo que lo simplifique y lo formalice. Entonces, considerando que el lenguaje está relacionado con el pensamiento humano, es necesario desarrollar un modelo del pensamiento donde la "idea" es la unidad básica. En este caso, para que las ideas sean accesibles por las computadoras se realizó una formalización llamada notación de ideas. Esto considera dos tipos de ideas: las ideas atómicas que requieren la cantidad mínima de información y se describen como un solo término y las ideas complejas que se describen como una lista de todas las ideas que las componen. La construcción del modelo permitió realizar una traducción de las ideas del lenguaje natural a código fuente. Por otra parte, se comprobó que es posible modelar el lenguaje natural con el uso de artefactos adecuados para este nuevo paradigma. A continuación se presenta el método naturalístico de desarrollo de software, así como el uso de los artefactos que propician el modelado del pensamiento naturalístico, ver Figura 3.12.

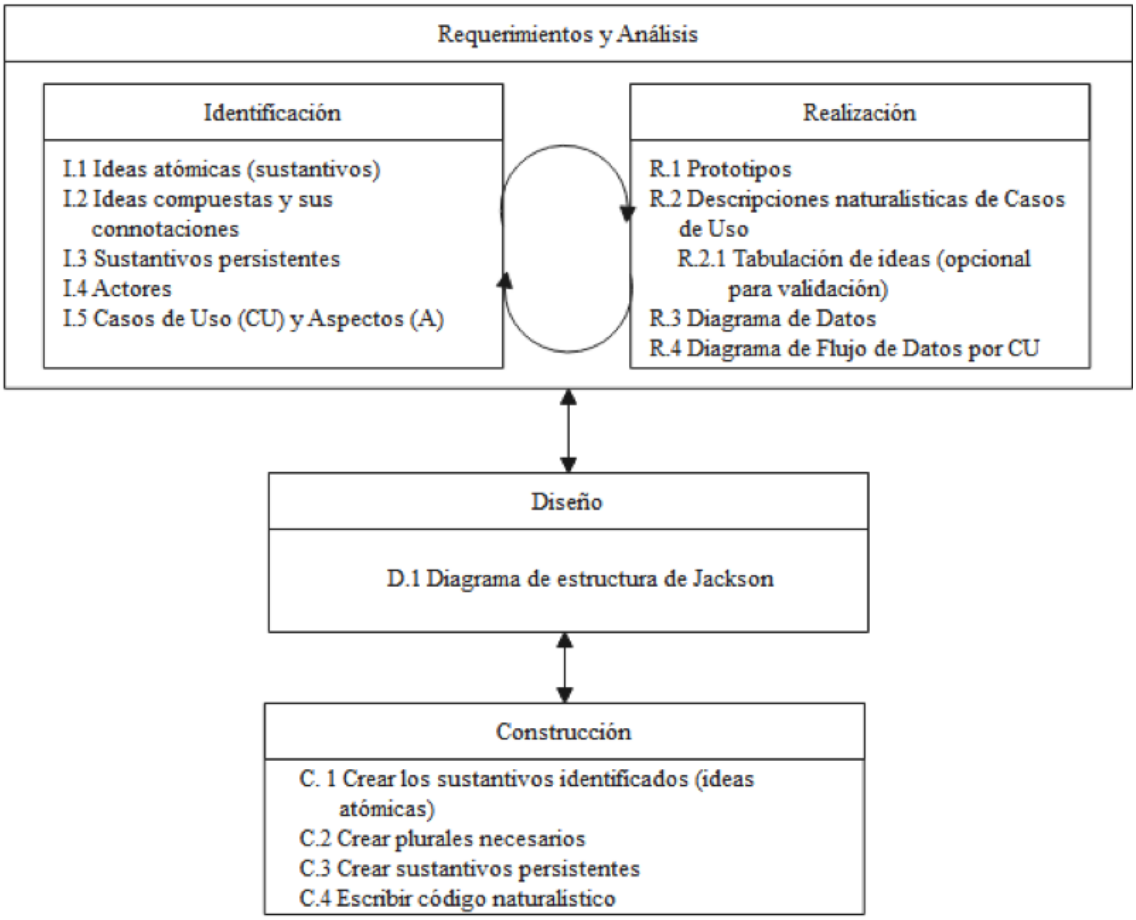


Figura 3.12: Método de desarrollo naturalístico.

3.2.1. Requerimientos y análisis

Como todo método de desarrollo de software el método naturalístico parte del análisis de requerimientos. El análisis de requerimientos permite conocer y entender las necesidades que el software debe atender. Esta etapa define qué se requiere del sistema.

Etapa de identificación

La etapa de identificación permite al desarrollador tener un entendimiento claro de lo que el software necesita resolver antes de realizarlo. Esta etapa reconoce el contexto en que el software debe realizarse, los conceptos específicos que se abordarán en la realización, el diseño, la construcción y el orden en que se efectuará el trabajo. El trabajo realizado en la

etapa de identificación afectará la información, las funciones y los comportamientos que tendrá el diseño resultante. En seguida se definen los pasos que componen a la etapa de identificación.

- **Identificación de ideas atómicas (sustantivos), compuestas y sus connotaciones.** El término idea en el modelo de desarrollo naturalístico surge como una forma de expresar el pensamiento en pequeñas unidades de información mínima que a través de una notación específica que permite sea procesada por las computadoras. Existen dos tipos de ideas, las ideas atómicas y las ideas complejas. La forma de una idea atómica es un sustantivo mientras que una idea compleja se caracteriza por tener asociaciones con otras ideas atómicas o complejas. En la notación de ideas, las ideas atómicas se describen como un solo término, mientras que las ideas complejas se describen como una lista de todas las ideas que las componen [4]. De esta forma se explica el pensamiento, como un conjunto de ideas simples y complejas, relacionadas entre sí. En la Figura 3.13 se muestra una representación gráfica de la notación de ideas para las ideas complejas *madera* y *mesa*, se observa que la concepción de las ideas es un conjunto de relaciones entre diferentes ideas complejas y atómicas. La representación anterior, aunque ilustrativa no es práctica para ser procesada por las computadoras, la notación más adecuada se muestra en la Figura 3.14 que señala la representación de las ideas usando un lenguaje formal adecuado. Es posible notar que la idea *mesa* está compuesta de la idea compleja *madera*, que a su vez está compuesta de diferentes ideas atómicas, para este caso es posible representar la idea *mesa* con la idea *madera* extendida dentro de sí, como se observa en la Figura 3.15. En este punto, es importante considerar que las ideas compuestas identificadas formarán parte de la descripción de los casos de uso, mientras que las ideas atómicas pasarán a ser sustantivos dentro del lenguaje de Cal-4700 [20].

- **Identificar los sustantivos que serán actores del sistema o persistentes.** De los sustantivos localizados en el paso anterior se deben separar los que son actores del sistema y los que son persistentes, es decir, que permanecerán aún después de

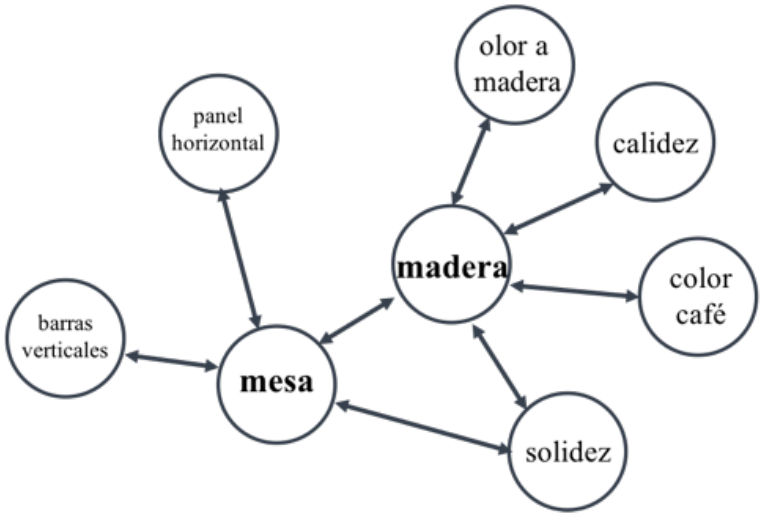


Figura 3.13: Representación gráfica de la notación de ideas.

Madera: [olor a madera, calidez, solidez, color café]

Mesa: [panel horizontal, barras verticales, madera, solidez]

Figura 3.14: Notación de ideas complejas.

Mesa: [panel horizontal, barras verticales, [olor a madera, calidez, solidez ,color café], solidez]

Figura 3.15: Notación de ideas extendida.

concluida la ejecución del programa. La forma de identificarlos es dividir los sustantivos que solo son actores del sistema de los que tienen otro dato o cosa (*Thing*) dentro del programa.

- **Identificar los casos de uso (CU) y aspectos.** Siguiendo los pasos anteriores, a este punto se tiene claro cuáles elementos formarán parte de los casos de uso. Entre ellos es importante identificar cuáles serán aspectos (*concerns*, que hacen un corte transversal a la funcionalidad del sistema). Los casos de uso definidos más adelante describirán el comportamiento del sistema en las distintas condiciones en las que el sistema responde a las peticiones de alguno de sus participantes. Los casos de uso en el desarrollo de software son bien conocidos porque mejoran la comprensión de los requerimientos de los participantes del sistema al mostrar descripciones en lenguaje natural, característica compatible con la filosofía del paradigma naturalístico.

Etapa de realización

En la etapa de realización se modelan los datos de manera naturalística. Esta tarea de modelado conduce a la especificación de requerimientos y la representación de un diseño lógico del software que se va a elaborar. En seguida se presentan los artefactos de modelación del método naturalístico:

- **Descripción naturalística de los casos de uso y tabulación de ideas.** Después de la etapa de identificación se tiene clara la visión de las funciones y características del sistema. Para continuar hacia las actividades técnicas es necesario entender cómo emplearán los actores del sistema dichas funciones y características. La descripción naturalística de los casos de uso redacta los flujos de datos normal, alternos y excepciones para cada caso de uso, narra una historia sobre cómo interactúa un actor con el sistema en circunstancias específicas, en este caso la historia es un texto narrativo. La redacción de los casos de uso permite contener los elementos del paradigma naturalístico como son las anáforas y catáforas. Es importante aclarar que un actor y un usuario no son necesariamente lo mismo, un actor también es una

idea o un aspecto. En la Figura 3.16 se muestra un ejemplo de descripción de casos de uso para el caso de uso "inscribir a un estudiante" en un sistema hipotético de inscripciones en un colegio. De manera adicional en este paso se propone presentar las ideas en forma de tabla que permita realizar un seguimiento y validación de las ideas hasta su construcción y validación. En la Figura 3.17 se muestra un ejemplo de una tabulación de ideas relacionado con el sistema de inscripciones que se mencionó previamente.

- **Diagrama de flujo de datos (DFD) por cada caso de uso.** El DFD [21] permite una representación gráfica de los datos, los actores y el flujo de la información entre estos. El enfoque del DFD es de tipo entrada-proceso-salida. Es decir, la información que entra al sistema, se transforma por el proceso y los datos que resultan salen del sistema. El DFD se compone de los siguientes elementos:

  - Burbujas: representan las acciones o procesos que hará el sistema.
  - Cuadrados: representas las entidades externas que producen la información para el uso del sistema y que además consumen la información generada por este.
  - Flechas: que representan el flujo de datos, estos son comandos de configuración o de activación/desactivación o datos de usuario.
  - Líneas dobles: representa el almacenamiento de datos.

  Para realizar este proceso es necesario hacer un análisis de la descripción de los casos de uso, donde se aíslan los sustantivos y los verbos. Los verbos son los procesos y se representan como burbujas mientras que los sustantivos son entidades y se representan como cuadros. Los sustantivos y verbos van asociados entre sí para cumplir el contrato del funcionamiento del sistema, esta asociación se representa con flechas que muestran el flujo de información entre los elementos. En la Figura 3.18 se aprecia el DFD para el caso de uso "inscribir estudiante", se observa el actor involucrado, las acciones y los almacenes de datos correspondientes, además es visible el aspecto "registrar seguimiento", que está representado con una línea punteada.

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 pregunta ¿Deseas registrar otro curso? 6.a Si: regresar a paso 4 6.b No: termina CU 7. 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.
Entradas:	El número de control del estudiante así como el número de cada curso a inscribir.
Salidas:	Lista de cursos inscritos.
Incluye: (relación include)	
Extiende: (relación extend)	
Lanzar: (relación trigger)	Registrar en archivo Seg
Prioridad:	Alta

Figura 3.16: Descripción del caso de uso Inscribir Estudiante.

No.	Enunciado/ Statement (acción)	Comando/ Command (frase)	Condición/ Condition	Agente	Anáfora	Objeto directo/ Direct object	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			Preguntar	Sistema		Curso	
6a		Si: Ejecutar		Sistema		Paso 4	
6b		No: Salir		Sistema			
7	Imprimir			Sistema	Cursos inscritos		

Figura 3.17: Tabulación de ideas.

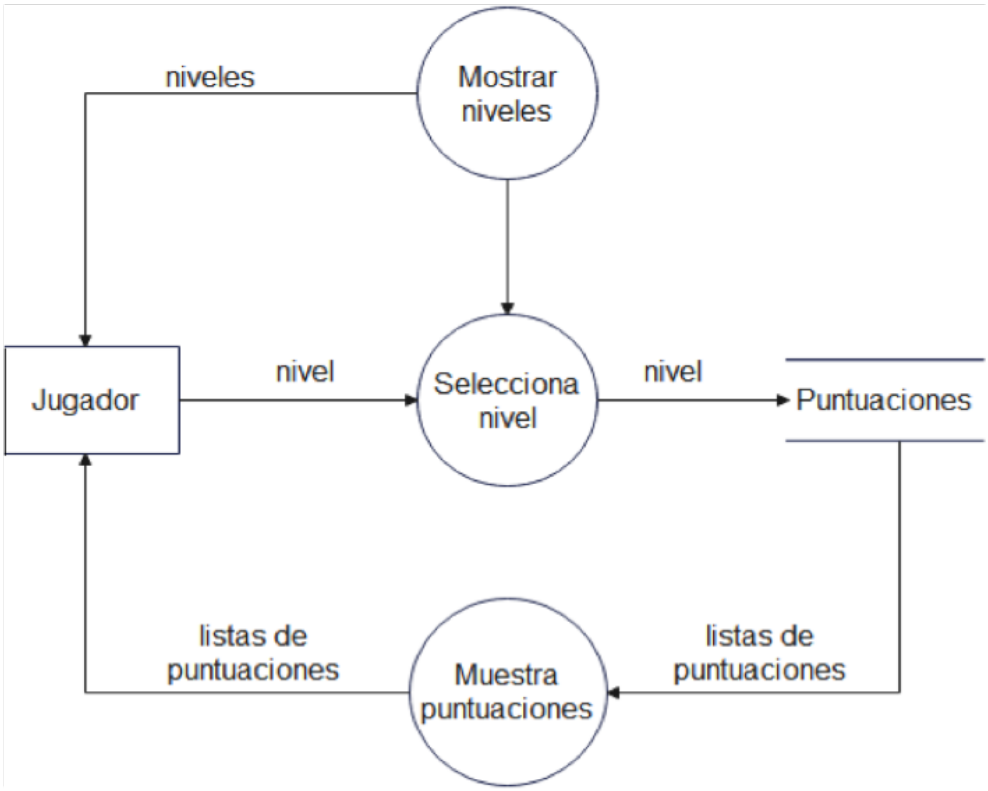


Figura 3.18: DFD del caso de uso Inscribir estudiante.

En la descripción del CU se incluye el aspecto "registrar seguimiento" como una relación de lanzamiento (trigger).

- **Prototipos.** Un prototipo [22] es la representación de una parte o de todo el sistema, sirve como modelo para fabricar la parte que representa. La creación de prototipos ayuda a los participantes a mejorar la comprensión de lo que se tiene que construir en caso de que los requerimientos no sean claros. Los prototipos están centrados en la representación de las partes del software que serán visibles para los usuarios finales, (por ejemplo, una interfaz gráfica o formatos de pantalla de salida). En la Figura 3.19 se muestra un prototipo para el caso de uso "Inscribir Estudiante", este incluye los elementos mínimos necesarios que se usaron para el modelado, además de otros datos descriptivos.

- **Diagrama de datos.** El diagrama de datos representa un esquema de persistencia

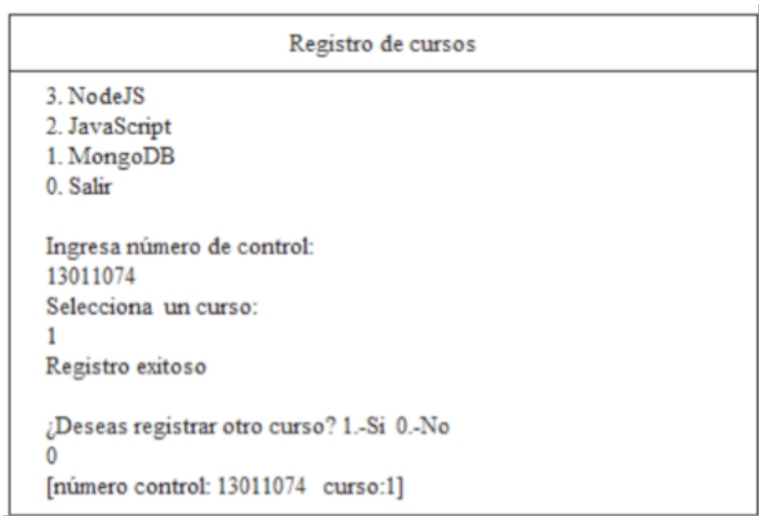


Figura 3.19: Prototipo del CU Inscribir Estudiante.

de la información, en esta sección normalmente se considera un diagrama de base de datos relacionales. Cal-4700 cuenta con limitaciones respecto a este tópico debido a que sus autores no consideraron un esquema de persistencia tradicional, sino mediante el uso de archivos de texto plano. Por tal motivo, es necesario investigar, como trabajo futuro, la forma apropiada y conveniente de modelar la información persistente dentro de los archivos de texto. A ese respecto, en este trabajo de investigación se considera la elaboración de un esquema que muestre la estructura y relación de los diferentes archivos de texto que funcionen para almacenar la información.

3.2.2. Etapa de diseño

Esta etapa agrupa los principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. El objetivo principal del diseño es producir un modelo resistente y funcional. La etapa de diseño inicia una vez que se ha realizado la fase de requerimientos y análisis, forma parte de la actividad de modelado y prepara la etapa de construcción. Para la etapa de diseño se propone realizar el diagrama de estructura de *Jackson* [23], que describe la solución de forma secuencial, en términos de tres formas de estructura: secuencia, selección e iteración. Su representación utiliza frases en lenguaje

natural, conforme al enfoque naturalístico. En la Figura 3.20 se observa el diagrama de estructura de *Jackson* para el CU "Inscribir estudiante", en este se aprecia la creación del sustantivo "Estudiante" dado que es una decisión de diseño. También, se observa el aspecto "registrar seguimiento", nuevamente identificado con una relación del tipo «trigger».

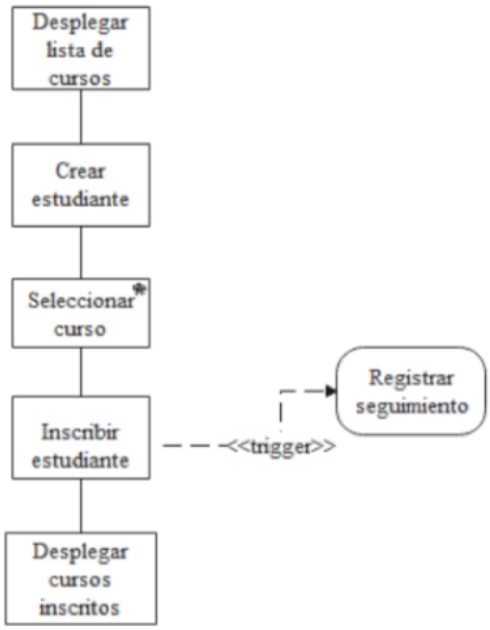


Figura 3.20: Diagrama de estructura de *Jackson* del CU Inscribir estudiante.

3.2.3. Etapa de construcción

La etapa de construcción considera un conjunto de tareas de codificación y pruebas que llevarán a un software a estar listo y operable. Para esta etapa se consideran las siguientes actividades:

- Crear los sustantivos identificados como ideas atómicas.
- Crear los plurales necesarios (colecciones).
- Crear los sustantivos persistentes, en caso de ser necesarios.
- Escribir el código naturalístico en Cal-4700.

El método naturalístico surge como respuesta a la necesidad de modelar los requerimientos de software con enfoque en el lenguaje natural. El lenguaje natural es ambiguo, sin embargo, permite expresar de mejor manera las ideas y es por eso que se considera como la alternativa adecuada para modelar ideas que se conviertan en software. Modelar el lenguaje natural es una tarea que no atienden aún las metodologías de desarrollo actuales y que trata de evitar la deformación de los requerimientos que ocurre con las técnicas de modelado actuales. En este trabajo se usa Cal-4700 como lenguaje de programación y su enfoque de desarrollo requiere una metodología adecuada a sus características, se ha seleccionado el método naturalístico para representar sus procesos puesto que de momento es el único que está directamente pensado en el paradigma de programación naturalístico.

Capítulo 4

Resultados

En este capítulo se presenta el desarrollo del caso de estudio que considera la elaboración de una aplicación lúdica usando Cal-4700 y el método naturalístico, que demuestra los beneficios de programar usando el paradigma de programación naturalística.

4.1. Caso de estudio

En este mundo globalizado la tecnología ha tomado importancia como herramienta fundamental en el proceso de aprendizaje. El uso de software lúdico es relevante porque despierta el interés de las personas en desarrollar habilidades y vencer dificultades. Las investigaciones realizadas en [24] han demostrado que el uso de aplicaciones interactivas son capaces de ayudar terapéuticamente a niños de entre 3 y 7 años que presenten dificultades psicomotrices.

Considerando lo anterior expuesto, se desarrolla el presente caso de estudio lúdico con el fin de favorecer el desarrollo de la motricidad fina de niños mayores de 5 años. Se revisaron diferentes herramientas educativas para definir la clase de aplicación lúdica a desarrollar. Se eligió una aplicación de rompecabezas al destacar el uso del ratón al momento de resolverlo. Para los requerimientos se consideraron las características de los rompecabezas que implementa el software educativo *GCompris*[1] en la sección de rompecabezas y la

[1]Software Educativo GCompris - https://gcompris.net/ [Accedido: 20-mayo-2022]

herramienta para creación de rompecabezas *Jigsaw Planet*[2].

4.1.1. Descripción del caso de estudio

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, 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.

4.1.2. Requerimientos del caso de estudio

Los requerimientos de la aplicación lúdica se listan 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

[2]Software Jigsaw Planet - https://www.jigsawplanet.com/ [Accedido: 20-mayo-2022]

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.

4.1.3. Ideas atómicas (sustantivos)

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

- **Jugador (*Player*).** Determinado en el requerimiento R1. Sustantivo que identifica al usuario que va a interactuar con el sistema. Esté es el único actor del sistema y su importancia es altamente relevante. Contiene como atributo el dato tipo cadena: nombre (*name*) que representa al **nombre** del jugador o su *nickname*. En la Figura 4.1 se observa su representación gráfica. En el Código 4.1 se observa el desarrollo naturalístico de la idea en Cal-4700.

Código 4.1: Implementación de la idea atómica jugador con Cal-4700.

```
1       A player has a name string.
```

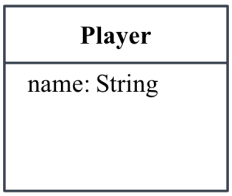


Figura 4.1: Idea atómica jugador.

- **Nivel (*Level*)**. Expresado en el requerimiento R2. Sustantivo que identifica el nivel de dificultad. Contiene el atributo **dificultad** (*difficulty*) de tipo cadena que representa el nivel de dificultad: fácil, medio y 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 4.2 se aprecia la representación gráfica de la idea. En el código 4.2 se observa la implementación de la idea *nivel*.

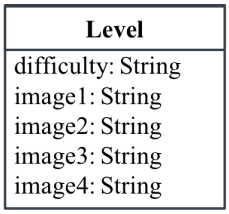


Figura 4.2: Idea atómica nivel.

Código 4.2: Implementación de la idea atómica *Nivel* con Cal-4700.

```
1 A level has a difficulty string, an image1 string, an image2
    string, an image3 string and an image4 string.
```

- **Puntuación (*Score*)**. Expuesto en el requerimiento R4. Esta idea atómica representa la puntuación del jugador. Contiene un dato tipo numérico llamado puntuación (*score*), que representa el registro numérico de la puntuación del jugador. Además, contiene el dato criterio (*criterion*) de tipo cadena que atiende al requerimiento R6 donde se específica 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 4.3, donde se aprecian las referencias a estas ideas atómicas. En el Código 4.3 se presenta la implementación de la idea *puntuación*.

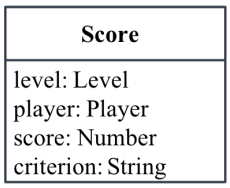


Figura 4.3: Idea atómica puntuación.

Código 4.3: Implementación de la idea atómica *Puntuación* con Cal-4700.

```
1 A score has a level Level, a player Player, a score number and a
    criterion string.
```

4.1.4. Casos de uso (CU) y aspectos (A)

El diagrama de CU para la aplicación lúdica se observa en la Figura 4.4. El requerimiento **R1** específica 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 en cualquier momento el juego iniciado. Una vez que las piezas del rompecabezas están ordenadas el juego se considera terminado (**CU Terminar juego**) y se registra la puntuación del juego (**A Registrar datos de puntuaciones**). El jugador tiene la facultad de consultar en cualquier momento las puntuaciones registradas (**CU Consultar puntuaciones**).

El **A 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 transversal a los requerimientos **R4, R5, R6** y **R8**.

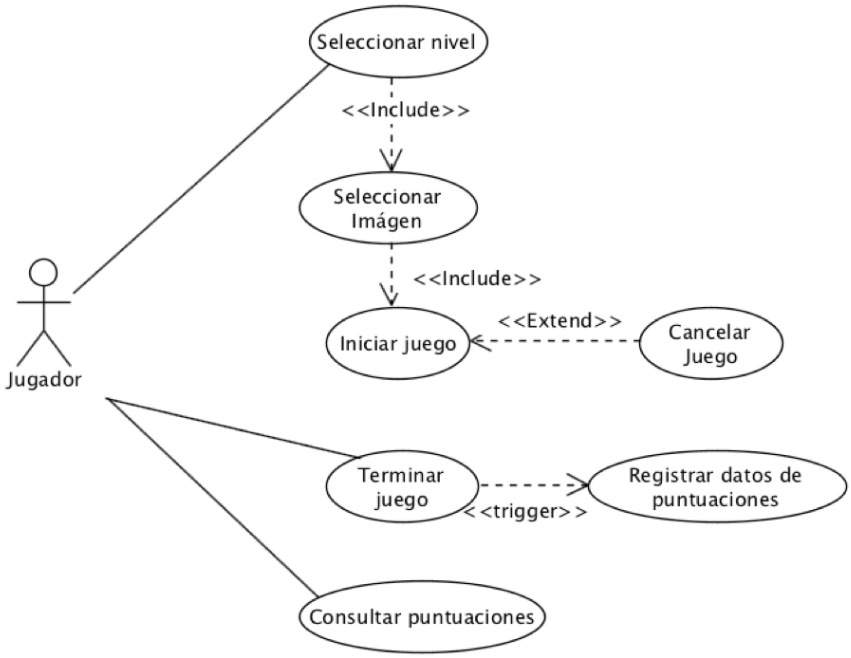


Figura 4.4: Diagrama de casos de uso de la aplicación lúdica.

Desarrollo de los casos de uso y aspectos

A partir del previo análisis se desarrolla cada CU y se presentan los artefactos de modelación elaborados así como su implementación con Cal-4700. Estas actividades corresponden a las etapas de realización, diseño y construcción del método naturalístico.

4.1.5. CU Seleccionar nivel

En el presente caso de uso se considera que el *jugador* tiene la facultad de seleccionar entre tres niveles disponibles de dificultad: fácil, medio y difícil. Este CU da inicio al camino principal de la aplicación. En la Tabla 4.1 se muestra la descripción del CU. En la Tabla 4.2 se observa la tabulación de ideas. La Figura 4.5 corresponde al DFD. La Figura 4.6 considera el prototipo. En la Figura 4.7 se observa el diagrama de estructura de *Jackson*.

Tabla 4.1: 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

Tabla 4.2: 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			

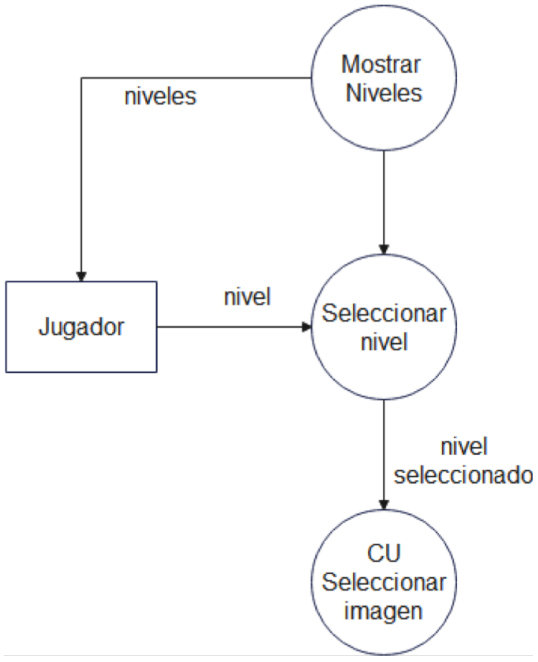


Figura 4.5: DFD del *CU Seleccionar nivel*.

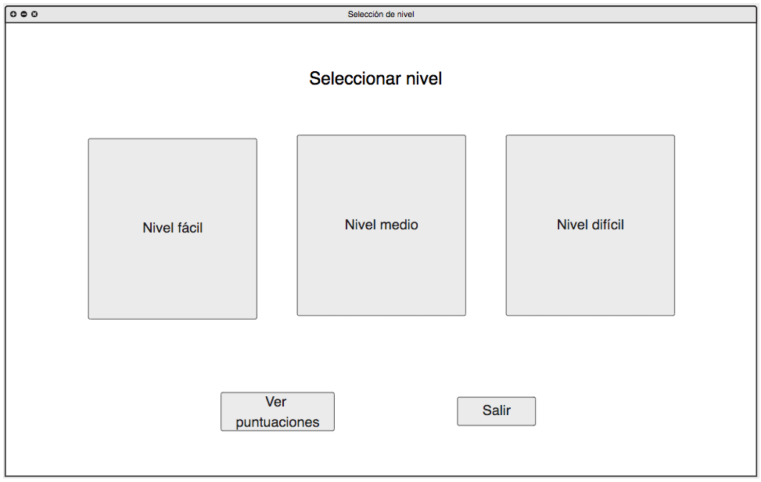


Figura 4.6: Prototipo del *CU Seleccionar nivel*.

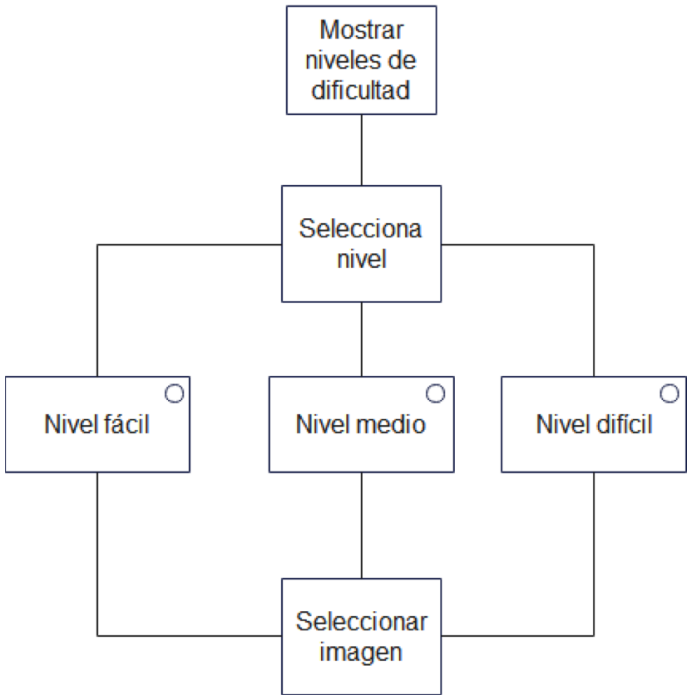


Figura 4.7: Diagrama de estructura de *jackson* del *CU Seleccionar nivel*.

En el Código 4.4 se observa la implementación del *CU Seleccionar Nivel* haciendo uso de las herramientas de Cal-4700. El desarrollo considera la definición de la rutina `show main interface` la cual realiza la función de dibujar la interfaz de usuario en la pantalla y se describe en el Apéndice B. En la Figura 4.8 se aprecia el resultado de la implementación del presente CU.

Código 4.4: Implementación del *CU Seleccionar Nivel* con Cal-4700.

```
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 4.8: Interfaz de usuario de la implementación del *CU Seleccionar nivel*.

4.1.6. CU 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 para jugar con ella. Se muestran las cuatro imágenes disponibles en miniatura. A partir del presente CU el jugador es capaz de volver al *CU Seleccionar nivel* mediante el uso de un botón. En la Tabla 4.3 se muestra la descripción de CU. En la Tabla 4.4 se observa la tabulación de ideas. La Figura 4.9 corresponde al DFD. La Figura 4.10 considera el prototipo. Y finalmente, en la Figura 4.11 se observa el diagrama de estructura de *Jackson*. Todos los artefactos mencionados corresponden al *CU Seleccionar imagen*.

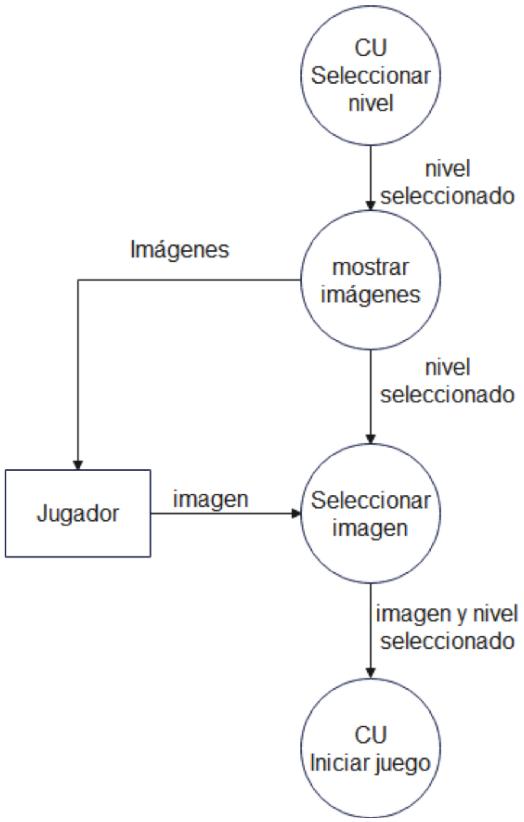


Figura 4.9: DFD del *CU Seleccionar imagen*.

Tabla 4.3: 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 4.4: 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			

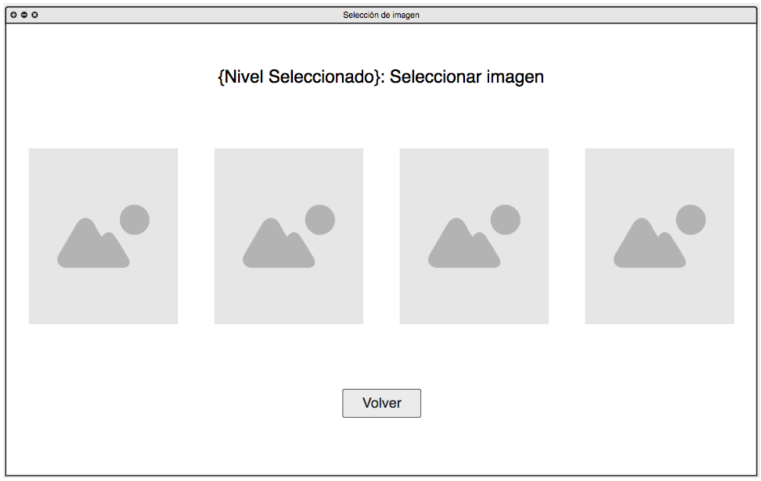


Figura 4.10: Prototipo del *CU Seleccionar imagen*.

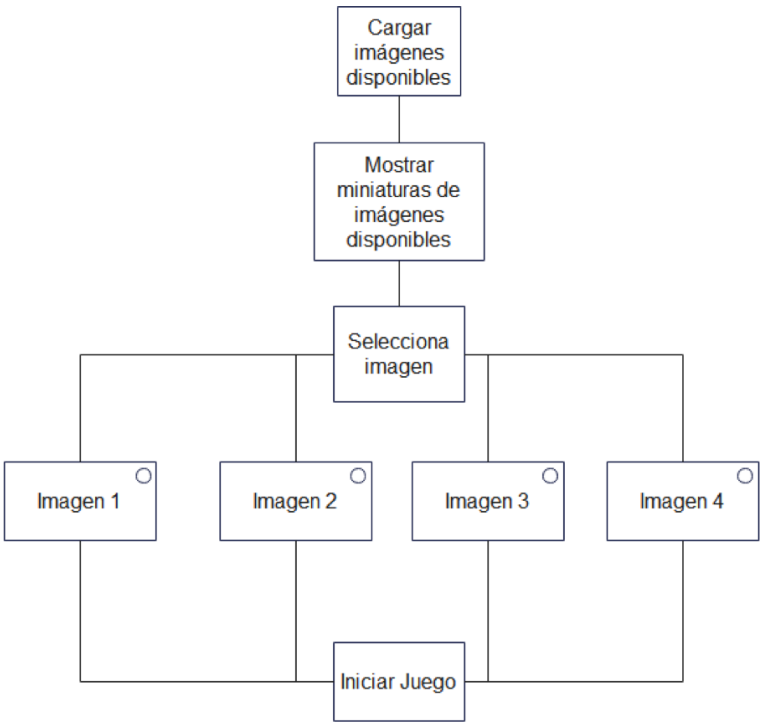


Figura 4.11: Diagrama de estructura de *Jackson* del *CU Seleccionar imagen*.

En el Código 4.5 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 y administrar lo que ocurre cuando el usuario selecciona alguna de éstas. Las rutinas que muestran las imágenes del nivel medio y difícil se encuentran en el Apéndice D, la rutina que construye la interfaz de usuario, `show level thumbnails given the easy level' difficulty` se localiza en el Apéndice B. En la Figura 4.12 se observa la interfaz de usuario de la implementación del *CU Seleccionar imagen* en el nivel fácil realizado con Cal-4700.

Código 4.5: Rutina en Cal-4700 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, go to easy
      difficulty puzzle given the image list' location one.
8  If the event's spot is in the picture button two, go to easy
      difficulty puzzle given the image list' location two.
9  If the event's spot is in the picture button three, go to easy
      difficulty puzzle given the image list' location three.
10 If the event's spot is in the picture button four, go to easy
      difficulty puzzle given the image list' location four.
11 If the event's spot is in the return button, show levels; exit.
12 repeat.
```

Figura 4.12: Interfaz de usuario de la implementación del *CU Seleccionar imagen* - nivel fácil.

4.1.7. CU Iniciar juego

El presente CU ocurre después que el *jugador* ha seleccionado el nivel de dificultad y la imagen. Al iniciar el *CU Iniciar juego* el sistema solicita al *Jugador* su nombre y éste lo escribe, posterior inicia el juego. Al comenzar el juego los contadores de número de movimientos y tiempo transcurrido (en segundos) se reinician en cero, se carga la imagen seleccionada y se divide en partes. Una vez que el juego está iniciado el *jugador* tiene la facultad de cancelar el juego, al hacerlo el sistema solicita confirmación, si la confirmación se recibe, el sistema vuelve al *CU02 Seleccionar imagen.* En la Tabla 4.5 se muestra la descripción del CU. En la Tabla 4.6 se observa la tabulación de ideas. La Figura 4.13 corresponde al DFD. Para éste CU se consideran dos prototipos puesto que el CU toma en cuenta dos actividades importantes: Solicitar el nombre de jugador, Figura 4.14 y comenzar el juego con la imagén dividida en secciones, Figura 4.15. En la Figura 4.16 se presenta el diagrama de estructura de *Jackson.*

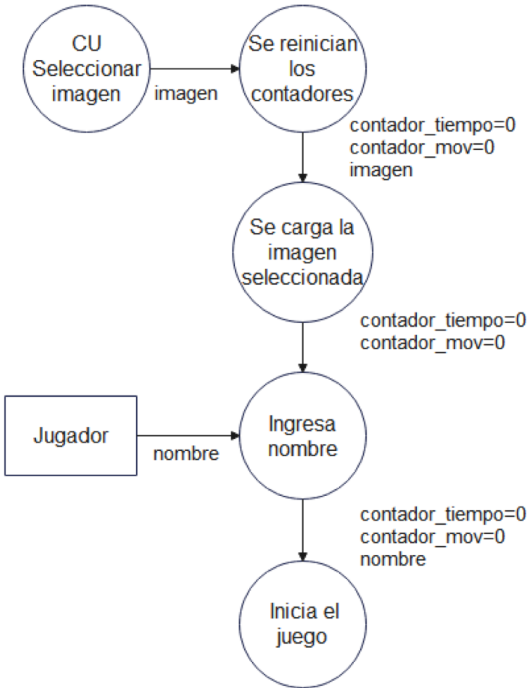


Figura 4.13: DFD del *CU Iniciar juego*.

En el Código 4.6 se observa la definición de la rutina encargada de iniciar el rompecabezas, para este ejemplo se presenta el nivel fácil. Las rutinas que inician el rompecabezas para los otros niveles de dificultad se encuentran en el Apéndice E. En el Código 4.7 se muestra la rutina que solicita el nombre al *jugador*. En el Código 4.8 se observa la rutina que construye el rompecabezas dado una dirección de imagen y un tamaño de pieza. La rutina en el Código 4.8 se encarga de crear los rompecabezas al recibir una dirección de imagen y un tamaño de pieza.

En la Figura 4.17 se muestra la interfaz de usuario que solicita el nombre del jugador. La Figura 4.18 presenta la interfaz de usuario con el juego iniciado en el nivel fácil.

Código 4.6: Rutina en Cal-4700 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.
```

```
Display the puzzle.
Loop.
Deque an event.
If the event is nil, break.
If the event's kind is not "left click", repeat.
If the event's spot is in the check button, check results.
If the event's spot is in the back button, quit; go to level easy;
    exit.
Track the mouse on the puzzle.
Repeat.
Destroy the puzzle.
```

Código 4.7: Rutina en Cal-4700 que solicita el nombre del jugador.

```
To request player name:
Show the arrow cursor.
Fill the request box with the white color.
Draw the request box.
Use small letters.
Write "Escribe tu nombre: " with the black pen at the top of the
   request box.
Draw the accept button.
Draw the text's box.
Refresh the screen.

The text has a box and a string.
```

Código 4.8: Rutina en Cal-4700 que crea los rompecabezas.

```
To create a puzzle from a path given a piece size:
Allocate memory for the puzzle.
```

```
Make the puzzle's box 6 inches by 6 inches.
Center the puzzle's box in the screen's box.
Fetch the puzzle's picture from the path.
Resize the puzzle's picture to 6 inches by 6 inches.
Center the puzzle's picture in the puzzle's box.
Cut the puzzle into pieces with the piece size.
Splatter the puzzle's pieces.
```

Tabla 4.5: Descripción de *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 *nickname*.
Flujo Normal:	1. Se reinician los contadores de número de movimientos y segundos transcurridos. 2. El sistema solicita el nombre o *nickname* 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 *nickname* del jugador.
Salidas:	1. La imagen seleccionada dividida en secciones.
Incluye: **(relación include)**	
Extiende **(relación extend)**	CU Terminar juego
Lanzar: **(relación trigger)**	
Prioridad:	Alta

Tabla 4.6: 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			

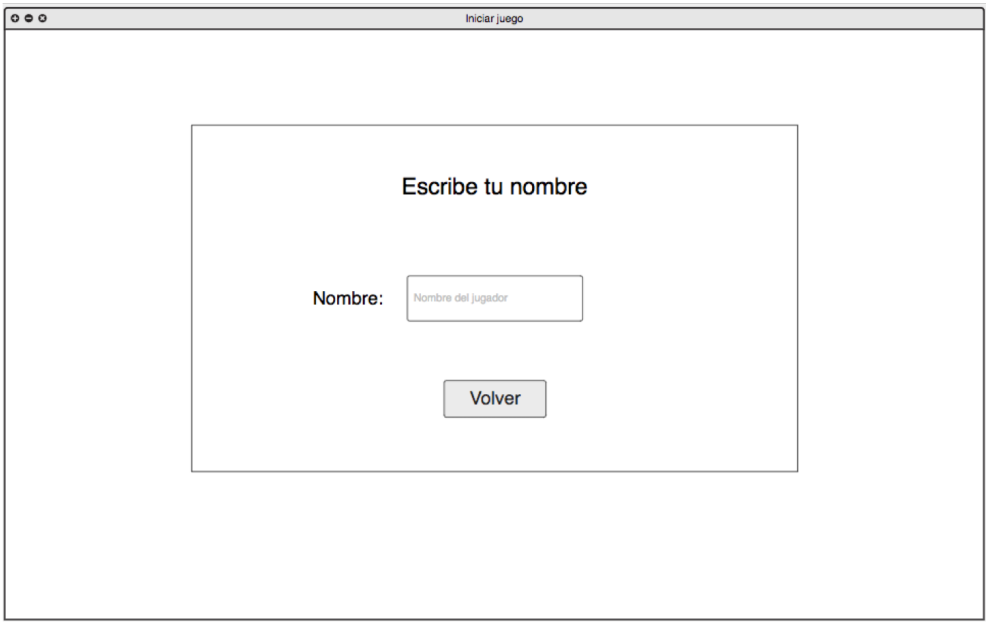


Figura 4.14: Prototipo. Solicitud de nombre a jugador.

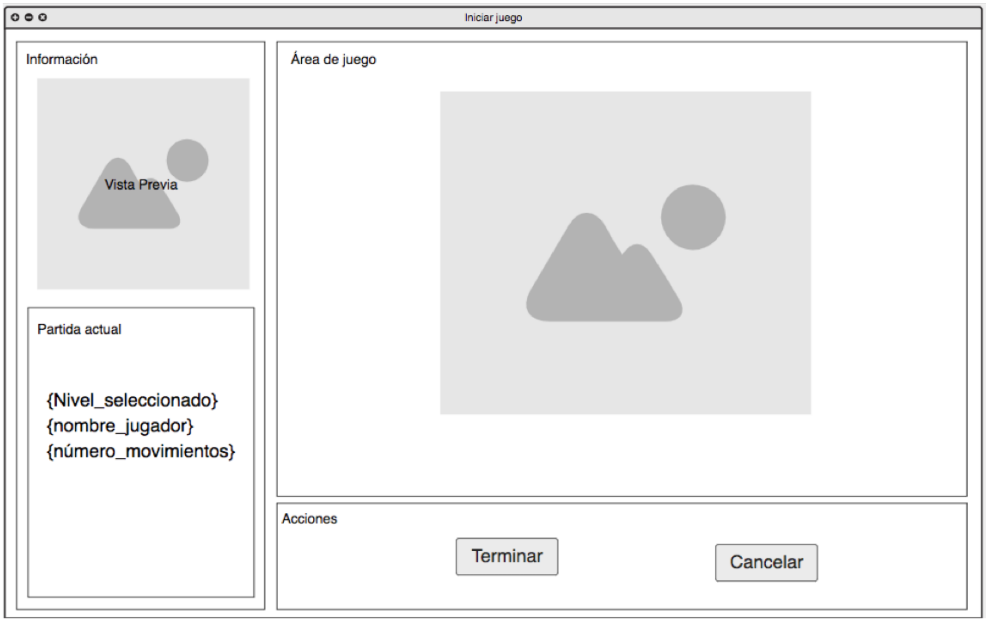


Figura 4.15: Prototipo. Juego inicia.

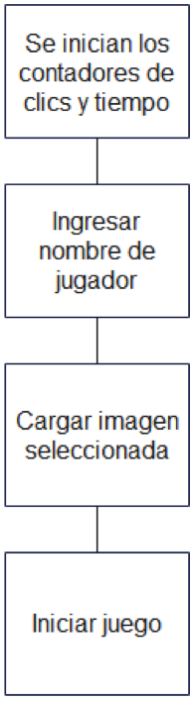


Figura 4.16: Diagrama de estructura de *Jackson* para el *CU Iniciar juego*.

Figura 4.17: Interfaz de usuario para la solicitud del nombre de jugador.

Figura 4.18: Interfaz de usuario de la implementación del *CU Iniciar juego* en el nivel fácil.

4.1.8. CU Terminar juego

El *CU Terminar juego* tiene lugar cuando el rompecabezas está completo. A través del presente CU mediante la transacción Registra datos de puntuaciones se lanza el aspecto. Al concluir el rompecabezas se muestra la puntuación de la partida actual.

En la Tabla 4.7 se muestra la descripción de CU. En la Tabla 4.8 se observa la tabulación de ideas. La Figura 4.19 corresponde al DFD. La Figura 4.20 considera el prototipo. En la Figura 4.21 se observa el diagrama de estructura de *Jackson*.

Tabla 4.7: Descripción de *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

Tabla 4.8: 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			

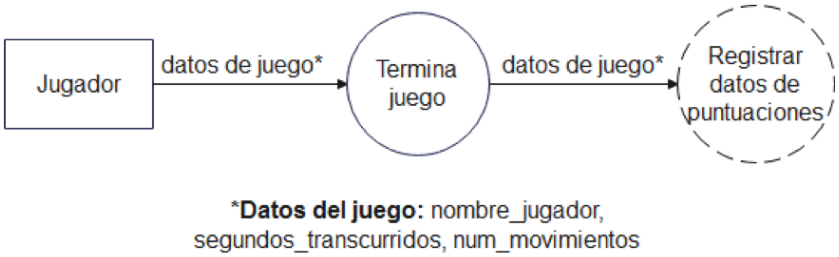


Figura 4.19: DFD del *CU Terminar juego*.

Figura 4.20: Prototipo del *CU Terminar juego*.

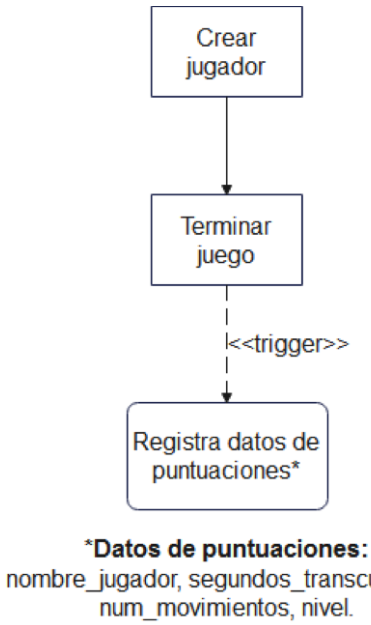


Figura 4.21: Diagrama de estructura de *Jackson* del *CU Terminar juego*.

En el Código 4.9 se realiza la implementación del *CU Terminar juego*. Por medio de la instrucción `Register the score` se realiza el registro de puntuaciones a través de la funcionalidad transversal del aspecto *Registrar datos de puntuaciones*, para consultar el funcionamiento ir a la Sección 4.1.10. En la Figura 4.22 se aprecia la interfaz que muestra el puntaje del juego terminado.

Código 4.9: Implementación del *CU Terminar juego* con Cal-4700.

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

4.1.9. CU Consultar puntuaciones

Para el *CU Consultar puntuaciones* se toma en consideración 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 puntuación: una para la puntuación por número de movimientos y otro para la puntuación por tiempo transcurrido. En la Tabla 4.9 se muestra la descripción del CU. En la Tabla 4.10 se observa la tabulación de ideas. La Figura 4.23 corresponde al DFD. La Figura 4.24 considera el prototipo. Y finalmente, en la Figura 4.25 se observa el diagrama de estructura de *Jackson*.

Figura 4.22: Interfaz de usuario de la implementación del *CU Terminar juego*.

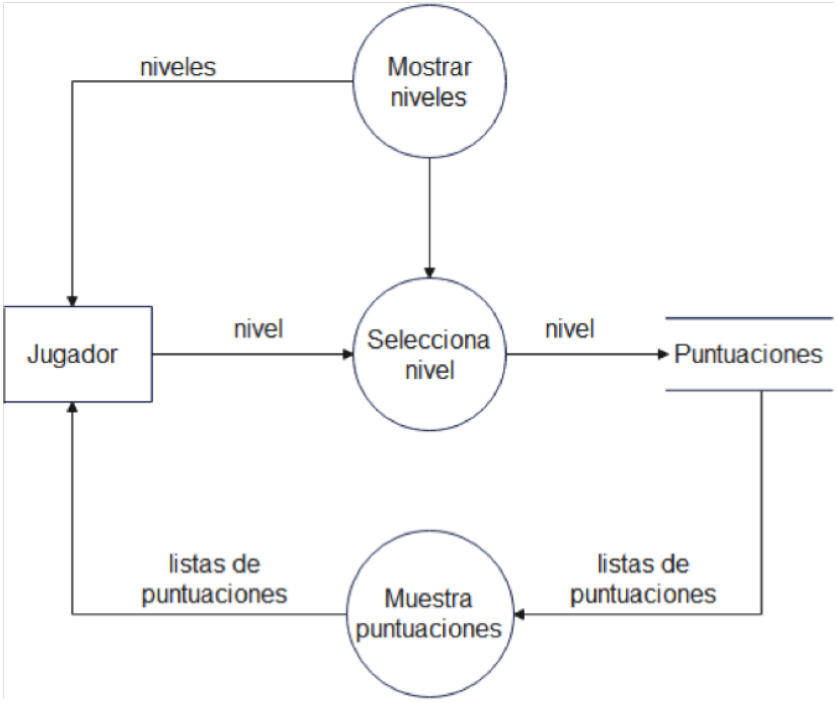


Figura 4.23: DFD del *CU Consultar puntuaciones*.

Tabla 4.9: 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

Tabla 4.10: 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			

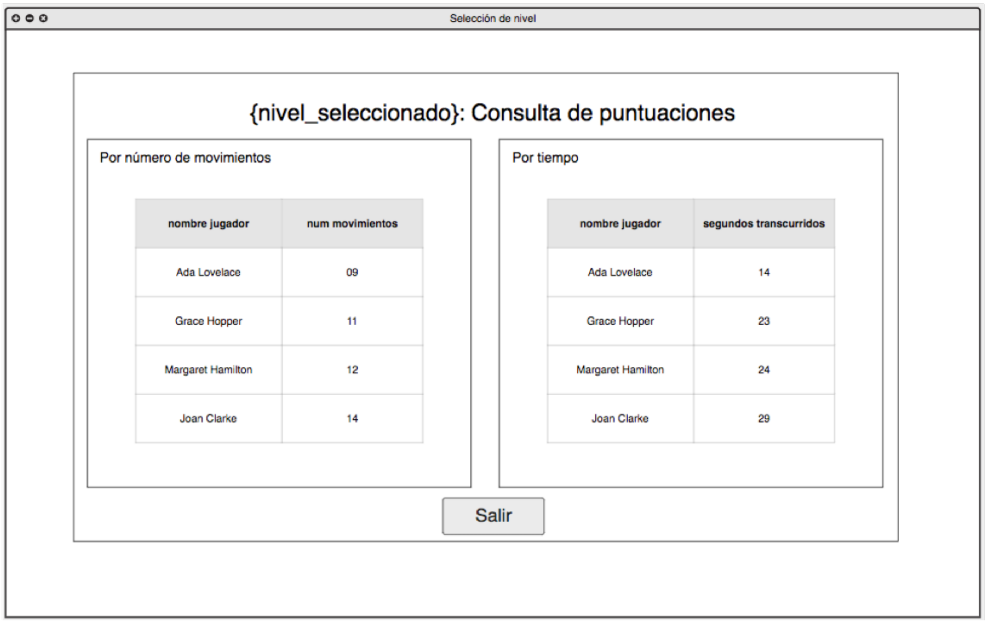


Figura 4.24: Prototipo para el *CU Consultar puntuaciones*.

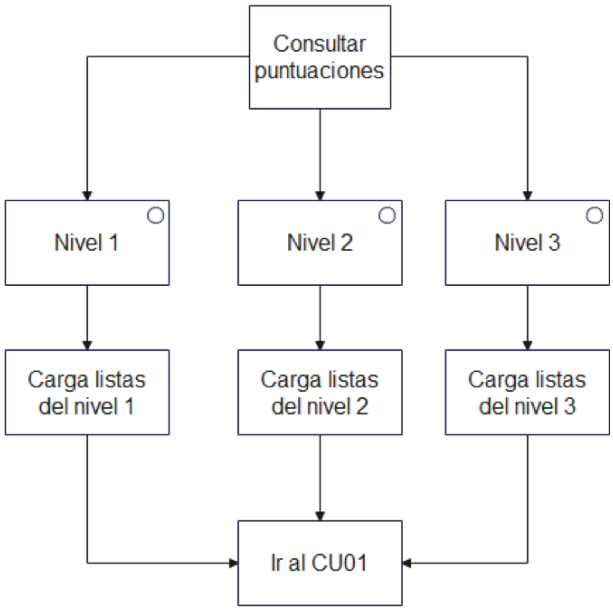


Figura 4.25: Diagrama de estructura de *Jackson* para el *CU Consultar puntuaciones*.

En el Código 4.10 se muestra la implementación del *CU Consultar puntuaciones*. Primero se muestran al jugador los tres niveles de dificultad (Figura 4.26), al elegir uno, el sistema carga las listas de puntuaciones (Código 4.11) y muestra como resultado las dos listas de puntuación: por número de movimientos y por tiempo (Figura 4.27).

Código 4.10: Rutina en Cal-4700 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, put "easy" into
      the score's level's difficulty; check score.
8  If the event's spot is in the medium level button, put "medium"
      into the score's level's difficulty; check score.
9  If the event's spot is in the hard level button, put "hard" into
      the score's level's difficulty; check score.
10 If the event's spot is in the back button, exit.
11 repeat
```

Código 4.11: Rutina en Cal-4700 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", put "C:\
      AplicacionLudica\easyScores.txt"  into a path.
6  if the score's level's difficulty is "medium", put "C:\
      AplicacionLudica\mediumScores.txt.txt"  into the path.
```

```
7  if the score's level's difficulty is "hard", put "C:\
       AplicacionLudica\hardScores.txt"  into the path.
8  read the path into a buffer called original. If the i/o error is
       not blank, debug the i/o error.
9  slap a rider on the original.
10 Loop.
11 get a string from the original given the rider.
12 if the string is blank, break.
13 get a token from the string.
14 if the token is "M", get the token from the string; put the scoreM
       then  the string into the scoreM.
15 if the token is "T", get the token from the string; put the scoreT
       then  the string into the scoreT.
16 repeat.
17 show score list.
```

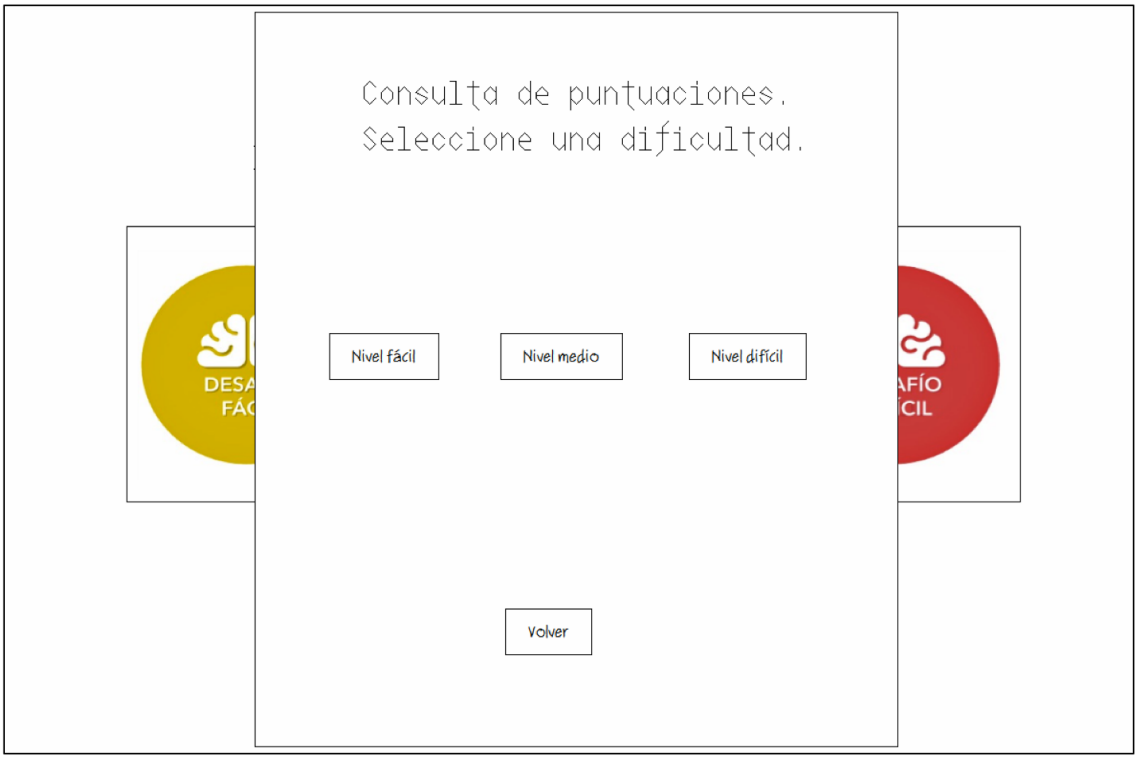


Figura 4.26: Interfaz para la consulta de puntuaciones a través del nivel de dificultad.

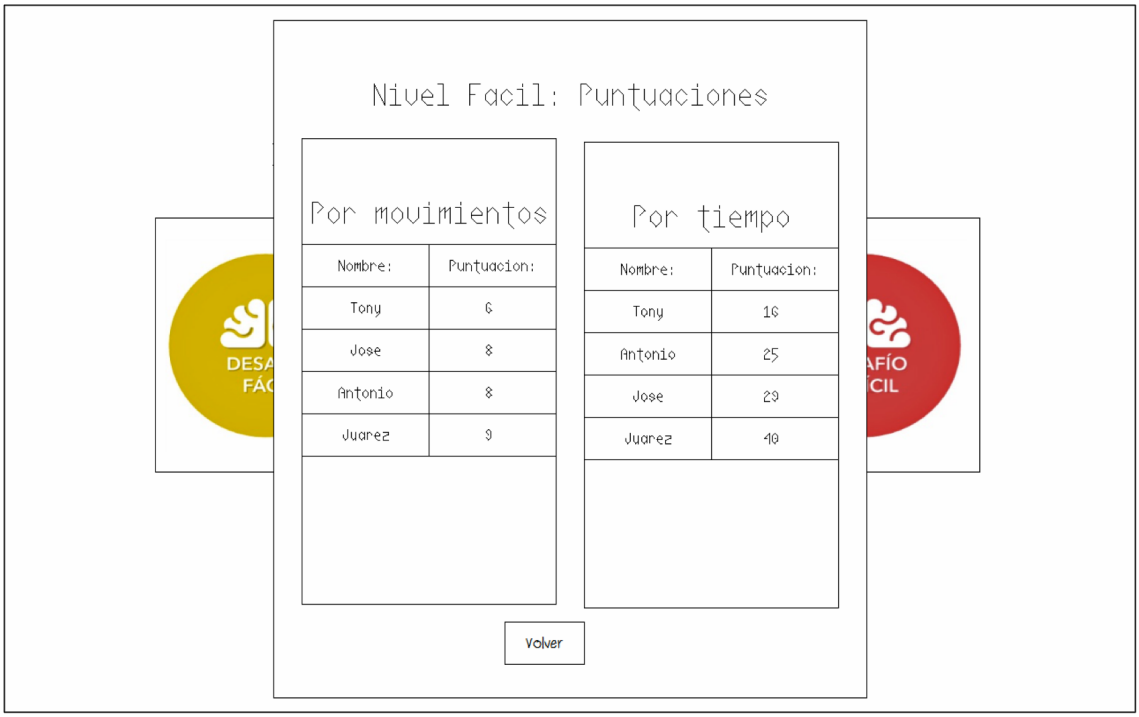


Figura 4.27: Interfaz de usuario que muestra las listas de puntuaciones del nivel fácil.

4.1.10. 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 de 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 4.11 se muestra la descripción del CU con los elementos adecuados al tipo de requerimiento. En la Tabla 4.12 se observa la tabulación de ideas. La Figura 4.28 muestra el DFD correspondiente. Finalmente, la Figura 4.29 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 4.11: 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

Tabla 4.12: 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			

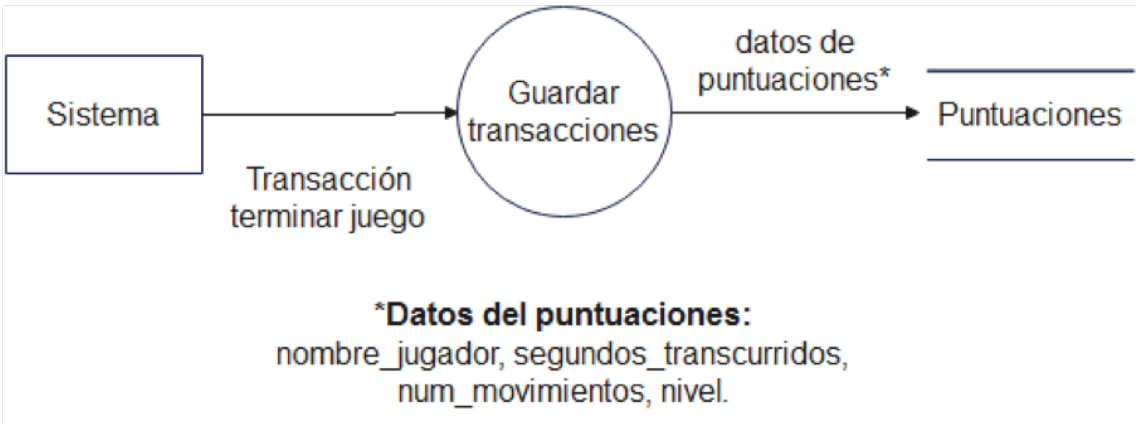


Figura 4.28: DFD del aspecto *Registrar datos de puntuaciones*.

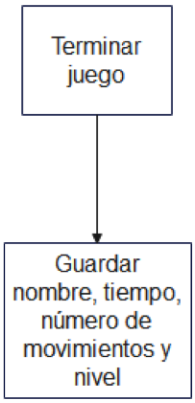


Figura 4.29: Diagrama de estructura de *Jackson* del aspecto *Registrar datos de puntuaciones*.

En el Código 4.12 se muestra la implementación del 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.

Código 4.12: Implementación del 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", put "C:\
       AplicacionLudica\easyScores.txt"  into a path.
5  if the score's level's difficulty is "medium", put "C:\
       AplicacionLudica\mediumScores.txt.txt"  into the path.
6  if the score's level's difficulty is "hard", put "C:\
       AplicacionLudica\hardScores.txt"  into the path.
7  if the path is not in the file system, create the path in the file
        system.
8  read the path into a buffer called original.
9  put the score's criterion then " - " then the  score's player's
       name then " - " then the score's score  into a string called
       auxiliar.
10 write the original then the auxiliar then the crlf string to the
       path.
11 If the i/o error is not blank, debug the i/o error.
```

4.1.11. Persistencia de datos

Cal-4700 maneja un sistema de persistencia basado en archivos de texto plano. Para la presente implementación se organizó la información en tres archivos de texto (*easyScores, mediumScores y hardScores*), cada archivo corresponde a las puntuaciones de un 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 carácter de *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 4.30 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 4.30: Esquema de persistencia de datos del sistema.

4.1.12. Estructura del proyecto en Cal-4700

Los proyectos de Cal-4700 se administran mediante un directorio único dentro del sistema operativo. Para el presente caso de estudio 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. 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 de texto con el código que pone en marcha al sistema. Dentro de él se encuentra la rutina principal `To run` y las rutinas inicialización y finalización de los elementos de la interfaz en memoria. Ver Apéndice A.

- **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. Ver Apéndice B.

- **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. Ver Apéndice C.

- **CU Seleccionar Nivel.** Archivo que contiene las definiciones para el *CU Seleccionar nivel*. Ver en Sección 4.1.5.

- **CU Seleccionar Imagen.** Archivo que considera las definiciones del *CU Seleccionar imagen*. Ver en la Sección 4.1.6 y en el Apéndice D.

- **CU Iniciar Juego.** Contiene las rutinas necesarias para solicitar el nombre al jugador e iniciar el juego. Ver Sección 4.1.7 y el Apéndice E.

- **CU Terminar juego.** Considera la rutina que controla lo que ocurre cuando el *Jugador* termina el rompecabezas. Ver Sección 4.1.8.

- **CU Consultar puntuaciones.** Un archivo que contiene el código que permite al *Jugador* consultar las listas de puntuaciones. Ver Sección 4.1.9.

- **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*. Ver en Sección 4.1.10.

- **The noodle.** Es una biblioteca de Cal-4700 que contiene definiciones de tipos y rutinas para la correcta compilación de los programas.

Capítulo 5

Conclusiones

En el presente capítulo se presentan las conclusiones correspondientes al trabajo de tesis.

5.1. Conclusiones

La programación naturalística busca reducir la brecha entre el dominio del problema y el dominio de la solución incorporando elementos del lenguaje natural en los lenguajes de programación. El caso de estudio abordado muestra que la utilización de un subconjunto controlado del idioma inglés *(plain english)*, como lenguaje de programación, permite alcanzar los objetivos de reducir complejidad y ofrecer mayor expresividad al código fuente. Cal-4700 y su editor son lo suficientemente robustos para aplicaciones pequeñas a medianas de propósito general, tal como se observa en el caso de estudio lúdico.

Usar los lenguajes naturalísticos favorece al crecimiento y fortalecimiento del paradigma naturalístico. Al ponerlo en práctica, se demuestra que existe una forma de analizar, modelar e implementar software usando los elementos del lenguaje natural y sin deformar las ideas originales. Es posible mejorar la documentación, ya que el código se auto-documenta y además, es más comprensible para el lector.

El paradigma naturalístico tiene mucho por avanzar y es imperativo difundirlo con aplicaciones que muestren los beneficios ya citados. Cal-4700 es extensible y se recomienda

estudiar ampliamente el archivo *the noodle* para la definición de rutinas propias y para la definición de rutinas equivalentes o alternativas, lo cual es una posibilidad.

5.2. Trabajo a futuro

Como trabajo a futuro se precisa extender las capacidades del sistema para generar aplicaciones más robustas. Esto es factible de realizar mediante la implementación de nuevas bibliotecas de contexto que permitan mejorar el alcance del lenguaje, lo cual permite Cal-4700. Un ejemplo claro de esto sería implementar una biblioteca para el manejo de persistencia de datos con archivos.

Además, es necesario implementar funciones que permitan generar aplicaciones para el entorno web y móvil, ya que actualmente se carece de esas capacidades.

Se requiere estudiar a detalle la biblioteca *The noodle* ya que en ella se encuentran implementaciones de rutina útiles para generar aplicaciones de escritorio de propósito general con acceso a internet.

Apéndice A

El archivo de inicio

En el Código A.1 se muestra el contenido del archivo de inicio que contiene la interfaz principal y las rutinas de inicialización y finalización de elementos en la memoria.

Código A.1: Contenido del archivo *inicio*. Elaborado con Cal-4700.

```
1  To run:
2  Start up.
3  Initialize our stuff.
4  show levels.
5  Finalize our stuff.
6  Shut down.
7
8  To initialize our stuff:
9  Create the background. Initialize the buttons.
10 initialize the level thumbnails.
11 initialize the name request interface.
12 initialize the text.
13
14 To finalize our stuff:
15 Destroy the background.
```

Apéndice B

El archivo de interfaz

En el Código B.1 se muestra el archivo de *interfaz* que contiene la codificación para la interfaz gráfica de la implementación.

Código B.1: Contenido del archivo *interfaz*. Elaborado con Cal-4700.

```
1        \fondo de pantalla
2  The background is a picture.
3
4  To create the background:
5  Draw the screen's box with the white color [borde] and the white
       color[relleno].
6  refresh the screen.
7  Extract the background given the screen's box.
8
9  \botones
10 A button has a box, a picture and a name.
11 A action button is a button.
12
13 To make a action button given a spot and a name:
14 Put the spot's x minus 1/2 inch minus the name's width into the
       action button's left.
```

```
Put the spot's y minus 1/2 inch into the action button's top.
Put the spot into the action button's right-bottom.
Put the name into the action button's  name.

To make a button given a spot and a name:
Put the spot's x minus 2 inch minus the name's width into the
   button's left.
Put the spot's y minus 3 inch into the button's top.
Put the spot into the button's right-bottom.
put the name into the button's name.

To draw a button:
Fill the button's box with the white color.
if the button's name is "Nivel facil.", create picture level with
   the button and "facil.jpg" .
if the button's name is "Nivel medio.", create picture level with
   the button and "medio.jpg" .
if the button's name is "Nivel dificil.", create picture level
   with the button and "dificil.jpg" .
if the button's name is "Imagen1", create picture level with the
   button and the image list' location one.
if the button's name is "Imagen2", create picture level with the
   button and the image list' location two .
if the button's name is "Imagen3", create picture level with the
   button and the image list' location three.
if the button's name is "Imagen4", create picture level with the
   button and the image list' location four .
Draw the button's name in the button's box with "center".
Draw  the button's box.

```

```
To create picture level given a button and a string:
allocate memory for the button's picture.
fetch the button's picture from the string.
Resize the button's picture to 2-1/2 inches by 2-1/2 inches.
Center the button's picture in the button's box.
draw the button's picture.

To decide if a spot is in a button:  [desisores, rutinas para
   tomar desisiones]
If the spot is in the button's box, say yes.
Say no.

\definicion de botones y ubicaciones
The easy level button is a button.
The medium level button is a button.
The hard level button is a button.
The score button is an action  button.
The quit button is an action button.
the check button is an action button.
the back button is an action button.

\inicializar los botones del CU seleccionar niveles
To initialize the buttons:
Put the screen's top plus 5-1/2 inch into a spot's y.
Put the screen's left plus 4 inch into the spot's x.
make the easy level button given the spot and "Nivel facil.".
Put the medium level button's right plus  7-1/2 inch into the spot
   's x.
make the medium level button given the spot and "Nivel medio.".
Put the hard level button's right plus 11 inch into the spot's x.
```

```
make the hard level button given the spot and "Nivel dificil.".
Put the screen's center-bottom into the spot.
Put the screen's bottom minus 1/2 inch into the spot's y.
Put the screen's left plus 8 inch into the spot's x.
make the quit button given the spot and "Salir".
Put the screen's left plus 5 inch into the spot's x.
make the score button given the spot and "Ver puntuaciones".

\Miniaturas de imagen.
To show level thumbnails given a string:
show the arrow cursor.
Draw the background.
Write the image message given the string.
Draw the picture button one.
Draw the picture button two.
Draw the picture button three.
Draw the picture button four.
Draw the return button.
refresh the screen.

To write the image message given a string:
Use large letters.
Write the string then ": Selecciona una imagen" with the black pen
    at the top of the screen's box.

\definiciones de los botones para miniaturas
The picture button one is a button.
The picture button two is a button.
The picture button three is a button.
The picture button four is a button.
```

```
94  The return button is a action button.
95
96  \inicializar las miniaturas del nivel
97  To initialize the level thumbnails:
98  imagine a box 20 inch by 20 inch.
99  Move the box to the top left corner of the screen's box.
100 Draw and fill the box with the gray color.
101 Put the screen's top plus 5-1/2 inch into a spot's y.
102 Put the screen's left plus 3-1/2 inch into the spot's x.
103 make the picture button one given the spot and "Imagen1".
104 Put the picture button two's right plus 6-1/2 inch into the spot's
        x.
105 make the picture button two given the spot and "Imagen2".
106 Put the picture button three's right plus 9-1/2 inch into the spot
       's x.
107 make the picture button three given the spot and "Imagen3".
108 Put the picture button four's right plus 12-1/2 inch into the spot
       's x.
109 make the picture button four given the spot and "Imagen4".
110 Put the screen's center-bottom into the spot.
111 Put the screen's bottom minus 1/2 inch into the spot's y.
112 Put the screen's left plus 6 inch into the spot's x.
113 make the return button given the spot and "Volver".
114
115 \inicializar la interfaz de solicitud del nombre
116 To initialize the name request interface:
117 imagine the request box 3 inch by 3 inch.
118 put the request box in the center of the screen.
119 Put the request box's top plus 2-2/3 inch into a spot's y.
120 Put the request box's left plus 2 inch into the spot's x.
```

```
make the accept button given the spot and "aceptar".

the accept button is a action button.
the request box is a box.

\muestra la interfaz de solicitud de nombre
to go to the name request interface:
show the arrow cursor.
Fill the request box with the white color.
Draw the request box.
Use small letters.
Write "Escribe tu nombre: " with the black pen at the top of the
   request box.
Draw the accept button.
Draw the text.
Draw the text's box.
refresh the screen.

\para manejar texto en la solicitud del nombre
The text has a box and a string.

To initialize the text:
Put the request box's left plus 1/2 inch into the text's left.
Put the text's left plus 2 inches into the text's right.
Put the request box's top plus 1-1/2 inch into the text's top.
Put the request box's top plus 1 inch into the text's bottom.

To draw the text:
Put the text's string then "_" into a string.
Draw the string in the text's box.
```

Apéndice C

El archivo *puzzle*

En el Código C.1 se muestra el contenido del archivo *puzzle* que contiene las definiciones de rutina para las actividades de creación y manipulación de los rompecabezas.

Código C.1: Contenido del archivo *puzzle*. Elaborado con Cal-4700.

```
1         \declaraciones del rompecabezas
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.
```

```
Put the puzzle's left-top into a spot.
Loop.
Allocate memory for a piece.
Append the piece to the puzzle's pieces.
Put the spot and the spot plus the piece size into the piece's box
    .
Extract the piece's picture using the piece's box.
Add the piece size to the spot's left.
If the spot's left is less than the puzzle's right, repeat.
Put the puzzle's left into the spot's left.
Add the piece size to the spot's top.
If the spot's top is the puzzle's bottom, break.
Repeat.

To splatter some pieces:
Make a main box 10-2/5 inch by 6-4/5 inch.
Move the main box to the right side of the screen's box.
make a work box 3-1/2 inch smaller than the main box.
center the work box in the main box(horizontally).
center the work box in the main box(vertically).
Loop.
Get a piece from the pieces.
If the piece is nil, break.
Pick a spot anywhere in the work box.
Round the spot to the nearest multiple of 1/4 inch.
Center the piece's box on the spot.
Center the piece's picture on the spot.
Repeat.

\imprimir imagen de muestra
```

```
To draw guide image given a puzzle:
Draw the puzzle's picture.

To display a puzzle:
Draw the interface.
Show everything in the puzzle.
Draw guide image with the puzzle.
Loop.
Get a piece from the puzzle's pieces.
If the piece is nil, break.
Draw the piece's picture.
Repeat.
Refresh the screen.

To draw the interface:
Draw the background.
imagine a box 3 inch by 10 inch.
Move the box to the top left corner of the screen's box.
Draw and fill the box with the gray color.
imagine a second box 15 inch by 1 inch.
move the second box to the bottom of the screen's box.
Draw and fill the second box with the gray color.
imagine a third box 2-1/2 inch by 3 inch.
center the third box in the box(vertically).
center the third box in the box(horizontally).
Draw and fill the third box with the white color.
\botones
Put the screen's center-bottom into a spot.
Put the screen's bottom minus 1/4 inch into the spot's y.
Put the screen's left plus 7 inch into the spot's x.
```

```
make the check button given the spot and "Verificar".
Put the screen's left plus 9 inch into the spot's x.
make the  back button given the spot and "Cancelar".

To track the mouse on a puzzle:
Imagine a big box 10-1/5 inch by 7 inch.
Move the big box to the right side of the screen's box.
Find a piece of the puzzle given the mouse's spot.
If the piece is nil, exit.
Loop.
If the mouse's left button is up, exit.
Put the mouse's spot into a spot.
Round the spot to the nearest multiple of 1/4 inch.
Center the piece's box on the spot.
Center the piece's picture on the spot.
Display the puzzle.
if the piece's box is not in the big box, exit.
Repeat.

To find a piece of a puzzle given a spot:
Get the piece from the puzzle's pieces (backwards).
If the piece is nil, exit.
If the spot is not in the piece's box, repeat.

To fetch a picture from a path:
Read the path into a buffer.
Create the picture from the buffer.
```

Apéndice D

El archivo CU Seleccionar imágenes

En el Código D.1 se muestra el contenido del archivo *CU Seleccionar Imagen* que considera las definiciones de rutina que muestran al jugador las imágenes disponibles por nivel de dificultad. Se consideran las rutinas del nivel medio y difícil.

Código D.1: Contenido del archivo *CU Seleccionar imagen*. Elaborado con Cal-4700.

```
1   \Definiciones de imagen.
2   the image list has a location one string, a location two string, a
        location three string and a location four string.
3   the image list' location one is "cat1.jpeg".
4   the image list' location two is "torreEifel.jpg".
5   the image list' location three is "nocheEstrellada.jpg".
6   the image list' location four is "mar2.jpg".
7
8   \Nivel medio
9   to go to level medium:
10  show level thumbnails given the medium level' difficulty .
11  Loop.
12  Deque an event.
13  If the event is nil, break.
14  If the event's kind is not "left click", repeat.
```

```
15 If the event's spot is in the picture button one, go to medium
      difficulty puzzle given the image list' location one.
16 If the event's spot is in the picture button two,  go to medium
      difficulty puzzle given the image list' location two.
17 If the event's spot is in the picture button three,  go to medium
      difficulty puzzle given the image list' location three.
18 If the event's spot is in the picture button four, go to medium
      difficulty puzzle given the image list' location four.
19 If the event's spot is in the return button, show levels; exit.
20 repeat.
21
22 \Nivel dificil.
23 to go to level hard:
24 show level thumbnails given the hard level' difficulty .
25 Loop.
26 Deque an event.
27 If the event is nil, break.
28 If the event's kind is not "left click", repeat.
29 If the event's spot is in the picture button one, go to hard
      difficulty puzzle given the image list' location one.
30 If the event's spot is in the picture button two, go to hard
      difficulty puzzle given the image list' location two.
31 If the event's spot is in the picture button three, go to hard
      difficulty puzzle given the image list' location three.
32 If the event's spot is in the picture button four, go to hard
      difficulty puzzle given the image list' location four.
33 If the event's spot is in the return button, show levels; exit.
34        repeat.
```

Apéndice E

El archivo CU Iniciar juego

Rutinas que inician el rompecabezas para los niveles de dificultad medio y difícil. En el Código E.1 se presenta el contenido de archivo *CU Iniciar Juego* que considera las definiciones de rutina para administrar lo que ocurre cuando se inicia un rompecabezas con la dificultad media y difícil.

Código E.1: Contenido del archivo *CU Iniciar juego*. Elaborado con Cal-4700.

```
\Nivel medio
To go to medium difficulty puzzle given a location:
Restart counters.
Request player name.
Create a puzzle from the location with the medium level' piece
   size.
Display the puzzle.
Loop.
Deque an event.
If the event is nil, break.
If the event's kind is not "left click", repeat.
If the event's spot is in the check button, check results.
If the event's spot is in the back button, quit; go to level
   medium; exit.
```

```
13 Track the mouse on the puzzle.
14 Repeat.
15 Destroy the puzzle.
16
17 \Nivel dificil
18 Ro go to hard difficulty puzzle given a location:
19 Restart counters.
20 Request player name.
21 Create a puzzle from the location with the hard level' piece size.
22 Display the puzzle.
23 Loop.
24 Deque an event.
25 If the event is nil, break.
26 If the event's kind is not "left click", repeat.
27 If the event's spot is in the check button, check results.
28 If the event's spot is in the back button, quit; go to level hard;
       exit.
29 Track the mouse on the puzzle.
30 Repeat.
31 Destroy the puzzle.
```

Bibliografía

[1] "Cambridge Dictionary : Cambridge University Press," [En línea]. Disponible en: https://dictionary.cambridge.org/es/. [Accedido: 30-jun-2022].

[2] O. Pulido Prieto and U. Juarez Martinez, "Naturalistic programming: Model and implementation," vol. 18, no. 7, pp. 1230–1237, 2019. DOI: 10.1109/TLA.2020.9099764.

[3] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr, "Beyond aop: toward naturalistic programming," *ACM Sigplan Notices*, vol. 38, no. 12, pp. 34–43, 2003. DOI: 10.1145/966051.966058.

[4] R. Knöll and M. Mezini, "Pegasus: first steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 542–559, 2006. DOI: 10.1145/1176617.1176628.

[5] L.-M. Alducin-Francisco, U. Juarez-Martinez, S. G. Pelaez-Camarena, L. Rodriguez-Mazahua, M.-A. Abud-Figueroa, and O. Pulido-Prieto, "Perspectives for software development using the naturalistic language sn," in *2019 8th International Conference on Software Process Improvement (CIMPS)*, pp. 1–11, IEEE, 2019. DOI: 10.1109/CIMPS49236.2019.9082428.

[6] W. Serrano Gómez, "¿Qué constituye a los lenguajes natural y matemático?," *Sapiens*, vol. 6, no. 1, pp. 47–60, 2005.

[7] "Diccionario de la Real Academia Española: Asociación de Academias de la Lengua Española," [En línea]. Disponible en: https://dle.rae.es/. [Accedido: 21-mayo-2022].

[8] F. G. Bellas, R. M. Unanue, and V. D. F. Fernández, *Lenguajes de programación y procesadores*. Editorial Centro de Estudios Ramón Areces SA, 2016.

[9] O. Pulido-Prieto and U. Juárez-Martínez, "A survey of naturalistic programming technologies," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–35, 2017. DOI: 10.1145/3109481.

[10] M. T. Espinal, J. Macia, J. Mateu, and J. Quer, *Semántica*. Ediciones AKAL, 2020.

[11] A. Massolo, "Sesgos de razonamiento, lenguajes formales y enseñanza de la lógica," *Límite (Arica)*, vol. 14, pp. 0–0, 2019. DOI: 10.4067/s0718-50652019000100210.

[12] U. Juarez Martinez, *Programación Orientada a Aspectos*. Tecnológico Nacional de México, Instituto Tecnológico de Orizaba, primera ed.

[13] O. P. Prieto and U. J. Martínez, "Naturalistic programming: Model and implementation," *IEEE Latin America Transactions*, vol. 18, no. 07, pp. 1230–1237, 2020. DOI: 10.1109/TLA.2020.9099764.

[14] R. Knöll, V. Gasiunas, and M. Mezini, "Naturalistic types," in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 33–48, 2011. DOI: 10.1145/2048237.2048243.

[15] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Towards naturalistic programming: mapping language-independent requirements to constrained language specifications," *Science of Computer Programming*, vol. 166, pp. 89–119, 2018. DOI: 10.1016/j.scico.2018.05.006.

[16] L. A. Hernández-González, U. Juárez-Martínez, and L. M. Alducin-Francisco, "Evolution of naturalistic programming: A need," pp. 185–198, 2020. DOI: 10.1007/978.

[17] G. Rzeppa and D. Rzeppa, "The osmosian order of plain english programmers," [En línea]. Disponible en: http://www.osmosian.com/. [Accedido: 30-jun-2022].

[18] "Software edraw," [En línea]. Disponible en: https://www.edrawsoft.com/es/. [Accedido: 20-mayo-2022].

[19] "Balsamiq. rapid, effective and fun wireframing software," [En línea]. Disponible en: https://balsamiq.com/. [Accedido: 20-mayo-2022].

[20] L. A. Hernández González, *Método de desarrollo de software para el paradigma naturalístico. (Tesis en progreso).* PhD thesis, 2022.

[21] T. DeMarco, *Structured Analysis and System Specification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[22] E. Suárez, "Prototipo, contexto e ingeniería del software," *Estudios de Postgrado, en Sistemas de Información*, 2017.

[23] M. Jackson, "The jackson development methods," *Wiley Encyclopaedia of Software Engineering*, 1992.

[24] S. Y. Bermeo Bonete, "Diseño y desarrollo de una aplicación móvil lúdico-interactiva para brindar soporte en el diagnóstico y la intervención de dificultades en la motricidad fina en niños de 3 a 7 años," 2019.