Universidad de las Ciencias Informáticas

Facultad 9

Título: Componente Evaluador de Expresiones Matemáticas.

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas.

Autores:
Cesar Santos Sanabria

Luisdey Colomina Curbelo

Tutor:
Yesnier Bravo García

Dedicatoria

A mis padres y mis hermanos por su confianza, paciencia y entrega absoluta.

Luisdey

A mi abuela Nana, por haber sido la persona más extraordinaria que haya conocido. Por desgracia mi abuela falleció antes de graduarme, pero estoy seguro que hubiese estado muy feliz. Hoy comienza una nueva etapa en mi vida sin ti (simplemente no es lo mismo). Te echo de menos vieja.

Cesar

DECLARACIÓN DE AUTORÍA

Declaramos que somos los únicos autores de este trabajo y autorizamos a la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio. Para que así conste firmamos la presente a los ____ días del mes de ________ del año ________.

______________________________ ______________________________

Firma del Autor
Cesar Santos Sanabria

Firma del Autor
Luisdey Colomina Curbelo

____

Firma del Tutor
Lic. Yesnier Bravo García

Tabla de contenido

Resumen....VI
Introducción. 1
Capitulo 1: Fundamentación Teórica 5
1.1 Introducción.... 5
1.2 Objeto de Estudio 5
1.2.1 Descripción de la situación problemática actual. 5
1.3 Análisis de Otras Soluciones Existentes.... 6
1.4 Lenguajes de Programación.... 8
1.4.1 Definición. 8
1.4.2 Clasificación.... 8
1.4.3 Lenguajes de Dominio Específico. 9
1.4.4 Especificación de un lenguaje. 12
1.5 Compiladores e Intérpretes. 13
1.5.1 Principales definiciones y conceptos. 13
1.5.2 Estructura general de un Intérprete. 16
1.5.3 Etapas de un intérprete 17
1.6 Propuesta de Solución.... 23
1.7 Conclusiones Parciales. 23
Capítulo 2: Tecnologías y tendencias actuales a desarrollar. 25
2.1 Introducción.... 25
2.2 El Lenguaje Unificado de Modelado (UML). 25
2.2.1 Definiendo el lenguaje utilizado para la modelación.... 25
2.3 El proceso unificado de desarrollo de software (RUP) 27
2.3.1 ¿Agile UP? 28
2.3.2 Selección de la metodología de desarrollo. 29
2.4 Selección del Lenguaje de Programación.... 29
2.4.1 Definiendo C++.... 29
2.4.2 Justificación del lenguaje de programación seleccionado.... 30
2.5 NetBeans.... 30
2.5.1 Selección del entorno de desarrollo. 31
2.6 Visual Paradigm.... 31
2.6.1 Selección de la herramienta de Modelado..... 32
2.7 Conclusiones Parciales 32
Capítulo 3: Presentación de la solución propuesta 33
3.1 Introducción.... 33
3.2 Modelo del Dominio 33

3.3 Requerimientos funcionales 35
3.4 Requerimientos no funcionales 35
3.5 Descripción del Sistema Propuesto 36
3.6 Modelo de Casos de uso del Sistema 36
3.6.1 Descripción Textual de los Casos de Uso 37
3.7 Lenguaje Lince 40
3.8 Conclusiones parciales 42
Capítulo 4: Construcción de la solución propuesta 43
Introducción 43
4.1 Arquitectura 43
4.2 Patrones 45
4.2.1 Instancia Única (Singleton) 45
4.2.2 Patrón Fábrica Simple 46
4.2.3 Patrón Visitador (Visitor) 47
4.2.4 Patrón Estrategia (Strategy) 48
4.2.5 Patrón Fachada (Facade) 49
4.3 Diagrama de Paquetes 49
4.4 Diagrama de clases del diseño. 51
4.5 Diagramas de Interacción del Diseño (Secuencia) 56
4.6 Validación del Componente 57
4.7 Conclusiones parciales 60
Capítulo 5: Estudio de la Factibilidad. 61
5.1 Introducción. 61
5.2 Análisis de Puntos de Casos de Uso. 61
5.3 Análisis de costos y beneficios 66
5.4 Conclusiones Parciales 66
Conclusiones Generales 67
Recomendaciones 67
Trabajos citados 68
Bibliografía 71
Anexo 1. Gramática del lenguaje Lince 72

Resumen

El presente trabajo de diploma de la Universidad de las Ciencias Informáticas (UCI), tiene como objetivo el desarrollo de un componente de software que permita la especificación y evaluación de fórmulas matemáticas, debido a la falta de flexibilidad que presentan los sistemas informáticos actuales en la implementación de los modelos matemáticos. El uso de este componente implica que el desarrollo de futuras aplicaciones sea más sencillo y en un menor plazo de tiempo. El mismo constituye una poderosa herramienta para el equipo de desarrollo, delegando la implementación de los modelos matemáticos al componente. Este documento recoge el estudio del estado del arte, así como la investigación realizada. Luego detalla el proceso de desarrollo de software seguido para la creación de la solución y los aspectos técnicos de la creación del mismo.

Palabras Claves: Componente, Compilador, Intérprete, Fórmula, Fases de Compilación, Lexer, Parser, Análisis Lexicográfico, Análisis Sintáctico, Análisis Semántico, Formas Intermedias, Árbol de Sintaxis Abstracta.

Introducción.

Hoy en día las investigaciones no tienen fronteras, el estudio de un tópico específico implica muchas personas de diferentes disciplinas. Un trivial problema del derrame de químicos en un lago involucra ingenieros químicos, biólogos, médicos, economistas, planificadores urbanos, políticos, etc. Solo se halla una solución comprensiva, si todas estas personas, cada una desde su disciplina cooperan tal como si ellos tuviesen un enemigo común.

Una importante herramienta en la investigación de problemas científicos como el planteado anteriormente es el ordenador, que ha tenido un desarrollo espectacular en los últimos años irrumpiendo en todos los campos de la ciencia. En la actualidad es difícil imaginar alguna rama de la ciencia que no use el computador con frecuencia, esto obliga a los científicos de diversos ámbitos a relacionarse y establecer una comunicación con los ordenadores, que trabajan exclusivamente sobre objetos matemáticos realizando diversas operaciones de cálculo sobre los mismos, lo que ha provocado que muchas ciencias que poco tenían que ver con las matemáticas se hayan matematizado enormemente (1).

Todo esto trajo consigo el desarrollo de una rama de la computación conocida como Matemática Computacional que ha obtenido logros significativos, por ejemplo en el año 2002 se obtuvo la descripción completa de la secuencia del genoma humano detrás de la cual se encuentra un algoritmo matemático que redujo drásticamente el tiempo necesario para completar la secuencia, y que fue principalmente desarrollado por el matemático Eugene Myers (1). Otro ejemplo tangible se puede encontrar en la rama de la medicina, la tomografía computarizada y la resonancia magnética que, con la ayuda de los rayos X u otras técnicas, más la potencia de cálculo de las computadoras actuales, son verdaderos artefactos matemáticos donde el problema consiste precisamente en reconstruir una imagen conociendo la atenuación y el ángulo de los rayos. Podemos encontrar centenares de ejemplos con un considerable peso en el avance de diferentes aéreas de la ciencia, y continúan surgiendo muchos más en la actualidad.

La Matemática Computacional es un área importante de la investigación en la ciencia y la ingeniería. Esta es ampliamente implementada en el software de aplicación que se desarrolla para el modelado de problemas y su simulación, este tipo de software está respaldado por un modelo matemático que fue elaborado a priori y puede estar sujeto a

cambios en el futuro, lo que puede provocar un impacto negativo en el producto. Ordinariamente se programa haciendo uso de lenguajes de alto nivel como C#, C++ o Java, entre otros; estos programas son estáticos por naturaleza, esto implica que no pueden ser cambiados en tiempo de ejecución. Consecuentemente el software desarrollado con esta tecnología es poco flexible, existiendo la posibilidad de que ocurra un cambio en el modelo matemático, está como mínimo, sujeto a una re-compilación del mismo, lo cual no es deseable.

A esta problemática anterior, se le adiciona otro aspecto relevante en este medio, la comunicación hombre-máquina. Desde el surgimiento de la computación, este ha sido un obstáculo en el desarrollo de esta ciencia. En el principio constituía una tediosa labor "explicarle" a un ordenador lo que debía hacer para solucionar un problema determinado, este proceso solo podía realizarse por expertos en la materia, cambiando manualmente la memoria del ordenador mediante clavijas e interruptores. Luego de una serie de avances, uno significativo fue la invención de los programas intérpretes, estos permitían a las personas comunicarse con las computadoras utilizando medios distintos a los números en binarios de antaño, y más parecidos al lenguaje natural. En la década del 50 aparece el primer compilador primitivo para un lenguaje llamado FORTRAN (**Fórmula**-**Tran**slator), creado por John Backus con el objetivo de expresar cálculos matemáticos y científicos, en un lenguaje más apetecible sin perder de vista la eficiencia de la máquina. La importancia de los lenguajes de programación se halla, en que un mayor número de personas que no son especialistas en la materia puedan establecer una comunicación con el computador.

Hasta la fecha una tarea importante es el estudio y elaboración de lenguajes cada vez más cercanos al hombre, también conocidos como "Lenguajes de Alto Nivel". Un subconjunto de este son los lenguajes de dominio específico (DSL por sus siglas en ingles), que al ser más expresivos en un dominio en particular permiten expresar los problemas de una forma más viable, sin que sean necesarios vastos conocimientos de programación. Los DSL pueden ser usados con fines diferentes, uno de ellos constituye un esfuerzo por aislar la lógica del negocio aspirando hacer el software más flexible, permitiendo que sus usuarios modifiquen a gusto estos subprogramas que representan especificidades del dominio. Es muy común incluir funcionalidades como estas en aplicaciones donde se realizan cálculos por medio de fórmulas con cierto grado de complejidad. En particular se encuentran aquellas que requieren acceso a campos de diversas fuentes de datos o que utilicen otras fórmulas anteriormente definidas; y pueden

proporcionar como resultado los mismos valores numéricos, booleanos, cadenas de texto o matrices.

Algunas herramientas han facilitado la declaración de tales expresiones en algún lenguaje básico de dominio específico para establecer campos calculados en reportes, pero no como soporte general para la implementación de cualquier componente de software que lo requiera. De esta forma se arrastra muchas veces rigidez en los requerimientos que involucran fórmulas con posibilidad de cambiar en un futuro, lo cual implicaría en muchos casos, como se mencionaba anteriormente, la re-compilación de la solución.

Partiendo de la situación anterior, se ha definido como **problema a resolver:** Carencia de flexibilidad en la implementación de los modelos matemáticos en aplicaciones informáticas.

En vista a resolver el problema descrito se hace necesario: Diseñar e implementar un componente de software que permita la especificación y evaluación de fórmulas matemáticas en los productos de software elaborados, lo cual figura como **objetivo general** de esta investigación.

La investigación estará centralizada fundamentalmente en el Proceso de especificación y evaluación de fórmulas en los sistemas de software, lo cual constituye el **objeto de estudio**.

Además la investigación estará enmarcada en la Modelación e Implementación de componentes para la especificación y evaluación de fórmulas matemáticas en los sistemas de software, representando el **campo de acción**.

Para el cumplimiento del objetivo trazado se conciben las siguientes tareas:

1. Investigar las tecnologías y herramientas existentes en la actualidad para la especificación y evaluación de fórmulas.
2. Confeccionar un lenguaje de dominio específico a partir de la asimilación de las estructuras sintácticas existentes en los lenguajes básicos que soportan las herramientas de propósito específico.
3. Confeccionar un analizador léxico para el lenguaje elaborado.
4. Confeccionar un analizador sintáctico para el lenguaje elaborado.

5. Confeccionar un analizador semántico para el lenguaje elaborado.

6. Confeccionar un intérprete de código intermedio.

7. Implementar las funciones matemáticas predefinidas.

El desarrollo exitoso de las tareas expuestas anteriormente contribuirá al cumplimiento de la **idea a defender** la cual propone que: si se dispone de un componente de software para la especificación y evaluación de fórmulas, se logrará un mayor dinamismo en la implementación de los modelos matemáticos en aplicaciones informáticas.

Para la realización de la investigación se hizo necesario aplicar algunos métodos científicos dentro de los que se incluyen métodos teóricos y empíricos los cuales facilitaron la recopilación de la información necesaria para la modelación del diseño de la solución propuesta. De los teóricos se utilizaron el histórico-lógico, analítico-sintético e Inductivo-deductivo. De los empíricos se utilizó la observación.

El método **Histórico-Lógico** se aplica para analizar a nivel nacional e internacional, las Aplicaciones que de una forma u otra han usado intérpretes de código empotrado y observar sus características y comportamiento, así como investigaciones realizadas anteriormente sobre el tema.

El método **Analítico-Sintético**, se define con el objetivo de modelar el funcionamiento de los procesos de compilación e interpretación de lenguajes matemáticos, así como obtener, resumir y describir los elementos más importantes relacionados con estos procesos para construir el Modelo de Dominio y realizar el Modelo de Diseño del sistema propuesto.

El método **Inductivo-Deductivo**, se aplica en la revisión y justificación del modelo y la tecnología para diseñar intérpretes. Mediante el análisis de casos aislados se alcanzarán proposiciones generales, que luego serán empleadas en la inferencia de la tecnología y el modelo más adecuado a emplear en el diseño.

Capitulo 1: Fundamentación Teórica

"La práctica debe siempre ser edificada sobre la buena teoría."
Leonardo Da Vinci

1.1 Introducción.

En este capítulo se abordarán conceptos asociados al dominio del problema, que resultan necesarios para una mejor comprensión del tema a desarrollar. Entre los cuales figuran las principales cuestiones y elementos teóricos en la elaboración de lenguajes de alto nivel e intérpretes de código, sus tendencias actuales, y un análisis de algunos componentes de software más importantes relacionados con este campo.

1.2 Objeto de Estudio

El objeto de estudio es la parte de la realidad objetiva sobre la cual actúa el sujeto, tanto desde el punto de vista práctico como teórico, con vista a la solución del problema planteado (2).

1.2.1 Descripción de la situación problemática actual.

Actualmente la informática es una herramienta de ayuda en casi todos los campos que el hombre conoce. La industria del software ha evolucionado para cubrir estas necesidades creando nuevas técnicas y metodologías para desarrollar sus productos en un intento de hacer el software más robusto, flexible y con un menor costo. Un aspecto relevante en este ámbito es el uso de modelos matemáticos en el software empresarial, estos tienen un gran impacto en el mismo, debido a que son dinámicos y están a expensas de cambios en cualquier momento. Sin embargo, hoy en día la mayor parte de los programas tienen un carácter "singular", o sea, que no están preparados para los posibles cambios. Algunos pasos a seguir por un desarrollador para resolver un problema en concreto son los siguientes:

- Definir el problema.
- Plantear el sistema de ecuaciones o fórmulas (modelo matemático).
- Desarrollar un algoritmo (procedimiento de cálculo) para su resolución.
- Desarrollar el software (teniendo en cuenta lo anterior).

Este es un procedimiento largo y complejo y en la mayoría de los casos requiere de especialistas en el sector para la elaboración del problema, y especialistas en informática

para la elaboración del software. Esto es difícil para ambos tipos de especialistas ya que deben ponerse uno a nivel del otro para poder comunicarse. Además de esto el software como se ha mencionado anteriormente, debido a la naturaleza de los lenguajes de programación y las herramientas utilizadas, es **estático**, lo que significa que un mínimo cambio en el modelo matemático tiene un impacto significativo en él.

Por otra parte existen técnicas más avanzadas que están en desarrollo, estas se fundamentan en aislar las funciones y algoritmos que se presentan repetidamente o que son específicos del negocio en cuestión. De esta forma se obtiene una separación entre el software y el modelo matemático, lo que permite que este último sea introducido o cambiado por los usuarios de la aplicación sin que tenga un impacto en el software. Todo esto se comprende en lo definido en esta investigación como “Proceso de especificación y evaluación de fórmulas en los sistemas de software”. Las técnicas y métodos usados en este proceso no son nuevas, ya existen algunos trabajos sobre estos temas, pero de forma separada y no con los mismos fines que los ya propuestos aquí. De forma general, todas las tareas de este proceso se pueden agrupar en dos grandes tareas, para una mejor comprensión del mismo:

- Especificación de fórmulas
- Evaluación de fórmulas

La primera tarea propone un conjunto de sub-tareas para lograr como objetivo un lenguaje de especificación de fórmulas, así como las herramientas relacionadas con la edición de código escrito en dicho lenguaje. Esta parte está relacionada con los lenguajes de programación, su diseño y especificación. Más adelante en este capítulo se pueden encontrar elementos teóricos que reflejan las tendencias actuales en este campo.

La segunda sub-tarea está relacionada con la elaboración de un evaluador para las fórmulas propuestas en el paso anterior, lo que desde el punto de vista informático está estrechamente relacionado con la teoría de compiladores e intérpretes de código.

1.3 Análisis de Otras Soluciones Existentes

En la actualidad existen otros componentes de software y librerías que realizan el proceso de evaluación de fórmulas. Entre los cuales podemos encontrar:

“#Calculation Component”, este es un motor de cálculo. Este componente ActiveX integra análisis de la expresión y su evaluación. Es útil en dos áreas principales: la primera,

cuando una fórmula tiene que ser definida y evaluada en tiempo de ejecución (por ejemplo, cuando al usuario final se le permite introducir una fórmula que será evaluada y usada); la segunda, cuando se define un juego de fórmulas que dependen una de otra, y son configuradas y evaluadas en tiempo de ejecución. Soporta matemática convencional, cadenas, fecha y hora, operadores y funciones lógicas y es apropiado para procesamiento numérico pesado. También permite definición de variables, operaciones Matriciales y de Arreglos (tal como MATLAB), comentarios, diferentes sistemas numéricos y funciones personalizadas. (3)

Este componente a pesar de estar concurrido de funcionalidades tiene varias desventajas. Entre ellas se encuentra que es un software privativo, cuenta con muy poca documentación y no tiene soporte para la interacción con fuentes de datos lo cual constituye uno de los objetivos fundamentales de esta investigación.

Existen versiones de los intérpretes de Python y Ruby que pueden ser embebidos en software de aplicación, estos lenguajes son muy potentes y tienen un formidable conjunto de funcionalidades que pueden ser usadas, pero como todos los lenguajes de propósito general, la curva de aprendizaje es muy grande para personas con pocos conocimientos de programación. Además estos intérpretes fueron hechos sobre el lenguaje C/C++ y solo pueden ser embebidos en aplicaciones que sean programadas en este lenguaje. Aunque en la actualidad se está trabajando en implementar el intérprete de Ruby, en otros lenguajes como Java[1] no son estables aún, lo mismo sucede con el intérprete de Python. También se hizo un análisis de algunos productos de software de tipo asistente matemático, como Matlab, Genius, Maxima, etc.

El software analizado tiene funcionalidades atractivas en vistas del diseño de un evaluador de fórmulas matemáticas. Haciendo un análisis de los servicios que presta se puede establecer las bases para el diseño del sistema, de manera que sea conveniente, un lenguaje con una pequeña curva de aprendizaje, y un intérprete sobre plataformas libres y de fácil implementación para un conjunto de lenguajes.

[1] JRuby es una implementación de Ruby hecha en Java 100%. También funciona como lenguaje embebido dentro de la máquina virtual de Java para más información visite el sitio oficial http://jruby.org/.

1.4 Lenguajes de Programación

Los lenguajes de programación, igualmente que el lenguaje natural son el instrumento para la comunicación entre seres humanos, son los que permiten establecer una comunicación hombre-máquina. Desde el surgimiento de los mismos han evolucionado extraordinariamente, quedando cerca del lenguaje natural. El objetivo de estos es la construcción de programas de computadoras que generalmente son escritos por humanos, por lo que deben ser comprendidos tanto por los hombres como por las máquinas.

1.4.1 Definición.

Los lenguajes de programación son una solución de compromiso entre las necesidades del programador y la máquina. De esa forma, las declaraciones de los diferentes elementos que componen un programa, ya sean tipos, variables, nombres simbólicos, etc. son concesiones de los diseñadores de lenguajes para que los humanos podamos entender mejor lo que se ha escrito.

De forma general la definición de **"Lenguaje de Programación"** es la siguiente:

- Conjunto de símbolos y reglas que permiten la comunicación con un computador.

En computación, un lenguaje de programación es cualquier lenguaje artificial, el cual, se utiliza para definir adecuadamente una secuencia de instrucciones que puedan ser interpretadas y ejecutadas en una computadora. Se asume que las instrucciones así escritas son traducidas luego, a un código que la máquina pueda "comprender". Los lenguajes de programación intentan conservar una similitud con el lenguaje humano, con la finalidad de que sean más naturales a quienes los usan. Establecen un conjunto de reglas sintácticas y semánticas, las cuales rigen la estructura del programa de computación que se escribe o edita. De esta forma, permiten a los programadores o desarrolladores, poder especificar de forma precisa los datos sobre los que se va a actuar, su almacenamiento, transmisión y demás acciones a realizar bajo las distintas circunstancias consideradas. (4)

1.4.2 Clasificación.

Los lenguajes de programación pueden clasificarse atendiendo a varios puntos de vista. Comúnmente se suele hablar de niveles y generaciones. Respecto a los niveles los lenguajes se clasifican en: **Alto nivel** y **Bajo nivel**, aunque algunos consideran la

existencia de un tercer nivel, conocido como "lenguajes de Medio Nivel". Los de bajo nivel son los que emplean las computadoras, estos se encuentran a nivel de circuitos y microprocesadores que solo son capaces de operar señales electrónicas binarias, mientras que los de alto nivel son los más parecidos al lenguaje natural de los humanos, existe un alto nivel de abstracción entre lo que se pide a la computadora y lo que realmente comprende, debido a esto las instrucciones son independientes de la máquina. Un programa escrito en un lenguaje de alto nivel, debe ser compilado o interpretado para traducir su código, en otro de bajo nivel (lenguaje máquina).

Igualmente que las computadoras (hardware), el software ha evolucionado en generaciones, siendo de mayor interés para esta investigación los de "**Cuarta generación".** Los lenguajes de cuarta generación (4GL) también conocidos como Lenguajes de propósito especial, son usados en programación de propósitos específicos, caracterizados por una mayor facilidad de uso comparado con los de la tercera generación (de propósito general). Teniendo como inconveniente la pérdida de flexibilidad de la que se disponía en lenguajes anteriores. (5)

1.4.3 Lenguajes de Dominio Específico.

En los últimos años se ha popularizado el término de "Lenguaje de Dominio Específico" (DSL, Domain Specific Language) en la industria del software para indicar a un lenguaje de programación o especificación dedicado a un dominio particular o una técnica particular de solución de un problema. Este concepto no es algo nuevo, pero se ha acentuado su uso últimamente por las ventajas que proporciona. Los lenguajes de dominio específico pertenecen a la cuarta generación (4GL), y sus usos conocidos hasta el momento son extensos, desde fórmulas y macros en hojas de cálculo, lenguaje de procesamiento de sonidos como el Csound[2], hasta lenguajes de especificación de gramáticas para la creación de compiladores e intérpretes como el YACC[3].

De forma concreta podemos definir un lenguaje de dominio específico como: "Un lenguaje de programación o lenguaje de especificación ejecutable que ofrece potencia expresiva

[2] **Csound** es un paquete de software orientado a crear, editar, analizar y componer música y sonido. También es llamado así el propio lenguaje de programación que se usa para controlar el software.

[3] **Yacc** es un programa para generar analizadores sintácticos. Las siglas del nombre significan *Yet Another Compiler-Compiler*, es decir, "Otro generador de compiladores más".

enfocada y restringida a un dominio de problema concreto" (6). Entre las principales características de los DSL se pueden encontrar:

- Generalmente estos lenguajes son pequeños y ofrecen un restringido conjunto de notaciones y abstracciones, aunque en ocasiones, pueden contener un completo sub-lenguaje de propósito general.
- Suelen ser lenguajes declarativos, pudiendo considerarse lenguajes de especificación, además de lenguajes de programación.
- Un objetivo común de muchos de estos lenguajes es la programación realizada por el usuario final, que ocurre cuando son los usuarios finales los que se encargan de desarrollar sus programas. Estos usuarios no suelen tener grandes conocimientos informáticos pero pueden realizar tareas de programación en dominios concretos con un vocabulario cercano a su especialización.

Los DSL son lenguajes con objetivos específicos, tanto en su diseño como en su implementación. Estos pueden ser tanto un lenguaje de diagramación visual (como el creado por Generic Eclipse Modeling System, para más detalles ver Anexo 1), o lenguajes textuales, como el AWK[4] que es un lenguaje de búsqueda y procesamiento de patrones, que está especialmente elaborado para trabajar con archivos estructurados y patrones de texto.

Existen varias técnicas para desarrollar lenguajes de dominio específico. La primera opción podría ser la elaboración de un lenguaje DSL independiente y procesarlo como cualquier lenguaje de programación ordinario, esta opción requiere del diseño completo del lenguaje y la implementación de un intérprete o un compilador siguiendo las técnicas tradicionales. Otra posibilidad es empotrar el lenguaje específico en otro de propósito general. Mediante el pre-procesamiento o proceso de macros, se empotra un lenguaje de propósito específico en otro lenguaje incluyendo una fase intermedia que convierte el lenguaje específico en el lenguaje anfitrión. Por ejemplo, el preprocesador de C++ puede utilizarse para crear un completo lenguaje específico para tareas concretas. Finalmente, otra opción podría ser desarrollar un intérprete extensible que incluya elementos que permitan modificar el intérprete, para que analice lenguajes diferentes.

[4] Este lenguaje, por sus ventajas, y la elevada cantidad de aplicaciones es considerado por algunos como un lenguaje de propósito general.

Los lenguajes de dominio específico son herramientas con un propósito limitado. El objetivo de estos es suministrar una forma de comunicación más clara con una parte del sistema. Los DSL tienen el potencial de obtener ciertos beneficios así como desventajas. Cuando se está considerando el uso de uno, debe hacerse un análisis de estos para obtener una decisión acertada sobre su uso, algunas de las cuestiones a analizar podrían ser (7):

Ventajas:

- Los DSL permiten expresar soluciones usando los términos y el nivel de abstracción apropiado para el dominio del problema. En consecuencia, los mismos expertos del dominio pueden comprender, validar, modificar y a menudo desarrollar programas en DSL.
- Es código auto-documentado.
- Los DSL mejoran la calidad, productividad, confianza, portabilidad y reusabilidad de las aplicaciones.
- Los DSL permiten validaciones a nivel del dominio. Mientras las construcciones del lenguaje estén correctas, cualquier sentencia escrita puede considerarse correcta.

Desventajas:

- El costo de aprender un nuevo lenguaje contra su aplicación limitada.
- El costo de diseñar, implementar y mantener un DSL y las herramientas para trabajar con él.
- Encontrar, establecer y mantener el alcance adecuado.
- Dificultad para el balance de las ventajas y desventajas entre las construcciones de los DSL y de los lenguajes de propósito general.
- Potencial pérdida de eficiencia y rendimiento en comparación con el software escrito "a mano".

La creación de los lenguajes de dominio específico (DSL) difiere de manera fundamental a la de los lenguajes de programación tradicionales. Aunque la idea de dominio en determinados idiomas es más que maduro, su papel en la arquitectura, en el diseño e implementación de sistemas de software ha sido reconocido recientemente. Como se mencionaba anteriormente estos se encuentran clasificados dentro de los lenguajes de 4ta generación, que suelen combinar elementos procedimentales con elementos

declarativos, herramientas o interfaces visuales muy fáciles de utilizar, siguiendo el estilo de diálogos con ventanas, iconos y ratón, puesto de moda por las aplicaciones Windows. No sólo son útiles a los usuarios no informáticos, sino que facilitan mucho el trabajo a los usuarios informáticos, permitiendo la elaboración de aplicaciones más robustas, flexibles y de mayor satisfacción para los clientes.

1.4.4 Especificación de un lenguaje.

En la definición de lenguaje que se indicaba anteriormente se decía que un lenguaje es un conjunto de símbolos y reglas, estas reglas también conocidas como “reglas Sintácticas” y/o “reglas semánticas”, son las que determinan el lenguaje. La sintaxis de un lenguaje de programación es el conjunto de reglas formales que especifican la estructura de los programas pertenecientes a dicho lenguaje, mientras que la semántica es el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente válida (8 pág. 1).

Es posible hacer una descripción informal de un lenguaje de programación mediante el lenguaje natural. Esto hace que la especificación sea evidente para cualquier persona, aunque la experiencia dice que es una tarea muy compleja, si no imposible, el describir todas las características de un lenguaje de programación de un modo preciso usando el lenguaje natural (8 pág. 5). Por eso se han elaborado formas rigurosas de confeccionar la especificación de un lenguaje, tanto para la sintaxis como para la semántica.

La especificación formal de un lenguaje desde el punto de vista sintáctico se suele llevar a cabo mediante la descripción estándar de su gramática en notación BNF (Backus-Naur-Form). La misma está compuesta por:

Terminales: Símbolos básicos con los que se forman las cadenas. También son conocidos como componentes léxicos, ejemplo **if**, **then**, **while**.

No Terminales: Son variables sintácticas que denotan conjuntos de cadenas.

Símbolo Inicial: Es un no terminal, y el conjunto de cadenas que representa es el lenguaje definido por la gramática.

Producciones: Las producciones especifican como pueden combinarse los terminales y no terminales para formar cadenas.

Por ejemplo la gramática siguiente define expresiones aritméticas simples.

```
(1)   expresion     →   expresion   +   termino
(2)                 |   expresion   -   termino
(3)                 |   termino
(4)   termino       →   termino   *   factor
(5)                 |   termino   /   factor
(6)                 |   factor
(7)   factor        →   -   factor
(8)                 |   (   expresion   )
(9)                 |   CTE_ENTERA
```

Figura 1 Notación Backus Naur Form (BNF)

En el caso de la especificación semántica de un lenguaje no existe ningún método estándar globalmente extendido y cada uno la describe desde diferentes puntos de vista. Entre los principales métodos se encuentran: la Semántica operacional, Semántica denotacional, Semántica axiomática, Semántica algebraica, entre otras. Además de estos existe otro método de especificar la semántica de un lenguaje mediante el uso de **Gramáticas Atribuidas**. Las gramáticas atribuidas asignan atributos a las construcciones sintácticas del lenguaje, que describen información semántica, por ejemplo el tipo de una expresión, aunque pueden emplearse también para representar cualquier otra propiedad como la evaluación de una expresión o incluso su traducción a una determinada plataforma. Al no estar directamente ligadas al comportamiento dinámico (en ejecución) de los programas, no suelen clasificarse como otro tipo de especificación formal de semántica de lenguajes. Sin embargo, su uso tan versátil hace que estén, de un modo directo o indirecto, en cualquier implementación de un procesador de lenguajes (8 pág. 8).

1.5 Compiladores e Intérpretes.

1.5.1 Principales definiciones y conceptos.

En este sub-epígrafe se abordan las principales definiciones de los “elementos” que se usan en la teoría de compilación. Estos generalmente están relacionados fuertemente unos con otros, aunque existen diferencias sustanciales entre los mismos que se necesitan conocer, ya que estos en algunos casos son tan parecidos que tienden a confundirse unos con otros. Algunas de estas definiciones son:

Concepto de traductor: Un traductor se define como un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino, produciendo, si cabe, mensajes de error (9 pág. 2).

Figura 2: Representación de Traductor de código.

La figura 2 muestra de forma general un esquema básico de lo que constituye un traductor. Un traductor es un concepto general, que engloba tanto a compiladores como intérpretes, donde el "Programa de salida" en los primeros suele ser código de máquina, mientras que en los segundos está constituido por una serie de acciones atómicas que serán ejecutadas posteriormente.

Concepto de Compilador: Es aquel **traductor** que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, realiza una traducción de un código de alto nivel a código máquina (también se entiende por compilador aquel programa que proporciona un fichero objeto en lugar del ejecutable final) (9 pág. 3).

Concepto de Intérprete: Existen múltiples definiciones de intérpretes, las cuales no difieren de forma notable, entre las que se destacan:

- Es como un compilador, solo que la salida es una ejecución. El programa de entrada se reconoce y ejecuta a la vez. No se produce un resultado físico (código máquina) sino lógico (una ejecución) (9 pág. 3).
- Un intérprete es un traductor de lenguaje, igual que un compilador, pero difiere de éste, en que ejecuta el programa fuente inmediatamente, en vez de generar un código objeto que se ejecuta después de que se completa la traducción (10 pág. 4).
- En lugar de producir un programa objeto como resultado de una traducción, un intérprete realiza las operaciones que implica el programa fuente (11 pág. 4).
- Un intérprete es un programa que analiza y ejecuta simultáneamente un programa escrito en un lenguaje fuente (6 pág. 3).

En la Ilustración 2 se presenta el esquema general de un intérprete visto como una caja negra. Cualquier intérprete tiene dos entradas: un programa P escrito en un lenguaje fuente LF (en lo sucesivo, se denotará P/LF) junto con los datos de entrada, a partir de dichas entradas, mediante un proceso de interpretación va produciendo unos resultados:

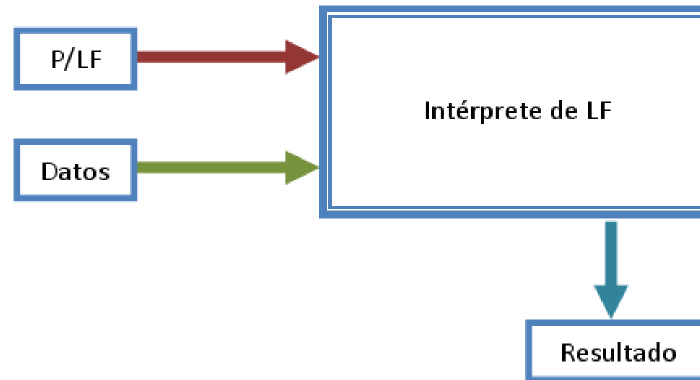

Figura 3: Esquema general de un Intérprete

Los lenguajes de programación, en principio, pueden ser interpretados o compilados. Pero puede preferirse uno que otro dependiendo de la situación en que se encuentre la traducción. Cada uno tiene sus ventajas y desventajas que lo hacen más aceptables para situaciones específicas, por ejemplo los lenguajes funcionales como el LISP, o de "muy" alto nivel como Visual Basic, JavaScript, PHP, Ruby, etc. tienden a ser interpretados, mientras que otros como FORTRAN, C, C++, Pascal, etc. son compilados. Los primeros tienen sus ventajas y desventajas sobre los segundos, que se encuentran acentuadas por la influencia en el tipo de proceso utilizado para la traducción. Los lenguajes Interpretados se caracterizan por ser independientes de la plataforma, el uso de reflexión[5] que constituye una forma de meta-programación (donde un programa manipula su código fuente en ejecución), además los tipos de datos y la gestión de memoria es dinámica. Sin embargo los programas interpretados son menos eficientes que los compilados, aunque en la actualidad se han presentado varias soluciones que cubren parcialmente este problema, como las Máquinas Virtuales, por ejemplo la JVM (Java Virtual Machine), además estos necesitan que el intérprete se encuentre en la máquina local para poder ejecutar los programas, no siendo así con los compilados, que están listos para ejecutarse sin necesidad de terceros.

[5] En informática, reflexión (o reflexión computacional) es la capacidad que tiene un programa de ordenador para observar y opcionalmente modificar su estructura de alto nivel.

1.5.2 Estructura general de un Intérprete.

Un intérprete de código es un software de una abrumadora complejidad, según afirma Booch "Cuando se diseña un sistema de software complejo, es esencial descomponerlo en partes más y más pequeñas, cada una de las cuales se pueden refinar entonces de forma independiente" (12). Así mismo ocurre en este caso. Donde se puede encontrar que generalmente los intérpretes están compuestos por los siguientes módulos:

- **Traductor a Representación Interna:** Toma como entrada el código del programa P en Lenguaje Fuente (P/LF), lo analiza y lo transforma a la representación interna correspondiente a dicho programa P.
- **Representación Interna (P/RI):** La representación interna debe ser consistente con el programa original. Entre los tipos de representación interna, los árboles sintácticos son los más utilizados y, si las características del lenguaje lo permiten, pueden utilizarse estructuras de pila para una mayor eficiencia.
- **Tabla de símbolos:** Durante el proceso de traducción, es conveniente ir creando una tabla con información relativa a los símbolos que aparecen. La información a almacenar en dicha tabla de símbolos depende de la complejidad del lenguaje fuente. Se pueden almacenar etiquetas para instrucciones de salto, información sobre identificadores (nombre, tipo, línea en la que aparecen, etc.) o cualquier otro tipo de información que se necesite en la etapa de evaluación.
- **Evaluador de Representación Interna:** A partir de la Representación Interna anterior y de los datos de entrada, se llevan a cabo las acciones indicadas para obtener los resultados. Durante el proceso de evaluación es necesario contemplar la aparición de errores.
- **Tratamiento de errores:** Durante el proceso de evaluación pueden aparecer diversos errores ya sean de tipo sintácticos o semánticos, así como el desbordamiento de la pila, divisiones por cero, etc. que el intérprete debe contemplar.

De forma general, la ilustración siguiente muestra la estructura general de un intérprete, sus módulos y cómo interactúan sus partes fundamentales.

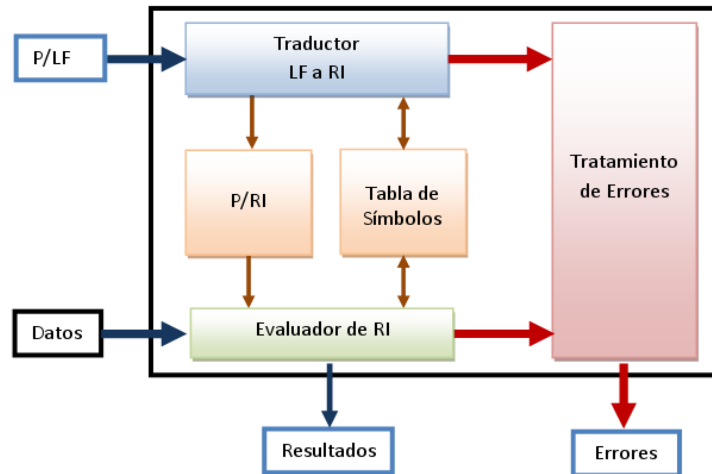

Figura 4: Organización interna de un intérprete

1.5.3 Etapas de un intérprete

En principio los intérpretes al igual que los compiladores trabajan en Etapas[6], cada una realiza transformaciones de una representación a otra. La figura 5 muestra algunas de las principales:

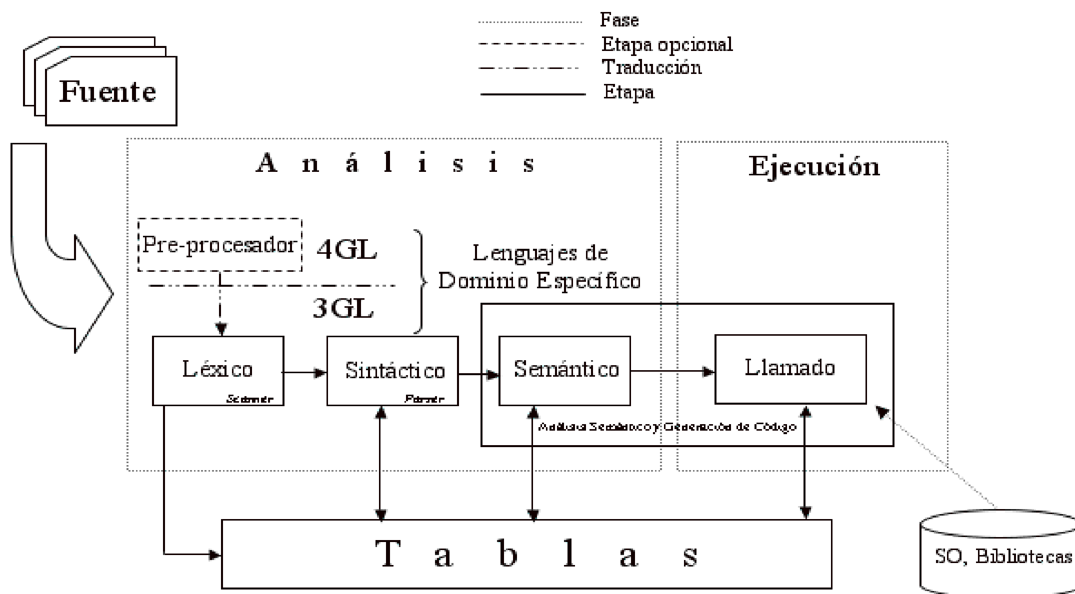

Figura 5: Etapas de un intérprete

6 Algunos autores llaman "Etapas" lo que se nombra aquí como "Fases".

Como se muestra en la figura, las etapas están agrupadas en dos fases “Análisis” y “Ejecución”, estas etapas están estrechamente relacionadas con los módulos descritos en el epígrafe anterior. A continuación se puede encontrar una descripción de cada etapa, destacando los detalles que se consideran más importantes para alcanzar el objetivo de la investigación.

Pre-procesador: Estos producen la entrada para el intérprete y pueden realizar varias funciones, como eliminar los comentarios en el código, la definición de macros por parte del usuario, las macros son abreviaturas de construcciones más grandes. El pre-procesador realiza tareas que no son indispensables o que pueden ser movidas a otra etapa, por ejemplo la etapa de análisis léxico puede encargarse de eliminar los comentarios, debido a esto se dice que es una etapa opcional y se puede prescindir de ella.

Análisis Léxico: En esta etapa se comienza a procesar verdaderamente el programa, este generalmente está formado por un flujo de caracteres, el cual se comienza a leer de izquierda a derecha (generalmente) formando unidades significativas denominadas **tokens**. La tarea realizada por un analizador léxico es un caso especial de coincidencia de patrones, se necesitan métodos especiales para su construcción, como las **Expresiones Regulares** y los **Autómatas Finitos**. Además de formar los tokens el analizador léxico puede realizar otras funciones, por ejemplo una relacionada con el **Tratamiento de Errores**, como contar la cantidad de caracteres de nueva línea para asociar el número de la línea con un error en caso de que exista.

Existen varias razones para hacer del análisis léxico una fase independiente y separarla del análisis sintáctico, entre las que se encuentran:

1. **Un diseño más sencillo**: La separación del análisis léxico y el análisis sintáctico permite simplificar ambas etapas. Por ejemplo un análisis sintáctico que incluya las convenciones de los comentarios y espacios en blanco es bastante más complejo que si estos fuesen eliminados por un analizador sintáctico a priori.
2. **Eficiencia**: Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para esta función. Se consume una gran parte del tiempo en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de buffers para la lectura de los

caracteres de entrada y procesamiento de componentes léxicos, se obtiene una mejora significativa del rendimiento en general.

3. **Mejor Transportabilidad**: Las peculiaridades del alfabeto de entrada y otras anomalías propias pueden limitarse al analizador léxico, la representación de símbolos especiales o no estándar como el "↑" en Pascal, pueden aislarse en esta etapa.

Con respecto al tratamiento de errores en esta etapa, solo pueden ser detectados los errores de nivel léxico, debido a su visión restringida del programa fuente. Por ejemplo en el caso de que el programa fuente "**fi(a == f(x))**" el analizador léxico no puede distinguir si **fi** es un error de escritura de la palabra clave **if,** pero ya que es un identificador valido, este devuelve el componente léxico de un identificador dejando este error a ser detectado en etapas posteriores.

Análisis Sintáctico: El Analizador Sintáctico obtiene una cadena de componentes léxicos del Analizador Léxico, y comprueba si la cadena puede ser generada por la gramática del lenguaje fuente. Este debe informar de cualquier error de sintaxis de manera inteligible. También debe recuperarse de los errores que ocurren y seguir procesando el resto de su entrada.

Existen varios tipos de analizadores sintácticos para gramáticas, pero los que generalmente se usan para su construcción son dos[7] de ellos, debido a que son más eficientes y menos costosos de construir. Estos métodos son el Ascendente y Descendente.

Se conoce como **análisis sintáctico por descenso recursivo** el que se realiza para la clase de gramáticas **LL(k)**, este se considera como un intento de encontrar una derivación por la izquierda de una cadena de entrada, construyendo el árbol de análisis sintáctico, comenzando desde la raíz y creando los nodos del árbol en orden previo. Este analizador se llama predictivo cuando no necesita hacer un retroceso. Para construir un analizador sintáctico predictivo se debe conocer dado el símbolo actual **a** de entrada y el no terminal **A** a expandir cual de las alternativas de la producción: $\mathbf{A \rightarrow \alpha_1 \mid \alpha_2 \ldots \mid \alpha_n}$ es la única alternativa que da lugar a una cadena que comience con **a**.

[7] Existen otros métodos de análisis sintácticos que están en desarrollo actualmente.

Un analizador LR es el reverso de un analizador LL, este intenta encontrar las derivaciones por la derecha. Este tipo de análisis es atractivo por varias razones (11 pág. 221):

- Se pueden construir analizadores sintácticos LR para reconocer prácticamente todas las construcciones de los lenguajes de programación para los que se pueden escribir gramáticas de libres contexto.
- Es el método de análisis por desplazamiento y reducción sin retroceso más general que se conoce.
- La clase de gramáticas que puede analizarse con los métodos LR es un supra conjunto de le la clase de gramáticas que se pueden analizar con los analizadores sintácticos predictivos.

El principal inconveniente de los analizadores LR es que son muy difíciles de construir de forma manual, se necesitan herramientas específicas y que solo están disponibles para algunos lenguajes. Por otra parte los analizadores sintácticos LL son fáciles de construir manualmente y su funcionamiento encaja con la forma en que los humanos pensamos al analizar un texto determinado. Además la forma en que trabajan es ventajosa para la recuperación de errores ya que siempre se conoce el próximo token que debe hallarse, en caso de ser incorrecto se puede informar exactamente lo que escribió incorrecto y lo que debería haber escrito.

Finalmente, sin que afecte el método de análisis usado, ya sea LL o LR u otro, el resultado del analizador sintáctico no es más que el Árbol de Sintaxis Abstracta. Tales árboles representan abstracciones que contienen toda la información del código fuente original.

Análisis Semántico: La fase de análisis semántico de un procesador de lenguaje es la que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación. (8 pág. 1). Esta etapa obtiene su nombre debido a que la información que requiere es relativa al significado del lenguaje, que no puede ser representada por las gramáticas de libre contexto, como por ejemplo que la utilización de una variable en el lenguaje Pascal ha de estar previamente declarada.

El objetivo principal del analizador semántico de un procesador de lenguaje es asegurarse de que el programa analizado satisfaga las reglas requeridas por la especificación del lenguaje, para garantizar su correcta ejecución (8 pág. 2). El análisis semántico no modela la semántica de los programas construidos con el lenguaje de programación, si no que usa información parcial de su comportamiento para realizar una serie de comprobaciones necesarias (que no pueden ser ejecutadas por el analizador sintáctico) para comprobar que el programa pertenece al lenguaje. Existen una multitud de comprobaciones que se pueden realizar en un analizador semántico. Aunque estas dependen de las características del lenguaje, generalmente se realizan las siguientes comprobaciones:

- **Declaración de identificadores y reglas de ámbitos**: Este se asegura de que las variables han sido declaradas, antes de hacer uso de las mismas. Esta comprobación se aplica en lenguajes como Java, C++, C# y otros, sin embargo puede ser que el lenguaje no requiera que una variable haya sido declarada con anterioridad como el PHP.
- **Comprobaciones de unicidad**: Existen multitud de elementos en lenguajes de programación cuyas entidades han de existir de un modo único, es decir, no se permite que estén duplicadas. Entre ellas podemos encontrar: Constantes de cada caso (case) en Pascal, C o Java. Los valores de un tipo enumerado de Pascal o C, la declaración de un identificador en un ámbito, entre otras.
- **Comprobaciones de tipo**: Esta es una de las comprobaciones más exhaustivas y amplias del análisis semántico, y es necesaria en todo lenguaje de alto nivel. Ya sea en modo estático (en tiempo de compilación) o dinámico (en tiempo de ejecución). Al realizar la comprobación de tipos, el analizador ejecutará dos tareas como mínimo, la primera de ellas, debe ser comprobar las operaciones que se pueden aplicar a cada construcción del lenguaje. Y la segunda es inferir el tipo de cada construcción del lenguaje.

Luego de haber visto el ejemplo anterior se puede llegar a la conclusión de que resulta ventajoso (en cuanto a eficiencia) usar lenguajes fuertemente tipados. A continuación se listan algunas de las ventajas del uso de tipos, en lenguajes de programación (8 pág. 74):

- **Fiabilidad**: La comprobación estática de tipos reduce el número de errores que un programa puede generar en tiempo de ejecución.

- **Abstracción**: Otra ventaja de emplear tipos en los lenguajes de programación, es que su uso, fuerza al programador a dividir el problema en diversos tipos de módulos de un modo disciplinado.

- **Legibilidad**: Un tipo de una entidad (variable, objeto o función) transmite información acerca de lo que se intenta hacer con ella, constituyendo así un modo de documentación del código.

- **Eficiencia**: Una entidad de un programa declarada con un tipo específico, indica información relativa a lo que se intenta hacer con ella. De este modo, al conocer el tipo de las construcciones del lenguaje, se podrá generar código de carácter más específico y eficiente que si no se contara con esa información.

Los algoritmos para la implementación de los analizadores semánticos no son claramente expresables como los del analizador léxico o el sintáctico. Si el análisis semántico se puede suspender hasta que todo el análisis sintáctico (y la construcción de un árbol sintáctico abstracto) esté completo, entonces la tarea de implementar el análisis semántico se vuelve considerablemente más fácil, y consiste en esencia en la especificación de orden para un recorrido del árbol sintáctico, junto con los cálculos a realizar cada vez que se encuentra un nodo en el recorrido.

Llamado o Ejecución: Esta etapa es la única en la fase de Ejecución, es importante mencionar que existen dos métodos básicos para su implementación. Este puede elaborarse usando interpretación iterativa o recursiva. La interpretación iterativa es apropiada para los lenguajes sencillos donde se analiza y ejecuta cada expresión de forma directa, como podrían ser los códigos de máquinas abstractas o lenguajes de sentencias simples. Consiste en un ciclo básico de búsqueda, análisis y ejecución de instrucciones. El algoritmo básico sería el siguiente.

```
Inicializar
REPETIR
  Buscar siguiente Instrucción i
  SI encontrada ENTONCES
                    Analizar i
                    Ejecutar i
HASTA (que no haya más instrucciones)
```

La interpretación recursiva generalmente es usada para la construcción de prototipos de lenguajes. En esta suele utilizarse un modelo de interpretación recursiva donde las sentencias pueden estar compuestas de otras sentencias y la ejecución de una sentencia puede lanzar la ejecución de otras sentencias de forma recursiva. Algunos expertos en el tema coinciden en que los intérpretes recursivos no son apropiados para aplicaciones prácticas debido a su ineficiencia, aunque en la actualidad se están desarrollando estudios que intentan demostrar la veracidad de este planteamiento, mientras en otros casos ya se han puesto en marcha lenguajes con interpretación recursiva como el lenguaje Ruby, que usa como representación interna el AST y su evaluación consiste en un recorrido de forma recursiva por el mismo. Se han realizado pruebas de intérpretes de java basados en AST demostrando ser más rápidos que un interpretador similar basado en *bytecode* (13), debido a que se puede realizar una mejor optimización al tener la estructura completa del programa, así como los tipos de datos de alto nivel con los que se cuenta durante la ejecución de un programa.

1.6 Propuesta de Solución.

Luego de analizar la situación del software con respecto a las fórmulas matemáticas, la obtención de datos desde diferentes fuentes de datos, y algunos elementos teóricos necesarios para la construcción de este tipo de herramientas, se propone elaborar un componente de software para la especificación y evaluación de fórmulas. Además se pretende desarrollarlo sobre plataformas libres, usando técnicas estándar, de forma que pueda ser implementado en un conjunto de lenguajes, con el objetivo de tener una versión del componente para diferentes plataformas de desarrollo.

Se espera con esta propuesta de solución brindar a los desarrolladores de software una potente herramienta, con la idea de crear software más robusto, flexible ante los cambios del modelo matemático de un negocio determinado. Además de obtener una separación del conocimiento, aislando al informático de entender la complejidad del negocio en cuanto a fórmulas matemáticas, que a veces puede resultar engorroso.

1.7 Conclusiones Parciales.

Puede que al estudiar toda la teoría relacionada con la solución propuesta, como la elaboración y especificación de lenguajes, además de la construcción de intérpretes, se piense que es una tarea titánica, que solo puede llevarse a cabo por empresas que dominan muchos recursos. Pero en la práctica se ha demostrado que esto no es cierto, un

lenguaje y la construcción de un intérprete pueden ser realizados por un hombre en un plazo de tiempo razonable. Un vivo ejemplo de esto, es el lenguaje Python y su intérprete, que fue desarrollado por Guido Van Rossum[8].

En varias ocasiones buscamos incesantemente soluciones para los nuevos problemas que surgen, obviando por completo las soluciones que ya se han encontrado a algunos problemas del pasado, entonces se pasa por alto que la posible solución está en la teoría que ya se conoce, vista desde otro ángulo o combinadas entre sí de alguna forma útil. Esta investigación propone usar técnicas convencionales como los intérpretes de código y lenguajes de dominio específico, que se complementan bajo el concepto de componente de software, para solucionar la problemática actual de los desarrolladores de software, frente al impacto que provoca los cambios en el modelo matemático sobre el cual se concibió un producto determinado.

[8] Para más información diríjase al sitio oficial de Python http://www.python.org, también puede ver la página personal de Guido Van Rossum en: http://www.python.org/~guido/

Capítulo 2: Tecnologías y tendencias actuales a desarrollar.

2.1 Introducción.

Desarrollar un buen software depende de varios factores, elegir la metodología más apropiada, los lenguajes más adecuados, ya sean de modelado o de programación, la utilización del tiempo, entre otros; por tanto se hace necesario tener conocimiento de los mismos para lograr un mejor resultado. En este capítulo se realizará un análisis de las tecnologías y tendencias actuales que pudieran ser útiles en el desarrollo de la solución propuesta. Así como la posible metodología de desarrollo a emplear, el lenguaje de programación y el lenguaje de modelado, entre otros aspectos.

2.2 El Lenguaje Unificado de Modelado (UML).

2.2.1 Definiendo el lenguaje utilizado para la modelación

UML, por sus siglas en inglés, *Unified Modeling Language* es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software (14), es el más conocido y utilizado en la actualidad. UML ofrece un estándar para describir el sistema, además de poseer características y propiedades que son las que lo validan como el más aceptado en la actualidad.

Conceptos y modelos de UML

UML posee conceptos y modelos que pueden llegar a ser áreas conceptuales, las mismas son: (14)

Estructura estática: Todo modelo debe definir su propio universo, o sea, los conceptos claves de las aplicaciones, sus propiedades internas, y las relaciones entre cada una de ellas.

Comportamiento dinámico: En UML existen dos formas de modelar el comportamiento. Una es la historia de la vida de un objeto, que muestra la forma en que interactúa con el resto del mundo y la a otra son los patrones de comunicación de un conjunto de objetos conectados, que muestra cómo interactúan para implementar su comportamiento.

Construcciones de implementación: Los modelos de UML son significativos para el análisis lógico y para la implementación física. Ciertos constructores representan elementos de implementación.

Organización del modelo: Los paquetes son unidades organizativas, jerárquicas, y de propósito general de los modelos de UML. Pueden usarse para almacenamiento, control de acceso, gestión de la configuración, y construcción de bibliotecas que contengan fragmentos de código reutilizable.

Características de UML

Algunas de las propiedades de UML como lenguaje de modelado estándar son (14):

- Concurrencia, es un lenguaje distribuido y adecuado a las necesidades de conectividad actuales y futuras.
- Ampliamente utilizado por la industria desde su adopción por OMG[9].
- Reemplaza a decenas de notaciones empleadas con otros lenguajes.
- Modela estructuras complejas.
- Las estructuras más importantes que soportan tienen su fundamento en las tecnologías orientadas a objetos, tales como objetos, clase, componentes y nodos.
- Emplea operaciones abstractas como guía para variaciones futuras, añadiendo variables si es necesario.
- Comportamiento del sistema: casos de uso, diagramas de secuencia y de colaboraciones, que sirven para evaluar el estado de las máquinas.

El modelado de la aplicación se realizará con el Lenguaje Unificado de Modelado para así lograr un mayor entendimiento del sistema, ya que es uno de los más utilizados mundialmente y esto permite tener una amplia documentación del mismo, UML admite describir por pasos los procesos que se llevan a cabo a lo largo del desarrollo de software, además de ser, como antes se mencionaba el más aceptado actualmente, llegando a ser el estándar a nivel mundial.

[9] La OMG (Object Management Group) es una asociación sin fines de lucro formada por grandes corporaciones, muchas de ellas de la industria del software, como IBM, Apple, Sun Microsystems y HP.

2.3 El proceso unificado de desarrollo de software (RUP)

"La Ingeniería de Software es el establecimiento y uso de principios sólidos de ingeniería, orientados a obtener software económico que sea fiable y trabaje de manera eficiente en máquinas reales" (Fritz Bauer, 1968 (conferencia NATO)).

Realmente, construir software de computadora es un proceso de aprendizaje iterativo, y el resultado, algo que Baetjer podría llamar <<capital del software>>, es el conjunto del software reunido, depurado y organizado mientras se desarrolla el proceso. (15)

Definiendo RUP

RUP es un proceso de desarrollo dirigido por casos de uso, iterativo e incremental y centrado en la arquitectura. Es el resultado de la evolución e integración de diferentes metodologías de desarrollo, es un proceso pesado, donde una de sus debilidades es la gran cantidad de documentos que se genera a lo largo del desarrollo del software. Es uno de los procesos de desarrollo de software más generales de los que existen actualmente ya que está pensado para adaptarse a cualquier proyecto, incluso proyectos que van más allá del desarrollo de software. (16)

Principales características de RUP

RUP posee características que lo hacen sobresalir por encima de las demás metodologías de desarrollo existentes, algunas de ellas son: (17)

- **Las mejores prácticas y más probadas de la industria**: Son las mejores prácticas de desarrollo adoptadas en cientos proyectos mundialmente y enseñadas como parte de la materia en cientos de universidades, la metodología RUP se convirtió rápidamente en el estándar de facto para el proceso de desarrollo en la industria de software.
- **Proceso hecho práctico:** A diferencia de otras metodologías comerciales, la plataforma RUP hace que el proceso sea práctico con bases de conocimiento y guías para ayudar en el despegue de la planificación del proyecto, integrar rápidamente a los miembros del equipo y poner en acción el proceso personalizado.
- **Se adapta a las necesidades de los proyectos**: Solo la plataforma RUP nos brinda un marco de proceso configurable que permite seleccionar e implantar los

componentes específicos de procesos necesarios para proporcionar un proceso consistente y personalizado para cada equipo y proyecto.

2.3.1 ¿Agile UP?

En el caso de esta investigación se utilizará como metodología de desarrollo para la misma, la versión ágil de RUP o Agile Unified Process (AUP, por sus siglas en inglés), como se conoce mundialmente en el desarrollo de software.

El Agile UP (AUP) describe un enfoque simple y fácil de entender para el desarrollo de software usando técnicas y conceptos que aún se mantienen vigentes en RUP. Los enfoques aplican técnicas ágiles incluidas en el Desarrollo Dirigido por Pruebas (TDD, por sus siglas en inglés), Desarrollo Dirigido por Modelado Ágil (AMDD, por sus siglas en inglés), administración de cambios ágil, y refactorización de bases de datos para mejorar la productividad. (18)

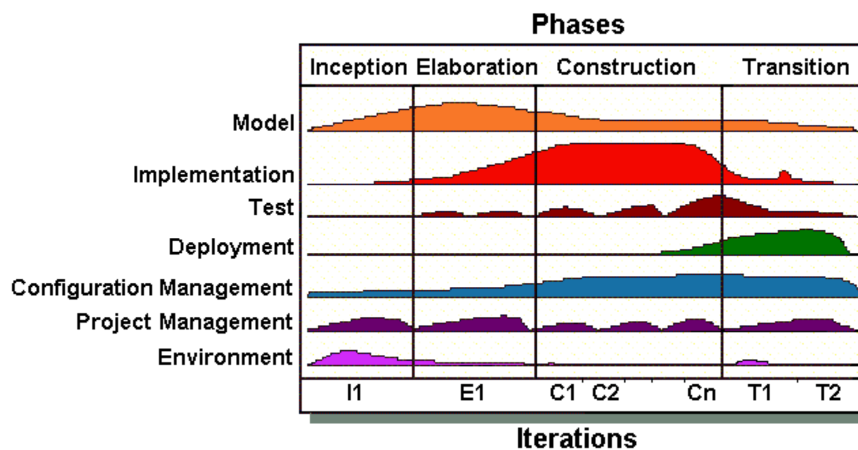

Figura 6: Fases de la metodología de desarrollo RUP

Algunas características de Agile Up

Esta metodología ha emergido a la industria mundial como un estándar para los actuales procesos, grupos dentro de varias compañías líderes la están adoptando, como por ejemplo: IBM, INTEL, ERICSSON. AUP es cohesivo y está bien documentado. Está constituido por siete disciplinas: Modelo; Implementación; Prueba; Despliegue; Gestión de la Configuración; Entorno. Y al ser una versión de RUP posee las mismas cuatro fases: Inicio; Elaboración; Construcción; Transición.

2.3.2 Selección de la metodología de desarrollo.

Como se ha demostrado, y como su nombre lo señala es una metodología ligera pero muy abarcadora, ya que está basada fundamentalmente en los conceptos de RUP, la cual es una de las metodologías más generales que existen mundialmente. Lo más apreciable de Agile UP es precisamente que es más rápida y no necesita toda la cantidad de artefactos que genera RUP, pero a la vez es muy eficiente, ya que sus fases y disciplinas están muy ligadas con las de esta robusta metodología, a la vez que se puede añadir que Agile UP es una modificación o especificación de RUP, pero con la ventaja que ya ha sido probada mundialmente.

2.4 Selección del Lenguaje de Programación

¿Por qué usar un lenguaje orientado a objetos?

La programación orientada a objetos es una de las formas más populares de programar y tiene gran aceptación en el desarrollo de proyectos de software. Esto se debe a sus grandes capacidades y ventajas frente a otros paradigmas de programación. La tecnología orientada a objetos ya no se aplica solamente a los lenguajes de programación, sino que se aplica también en el análisis y diseño con mucho éxito, al igual que en las bases de datos. (19)

Ventajas de los lenguajes orientados a objetos:

Los lenguajes orientados a objetos tienen algunas ventajas y características específicas que los hacen representar el mundo real muy detalladamente. Fomentan la reutilización y extensión del código, permitiendo la creación de sistemas más complejos y relacionados con el mundo real. Facilitan la creación de programas visuales y permite la construcción de prototipos ya sea de interfaz de usuario o prototipos de clases. La POO facilita el trabajo en equipo y de esta forma se agiliza el desarrollo del software y el mantenimiento del mismo. (19)

2.4.1 Definiendo C++

El lenguaje C++ fue inventado en 1980 por Bjarne Stroustrup en los Laboratorios Bell en Murray Hill, New Jersey. Inicialmente recibió la denominación de << C con clases >>. En 1983 se le nombró C++. Desde 1980 C++ ha sufrido dos importantes revisiones, una en

1985 y otra en 1990. Este lenguaje es una versión ampliada del lenguaje C, incluye todo lo que forma parte de C y añade soporte para la programación orientada a objetos. (20)

Características de C++

Entre las características principales de este lenguaje podemos encontrar que:

- Proporciona grandes facilidades para la programación Orientada a Objetos.
- Permite el uso de plantillas o programación genérica.
- Nos permite redefinir los operadores o sobrecargarlos como también se conoce.
- Permite la identificación de tipos en tiempo de ejecución.

2.4.2 Justificación del lenguaje de programación seleccionado

El lenguaje C++ es potente, esto se puede observar en que la mayoría de los compiladores e intérpretes en el mundo han sido desarrollados en C++. Además permite la programación tanto a bajo nivel como a alto nivel. En comparación con java se puede decir que la rapidez de ejecución es mayor, es más potente en el manejo de memoria, es muy útil en la programación orientada a objetos, y la programación genérica.

2.5 NetBeans

El IDE NetBeans es un entorno de desarrollo integrado, disponible para Windows, Mac, La herramienta seleccionada para la implementación de la solución propuesta fue NetBeans un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés), disponible para Windows, Mac, Linux y Solaris. NetBeans es un IDE de código abierto y una plataforma de aplicaciones que permite a los desarrolladores la creación de páginas Web, aplicaciones de escritorio y aplicaciones móviles utilizando fundamentalmente la plataforma Java. (21)

Características de NetBeans

El IDE que se utilizará posee algunas características que lo hacen destacado entre los más utilizados en el desarrollo de software: (21)

- Permite la creación de proyectos Web con Java EE 6 y EJB.
- Utiliza como soporte EJB 3.1.
- JPA 2.0, implementación, depuración y perfilado con servidor de aplicaciones GlassFish v3.
- Completamiento de código, pistas de error, ventanas emergentes de documentación, etc.

- Editor de apoyo para las bibliotecas Facelets, componentes compuestos, lenguaje de expresión, incluyendo generadores para JSF y HTML.
- Paleta de componentes personalizable.
- Añadido soporte para el último SDK de JavaFX 1.2.1.

2.5.1 Selección del entorno de desarrollo.

Para la implementación del componente se utilizará NetBeans por las características antes expuestas, aunque también es válido destacar que existen otros IDE's disponibles y muy tentadores como Eclipse. Tanto NetBeans como Eclipse se integran con Visual Paradigm, pero NetBeans posee mejor completamiento de código con C++, es necesario recalcar que NetBeans además de generar código, también genera documentación del mismo, lo que aporta una buena utilidad a ser tomada en cuenta en el momento de desarrollar el software.

2.6 Visual Paradigm

Visual Paradigm es una herramienta de diseño que sirve para realizar modelado UML, soporta todos los diagramas del mismo. Soporta también SysML y diagramas de entidad-relación. (22)

Esta herramienta posee unas características graficas muy cómodas que facilitan la realización de los diagramas de modelado que sigue el estándar de UML y los utilizados para el diseño de la aplicación: (23)

- Diagramas de clases
- Diagramas de Casos de Uso
- Diagramas de Secuencia
- Diagramas de Componentes

Entre otras características importantes que tiene se encuentran: (23) (22)

- Integración con diversos IDE's como son: NetBeans, JDeveloper, Eclipse, JBuilder.
- Ingeniería Inversa para: JAVA, .NET, XML, Hibernate.
- Exportación de imágenes jpg, png y svg (w3g estándar).
- Genera la documentación del sistema en formato PDF, HTML y MS Word.
- Genera código.

- Disponibilidad en múltiples plataformas.
- Genera documentación.

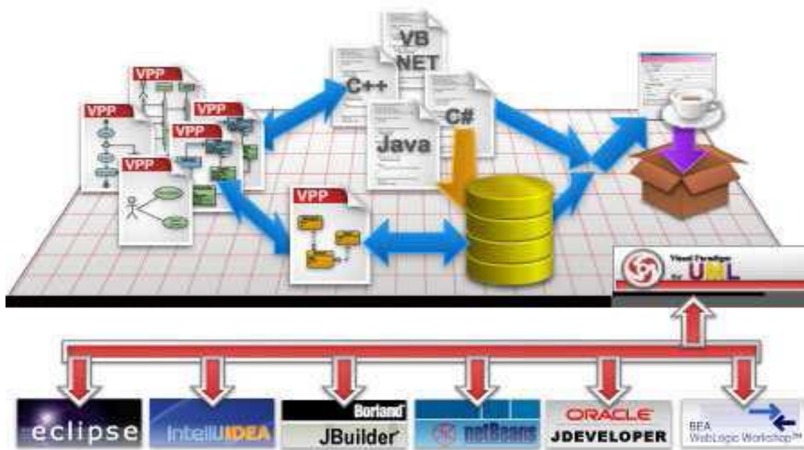

Figura 7: Integración de Visual Paradigm en el proceso de desarrollo.

2.6.1 Selección de la herramienta de Modelado.

Otra herramienta de diseño disponible y muy aceptable en el modelado con UML es el *Rational Rose*, pero Visual Paradigm posee un área de trabajo que brinda mayores opciones y utilidades al usuario, su forma de organización de los diagramas es muy cómoda y organizada. Como se ha visto en sus características, Visual Paradigm se integra con varios IDE's como por ejemplo NetBeans, que es el designado para el desarrollo del componente. Además como se puede apreciar el Visual Paradigm es multiplataforma lo que permite ampliar el campo de desarrollo de la investigación y el componente. Por estas razones se llegó a la conclusión que la herramienta CASE a utilizar en este trabajo será Visual Paradigm.

2.7 Conclusiones Parciales

Con el análisis realizado a través de este capítulo se ha logrado la selección de la metodología que regirá el desarrollo del componente, así como las herramientas que darán solución al mismo. Se concluyó que la metodología a usar sería Agile UP por estar muy complementada con RUP y a la vez ser más ágil; el lenguaje de modelado para describir todos los procesos y el negocio en general será UML, y Visual Paradigm la herramienta CASE para modelar la aplicación. El lenguaje de programación escogido es C++ por sus potencialidades y como IDE de desarrollo se utilizará NetBeans por su buen completamiento de código entre otras importantes características.

Capítulo 3: Presentación de la solución propuesta

3.1 Introducción

El objetivo de este trabajo, es obtener un componente de software que facilite la especificación y evaluación de fórmulas matemáticas. En este capítulo se realizará una descripción de la solución propuesta acorde con la metodología de desarrollo escogida. El propósito es asegurar la calidad del producto final a partir de una guía adecuada del proceso de desarrollo del software.

3.2 Modelo del Dominio

Luego de un estudio de los procesos del negocio (1.2.1) se llega a la conclusión que no tienen fronteras bien establecidas, a partir de la cual se logren identificar las personas que inician o desarrollan las actividades. Por lo que se decide elaborar un modelo de Dominio (ver figura 8).

Para esto se identifican los conceptos presentes en el modelo de dominio mediante un Glosario de términos del Modelo del Dominio:

- **Token:** Unidad básica de un compilador, que puede representar un símbolo o conjunto de símbolos con un significado semántico.
- **Árbol de Sintaxis Abstracta** (**AST,** por sus siglas en inglés): Estructura en forma de árbol que posee sus componentes teniendo en cuenta solamente los terminales que aportan algún valor semántico, es decir, se ignoran los símbolos separadores y de agrupación (tales como punto y coma, coma, paréntesis, etc.).
- **Analizador Léxico (***Lexer* o *Scanner* en su término en inglés**)**: Componente del sistema que lee el código escrito por el usuario y produce como salida una secuencia de *tokens,* que es utilizada por el Analizador sintáctico para crear el **AST**. También realiza otras funciones, como la eliminación de los comentarios, caracteres de tabulación, espacios en blanco, etc.
- **Analizador Sintáctico** (**Parser** en su término en ingles): Componente del sistema que toma como entrada una lista de tokens y que valida el orden en que estos aparecen, dándole su significado semántico cada vez que se reduce una regla.
- **Cadena**: Secuencia de símbolos de cierto alfabeto colocados uno a continuación del otro.

- **Usuario**: Es la persona (o sistema) que interactúa con el componente.
- **Código Intermedio**: Es una interpretación que genera un compilador de las cadenas entradas, que sirve para ser trasladada o interpretada por otros sistemas (el AST puede ser usado de código intermedio).
- **Máquina Virtual**: Es la fase final del intérprete que ejecuta el código intermedio.
- **Analizador Semántico**: Etapa en la que se comprueba la corrección semántica de las instrucciones y se detectan errores semánticos.
- **Librerías Matemáticas**: Conjunto de funcionalidades para dar soporte a los cálculos matemáticos elementales.
- **Librerías de Apoyo del Lenguaje**: Conjunto de funcionalidades que dan soporte al sistema de tipos y operadores que se usan en el lenguaje.

El modelo del dominio se describe en la Figura 11, utilizando el Lenguaje Unificado de Modelado (UML).

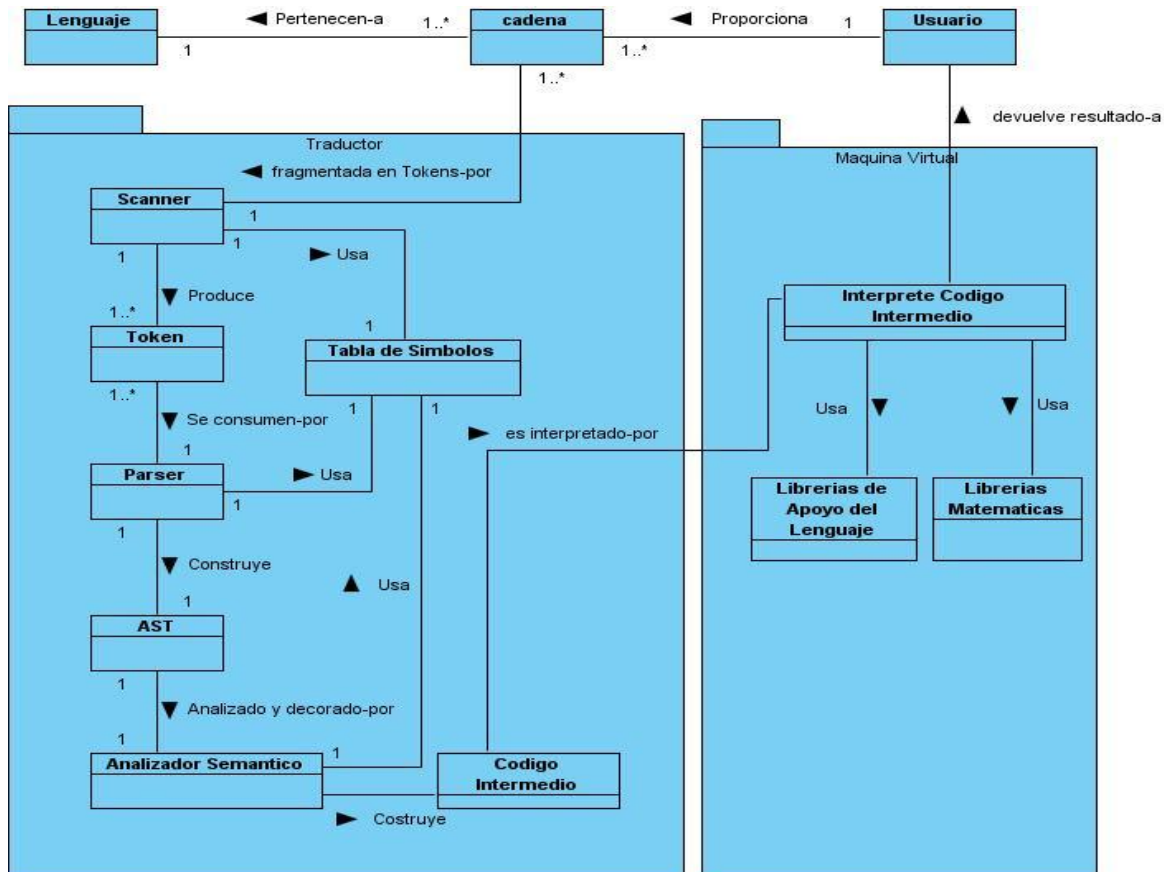

Figura 8: Modelo de dominio

Luego de conocer los conceptos asociados al objeto de estudio, queda abierta la pregunta: ¿Qué debe hacer el sistema para que se cumplan los objetivos planteados? La respuesta la podemos encontrar en una lista enumerada de requerimientos.

3.3 Requerimientos funcionales

Los requisitos definen el comportamiento del software. Se pueden encontrar dos tipos de requisitos, Requisitos funcionales y los No funcionales. Los Requisitos Funcionales no son más que las condiciones o capacidades que deben estar presentes en un sistema o componentes de sistema. Para dar respuesta a la pregunta ¿Qué debe hacer el sistema para que se cumplan los objetivos planteados? Se confeccionó una lista enumerada de requerimientos:

RF1 El sistema debe ser capaz de reconocer y analizar las cadenas pertenecientes al lenguaje elaborado (operaciones aritméticas, expresiones lógicas, instrucciones, consultas, etc.)

RF2 El sistema debe ser capaz de recuperarse de los errores de compilación y seguir el proceso.

RF3 El sistema debe ser capaz de devolver una lista con los errores encontrados durante el proceso de compilación (Errores Lexicográficos, Sintácticos, Semánticos, y Lógicos).

RF4 El sistema debe ser capaz de compilar las fórmulas en un lenguaje intermedio, para ser almacenado.

RF5 El sistema debe ser capaz de evaluar una fórmula compilada, previamente almacenada, y generar resultados.

RF6 El sistema debe ser capaz de obtener datos desde varias fuentes de datos al evaluar alguna fórmula que así lo requiera.

3.4 Requerimientos no funcionales

Los requisitos No Funcionales son aquellos que describen las cualidades, características y propiedades del producto.

Requerimiento de usabilidad:

- El sistema debe tener interfaces amigables y homogéneas, que cumplan con un estándar de código elaborado para la implementación de la solución propuesta.
- El sistema debe generar los mensajes de Error de forma inteligible.

Requerimiento de Soporte:

- El sistema debe ser capaz de dar las mismas salidas para diferentes ambientes.
- El sistema debe ser flexible ante la necesidad de cambios de sus salidas.
- Los errores del sistema no pueden afectar a los sistemas clientes.

Requerimiento de portabilidad:

- El sistema debe ser compatible con los Sistemas Operativos: Windows XP, Windows Vista, Windows 7 y GNU/Linux.

Requerimiento de rendimiento:

- El sistema debe garantizar un tiempo de respuesta de 3 a 5 segundos en dependencia de la complejidad de las fórmulas a evaluar.

3.5 Descripción del Sistema Propuesto

Teniendo en cuenta el modelo de dominio, los requerimientos planteados anteriormente y los objetivos de esta investigación, se propone la elaboración de un Traductor de fórmulas a lenguaje Intermedio o fórmula compilada, y una Máquina Virtual o intérprete de código Intermedio. Por otra parte se considera la existencia de un solo rol, el usuario del sistema, el cual es otro sistema que se beneficia de los resultados del componente.

3.6 Modelo de Casos de uso del Sistema

Actores	Justificación
Usuario	Representa un **sistema** que tiene la capacidad de usar las diferentes funcionalidades que brinda el componente.

A continuación se presentan los casos de uso determinados para satisfacer los requerimientos funcionales del sistema:

CU-1	**Compilar Fórmula**
Actor	Usuario
Descripción	El usuario proporciona el código fuente de la fórmula que se desea compilar, y se realiza el análisis léxico, sintáctico y semántico de la misma, generando el código intermedio o Fórmula Compilada (**FC**)
Referencia	RF1, RF2, RF3, RF4
CU-2	Evaluar Fórmula Compilada
Actor	Usuario
Descripción	El usuario indica la fórmula que quiere evaluar así como los parámetros de entrada (en caso que lo requiera) y el sistema genera el resultado.
Referencia	RF5, RF6

Diagrama de casos de Uso del Sistema:

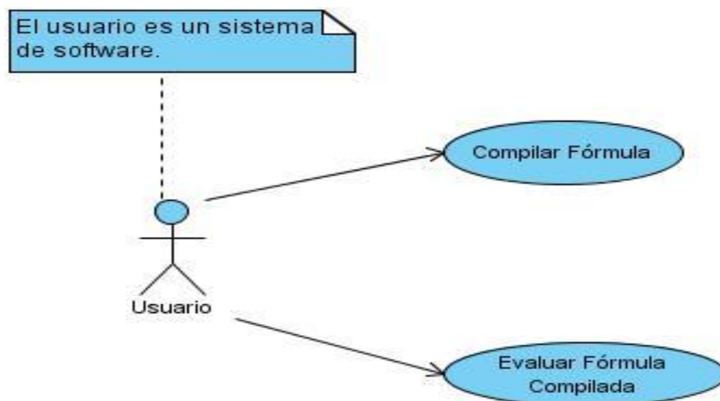

Figura 9: Diagrama de casos de uso

3.6.1 Descripción Textual de los Casos de Uso

En la descripción textual de los casos de uso del negocio se detallan las actividades que se realizan dentro de los procesos, dando así un mayor entendimiento y seguimiento del mismo.

Caso de Uso del Sistema:	**Compilar Fórmula**
Actor(es):	Usuario
Propósito	Reconocer y almacenar las fórmulas proporcionadas por el usuario mientras estas pertenezcan al lenguaje.

<table>
<tr><td>Resumen:</td><td colspan="2">El caso de uso se inicia cuando el actor proporciona el código fuente de una fórmula con el objetivo de compilarla para almacenar el código intermedio de la misma.</td></tr>
<tr><td>Precondiciones:</td><td colspan="2">-</td></tr>
<tr><td>Referencias:</td><td colspan="2">RF1, RF2, RF3, RF4</td></tr>
<tr><td>Prioridad:</td><td colspan="2">Crítico.</td></tr>
<tr><td colspan="3">Flujo Normal de los Eventos</td></tr>
<tr><td colspan="2">Acción del Actor</td><td>Respuesta del Sistema</td></tr>
<tr><td colspan="2">1. El usuario proporciona el código fuente de la fórmula que desea compilar.</td><td></td></tr>
<tr><td colspan="2">2. El usuario invoca la acción de compilar fórmula.</td><td>3. El sistema inicia el proceso de compilación del código fuente de la fórmula.</td></tr>
<tr><td colspan="2"></td><td>4. El sistema realiza el análisis léxico al código fuente de la fórmula proporcionada por el usuario.</td></tr>
<tr><td colspan="2"></td><td>5. El sistema realiza el análisis sintáctico usando los tokens generados por el analizador léxico.</td></tr>
<tr><td colspan="2"></td><td>6. El sistema realiza el análisis semántico (decorando y comprobando las reglas semánticas) sobre el árbol de sintaxis abstracta generado por el Analizador sintáctico.</td></tr>
<tr><td colspan="2"></td><td>7. Si se encuentran Errores ir a la Sección: Se detectan errores.

Si no se encuentran Errores ir a la Sección: No se detectan errores.</td></tr>
<tr><td colspan="3">Sección: Se detectan errores.</td></tr>
<tr><td colspan="2"></td><td>8. El sistema retorna la lista de errores encontrados durante el proceso de compilación.</td></tr>
<tr><td colspan="2">9. El usuario debe corregir los errores y comenzar nuevamente el proceso de compilación de la fórmula,</td><td></td></tr>
</table>

finalizando el caso de uso.	
Sección: No se detectan errores.	
	10. El sistema Genera el código intermedio o fórmula compilada.
	11. El sistema informa la correcta compilación de la fórmula y se almacena su código intermedio, finalizando el caso de uso.
Flujos Alternos	
Acción del Actor	**Respuesta del Sistema**
Pos condiciones:	Se logra realizar la compilación de la fórmula y queda almacenado el código intermedio de la misma.

Caso de Uso del Sistema:	**Evaluar Fórmula Compilada**
Actor(es):	Usuario
Propósito	Obtener un resultado a partir de la evaluación de una fórmula.
Resumen:	El caso de uso se inicia cuando el actor indica que fórmula desea evaluar
Precondiciones:	Deben existir almacenada al menos una fórmula compilada.
Referencias:	RF5, RF6
Prioridad:	Crítico.
Flujo Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El usuario invoca la acción de evaluar fórmula compilada, indicando la fórmula que desea calcular.	2. El sistema inicia el proceso de evaluación del código compilado de la fórmula, haciendo uso de las librerías matemáticas y de acceso a fuentes de datos externas. **Flujos alternos**: • Si la fórmula no está almacenada ir a **2a** • Si la fórmula necesita parámetros que no se le suministraron ir a **2b** • Si se produce un error en tiempo de ejecución al evaluar la fórmula ir a **2c**

	3. El sistema retorna el valor resultante de evaluar la fórmula escogida, finalizando el caso de uso.
Flujos Alternos	
Acción del Actor	**Respuesta del Sistema**
	2a [La fórmula no está almacenada] El sistema informa al usuario que la fórmula indicada no se encuentra almacenada, finalizando el caso de uso.
	2b [La fórmula necesita parámetros que no se le suministraron] El sistema informa al usuario que faltan parámetros por suministrar a la fórmula que se desea evaluar, finalizando el caso de uso.
	2c [Se produce un error en tiempo de ejecución al evaluar la fórmula] El sistema informa al usuario que ocurrió un fallo en tiempo de ejecución informando la naturaleza del mismo si es posible, y finaliza el caso de uso.
Pos condiciones:	Se logra realizar la evaluación del código compilado de una fórmula y se obtiene el resultado.

3.7 Lenguaje Lince

En la elaboración del lenguaje de dominio específico para fórmulas matemáticas (Lince[10]), se analizaron una serie de aspectos relevantes que influyeron en su diseño. A continuación se describen brevemente algunos de ellos:

- **Concisión de notación:** Este aspecto es uno de los más importantes, y una de las principales características de los DSL. El mismo plantea que el lenguaje debe ayudar al programador, su sintaxis debe ser simple, unificada y legible por las personas que lo utilicen.
- **Ortogonalidad:** Se dice que dos características de un lenguaje son ortogonales si pueden ser comprendidas y combinadas de forma independiente. Cuanto más ortogonales son las características de un lenguaje, más comprensible es el mismo. Esto se debe a la existencia de menos situaciones especiales a memorizar por los programadores.

[10] Lince es el nombre que se designó al lenguaje elaborado en la propuesta de solución. No tiene un significado especial está inspirado en el animal que lleva ese nombre y se encuentra en peligro de extinción.

- **Abstracción:** Este aspecto estimula la reutilización, permitiendo que el programador identifique patrones repetitivos y automatice tareas mecánicas. Para cumplir con esta característica, el lenguaje permite utilizar fórmulas dentro de otras. De esta forma el programador puede poner en una fórmula las operaciones comunes y utilizarlas desde otras.
- **Portabilidad:** El lenguaje debe facilitar la creación de programas que funcionen en el mayor número de entornos computacionales. Con el objetivo de hacer el componente portable, se decidió programarlo en C++ Estándar, para el cual existen compiladores en diferentes plataformas como Windows y GNU/Linux.
- **Eficiencia:** En la implementación del intérprete se emplearon una serie de patrones de diseño, algoritmos y prácticas para elevar la eficiencia.
- ***Librerías e interacción con el exterior*:** Se desarrolló una estrategia que permite extender las funcionalidades del lenguaje, la misma permite crear nuevas fórmulas desde el lenguaje nativo C++ con acceso a datos internos del intérprete, lo que permite introducir una potencia a estas fórmulas que las programadas en Lince, un ejemplo puede ser la fórmula **getType(exp: variant): string** que evalúa y retorna el tipo de una expresión.

El chequeo de tipos del lenguaje Lince es estático, a pesar de tener la desventaja que hace las fórmulas más complejas, debido a que el programador tiene ocuparse de especificar los tipos de datos en cada variable[11], el mismo permite comprobar en tiempo de compilación que en tiempo de ejecución no se van a producir errores de tipos, además de ganar eficiencia, ya que no es necesario realizar comprobaciones de tipo en la fase de ejecución.

En cuanto al control de ejecución, para simplicidad del lenguaje se pensó una forma secuencial para ellos donde se hacen uso de algunos operadores:

- **Operador secuencial** (punto y coma): toma dos instrucciones como argumentos, ejecuta la primera instrucción y al finalizar ésta, ejecuta la segunda.
    - Ejemplo: x = 1**;** x = x + 2
- **El operador condicional**: evalúa la condición y, si se cumple, ejecuta las instrucciones <instrución_verdadero>, si no se cumple, ejecuta <instrución_falso>

[11] El chequeo de tipos estático hace que los lenguajes sean menos flexibles pero más seguros.

- **if** condición **then** <instrución_verdadero>
  else <instrución_falso>
  end

- También se encuentra el **While-do**, **do-While**, **For** (ver sintaxis anexo 1)**.**

En el anexo 4 se puede encontrar la definición formal del lenguaje, la cual se elaboró usando la notación *Extended Backus Naur Form* (EBNF).

3.8 Conclusiones parciales

Luego de hacer un profundo estudio se obtuvieron un conjunto de requisitos que pretenden asegurar que se ha especificado un sistema que recoge las necesidades del cliente y satisface sus expectativas. Se ha determinado que el sistema cuenta con dos procesos principales los cuales contienen todos los requisitos encontrados, ambos procesos se han descrito como Casos de Uso, especificando en detalle los pasos que los componen. Es importante señalar que si bien el sistema cuenta con dos casos de uso, los algoritmos implicados en cada paso tienen una relevante complejidad. También se realizó un estudio sobre las principales propiedades de los lenguajes de programación y de dominio específico que permitieron elaborar un lenguaje a la altura de los requerimientos.

Capítulo 4: Construcción de la solución propuesta

Introducción

En el presente capítulo se hace una descripción de la construcción de la solución propuesta, así como de los patrones utilizados en la elaboración de la aplicación, los modelos de implementación y pruebas realizadas.

4.1 Arquitectura

La arquitectura del software es el proceso de definición de una solución estructurada que cumple con los requisitos técnicos y operativos, optimizando al mismo tiempo atributos de calidad tales como: rendimiento, seguridad y manejabilidad. Se trata de una serie de decisiones basadas en una amplia gama de factores, donde cada una de estas decisiones puede tener un impacto considerable en la calidad, rendimiento, facilidad de mantenimiento y, en general, el éxito de la aplicación (24).

A lo largo del desarrollo del componente evaluador de expresiones matemáticas se han tomado una serie de soluciones mediante patrones de diseño con la finalidad de que los cambios o factores adversos que pudieran surgir, así como modificaciones o nuevas funcionalidades a incluir en futuras versiones, permitieran una fácil expansión, mantenimiento y modificación.

En la propuesta de solución, se encuentran implícitos una serie de requerimientos que inciden directamente en la selección de un estilo arquitectónico para el desarrollo del componente, algunos de estos requerimientos son:

- **Abstracción**: Con el objetivo de disminuir la complejidad es necesario elegir determinados niveles de abstracción para satisfacer necesidades particulares. Por ejemplo, es correcto que el analizador sintáctico detecte un error de un carácter que no pertenece al lenguaje, pero este nivel de abstracción no sería apropiado si se tratase de encontrar que las cadenas estén en correcto orden.
- **Encapsulación**: Debe considerarse la posibilidad de cambios en los algoritmos que se emplean en la solución de algunos problemas, como la multiplicación de matrices o las estructuras de datos que utiliza la tabla de símbolos para almacenar la información de los mismos, con el objetivo de mejorar el rendimiento u otra propiedad del sistema. Por lo que estas características no

deben estar expuestas, así como ninguna parte del sistema debe depender de los detalles internos de otras partes.

- **Separación entre funcionalidades**: Este requerimiento está emparentado con la Alta cohesión y el Bajo acoplamiento, es importante que las funcionalidades del componente sean independientes, así cuando se necesite cambiar una funcionalidad no tenga efectos colaterales, lo que permite modificarlas o remplazarlas fácilmente.
- **Bajo acoplamiento y Alta cohesión**: Estos dos factores son muy importantes en cualquier sistema de software. Siempre están presentes, y prácticamente son uno de los indicadores de la calidad con la que se desarrolla el sistema. El bajo acoplamiento nos indica un modo de dar soporte a una dependencia escasa y un aumento de la reutilización, mientras que la Alta cohesión intenta mantener la complejidad dentro de límites manejables.
- **Reusabilidad**: Es una de las propiedades más deseables de los componentes de software, y permite usar el mismo componente en varios sistemas, reduciendo el tiempo de desarrollo y pruebas, por ende la calidad del producto final.
- **Rendimiento**: Esta es una de las cuestiones de importancia que se mide en un lenguaje de programación, generalmente se resumen en la rapidez con la que se ejecutan sus programas y la memoria que consumen.

Para dar respuesta a estos requerimientos la arquitectura estructural que posee el Componente Evaluador de fórmulas matemáticas está sustentada por el estilo arquitectónico en **capas**. Además de responder a los requerimientos es una de las técnicas más comunes en la elaboración de sistemas de software complejos. Un ejemplo de ello se observa en la arquitectura de máquinas (ordenadores), donde las capas descienden desde un lenguaje de programación con llamadas al sistema operativo, controladores de dispositivos (drivers) y un conjunto de instrucciones de la **CPU**, en puertas lógicas dentro de los chips, del mismo modo en el que los sistemas de redes existe una capa **FTP** sobre **TCP** que esta sobre la **IP** a la vez que esta se encuentra sobre la **Ethernet**, etc.

En general la arquitectura de sistema cuenta con un conjunto de capas lógicas, que en su parte superior presenta dos estratos, el primero a la izquierda se encarga de compilar los

programas escritos en código lince mientras que el otro estrato (a la derecha) está relacionado con la ejecución de fórmulas previamente compiladas, ambos se encuentran sobre una capa “Proveedor de Datos” cuya funcionalidad es abstraer los programas de su ubicación física, así como las fuentes de datos que se usan en las consultas.

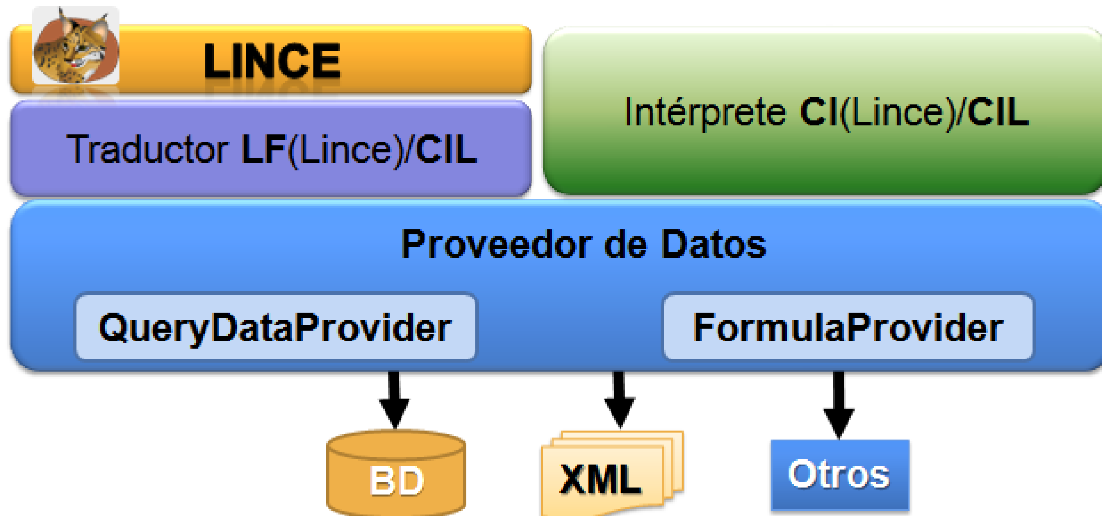

Figura 10: Arquitectura del Componente

4.2 Patrones

Los patrones de diseño son modelos de trabajo enfocados a dividir un problema en partes de modo que sea posible abordar cada una de ellas por separado para simplificar su resolución. Estos ayudan a construir el software basado en la experiencia colectiva de los ingenieros de software especializados.

4.2.1 Instancia Única (Singleton)

Existen algunas clases en el componente evaluador de expresiones matemáticas donde no existía la necesidad de crear más de una instancia, ya sea porque necesitaban compartirla entre varias clases como es el caso de **FormulaLoader** y **Configuration**, o porque dos instancias de una misma clase comparten el mismo estado y este nunca cambia, o sea la clase no posee métodos con efectos colaterales, como es el caso de las clases del paquete **TypeSystem**: **BoolType**, **MatrixType**, **NumericType**, **StringType**, **UndefinedType**, **VariantType** y **VoidType**, entre otras.

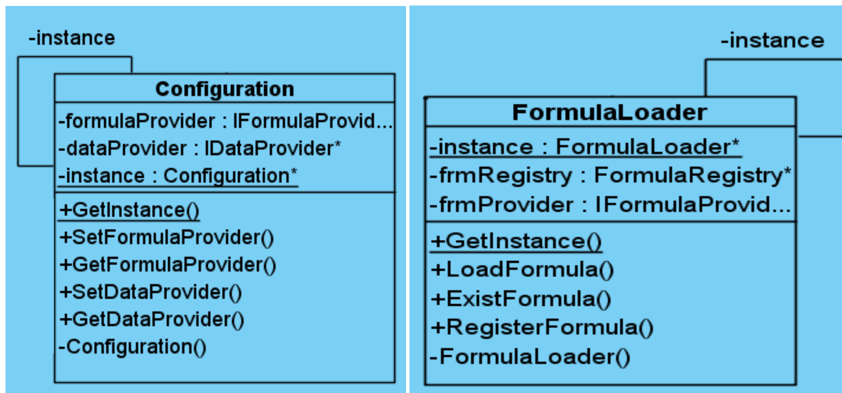

Figura 11: Clases que implementan el patrón Instancia Única

En este caso se decidió aplicar el patrón **Instancia Única**, que tiene como propósito garantizar que una clase sólo tenga una única instancia, proporcionando un punto de acceso global a la misma. Una variable global no previene de crear múltiples instancias de objetos. Una solución mejor es hacer que sea la propia clase la responsable de su única instancia, quien debe garantizar que no se pueda crear ninguna otra y proporcione un modo de acceder a ella. Esto se logró mediante la Implementación de un método estático **GetInstance()** en cada clase que implementa el patrón (ver figura 11).

De esta forma se solucionaron los problemas planteados, optimizando la cantidad de memoria innecesaria, y el peligro de que pudiese producirse algún lago de memoria por un mal manejo de la misma.

4.2.2 Patrón Fábrica Simple

El patrón Fábrica Simple, se usó para crear objetos a partir de un tipo enumerado. Esta es una buena técnica creacional de objetos, ya que se reutiliza el código de creación de los mismos que generalmente es un **switch** (en caso de C/C++) con varios casos según la cantidad de clases a crear. De la misma forma produce un código menos propenso a errores y más fácil de mantener en caso de tener que adicionar un nuevo tipo.

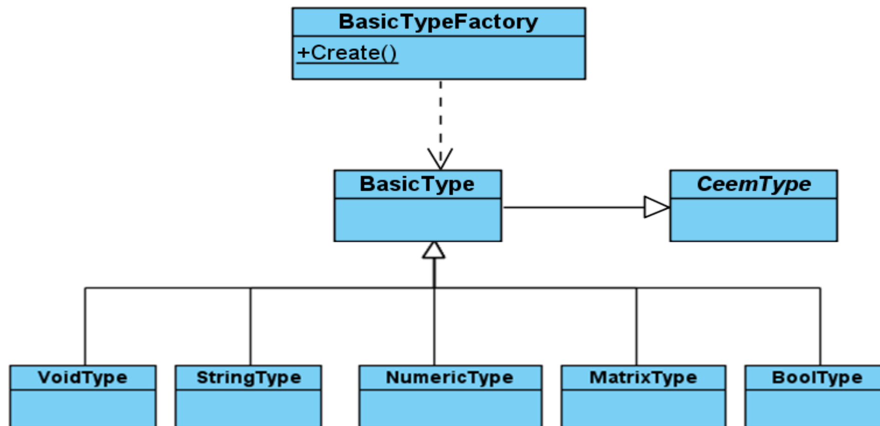

Figura 12: Clases que implementan el patrón Fábrica Simple

4.2.3 Patrón Visitador (Visitor)

El analizador semántico deberá utilizar el AST (obtenido por el analizador sintáctico) para llevar a cabo todas las comprobaciones necesarias y verificar la validez semántica del programa de entrada. Este proceso que también es conocido como Decoración del AST se realiza haciendo un recorrido sobre el mismo.

El AST es un árbol heterogéneo, una estructura de datos contiene muchas clases con diferentes interfaces, lo que hace difícil añadir nuevas funcionalidades, y estas se encuentran dispersas en los nodos del árbol. Con el objetivo de agrupar las funcionalidades y permitir definir y añadir un nuevo comportamiento sin necesidad de cambiar las clases de los elementos de la estructura del AST, se empleó el patrón Visitador en las clases encargadas de hacer el análisis semántico y ejecución, **TypeChecker** y **RuntimeEval** respectivamente.

Para su implementación se definió una clase abstracta **IVisitor** de la cual heredan los visitadores Concretos que implementan el patrón. De esta forma es fácil añadir nuevas operaciones al AST; sólo hay que crear un nuevo “Visitante”, en vez de cambiar todas las clases que se encuentren implicadas en la nueva funcionalidad.

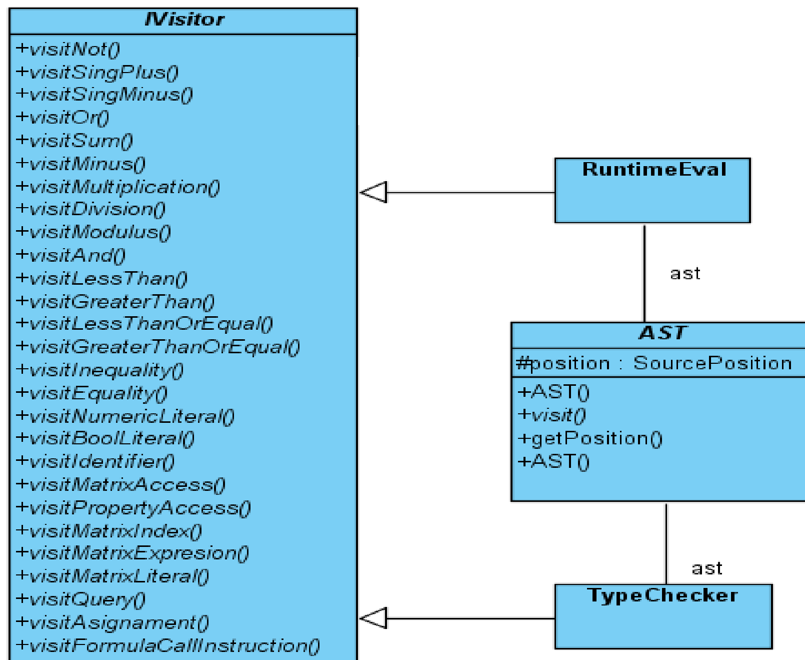

Figura 13: Clases que implementan el patrón Visitador

4.2.4 Patrón Estrategia (Strategy)

Con el motivo de hacer independiente el componente evaluador de expresiones matemáticas de las fuentes de datos para las consultas integradas en el lenguaje Lince y el almacenamiento de las Fórmulas en un medio físico, se usó el patrón Estrategia, este define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan. De esta forma se pueden implementar varias estrategias de obtención de datos sin que afecte las fórmulas.

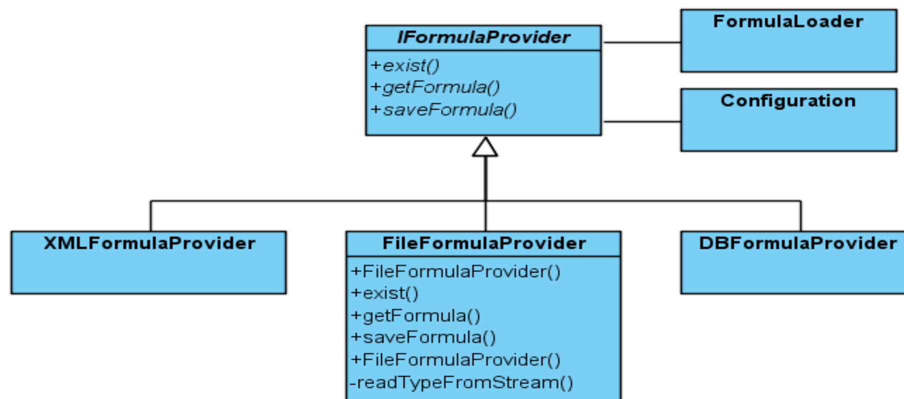

Figura 14: Clases que implementan el patrón Estrategia

4.2.5 Patrón Fachada (Facade)

El componente evaluador de fórmulas matemáticas contiene clases encargadas de realizar el análisis léxico, sintáctico, semántico, ejecución entre otras operaciones. No obstante solo algunas de las funcionalidades[12] de la aplicación que use el componente, necesitan acceder directamente a algunas de estas clases, mientras que para el resto se puede definir una interfaz general, que les permita sencillamente compilar sus programas.

Para dar solución a este problema se uso el patrón Fachada, el cual tiene como objetivo proporcionar una interfaz unificada de alto nivel representando a todo un subsistema. La "fachada" satisface a la mayoría de los clientes, sin ocultar las funciones de menor nivel a aquellos que necesiten acceder a ellas. El uso de este patrón trae como ventaja facilidad de uso del componente al reducir el número de objetos con los que el cliente trata, bajo acoplamiento entre el cliente y el componente lo que no constituye un obstáculo para usar otras clases del componente, permitiendo al cliente elegir entre facilidad de uso y generalidad.

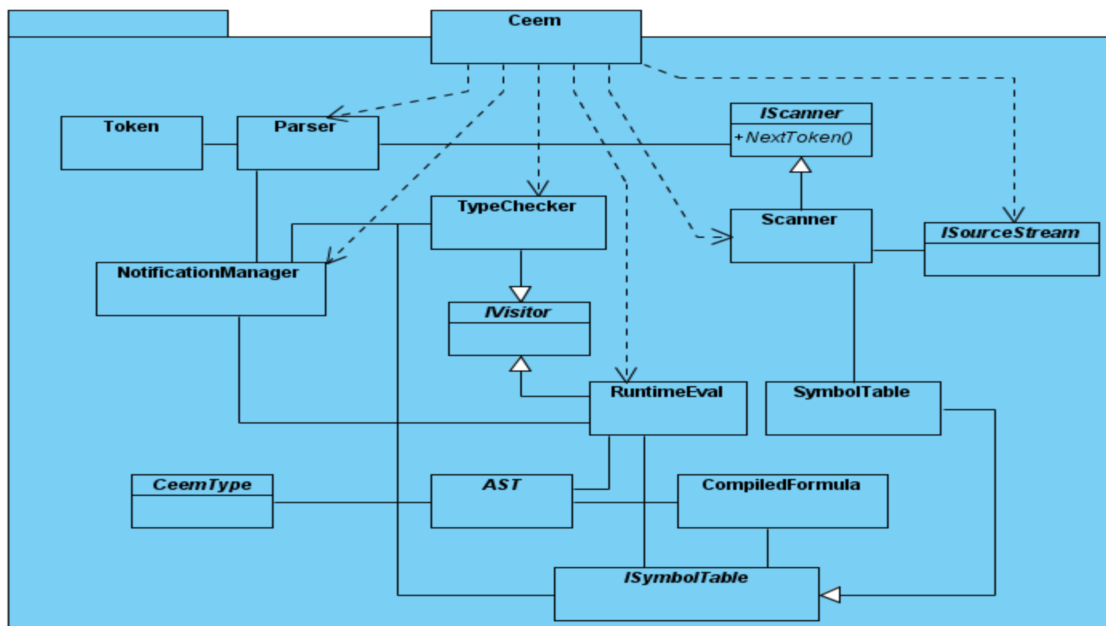

Figura 15: Uso del patrón Fachada

4.3 Diagrama de Paquetes

Con el objetivo de tener una mejor organización se decidió agrupar las clases en paquetes. Así como la arquitectura global del proyecto está compuesta por particiones

[12] Por ejemplo el coloreado de sintaxis puede concebirse haciendo uso del analizador léxico.

horizontales (capas), los paquetes pueden considerarse particiones dentro de una capa, y ofrecen la ventaja de separar los elementos detallados en abstracciones más amplias, lo cual brinda soporte a una vista de nivel superior y permite contemplar el modelo en agrupamientos más simples (25).

Para su elaboración se tuvieron en cuenta los siguientes criterios (25):

- Se agruparon los elementos para ofrecer un servicio común, intentando mantener un alto nivel de acoplamiento.
- Los paquetes tienen responsabilidades estrechamente relacionadas, lo que los hace cohesivos.
- Como consecuencia de los dos anteriores habrá relativamente bajo acoplamiento y colaboración entre los elementos de los paquetes.

Luego de aplicar los criterios antes mencionados, la distribución por paquetes del componente evaluador de fórmulas matemáticas queda de la siguiente manera:

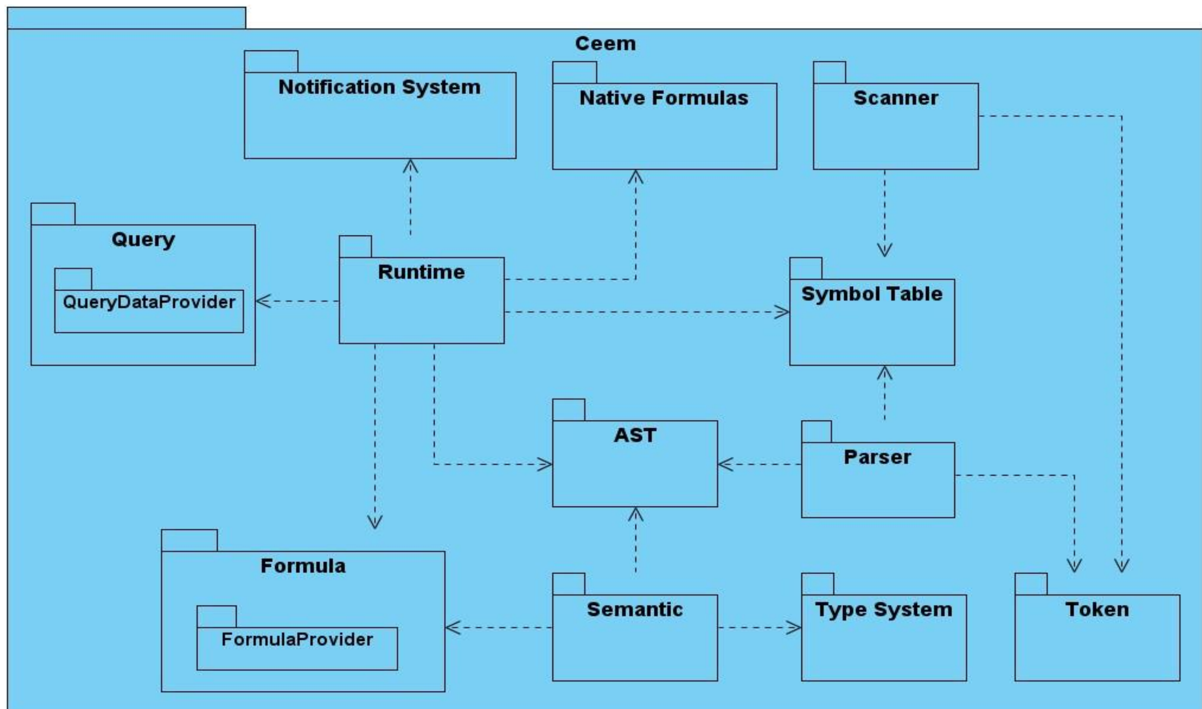

Figura 16: Diagrama de paquetes del Componente Evaluador de Fórmulas Matemáticas

Comentario sobre los paquetes y sus dependencias:

- **Ceem**: Es el paquete más general, este recoge todas las funcionalidades del componente.
- **SymbolTable**: Brinda la funcionalidad de almacenar información de los símbolos, ya sean variables o constantes.
- **Token**: Conjunto de clases que colaboran para almacenar la información de las unidades significativas denominadas Tokens, así como la posición que ocupan en el código del programa.
- **Scanner**: En este paquete se encuentran un conjunto de clases y tipos enumerados que brindan la funcionalidad de producir objetos de la clase Token a partir de los flujos de caracteres del programa que se esté analizando. El mismo depende del paquete SymbolTable ya que almacena los identificadores y las constantes literales a medida que son procesados.
- **Parser**: Valida que el programa este correctamente escrito, según la gramática definida para el lenguaje Lince, Al mismo tiempo crea el Árbol de Sintaxis Abstracta a partir de los Tokens que recibe del Scanner.
- **Semantic**: Realiza el análisis semántico al **AST**. decorándolo con los tipos de datos y otros atributos necesarios para la ejecución del programa.
- **TypeSystem**: Define el comportamiento de los tipos soportados por el lenguaje así como las operaciones que se pueden realizar entre los mismos.
- **Runtime**: Ejecuta el programa en forma de **AST** decorado devolviendo un resultado si es posible.
- **Formula y FormulaProvider**: Este paquete está relacionado con el almacenamiento y obtención de fórmulas desde un medio físico
- **Query y QueryDataProvider**: Provee al componente la funcionalidad de realizar consultas integradas al lenguaje Lince desde una fuente de datos.
- **NativeFórmulas**: Permite extender el lenguaje definiendo nuevas fórmulas en lenguaje nativo C/C++ que pueden ser usadas posteriormente desde programas Lince.
- **NotificationSystem**: Este paquete está relacionado con el almacenamiento de errores, tanto sintácticos como en tiempo de ejecución.

4.4 Diagrama de clases del diseño.

Para desarrollar una aplicación, también es necesario contar con descripciones detalladas y de alto nivel de la solución lógica y saber cómo satisface los requerimientos y las

restricciones. El Diseño pone de relieve una solución lógica: cómo el sistema cumple con los requerimientos (25 pág. 6). De la misma forma que se necesitan planos para construir un edificio, el software que es considerablemente más complejo, tiene sus planos, el modelo de diseño. La importancia del mismo se resume en la palabra <<calidad>>.

Un diagrama de clases del diseño muestra un conjunto de interfaces, colaboraciones y sus relaciones, principalmente se utilizan para modelar la vista de diseño estática de un sistema.

A continuación se muestra el diagrama de clases del diseño del componente evaluador de expresiones matemáticas, el mismo se encuentra fragmentado en partes para un mejor entendimiento, con el objetivo de seguir un hilo lógico, cada fragmento es consistente con un paquete del diagrama de paquetes del componente (ver figura 16). Para una mejor comprensión de los diagramas, puede consultar el anexo 3 que contiene la descripción de las clases que se muestran.

Diagrama de diseño del Paquete AST: Debido a las dimensiones del diseño de clases del AST la figura 17 muestra un **extracto** del diagrama de diseño obtenido, se ejemplifica con algunas clases la composición del mismo, se puede observar que existen dos grandes superclases: **Expression** de la cual heredan todas las expresiones del lenguaje como los identificadores, las operaciones unarias, y las binarias como las sumas multiplicaciones etc. y la superclase **SingleInstruction** de la cual descienden todas las instrucciones del lenguaje Lince.

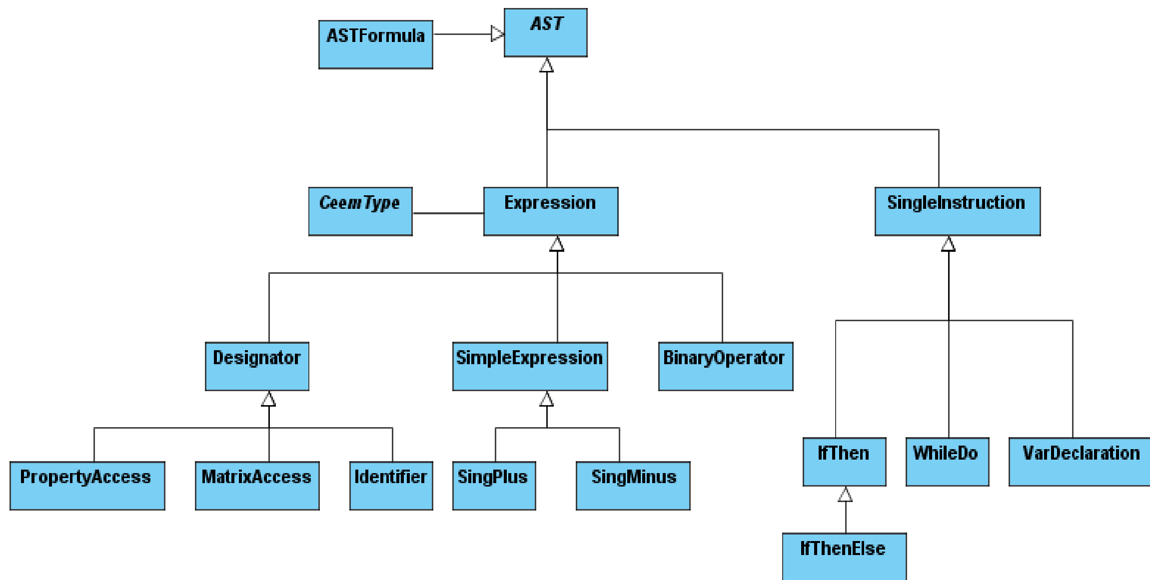

Figura 17: Diagrama de diseño de clases (paquete AST)

Diagrama de diseño del Paquete Scanner:

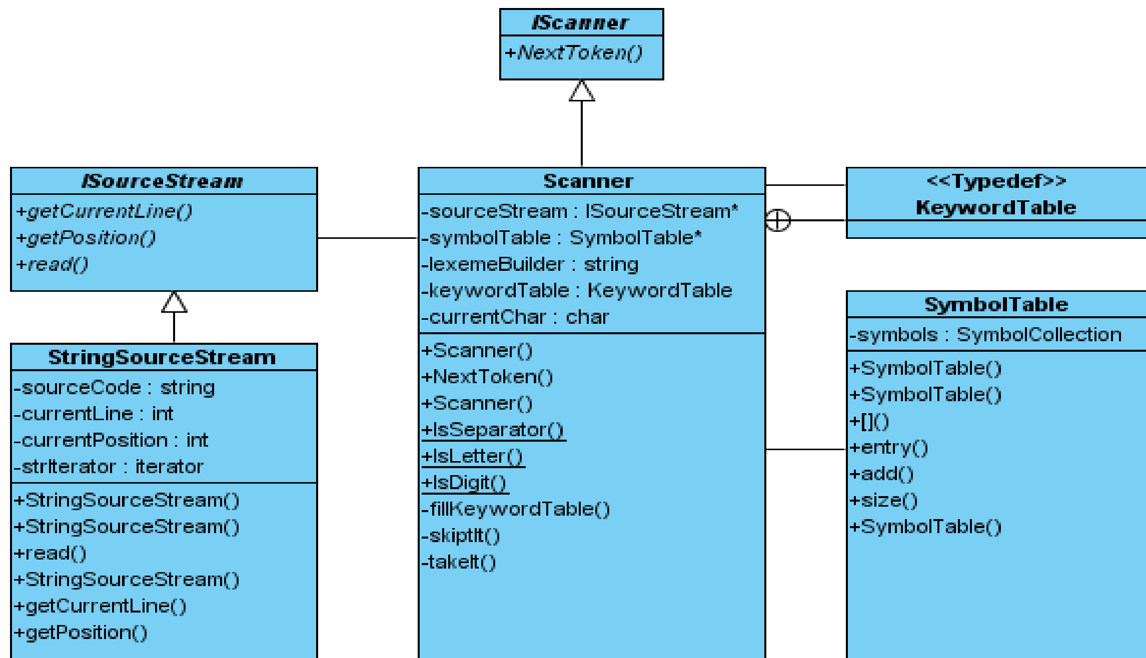

Figura 18: Diagrama de diseño de clases (paquete Scanner)

Diagrama de diseño del Paquete Parser:

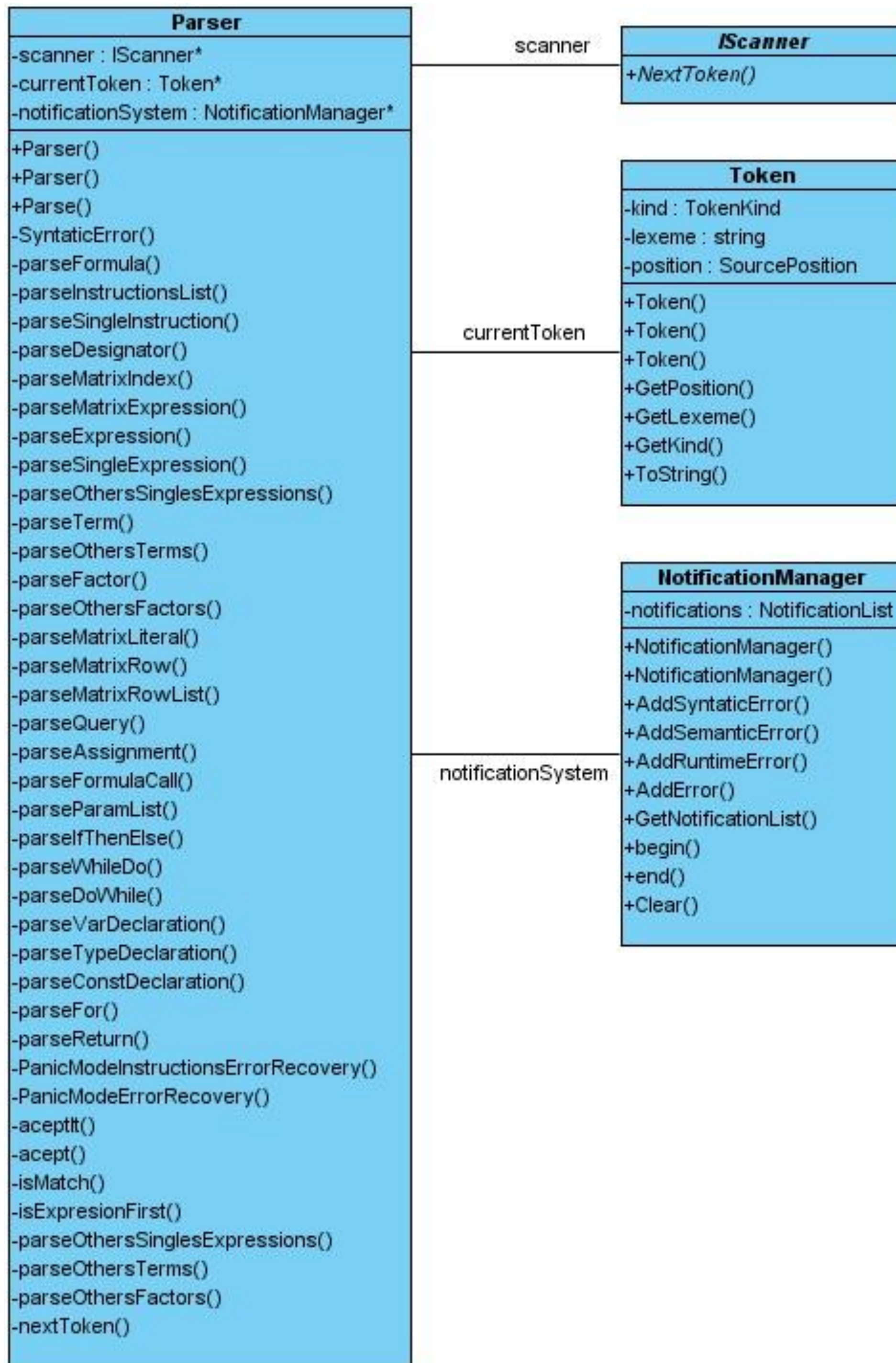

Figura 19: Diagrama de diseño de clases (paquete Parser)

Diagrama de diseño del Paquete Runtime:

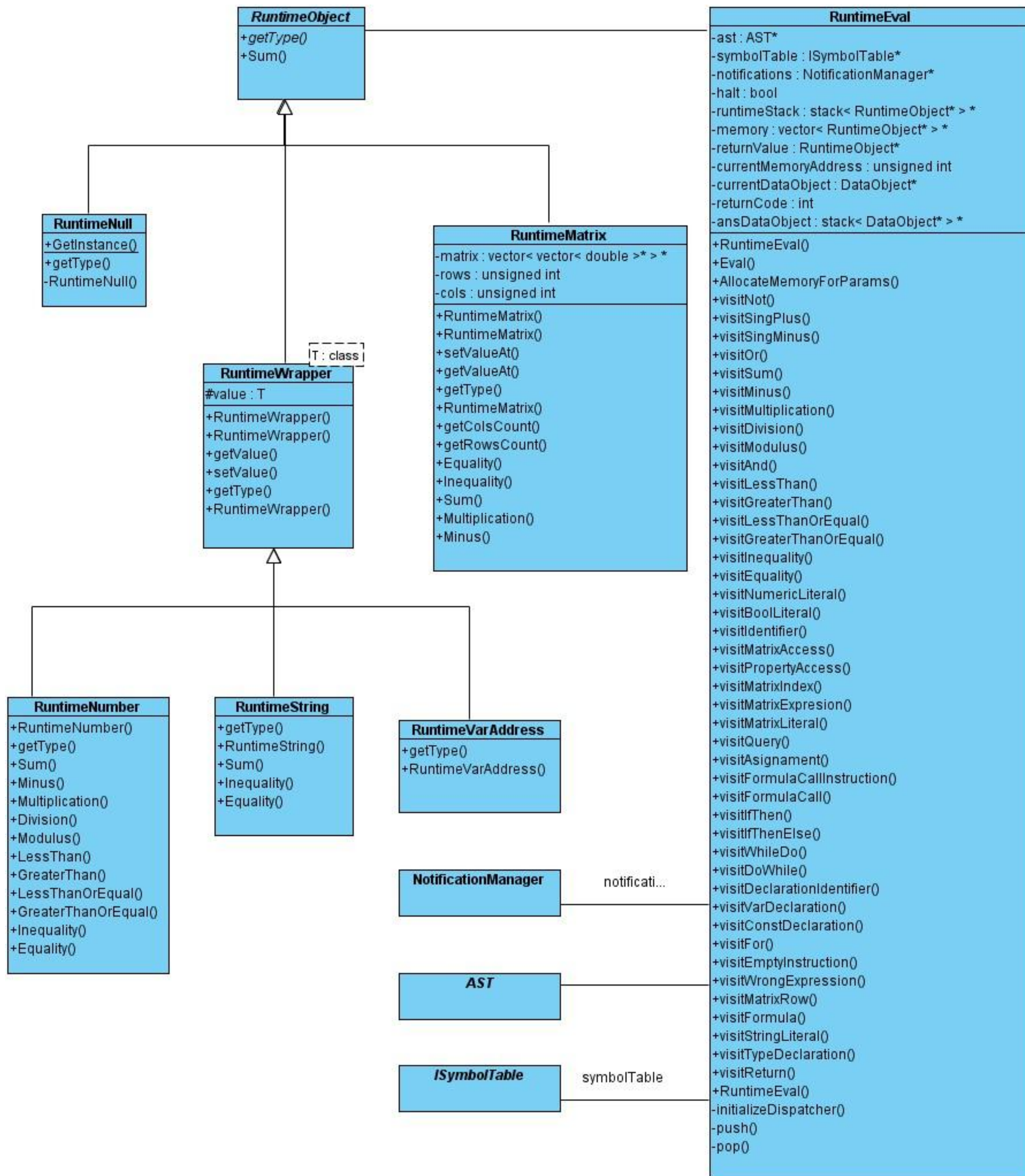

Figura 20: Diagrama de diseño de clases (paquete Runtime)

4.5 Diagramas de Interacción del Diseño (Secuencia)

El UML contiene diagramas de interacción que explican gráficamente como los objetos interactúan a través de mensajes para realizar las tareas (25). Un diagrama de secuencia muestra una interacción que está organizada como una secuencia temporal. En particular, muestra los objetos que participan en la interacción mediante sus líneas de vida y los mensajes que intercambian, organizados en forma de una secuencia temporal en determinado escenario de un caso de uso[13].

A continuación se muestra una generalización de los diagramas de secuencia correspondientes a los casos de uso **Compilar Fórmula** y **Evaluar Fórmula Compilada**.

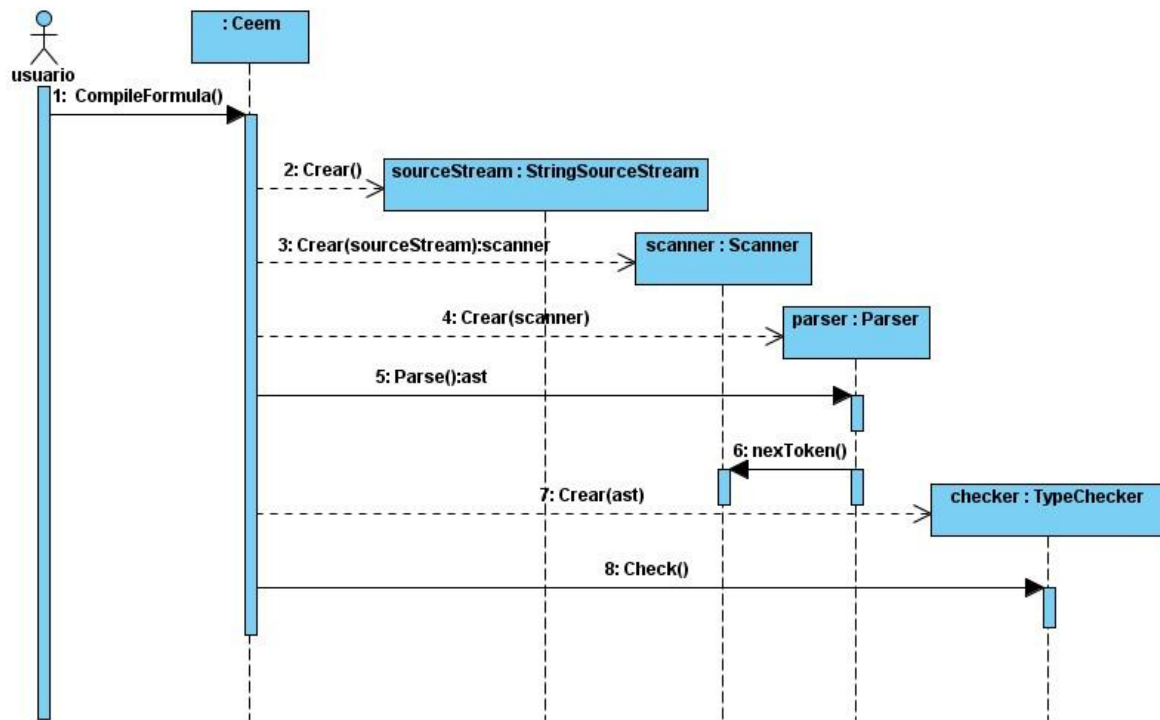

Figura 21: Diagrama de secuencia del caso de uso Compilar Fórmula.

[13] El escenario de un caso de uso es una instancia o trayectoria realizada por medio del uso: un ejemplo real de su ejecución.

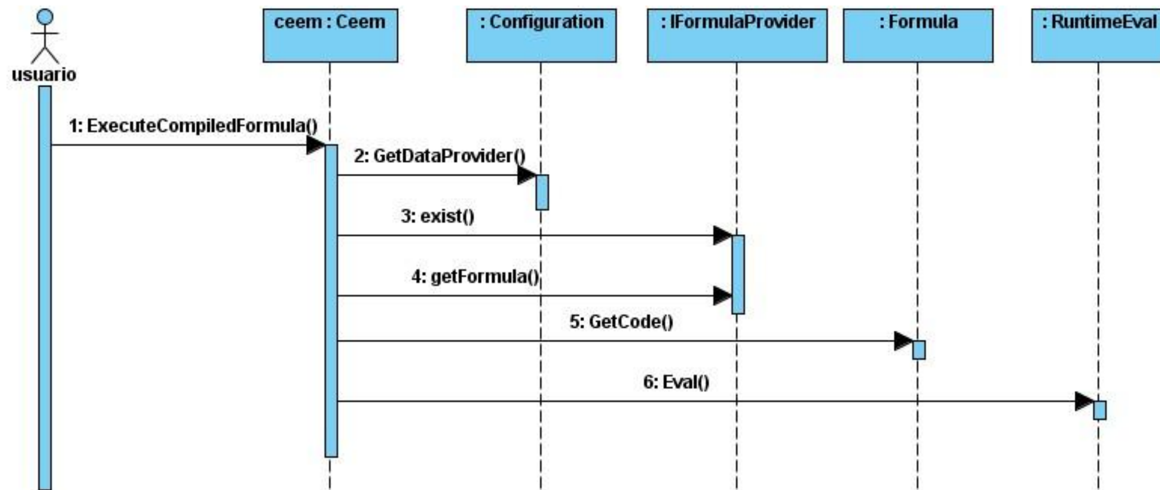

Figura 22: Diagrama de secuencia del caso de uso Evaluar Fórmula Compilada.

4.6 Validación del Componente

La complejidad inherente a los sistemas de software promueve innumerables posibilidades de que se produzcan situaciones erróneas, pueden darse resultados inesperados en las respuestas y comportamiento del mismo. La prueba y validación de los resultados no es un proceso que se realiza una vez desarrollado el software, sino que debe efectuarse en cada una de las etapas de desarrollo.

A continuación se muestran algunas de las pruebas realizadas a las funcionalidades del componente. Las pruebas que se muestran son de tipo **Caja Negra**, este tipo de pruebas están centradas en los requerimientos funcionales del software. Permiten derivar conjuntos de condiciones de entrada que ejerciten completamente todos los requerimientos funcionales de un programa.

Para realizar las pruebas se elaboró una interfaz visual que permite comprobar el funcionamiento del componente, así como datos experimentales en una fuente de datos. En el anexo 3 se puede observar algunas imágenes de la interfaz al realizarse las pruebas.

Caso de prueba: Reconocimiento y análisis de cadenas pertenecientes al lenguaje Lince.

Caso de Uso	Compilar Fórmula
Caso de prueba	Reconocimiento y análisis de cadenas pertenecientes al lenguaje Lince
Entrada	var cantidad: numeric = 0; var datos_alumnos : matrix;

	#Obtener nota de los
	datos_alumnos = from a in Alumnos
	where (a.matematica >= 3)
	&& (a.fisica >= 3) && (a.quimica >= 3)
	select [a.matematica, a.fisica, a.quimica,
	(a.matematica + a.fisica + a.quimica) /3];
	
	call var_dump(size(datos_alumnos)); #mostrar la cantidad de filas y columnas obtenidos de la consulta
	
	call var_dump(datos_alumnos); #mostrar los datos de los alumnos obtenidos de la consulta
	
	cantidad = rows_size(datos_alumnos); #cantidad de filas
	
	if(cantidad >= 3) then call var_dump("Existen " + cantidad + " alumnos aporbados"); end

Se espera	La fórmula no tiene errores.
Resultado	La fórmula no tiene errores.

Caso de prueba: Recuperarse de los errores de compilación y seguir el proceso y devolución de los errores encontrados.

Caso de Uso	Compilar Fórmula
Caso de prueba	Recuperarse de los errores de compilación y seguir el proceso y devolución de los errores encontrados.

Entrada	var var_01 : numeric = 1;
	var var_02 : boolean = true;
	var var_03: string ;
	var var_04: string;
	
	var_04 = var_03 + var_01;
	
	if var_no_declarada < 10 then
	var_02 = "No se puede asignar una cadena a un boolean";
	end
	
	var sintipo = 100;
	
	var_01 = 100 + sintipo;

Se espera	Error en la línea: 12: Se Esperaba: ':' y se encontró: '=' Error en la línea: 12: Se espera un tipo de variable

	Error en la línea: 6: Uso de la variable local no asignada Error en la línea: 8: Identificador 'var_no_declarada' no declarado Error en la línea: 8: Uso de la variable local no asignada Error en la línea: 8: Operación de comparación no admitida Error en la línea: 8: La condición tiene que ser un tipo boolean Error en la línea: 9: Los Tipos no coinciden Error en la línea: 14: Uso de la variable local no asignada Error en la línea: 14: No se puede convertir de tipo Undefined a Numeric: operación '+' no admitida entre tipos incompatibles Error en la línea: 14: Los Tipos no coinciden
Resultado	El resultado que se obtuvo fue el esperado.

Caso de prueba: Almacenamiento de fórmulas previamente compiladas y obtención de los datos desde una fuente de datos mediante consultas integradas al lenguaje Lince.

Para realizar esta se ejecutó la fórmula compilada en el caso de prueba anterior "caso_de_prueba1.lince". Además se implementó una fórmula "var_dump" que imprime el tipo y contenido de una variable o una expresión en la consola.

Caso de Uso	Ejecutar Fórmula
Caso de prueba	Almacenamiento de fórmulas previamente compiladas y obtención de los datos desde una fuente de datos mediante consultas integradas al lenguaje Lince.

Entrada	var cantidad: numeric = 0;
	var datos_alumnos : matrix;
	#Obtener nota de los
	datos_alumnos = from a in Alumnos
	where (a.matematica >= 3)
	&& (a.fisica >= 3) && (a.quimica >= 3)
	select [a.matematica, a.fisica, a.quimica,
	(a.matematica + a.fisica + a.quimica) /3];
	
	call var_dump(size(datos_alumnos)); #mostrar la cantidad de filas y columnas obtenidos de la consulta
	
	call var_dump(datos_alumnos); #mostrar los datos de los alumnos obtenidos de la consulta
	
	cantidad = rows_size(datos_alumnos); #cantidad de filas
	
	if(cantidad >= 3) then call var_dump("Existen " + cantidad + "

	alumnos aprobados"); end
Se espera	matrix: [3 4] matrix: [5 ,3 ,5 ,4.33333 3 ,4 ,3 ,3.33333 4 ,4 ,5 ,4.33333] string(Existen 3 alumnos aprobados)
Resultado	El resultado que se obtuvo fue el esperado.

Se debe ejecutar el programa antes de que llegue al cliente, con la intención especifica de descubrir todos los errores, de manera que el cliente no experimente la frustración asociada con un producto de baja calidad. Con el propósito de encontrar el mayor número posible de errores, las pruebas deben conducirse sistemáticamente, y los casos de prueba deben ser designados mediante técnicas disciplinadas (15 pág. 407).

Las pruebas realizadas al componente obtuvieron resultados satisfactorios, concluyendo que es un producto de calidad y apto para el uso.

4.7 Conclusiones parciales

Luego de un estudio de los requerimientos del componente evaluador de expresiones matemáticas, se concluyó que debía usarse una arquitectura en capas para dar respuesta a los mismos y obtener un producto estable de alto rendimiento. Se aplicaron un grupo de patrones de diseño con el objetivo de alcanzar soluciones factibles, así como para dar soporte al desarrollo, mantenimiento y evolución del componente. También se abordaron todos los temas referentes a la implementación de la solución y a la estrategia de pruebas a seguir durante la elaboración de la misma.

Capítulo 5: Estudio de la Factibilidad.

5.1 Introducción.

Al estudiar la factibilidad de un producto, ya sea de software o cualquier otra índole, se obtendrá como resultado una estimación del tamaño del producto, el esfuerzo necesario para desarrollarlo en un tiempo determinado, el presupuesto que requiere para su construcción y los recursos humanos y materiales utilizados en el mismo. En esta investigación se propone como variante de estimación para calcular todos estos factores, el Análisis de Puntos de Casos de Uso, que dará solución a todas estas interrogantes.

5.2 Análisis de Puntos de Casos de Uso.

El Análisis de Puntos de Casos de uso se trata de un método de estimación del período de desarrollo de un proyecto, en el cual se asignan valores numéricos a varios factores que afectan el progreso del mismo, para así de esta forma, contabilizar el tiempo total estimado. Con este método se obtiene un aproximado del esfuerzo en horas-hombre, teniendo en cuenta solamente la funcionalidad especificada en cada caso de uso.

Para la aplicación de este método se desarrollan una serie de pasos, tales como:

1. Calcular los Puntos de Casos de Uso sin ajustar.
2. Calcular los Puntos de Casos de Uso ajustados.
3. Estimar el esfuerzo a través de los puntos de casos de uso.
4. Calcular el esfuerzo del flujo de trabajo Implementación. Calcular el Esfuerzo Total de todo el proyecto.
5. Calcular el costo del proyecto.

1. Para calcular los Puntos de Casos de Uso sin ajustar se utiliza la siguiente fórmula**:**

UUCP = UAW +UUCW

Donde:

UUCP: Son los Puntos de Casos de Uso sin ajustar.

UAW: Es el factor de Peso de los Actores sin ajustar.

UUCW: Es el factor de Peso de los Casos de Uso sin ajustar.

Para calcular el factor de Peso de los Actores sin ajustar (**UAW**), es necesario tener en cuenta la cantidad de actores con que cuenta el sistema y el nivel de complejidad de los mismos.

Tipo de Actor	Descripción	Factores de peso	Actores	Total (Fact* CU)
Simple	Otro sistema que interactúa con el sistema a desarrollar mediante una interfaz de programación (API).	1	1	1
Medio	Otro sistema que interactúa con el sistema a desarrollar, mediante un protocolo o una interfaz basada en texto.	2	0	0
Complejo	Una persona que interactúa con el sistema mediante una interfaz gráfica.	3	0	0

Tabla 13. Factor de peso de los actores sin ajustar.

Se calcula mediante la siguiente ecuación:

UAW = $\sum (Factor_i * Actores_i) = 1$

Para calcular el Factor de Peso de los Casos de Uso sin ajustar (**UUCW**) es necesario tener en cuenta la cantidad de casos de uso con que cuenta el sistema y su nivel de complejidad. O sea cuantos casos de usos cumplen con la categoría (simple, medio y complejo), para esto se necesita saber la cantidad de transacciones con que cuenta cada caso de uso. Las transacciones se obtienen de la descripción de los casos de uso, cada acción del autor y la respectiva respuesta del sistema es una transacción.

Tipo de CU	Descripción	Factores de peso	Cantidad de CU	Total (Fact*CU)
Simple	Los casos de uso tienen de 1 a 3 transacciones.	5	2	10
Medio	Los casos de uso tienen de 4 a 7 transacciones.	10	0	0
Complejo	Los casos de uso tienen 8 o más transacciones.	15	0	0

Tabla 14. Factor de peso de los casos de uso sin ajustar.

Se calcula mediante la siguiente ecuación:

UUCW = $\sum (Factor_i * \text{Cantidad de } CU_i) = 10$

Entonces ya se pueden obtener los Puntos de Casos de Uso sin ajustar sustituyendo los valores en la fórmula:

UUCP = UAW +UUCW

UUCP = 1 +10 **=** 11

2. Para obtener los Puntos de Casos de Uso ajustados se utiliza la siguiente fórmula:

UCP= UUCP * TCF * EF

Donde:

UCP: Son los puntos de Casos de Uso ajustados.

UUCP: Son los puntos de Casos de Uso sin ajustar.

TCF: Es el factor de complejidad técnica.

EF: Es el factor de ambiente.

Para calcular el Factor de complejidad técnica (TCF) se utiliza la fórmula:

TCF = $0.6 + 0.01 * \sum (Peso_i * Valor_i)$

Donde Valor es un número del 0 al 5 con los siguientes significados:

0: No presente o sin influencia.

1: Influencia incidental o presencia incidental.

2: Influencia moderada o presencia moderada.

3: Influencia media o presencia media.

4: Influencia significativa o presencia significativa.

5: Fuerte influencia o fuerte presencia.

Factor	Descripción	Peso	Valor	Total (P*V)
F1	Sistema distribuido.	2	0	0
F2	Tiempo de respuesta.	1	5	5
F3	Eficiencia del usuario final.	1	1	1
F4	Procesamiento interno complejo.	1	5	5
F5	El código debe ser reutilizable.	1	5	5
F6	Facilidad de instalación.	0.5	0	0
F7	Facilidad de uso.	0.5	4	2
F8	Portabilidad.	2	4	8
F9	Facilidad de cambio.	1	5	5
F10	Concurrencia.	1	0	0
F11	Incluye objetivos especiales de seguridad.	1	0	0
F12	Provee acceso directo a terceras partes.	1	0	0
F13	Se requieren facilidades especiales de entrenamiento para los usuarios.	1	3	3

Total	34

Tabla 15. Factor de complejidad técnica.

Ahora se obtiene el factor de complejidad técnica sustituyendo los valores en la fórmula:

TCF = 0.6 + 0.01 * 34 = 0.94

Para calcular el Factor de Ambiente (EF) se utiliza la fórmula:

EF = 1.4 - 0.03 * $\sum$ ($Peso_i$ * $Valor_i$)

Donde Valor es un valor numérico del 0 al 5 con las mismas características que la fórmula anterior.

Factor	Descripción	Peso	Valor	Total (Peso*Valor)
F1	Familiaridad con el modelo de proyecto utilizado.	1.5	4	6
F2	Experiencia en la aplicación.	0.5	4	2
F3	Experiencia en la orientación a objetos.	1	5	5
F4	Capacidad del analista líder.	0.5	4	2
F5	Motivación.	1	5	5
F6	Estabilidad de requerimientos.	2	4	8
F7	Personal Part–Time.	-1	2	-2
F8	Dificultad del lenguaje de programación.	-1	4	-4
Total				22

Tabla Factor Ambiente

Al sustituir el valor obtenido en la fórmula se obtiene el factor de complejidad técnica:

EF = 1.4 - 0.03 * 22 = 0.74

Finalmente se tiene que:

UCP = UUCP * TCF * EF

UCP = 11 * 0.94 * 0.74 **= 7.6516**

3. Estimar el esfuerzo a través de los puntos de casos de uso

Para obtener el esfuerzo estimado en horas-hombre se hace uso de la fórmula:

E = UCP * CF

Donde:

E: Esfuerzo (horas-hombre).

UCP: Puntos de Casos de Uso ajustados.

CF: Factor de conversión.

Donde el Cálculo del Factor de conversión (CF).

CF = 20 horas-hombre (si Total EF ≤ 2)

CF = 28 horas-hombre (si Total EF = 3 ó Total EF = 4)

CF = abandonar o cambiar proyecto (si Total EF ≥ 5)

Como EF = 0.74 entonces CF = 20 horas-hombre.

Finalmente se tiene que:

E = UCP * CF = 7.6516* 20 **=** 153.032 horas-hombre

4. Calcular el Esfuerzo Total de todo el proyecto.

Actividad	Porciento	Horas-Hombre
Análisis	20	76,516
Diseño	25	95.645
Implementación	40	153.032
Prueba	15	57.387
Total	**100**	**382,58**

Tabla Distribución del esfuerzo entre las diferentes actividades.

El valor del esfuerzo calculado representa la implementación, los demás resultados se obtienen al sustituir los valores en la fórmula:

$$\frac{Parte}{Total} = \frac{\%}{100}$$

Suponiendo que se tienen 48 horas de trabajo semanales, pero en el caso del Componente Evaluador de Expresiones, son dos personas trabajando en la misma computadora, por lo que serían 24 horas semanales, entonces asumiendo que un mes tiene 4 semanas, la cantidad de horas mensuales de trabajo serian 96.

Si el esfuerzo total es de 382.58 horas-hombre y por cada 96 horas es un mes, entonces daría un esfuerzo total del proyecto de 3.98 mes-hombre.

5. Costo del producto.

Para obtener el costo del producto se hace uso de la fórmula:

Costo = CHM x ET

Donde:

ET: Esfuerzo total (mes-hombre), **CHM:** Costo hombre-mes

Para calcular el costo hombre-mes.

Donde:

SBM: Salario básico mensual

Salario básico de una persona: SBM = $100

CH: Cantidad de Hombres, **CH =** 2

Se calcula mediante la fórmula:

CHM = CH x SBM **=** 2 x $100 **=** $ 200

Finalmente se tiene que:

Costo = CHM x ET **=** $ 200 x 3.98

Costo = $ 796 CUP.

Costo = $ 31.8 CUC.

5.3 Análisis de costos y beneficios

Para desarrollar este componente no se requiere de grandes consumos de recursos, ni tampoco de mucho tiempo para su construcción (un costo de 796 pesos y casi 4 meses), por lo que se hace factible su desarrollo en vista a los beneficios que reporta para el progreso de futuras aplicaciones.

5.4 Conclusiones Parciales

El estudio de la factibilidad de un software o proyecto constituye una de las bases más importantes para evaluar el trabajo realizado y es una valiosa herramienta que permite establecer el alcance del mismo, y los diferentes aspectos que se deben tener en cuenta al hacer un análisis o una evaluación económica del proyecto.

En este capítulo se han detallado los costos necesarios para el desarrollo del componente en cuestión, los recursos humanos implicados, el tiempo de desarrollo, y otros importantes factores, llegando a la conclusión de que es factible la realización de esta aplicación debido a los bajos costos que genera su producción y el gran alcance que tendrá, aportando robustez y escalabilidad a futuras aplicaciones.

Conclusiones Generales

Luego de finalizar el desarrollo del presente trabajo, se concluye que se han alcanzado satisfactoriamente todos los objetivos propuestos. Se obtuvo un lenguaje de especificación de fórmulas que permite modelar un problema matemático sin amplios conocimientos de programación. También se combinaron satisfactoriamente métodos de matemática numérica con el intérprete elaborado, permitiendo usar complejos algoritmos como la multiplicación entre matrices de forma natural, como si fuesen valores numéricos cualesquiera. Además se ha comprobado la capacidad de extensión de la solución, la cual permite al componente adaptarse a diferentes entornos y usos.

Recomendaciones

Al concluir esta investigación y después de haber alcanzado los objetivos planteados se recomienda:

- Continuar el estudio de los algoritmos de cálculo numérico usados en las operaciones del componente (como multiplicación de matrices, resolución de ecuaciones, etc.)
- Incluir la fase de optimización en el intérprete para mejorar el tiempo de ejecución de las fórmulas.
- Continuar el análisis de nuevas formas intermedias con el objetivo de optimizar el proceso de interpretación.
- Agregar nuevas fórmulas predeterminadas para aumentar el alcance del componente.
- Utilizar la documentación generada como material de consulta, en caso de futuras actualizaciones de software.

Trabajos citados

1. **Dopico, Froilán M.** Matemáticas y sus Fronteras. *Matemática Computacional: Un nuevo pilar para el desarrollo científico y tecnológico.* [En línea] 25 de junio de 2007. [Citado el: 02 de diciembre de 2009.] http://weblogs.madrimasd.org/matematicas/archive/2007/06/25/68571.aspx.

2. **Hernández León, Rolando Alfredo y Coello González, Sayda.** *EL PARADIGMA CUANTITATIVO DE LA INVESTIGACIÓN CIENTIFICA.* La Habana : Editorial Universitaria, 2002. 959-16-0343-6.

3. **Team, XoYo Software Devom.** Devom - Give Power To Software Development - #Calculation Component - Powerful Calculation Engine And Math Expression Parser. [En línea] 2007. [Citado el: 18 de febrero de 2010.] http://www.devom.com/.

4. **Universidad de las Ciencias Informáticas.** Los paradigmas y la historia de algunos lenguajes de programacion. *Entorno virtual de Aprendizaje.* [En línea] 5 de octubre de 2009. [Citado el: 10 de diciembre de 2009.] http://eva.uci.cu/file.php/258/Bibliografia_Basica/Tema_2/TEMA_2_Los_paradigmas_y_la_historia_de_algunos_lenguajes_de_programacion.pdf.

5. RENa - Cuarta etapa - Informática - Lenguajes de programación. *Red Escolar Nacional - RENa.* [En línea] RENa, 2008. [Citado el: 18 de febrero de 2010.] http://www.rena.edu.ve/cuartaEtapa/Informatica/Tema13.html.

6. **Labra Gayo, José Emilio, y otros.** *Intérpretes y Diseño de Lenguajes de Programación.* 2004.

7. **Seta, Leonardo De.** Los lenguajes específicos de dominio. *http://www.dosideas.com.* [En línea] 27 de Marzo de 2009. [Citado el: 14 de Diciembre de 2009.] http://www.dosideas.com/actualidad/487-los-lenguajes-especificos-de-dominio.html.

8. **Ortín Soler, Francisco, y otros.** *Análisis Semántico en Procesadores de Lenguaje.* Oviedo : SERVITEC, 2004. 84-688-6208-8.

9. **Gálvez Rojas, Sergio y Mora Mata, Ángel.** *Java a tope: Traductores y Compiladores con LEX/YACC , JFLEX /CUP y JAVACC.* s.l. : Universidad de Málaga, 2005. 84-689-1037-6.

10. **Louden, Kenneth C.** *UNED Construccion de compiladores principios y practica.* s.l. : THOMSON PARANINFO,S.A., 2004. 9789706862990.

11. **Aho, Alfred V., Sethi, Ravi y Ullman, Jeffrey.** *COMPILADORES: PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS.* s.l. : PEARSON ADDISON-WESLEY, 1990. 9789684443334.

12. **Booch, Grady, y otros.** *Object-Oriented analysis and design with applications .* s.l. : Addison Wesley Professionals , 2007.

13. **Kasoft Software.** Java: Trees Versus Bytes. [En línea] Kasoft Software, 4 de Abril de 2005. [Citado el: 9 de 2 de 2010.] http://central.kaserver5.org/Kasoft/Typeset/JavaTree/Pt06.html#Head2869.

14. **Rumbaugh, James, Booch, Grady y Jacobson, Ivar.** *El lenguaje unificado de modelado.Manual de Referencia.* Madrid : Pearson Educación, 2000.

15. **Pressman, Roger S.** *Ingeniería del Software. Un enfoque práctico.* España : McGraw-Hill, 2002. 8448132149.

16. **Molpeceres, Alberto.** *Procesos de desarrollo: RUP, XP, FDD.* 2002.

17. Soluciones y Propuestas Rational. *Soluciones y Propuestas Rational.* [En línea] [Citado el: 9 de Febrero de 2010.] http://www.rational.com.ar/herramientas/rup.html.

18. DTIC-CGI. *Introduction to the Agile Up.* [En línea] 13 de mayo de 2006. [Citado el: 9 de Febrero de 2010.] http://cgi.una.ac.cr/AUP/html/overview.html.

19. Área temática de Java. Portal de Java - Ciberaula. *Tecnología Orientada a Objetos.* [En línea] 2006. [Citado el: 9 de Febrero de 2010.] http://java.ciberaula.com/articulo/tecnologia_orientada_objetos/.

20. **Schildt, Herbert.** *C++, guia de autoenseñanza.* España : McGRAW HILL, 2001. 0-07-882025-1.

21. Welcome to NetBeans. *NetBeans IDE 6.8 Release Information.* [En línea] 5 de Enero de 2010. [Citado el: 8 de febrero de 2010.] http://netbeans.org/community/releases/68.

22. UML tool, business process modeler and database designer for software development team. *UML tool for application design, documentation and code generation.* [En línea] [Citado el: 10 de Febrero de 2010.] http://www.visual-paradigm.com/product/vpuml/.

23. **Valdez Altamirano, Alfonso.** *Comparativo de Entornos de Desarrollo Integrados.*

24. *Application Architecture Guide.* s.l. : Microsoft Corporation., 2009. 9780735627109.

25. **Larman, Craig.** *UML y Patrones.* Mexico : PRENTICE HALL, 1999. 970-1 7-0261-1.

26. **Biografias y Vidas, S.C.P.** Biografias y Vidas. [En línea] [Citado el: 10 de Diciembre de 2009.] http://www.biografiasyvidas.com/biografia/b/babbage.htm.

27. **Universidad de las Ciencias Informáticas.** El nacimiento de la Computacion. *Entorno Virtual de Aprendizaje.* [En línea] 4 de mayo de 2009. [Citado el: 10 de diciembre de 2009.] http://eva.uci.cu/file.php/258/Bibliografia_Basica/Tema_1/El_nacimiento_de_la_Computacion.pdf.

28. **Universidad de las Ciencias Informáticas.** Entorno virtual de Aprendizaje. [En línea] 5 de octubre de 2009. [Citado el: 10 de diciembre de 2009.]

http://eva.uci.cu/file.php/258/Bibliografia_Basica/Tema_2/TEMA_2_Los_paradigmas_y_la_historia_de_algunos_lenguajes_de_programacion.pdf.

29. **Campbell, Christopher.** Lambda the Ultimate. [En línea] 18 de 05 de 2005. [Citado el: 05 de 02 de 2010.] http://lambda-the-ultimate.org/node/716.

30. **AIFTOP.** Asociación Ibérica de Fabricantes y Comerciantes de Transmisiones Oleo-hidráulicas Neumáticas. [En línea] [Citado el: 9 de Febrero de 2010.] http://www.aeftop.es/index.php?id_seccio_menu=33.

31. **Zayas, Carlos Alvarez de.** *METODOLOGIA DE LA INVESTIGACION CIENTIFICA.* SANTIAGO DE CUBA : CENTRO DE ESTUDIOS DE EDUCACION SUPERIOR "MANUEL F. GRAN", 1995.

32. **López, Angel.** *JAVA la programación del futuro.* Buenos Aires : MP Ediciones, 1997.

33. **Larman, Craig.** *An Agile Up: Introduction.* 2002.

34. Rational Rose Enterprise es el producto más completo de la familia Rational Rose, incluye soporte de Unified Modeling Language. *Soluciones y Propuestas RATIONAL, Servicios, Capacitación, Consultoría; Outsorcing de Testing - Software y Hardware IBM.* [En línea] Gsinnova. [Citado el: 3 de Marzo de 2010.] http://www.rational.com.ar/herramientas/roseenterprise.html.

35. **Ángel, Miguel.** Compiladores - Monografias.com. *Monografias.com - Tesis, Documentos, Publicaciones y Recursos Educativos.* [En línea] [Citado el: 20 de febrero de 2010.] http://www.monografias.com/trabajos11/compil/compil.shtml.

Bibliografía

Dopico, Froilán M. Matemáticas y sus Fronteras. *Matemática Computacional: Un nuevo pilar para el desarrollo científico y tecnológico.* [En línea] 25 de junio de 2007. [Citado el: 02 de diciembre de 2009.] http://weblogs.madrimasd.org/matematicas/archive/2007/06/25/68571.aspx.

Labra Gayo, José Emilio, y otros. *Intérpretes y Diseño de Lenguajes de Programación.* 2004.

Ortín Soler, Francisco, y otros. *Análisis Semántico en Procesadores de Lenguaje.* Oviedo : SERVITEC, 2004. 84-688-6208-8.

Gálvez Rojas, Sergio y Mora Mata, Ángel. *Java a tope: Traductores y Compiladores con LEX/YACC , JFLEX /CUP y JAVACC.* s.l. : Universidad de Málaga, 2005. 84-689-1037-6.

Louden, Kenneth C. *UNED Construccion de compiladores principios y practica.* s.l. : THOMSON PARANINFO,S.A., 2004. 9789706862990.

Aho, Alfred V., Sethi, Ravi y Ullman, Jeffrey. *COMPILADORES: PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS.* s.l. : PEARSON ADDISON-WESLEY, 1990. 9789684443334.

Booch, Grady, y otros. *Object-Oriented analysis and design with applications .* s.l. : Addison Wesley Professionals , 2007.

Pressman, Roger S. *Ingeniería del Software. Un enfoque práctico.* España : McGraw-Hill, 2002. 8448132149.

DTIC-CGI. *Introduction to the Agile Up.* [En línea] 13 de mayo de 2006. [Citado el: 9 de Febrero de 2010.] http://cgi.una.ac.cr/AUP/html/overview.html.

Schildt, Herbert. *C++, guia de autoenseñanza.* España : McGRAW HILL, 2001. 0-07-882025-1.

Application Architecture Guide. s.l. : Microsoft Corporation., 2009. 9780735627109.

Larman, Craig. *UML y Patrones.* Mexico : PRENTICE HALL, 1999. 970-1 7-0261-1.

Zayas, Carlos Alvarez de. *METODOLOGIA DE LA INVESTIGACION CIENTIFICA.* SANTIAGO DE CUBA : CENTRO DE ESTUDIOS DE EDUCACION SUPERIOR "MANUEL F. GRAN", 1995.

Larman, Craig. *An Agile Up: Introduction.* 2002.

Anexo 1. Gramática del lenguaje Lince

Nota para un mejor entendimiento de la gramática, los No-Terminales se encuentran entre paréntesis angulares mientras que los terminales están de color rojo.

<formula> ::= <instructions_list>

<instructions_list> **::=** <single_instruction> | [<single_instruction> <instructions_list>]

<single_instruction> **::=** <assignment > | <formula_call> | <if_then_else> | <while_do> | <var_declaration> | <do_while> | <const_declaration> |<for>;

<designator>**::=** id [“[”<matrix_index> “]” | .id]

<matrix_index> **::=** <expression> [,<expression>]

<matrix_expression> **::=** <expression> [: <expression>] [: <expression>]

<expression> **::=** <single_expression> <others_singles_expressions>

<others_singles_expressions> **::=** [<operators_level_0> <single_expression> <others_singles_expressions>]

<operators_level_0> **::=** > | < | >= | <= | != | ==

<single_expression> **::=** <term> <others_terms>

<others_terms> **::=** [<operators_level_1><term> <others_terms>]

<operators_level_1> **::=** + | - | "||"

<term> ::= <factor> <others_factors>

<others_factors> **::=** [<operators_level_2> <factor> <others_factors>]

<operators_level_2> **::=** * | / | % | &&

<factor> **::=** numeric_literal | string_literal |<bool_literal> | <designator> [“(“ <params_list> “)”] | ! <factor> | <sign> <factor> | (<expression>) | <matrix_literal> | <query>

<matrix_literal> **::=** “[”<matrix_columnList> “]”

<matrix_columnList> **::=** <matrix_row_list> [; <matrix_columnList>]

<matrix_row_list> **::=** <matrix_expression> [, <matrix_row_list>]

<bool_literal> **::=** true | false

<sign> **::=** + | -

<assignment > **::=** <designator> = <expression> ;

<formula_call> **::=** call id (<params_list>) ;

<params_list> **::=** [<expression> [,<params_list>]]

<if_then_else> **::=** if <expression> then <instructions_list> [else <instructions_list>] end

<while_do> **::=** while <expression> do <instructions_list> end

<do_while> **::=** do <expression> while <instructions_list> end

<var_declaration> **::=** var id : <type> [= <Expression>] ;

<type> **::=** numeric | boolean | string | matrix

<const_declaration> **::=** const id = (int_literal | float_literal | <bool_literal>) ;

<for> **::=** for id = <expression> to <expression> [steep <expression>] do <instructions_list> end