Entre Ciencia e Ingeniería, ISSN 1909-8367
Año 10 No. 20 - Segundo Semestre de 2016, página 15 - 23

Creación de una arquitectura utilizando Lenguaje de Modelado Unificado (UML) en la implementación de un Lenguaje Específico de Dominio Interno (LEDI): construcción de un LEDI para el modelado de problemas de optimización[1]

Creating an architecture using Unified Modeling Language (UML) in the implementation of an Internal Domain Specific Language (IDSL): construction of an IDSL for modeling optimization problems

Criação de uma arquitetura utilizando Linguagem de Modelagem Unificada (UML) na implementação de uma Linguagem Específica de Domínio Interno (LEDI): construção de uma LEDI para a modelagem de problemas de otimização.

A. Rodas, J. I. Ríos y G. R. Solarte

Recibido Octubre 16 de 2015 – Aceptado Mayo 30 de 2016

***Resumen*—El presente artículo muestra la creación de una arquitectura que ha sido diseñada para la implementación de un Lenguaje Específico de Dominio Interno (LEDI) orientado al modelado de problemas de optimización. Del mismo modo, se presenta la metodología C4 como la seleccionada para iniciar el proceso de diseño y cómo ella aplicada a través de la construcción de la arquitectura da como resultado diagramas en UML, así mismo la descripción de las tareas y propósitos que cumplen los componentes que conforman la base funcional del sistema.**

[1]Producto derivado del proyecto de grado de la Maestría en Ingeniería de Sistemas y Computación, dela Universidad Tecnológica de Pereira "Creación de una arquitectura utilizando Lenguaje de Modelado Unificado (UML), en la implementación de un Lenguaje Específico de Dominio Embebido (LEDE): Creación de LED Embebido en Ruby para el modelado de problemas de optimización".

A. Rodas. Docente Programa de Ingeniería de Sistemas y Computación, de la Universidad Tecnológica de Pereira, Pereira (Colombia), email: alejorodasvasquez@ utp.edu.co

J. I. Ríos. Director Maestría Ingeniería de Sistemas y Computación, de la Universidad Tecnológica de Pereira, Pereira (Colombia), email: jirios@ utp.edu.co

G. R. Solarte. Docente de Ingeniería de Sistemas y Computación, de la Universidad Tecnológica de Pereira, Pereira (Colombia), email: roberto@ utp.edu.co

***Palabras clave*— dominio-específico, desarrollo dirigido por modelo, Lenguaje Específico de Dominio Interno, Lenguaje Específico de Dominio Embebido, Ruby.**

***Abstract*— This article demonstrates how to create an architecture that is designed to implement an Internal Domain Specific Language (IDSL) oriented to the modeling of optimization problems. It also introduces the methodology C4 as the one selected to start the design process and how it can be applied through the building architecture, which gives diagrams in UML as a result, as well as the description of the tasks and objectives that meet the functional components that configures base of the system.**

***Key words*— specific domain, model directed development, Internal Domian Specific Language, Embedded Domain Specific Language, Ruby.**

***Resumo*- O presente artigo mostra a criação de uma arquitetura que foi desenhada para a implementação de uma Linguagem Específica de Domínio Interno (LEDI) orientado à modelagem de problemas de otimização. Do mesmo modo, se apresenta a metodologia C4 como a selecionada para iniciar o processo de desenho e como ela é aplicada através da construção da arquitetura dando como resultado diagramas em UML, da mesma**

forma a descrição das tarefas e propósitos que cumprem os componentes que conformam a base funcional do sistema.

***Palavras chave* - domínio específico, desenvolvimento dirigido por modelo, Linguagem Especifica de Domínio Interno, Linguagem específica de Domínio Embebido, Ruby.**

I. Introducción

Dentro de los diversos roles que debe cumplir un ingeniero de sistemas, uno de los más importantes es el ejercido como desarrollador de software. Es allí donde su capacidad de creatividad y manipulación de los lenguajes de programación se pone a prueba, no obstante existen ocasiones donde sería más apropiado contar con un lenguaje que fuera enfocado al dominio del problema que se está tratando de solucionar. Esta situación se presenta con los Lenguajes de Propósito General donde se necesita una buena cantidad de líneas de código para expresar la solución requerida, dichos lenguajes poseen elementos que no son propios del dominio (tal como paréntesis, palabras reservadas, entre otros) lo cual resta expresividad y añade complejidad al momento de depurar o buscar errores en el código. Por el contrario, los Lenguajes Específicos de Dominio son construidos de modo que abordan el problema en términos de su dominio, empleando sintaxis acorde con el contexto logrando una mejor abstracción alcanzado una mejor comunicación con los *expertos del dominio* y *mejorando la productividad en el desarrollo* [1].

II. Definición del problema

Desde el principio de los tiempos, el hombre ha utilizado señales, diagramas y dibujos para representar el mundo que lo rodea. La necesidad de representar sus ideas, lo ha conducido a un proceso de abstracción que dio como origen la creación del lenguaje hablado y escrito. En la actualidad, el hombre sigue en esta búsqueda donde las ciencias de la computación han permitido la creación de lenguajes artificiales, en especial los de programación, que ayudan a concretar dichas abstracciones, que se conciben en el análisis de un problema o situación en particular.

Cada uno de estos escenarios maneja su propio contexto el cual posee una semántica propia de su universo, es decir, cada entorno maneja su propia dinámica y términos que hacen posible la interacción dentro de este espacio, y es allí donde estos poseen una significancia válida, de modo que si se utilizasen en un espacio diferente, carecerían del significado que los hacen válidos.

Por ejemplo, imagine que ha ingresado a una organización dedicada a la intermediación financiera y se le pide que modele su sistema de intermediación, enfocándose en las actividades de comercio y liquidación. Según [12] un modelo es una representación abstracta de un sistema y la porción del mundo que interactúa con él. Sin embargo, para construir este modelo es necesario utilizar herramientas (como UML) y técnicas que permitan representar aquellos artefactos o componentes que constituyen el *dominio del problema* de modo que el modelo resultante pueda ser llevado al escenario llamado, *dominio de soluciones.*

Un *dominio de soluciones*, constituye el espacio donde aquellos componentes que pertenecen al dominio del problema son representados por medio de técnicas apropiadas [4]. Es decir, supongamos que ha escogido utilizar la metodología orientada a objetos, por lo tanto, las clases, objetos y métodos, conforman los principales artefactos del dominio de soluciones, y por medio de estos se puede realizar una mejor representación de los componentes de alto nivel del dominio del problema.

Así mismo, es en esta fase de construcción de artefactos en el dominio de soluciones, donde se utilizan, por lo general, los *lenguajes de programación de propósito general*, cuya característica principal radica en que su semántica y la gramática de las instrucciones que ellos poseen no están enmarcadas en la terminología y contexto de un dominio específico siendo este atributo un obstáculo cuando lo que se intenta es representar un dominio específico en su más pura expresividad.

Por tal motivo, cuando se llega a esta situación, donde las características del dominio del problema tiene particularidades que una herramienta genérica no puede abordar de forma efectiva [20], es necesario crear un lenguaje que sirva para el propósito requerido; esto es lo que se denomina como Domain-Specific Language (DLS) o Lenguaje Específico de Dominio (LED), donde la característica principal de los mismos, es proveer un lenguaje conciso, a medida, que sea fácil para ingenieros y expertos en el dominio de aprender, entender y aplicarlo para una clase específica de problema [20].

En efecto, durante la investigación que se planteó, la cual radica en la creación de un LEDI Orientado al Modelado de Problemas de Optimización, se encontró que existen los llamados Lenguajes de Modelado Algebraico, los cuales tienen como propósito servir para el modelado de problemas orientados a la optimización de funciones. No obstante, estos lenguajes solo puede ser comprensibles por personas que tienen formación como programador o aquellas que estarían dispuestas a invertir tiempo considerable en aprenderlos, ya que la forma cómo fueron concebidos, requiere un grado de conocimiento referente a la programación para lograr crear un modelo pertinente al problema planteado.

Así pues, los profesionales pertenecientes a las ramas de la Ingeniería Industrial, Administración Financiera y Negocios Internacionales, en fin, todas aquellas relacionadas con la Investigación de Operaciones y que necesitan la formulación de modelos dentro de su quehacer, y en el preciso caso la optimización de funciones, tienen una barrera al tratar de crear los modelos pertinentes empleando estos Lenguajes de Modelado Algebraico, lo cual les deja como alternativas la utilización de hojas de cálculo u otras herramientas no tan efectivas y fáciles de usar como se necesitan.

III. Justificación

Diseñar una función objetivo para su optimización es un proceso de abstracción donde se quiere expresar el comportamiento de cierto fenómeno por medio de variables representativas y algunas restricciones que están atadas a esta.

Para realizar esta abstracción es necesario contar con un lenguaje que permita la creación de dicho modelo. Dentro de esta categoría se encuentra OPL, AIMMS, OptimJ, GAMS, Zimp, entre otros, los cuales se denominan como Lenguajes de Modelado Algebraico para Optimización (Algebraic Modeling Languages for Optimization) para programación matemática.

Sin embargo, aunque ellos cuentan con una sintaxis que permite una amplia expresividad y emplean instrucciones como *maximize (maximizar)* y *subject to (sujeto a)*, que son propias del dialecto utilizado en la optimización de funciones, el resto del código es difícil de construir para una persona que no tenga conocimientos en programación.

Por el contrario, los Lenguajes Específicos de Dominio (LED) o Domain-Specific Language (DSL) son construidos de modo que el usuario del lenguaje especifique el comportamiento que desea en términos del dominio (contexto) del problema, eliminando, en la medida de lo posible, aquellos caracteres (como puntos y comas, definición de tipos de datos o definición de variables) que no son parte fundamental del dominio, de modo que el lenguaje fluya por medio de expresiones naturales propias del dominio sin presentar ambigüedad en sus términos y ofreciendo la expresividad necesaria que permita el modelado del problema por parte de un usuario que conozca del dominio del problema pero que no sea necesariamente un programador.

IV. Estado del Arte

A continuación se presentan algunos de los Lenguajes de Modelado Algebraico encontrados durante la investigación:

- GAMS (General Algebraic Modeling System) – última versión estable marzo 15 de 2016: es un sistema de alto nivel para el modelado de sistemas para programación matemática y optimización. GAMS, está adaptado para aplicaciones complejas de gran escala y permite construir modelos mantenibles que pueden ser adaptados a cualquier situación. [23]
- Pyomo – última versión estable enero 22 de 2015: es un paquete de software *open-source* basado en Python que soporta un conjunto diverso de funcionalidades para la formulación, resolución y análisis de modelos de optimización. [24]
- AIMMS (Advanced Interactive Multidimensional Modeling System) – última versión estable julio 7 de 2014: es un software diseñado para modelar problemas tanto de optimización como de planificación. Consiste en un Lenguaje de Modelado Algebraico y un ambiente integrado de desarrollo. [25]
- ASCEND – última versión estable abril 30 de 2012: ASCEND es un programa libre *open-source* modelos matemáticos. ASCEND puede resolver sistemas de ecuaciones no lineales, problemas de optimización lineales y no lineales y sistemas dinámicos expresados en la forma de ecuaciones diferenciales/algebraicas. [26]

VI. C4: Contexto, Contenedores, Componentes y Clases

Uno de los problemas que se presentó en la creación de la arquitectura, fue alcanzar una forma efectiva de comunicar el diseño sobre el cual se basará el LEDI. De forma, que a medida que se fuera construyendo la arquitectura se pudiera visualizar la evolución de la misma. Por tanto, se empleó UML como un medio de comunicación estandarizado que permite obtener este fin.

No obstante, UML presenta una serie de diagramas que permiten observar el sistema dinámica y estáticamente [21]. De modo que para analizar la forma o estructura que va tomando la arquitectura del LEDI a medida que va evolucionando es conveniente emplear los diagramas pertenecientes al área estructural particularmente los diagramas de clases y componentes.

Por otro lado, la sola escogencia de este enfoque no garantiza que se alcance automáticamente un diseño que comunique efectivamente la arquitectura del sistema, por lo tanto se hace necesario encontrar una metodología que permita alcanzar este objetivo y que permita, a la vez, construir diagramas que sean sencillos y sobre todo que expresen y transmitan la realidad del sistema.

Así pues, Simon Brown [22] presenta su metodología llamada C4, la cual propone que *un sistema de software está hecho por un número de contenedores, que a su vez se componen de una serie de componentes, que a su vez son implementados por una o más clases.* Esto permite visualizar la estructura de una arquitectura de software de forma simple, reflejada como una serie de bloques (Fig. 1) donde cada uno representa un nivel de abstracción dentro de la jerarquía.

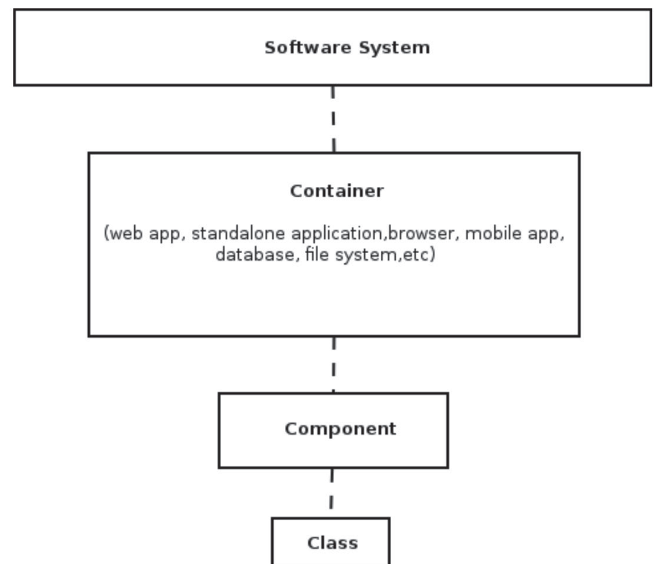

Fig. 1. Estructura jerárquica presentada como un modelo simple para la construcción de una arquitectura [22].

A continuación se presenta la definición de cada uno de los niveles [22].

- Sistema de Software (Software System): es el nivel más alto en la abstracción. En este nivel se observa el sistema en su forma general, de modo que se deben mostrar los actores que interactúan con el mismo, ya sean usuarios u

otros sistemas, los detalles técnicos no son importantes ya que lo que se pretende es mostrar un panorama general.

- Contenedores (Container): un diagrama de contenedor muestra las decisiones tecnológicas a un alto nivel, muestra cómo las responsabilidades son distribuidas a través de ellos y cómo los contenedores se comunican. Para la presente investigación se utilizó el diagrama de paquetes para representar este bloque, de modo que se pudiera identificar la estructura del sistema desde una perspectiva de capas como la que puede ofrecer este tipo de diagrama.
- Componentes (Component)*:* por cada contenedor, un diagrama de componentes permite observar los componentes lógicos claves y sus relaciones. Para la presente investigación se empleó el diagrama de componentes para mostrar esta fase del modelo C4, a través de este tipo de diagramas se mostró cómo cada uno de los paquetes que conforman el bloque de contenedores es conformado por componentes que desarrollan la implementación de una determinada funcionalidad.
- Clases (Classes): es un nivel opcional, es utilizado para explicar cómo un patrón o componente en particular será implementado. Para la investigación fue necesario llegar hasta este nivel, puesto que para lograr la implementación del Modelo de Dominio que sustenta el LEDI se necesitó emplear el diagrama de clases que soportan el mismo.

V. Patrones de construcción de un lenguaje específico de dominio interno (LEDI) y selección de uno de ellos para implementar en el proyecto

La clasificación que se hace de los Lenguajes Específicos de Dominio está relacionada con la forma cómo estos son implementados, es decir, cada una de los enfoques propuestos presenta características propias que se traducen en el diseño e implementación del LED. Según [1] existen dos enfoques principales de LED que son LED Interno (LEDI) y LED Externo. El primero ha sido seleccionado como objeto de la investigación realizada.

LED Interno (LEDI)

Un LED Interno es aquel que usa la infraestructura de un lenguaje de programación existente (también llamado lenguaje anfitrión) para construir la semántica de un dominio-específico [4]. Ampliando esta definición es útil especificar qué significa cuando se habla de *"usar la infraestructura de un lenguaje de programación existente"*.

Como ya es conocido, dentro de la composición que posee un lenguaje de programación no se puede olvidar mencionar su compilador, el cual posee diversas fases como son: analizador léxico, analizador sintáctico, analizador semántico, etc. Es decir, se cuenta con una infraestructura ya implementada la cual será utilizada por el LEDI.

Por tal motivo, un LEDI también puede ser llamado *Lenguaje Específico de Dominio Embebido*, refiriéndose al hecho que el LED, al utilizar la infraestructura de un lenguaje anfitrión, estará ligado a las restricciones de este, por lo tanto, cualquier expresión que se utilice debe ser un expresión legal en el lenguaje anfitrión, de ahí que sea importante escoger un *lenguaje anfitrión* versátil que cumpla con los criterios de expresividad requeridos; tal como lo ofrece el lenguaje de programación Ruby a través de su *Metraprogramación,* la cual está estrechamente ligada a la idea de la creación de Lenguajes Específicos de Dominio [2].

Antes de implementar un LEDI es necesario evaluar el tipo de sintaxis que se desea obtener, es decir, cuál será la estructura o apariencia que esta tendrá para ser presentada al usuario. Para lograr esto, Martin Fowler en [1] plantea tres patrones principales de implementación llamados Encadenamiento de Métodos (*Method Chaining*), Función anidada (*Nested Function*) y Secuencia de funciones (*Function Sequence*).

Sin embargo, haciendo un análisis de la forma y el tipo de sintaxis a la que se pretendía llegar en la investigación (como se muestra en la Fig. 2), se analizó una cuarta técnica también presentada en [1] llamada Cierres Anidados *(Nested Closures*), encontrando que esta técnica se adapta perfectamente al uso de Ruby como lenguaje anfitrión y se fundamenta esencialmente en el uso de *Bloques*.

$$
\begin{array}{l}
max('2x1 + 0.34x2').subjectTo\ \{ \\
\qquad restrictionA \quad '0.03x1 > 0.98' \\
\qquad restrictionB \quad '10.6x1 <= 0.002' \\
\qquad restrictionC \quad '2x1 >= 1.57'\ \}
\end{array}
$$

Fig. 2. Sintaxis resultante en la implementación del LEDI.
(Fuente: A. Rodas, J.I Ríos y G.R Solarte)

Así mismo, el patrón Cierre Anidado se muestra como una mejora de las Funciones Anidadas, Secuencia de Funciones y Encadenamiento de Métodos [1], ya que permite combinar estas técnicas creando un lenguaje versátil. Por consiguiente, se realizó la elección de esta técnica para ser la utilizada en la creación del LEDI objeto de la investigación.

VI. Definición de la arquitectura utilizando lenguaje de modelado unificado (UML), en la implementación de un lenguaje específico de dominio interno

Uno de los objetivos en la aplicación de la metodología C4 fue mejorar la abstracción en la representación del sistema mediante la creación de varios diagramas. Por tanto, siguiendo dicha metodología se inició la construcción de la arquitectura con un Diagrama de Contexto del Sistema (que puede ser equivalente a un Diagrama de Casos de Uso) permitirá identificar aquellos componentes que interactúan con el sistema observando a este como un solo bloque.

- *Actor Usuario:* este actor juega el papel de Usuario del LEDI e interactúa con los casos de uso Modelado de Problema de Optimización y obtener los valores de las variables de decisión. Donde el primero, se enfoca en la construcción de los componentes de la arquitectura que le permitirán al actor utilizar la infraestructura que soporta el modelo de dominio del LEDI; y el segundo

caso que ha sido diseñado para ejecutar todos aquellas operaciones que implican la resolución del modelado de la función a optimizar, entre estas la conexión entre el algoritmo de resolución de problemas de optimización creado en C y el propio LED implementado en Ruby.
- *Actor Sistema de Interpretación Gráfica de Resultados:* aunque este actor no tiene presencia obvia en la presente investigación, se ha tomado la decisión en incluirlo en el modelo puesto que, tomando en consideración trabajos futuros, es importante contar con un sistema que permita visualizar los resultados obtenidos de forma gráfica. Esta consideración se verá reflejada en la arquitectura, como se observará más adelante.

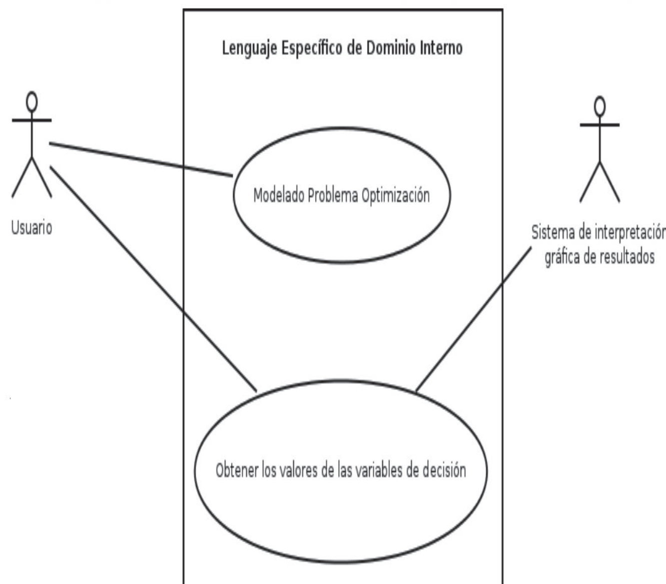

Fig. 3. Diagrama de Contexto del Sistema. (Fuente: A. Rodas, J.I Ríos y G.R Solarte)

Continuando con la aplicación de la metodología C4, la siguiente perspectiva a crear es una visualización del sistema por medio de lo que han sido llamados Contenedores *(Containers),* los cuales se representarán por medio de un Diagrama de Paquetes. Esta vista, permite descomponer el Diagrama de Contexto del Sistema en un conjunto de bloques que tendrán una comunicación entre ellos y responsabilidad individuales asignadas (Fig. 4).

- *Capa DSL Model:* esta capa contiene las estructuras que componen el Modelo de Dominio. Estas estructuras son representadas mediante clases, donde por medio de sus atributos y métodos, el usuario posee los elementos sintácticos provistos por el LEDI necesarios para la representación del modelo a optimizar. Del mismo modo, esta capa posee un componente dedicado a albergar los casos de prueba que ayudan a comprobar el Modelo de Dominio y los nuevos elementos que se añadan a este.
- *Capa SolverServices:* esta capa permite la comunicación entre las capas DSL Model y C Solver, las cuales están construidas en los lenguajes de programación Ruby y C, respectivamente. Para solventar este problema entre plataformas, se ha hecho uso de la librería FFI (*Foreign Function Interface*), la cual permite hacer llamados desde Ruby a funciones implementadas en C. Por lo tanto, la capa DSL Model puede llamar las rutinas algorítmicas alojadas en la capa C Solver.
- *Capa C Solver:* esta capa contiene los algoritmos que permiten resolver problemas de optimización, los cuales están construidos en el lenguaje de programación C, como va se ha mencionado.

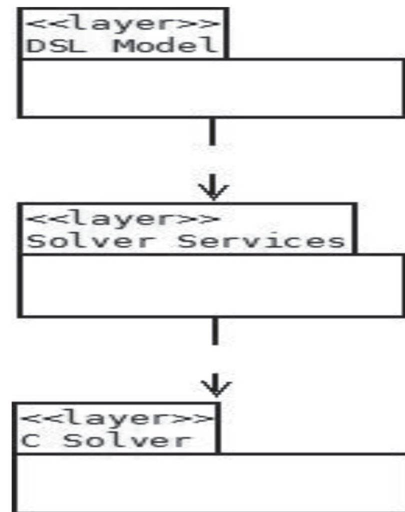

Fig. 4. Arquitectura de tres capas que sustenta el LEDI (Diagrama de Contenedores). (Fuente: A. Rodas, J.I Ríos y G.R Solarte)

Una vez obtenida la representación de la arquitectura del sistema por medio de un Diagrama de Contenedores (Fig. 4), el siguiente paso es descomponer cada Contenedor en bloques que han sido llamados Componentes (*Components*). Tal como se menciona en [22] un Contenedor representa el lugar en el cual los Componentes son ejecutados. En la Fig. 5 se muestra el sistema por medio de un Diagrama de Componentes, y cómo cada capa representada (por su correspondiente Contenedor) está conformada por uno o varios Componentes.

A continuación se realiza una descripción de los principales componentes.

- *Componente DSL Model:* contiene las estructuras que conforman el Modelo de Dominio, es decir, el LEDI propiamente dicho. Por lo tanto, es este Componente el que interactúa directamente con el actor Usuario
- *Componente Test DSL Model:* dedicado a los casos de prueba y tiene como finalidad comprobar el Modelo Semántico residente en el Modelo de Dominio, de modo que cualquier modificación o evolución de dicho modelo sea comprobable y validada a través de dichas pruebas o tests.
- *Componente Builder:* es el encargado de hacer el llamado a los métodos que yacen en el componente SolverServices (el cual está alojado en el Contenedor del mismo nombre).

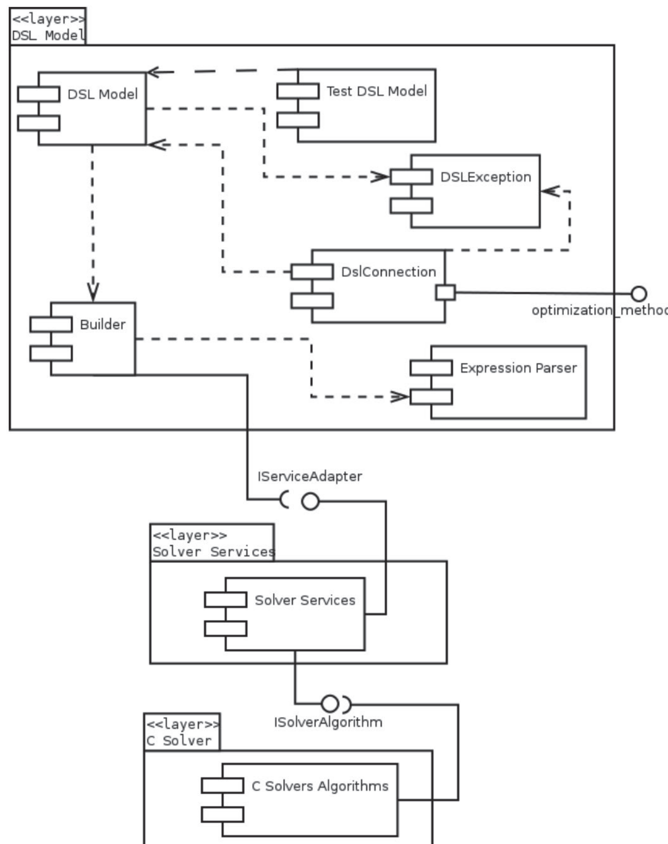

Fig. 5. Diagrama de Componentes. (Fuente: A. Rodas, J.I Ríos y G.R Solarte)

- *Componente DSLException:* tiene como función realizar las validaciones sobre los parámetros que reciben los métodos que conformar el LEDI y en caso de un error originar el mensaje apropiado. Contiene la clase *RestrictionException* la cual hereda de *Exception,* esta clase es propia de Ruby y responsable de manejar todas aquellas excepciones que se produzcan. Como ejemplo, algunas de ellas son: *NoMemoryError, RuntimeError, SecurityError, ZeroDivisionError, y NoMethodError* [31]. Así mismo, uno de los propósitos que se buscaba con esta herencia fue la creación de una clase diseñada exclusivamente para las necesidades del LEDI.
- *Componente Expression Parser:* tiene como función analizar cada una de las expresiones matemáticas (manifestadas en la función de optimización y sus restricciones) para posteriormente determinar en qué categoría (Programación Lineal o Programación No Lineal) se encuentra el problema que se está modelando, de esta manera el LEDI podrá utilizar el algoritmo de resolución apropiado.
- *Componente DslConnection:* tiene el propósito de ser el elemento que será invocado en el archivo (con extensión .rb) donde el usuario modelará su problema de optimización; este componente jugará el papel de ser la conexión entre dicho archivo y la arquitectura del LEDI.

Una vez descrita la arquitectura por medio de un Diagrama de Contenedores y de Componentes, se presenta a continuación el nivel de descripción más detallado de la misma por medio de un *Diagrama de Clases (*Fig. 6*)*. De igual forma, ya que el enfoque de implementación escogido para la construcción del LEDI radica en la combinación de los patrones Encadenamiento de Métodos y Cierre Anidado (Nested Closures), las características funcionales del lenguaje yacen en la creación de un Modelo de Dominio que por medio de la definición de clases y el empleo de sus métodos como elementos que proveen la sintaxis del LEDI, por tanto es conveniente analizar la arquitectura desde la perspectiva de un Diagrama de Clases.

Como se puede observar en la Fig. 6, *OptimizationFunction* y *Restriction* son clases que están asociadas por una relación de Composición; son estas clases las que conforman el Modelo de Dominio de la arquitectura. Se puede notar que la clase OptimizationFunction posee dos atributos que son: equation (permite almacenar la ecuación que conforma la función objetivo) y subject_to_restriccition (permite almacenar en forma de objeto la serie de restricciones a las que está sujeta la función a optimizar), este último atributo está relacionado con la clase Restriction, la cual posee el atributo llamado restriction_equation correspondiente a una estructura de tipo Hash destinada a almacenar las restricciones relacionadas con la función de optimización, donde la llave del arreglo Hash es el nombre de la restricción y el valor de dicha llave es la inecuación que expresa la restricción.

Como se mencionó anteriormente, toda la sintaxis que posee el LEDI yace en su Modelo de Dominio y es representada por los métodos que yacen en este. Sin embargo, ya que toda restricción lleva un nombre que la identifica (Fig. 2) es difícil saber cuál será el asignado por el usuario a cada restricción, haciendo imposible de esta forma, crear un método que sea residente permanente en el modelo. Esta dificultad es sorteada mediante la implementación del módulo *RestrictionBuilder*, el cual ha sido diseñado con el propósito de ser el encargado de construir, mediante la técnica de *Generación Dinámica de Métodos*, todas aquellas restricciones que el usuario ingrese.

Una vez dada la estructura apropiada al objeto de tipo *OptimizationFunction,* el cual representa el modelado del problema de optimización con sus respectivas restricciones, el usuario debe hacer uso del método *send_result_to_ solver*, el cual permite enviar dicho objeto hacia la clase *SolverConnectionBuilder*, la cual es la encargada de entablar la conexión con la capa *C Solver*, esto se hace invocando la implementación del método *algorithm_request*, la cual reside en la clase *ServiceSolverAdapter* y se define en la clase abstracta *IserviceAdapter*. De este modo se inicia el proceso que implica enviar el modelo de optimización hacia los algoritmos de resolución mediante el método *send_to_ adapter* el cual es llamado dentro del método *send_result_ to_solver.*

Del mismo modo, el componente *ExpressionParser* es conformado por los módulos *ExpressionExtensiones* y *ExpressionGrammar* (usado por la clase *SolverConnectionBuilder*), el cual utiliza el archivo de configuración *equation_grammar.treetop* junto con la librería *Treetop,* la cual debe ser instalada en el ambiente

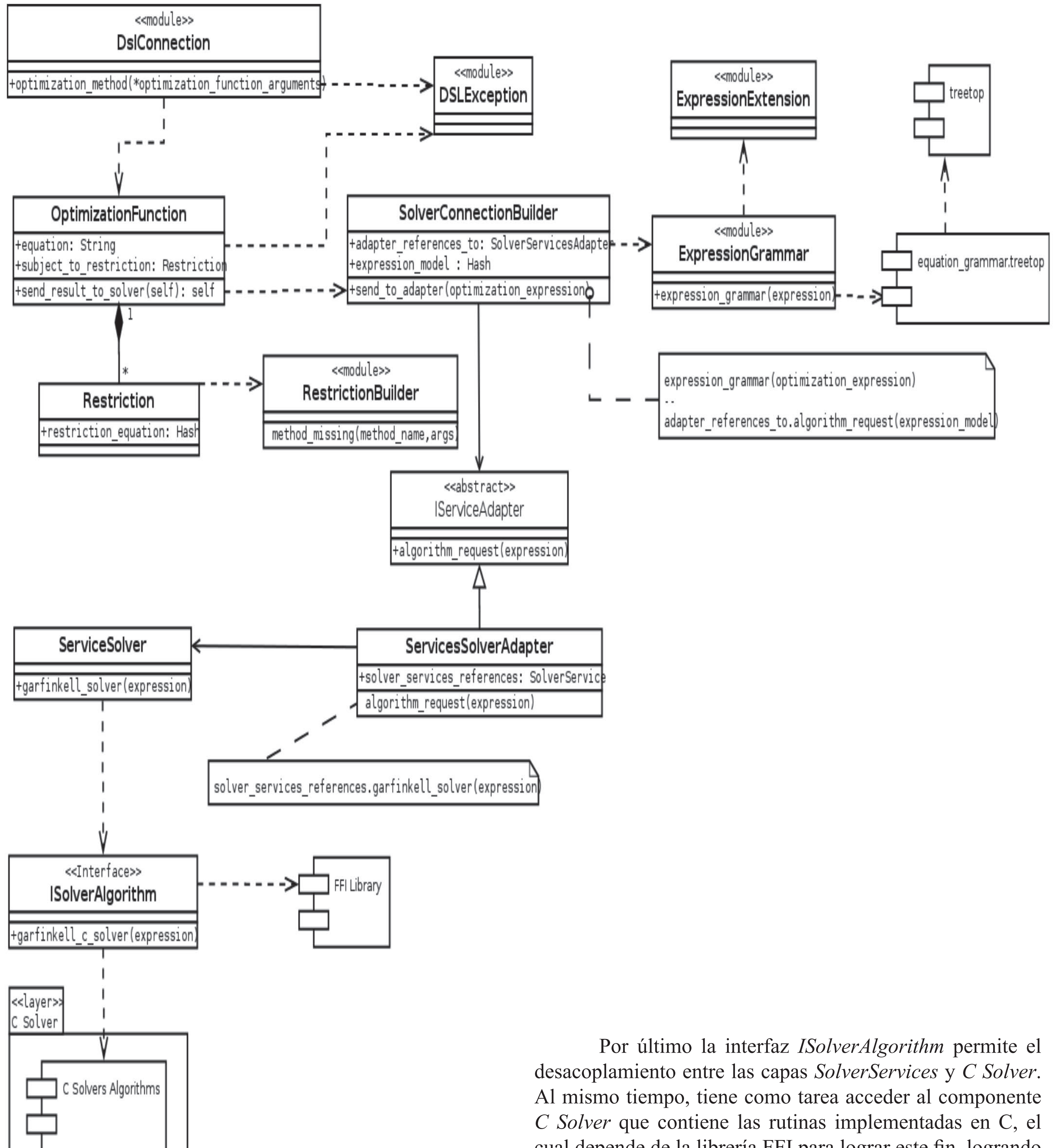

Fig. 6. Diagrama de Componentes. (Fuente: A. Rodas, J.I Ríos y G.R Solarte)

de desarrollo. A través de esta librería se logra realizar el análisis sintáctico de la expresión matemática, extrayendo y separando las variables y coeficientes de la misma, logrando identificar y categorizar el tipo de ecuaciones que el usuario está utilizando en el modelado del problema de optimización, esto con el fin de permitir que el LEDI pueda invocar internamente el algoritmo de resolución apropiado para generar el resultado esperado.

Por último la interfaz *ISolverAlgorithm* permite el desacoplamiento entre las capas *SolverServices* y *C Solver*. Al mismo tiempo, tiene como tarea acceder al componente *C Solver* que contiene las rutinas implementadas en C, el cual depende de la librería FFI para lograr este fin, logrando así la interconexión entre las dos tecnologías que conforman lactura (Ruby y C)

VII. Validaciones y pruebas de la arquitectura

Para la validación del LEDI se planteó un problema de optimización como caso práctico de forma que se pudiera evaluar el comportamiento que presenta la arquitectura. En las pruebas pertinentes se plantearon escenarios tales como el paso erróneo de número de parámetros y mal uso de la sintaxis del lenguaje. El formato empleado en la realización de las pruebas se muestra en la Tabla. I.

TABLA I
FORMATO DE CASOS DE PRUEBA

Caso práctico: expresión matemática de una función para optimizar.
Nombre caso de prueba: describe brevemente el caso de prueba.
Descripción: descripción detallada del caso de prueba.
Criterios: ítems que determinan si el caso ha sido satisfactorio o no.

Como ejemplo de una de las pruebas realizadas se muestran la Tabla II y Fig. 7.

TABLA II
MAL USO DE LA SINTAXIS DEL LENGUAJE

Caso práctico: Minimizar: z = 5x1 + 4x2 Sujeto a las restricciones: 6x1 + 4x2 <= 24 -2x1 + 2x2 <= 6
Nombre caso de prueba: mal uso de la sintaxis del lenguaje.
Descripción: el usuario olvida parametrizar las funciones de restricción a las que está sujeta la función a minimizar.
Criterios: la arquitectura debe mostrar un mensaje de error revelando que faltan las funciones de restricción, no están presentes.

```
prueba_dsl.rb   dsl_exception.rb
1 require_relative 'builder/dsl_connection'
2 include DslConnection
3
4 o_f(:min, produccion_pintura: '5x1+4x2').s_t
5
`validate_presence_of_restriccion': The restrictions are not present (DSLException::RestrictionException)
```

Fig. 7. Error: se deben especificar las restricciones a las que está sujeta la función de optimización. (Fuente: A. Rodas, J.I Ríos y G.R Solarte)

VIII. TRABAJOS FUTUROS

Dentro los proyectos relacionados se encuentran el *Modelado de una Arquitectura para la Construcción de una Herramienta la Visualización de Resultados Obtenidos Mediante un LEDI Orientado a la Definición de Problemas de Optimización*. Este proyecto presentará como resultado el modelo de una arquitectura empleando UML, enfocado en la creación de una herramienta gráfica que permita visualizar los resultados obtenidos del modelado de un problema de optimización, empleando el LEDI mostrado en el presente artículo.

Por otro lado, también se pretende que el LEDI construido pueda ser convertido en una librería que permita ser instalada en un Entorno Integrado de Desarrollo como Eclipse.

IX. CONCLUSIONES

Durante el proceso investigativo del proyecto se pudo constatar que el campo referente a la creación de Lenguajes Específicos de Dominio Interno ha sido explorado de diferentes formas, siendo el desarrollo web un nicho importante. En esta área se encontraron LEDI como Ruby on Rails y Rspec, el cual ha tomado fuerza en el sector de testing y pruebas de integración.

Al inicio de la investigación se tenía pensado que el campo de los Lenguajes Específicos de Dominio era un tema especializado y de difícil acceso, ya que usualmente, durante la formación que se recibe como ingeniero de sistemas el ámbito de los lenguajes de programación gira alrededor de los Lenguajes de Propósito General. Sin embargo, como se ha podido constatar por medio de la revisión bibliografía los Lenguajes Específicos de Dominio son utilizados y desarrollados no solo con fines investigativos sino industriales, por lo tanto valdría la pena ser incluidos dentro de los planes de curso.

UML ofrece una variedad de diagramas que permiten observar el software o una aplicación determinada desde varias Vistas. Sin embargo, al iniciar el planteamiento de la arquitectura del LEDI, se encontró que esta diversidad de diagramas en realidad planteaba cierta desventaja puesto que no se tenía claro cuál de todos seleccionar. En este punto era importante encontrar alguna metodología que brindara un punto de referencia. Es allí donde la metodología C4 ofrece dentro de su planteamiento una forma práctica y sencilla para construir un sistema, esto permitió crear la arquitectura con un enfoque funcional e incremental, como se puede evidenciar en el documento, a medida que se fue necesitando la incorporación de nuevos componentes que desempeñaran distintas funcionalidades, estos fueron acoplados sin problema en la arquitectura.

REFERENCIAS

[1] M. Fowler, Domain – Specific Languages. Ed. Boston: Addison Wesley, 2011.

[2] D. Flanagan and Y. Matsumoto, The Ruby Programming Language. Ed. California: O'Reilly, 2008.

[3] P. Cooper.,Beginning Ruby From Novice to Professional, ed 2nd . Ed New York: Apress, 2009.

[4] D. Ghosh, DSLs in Action. Ed. Stamford: Manning, 2010.

[5] S. Günther, Agile DSL-Engineering with Patterns in Ruby.

[6] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Ed. Addison Wesley, 2003.

[7] M. Voelter, DSL Engineering. Designing, Implementing and Using Domain-Specific Languages. [Online]. Available: http://dslbook.org.

[8] M. Mernik, "When and How to Develop Domain-Specific Languages". ACM Computing Surveys. vol 37, pp 316-344, december 2005.

[9] R. Fourer, D. M. Gay, B. W. Kernighan, AMPL: A Modeling Language for Mathematical Programming, 2da ed. 2003.

[10] IBM ILOG OPL Language User's Manual [Online]. Available: http://cedric.cnam.fr/~lamberta/MPRO/ECMA/doc/oplTutorial.pdf

[11] T. Halpin. UML Data Models From An ORM Perspective: Part 1.[Online]. Available: http://www.orm.net/pdf/ICMArticle1.pdf

[12] K. Czarnecki, "Overview of Generative Software Development", in Unconventional Programming Paradigms, 2005, pp. 326-341.

[13] J. Gärtner, X. GmbH, N. Musliu, W. Schafhauser and W. Slany. A Domain Specific Language for Modeling and Solving Staff Scheduling Problems. [Online]. Available: http://www.dbai.tuwien.ac.at/staff/musliu/CischedEMPLE.pdf.

[14] A. Mediratta. "A Generic Domain Specific Language For Financial Contracts," M.S thesis, Rutgers, The State University of New Jersey, 2007.

[15] H. Beck, K. Currie and A. Tate, A Domain Description Language for Job-ShopScheduling. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.269

[16] A. van Deursen, P. Klint and J. Visser. Domain-Specific Languages: An Annotated Bibliography. [Online]. Available: http://www.st.ewi.tudelft.nl/~arie/papers/dslbib.pdf

[17] R. Fourer, Algebraic Modeling Languages for Optimization. [Online]. Available: http://ampl.com/REFS/amlopt.pdf

[18] E. Gramma, R. Helm, R. Jhonson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Ed. Boston: Addison Wesley

[19] GarfinkeL, R.S. (1985). Motivation and Modeling, in LAWLER, E.L.; LENSTRA, J.K.; RINNOOY KAN, A.H.G.; SHMOYS, D.B. (eds.) *The Traveling Salesman Problem: A Guide Tour of Combinatorial Optimization*. Wiley. Chichester.

[20] J. de Lara y E. Guerra. "Domian-Specific Textual Meta-Modelling Languages form Model Driven Engineering".Modelling Foundations and Aplications. pp 316-344, July 2005.

[21] J. Rumbaugh, I. Jacobson y G. Booch. El Lenguaje Unificado de Modelado: Manual de Referencia. Ed. Madrid: Addison Wesley, 2000.

[22] S. Brown. Software Architecture for Developers. Ed. Leanpub, 2015.

[23] GAMS [Online]. Available: https://www.gams.com/

[24] Pyomo [Online]. Available: http://www.pyomo.org/

[25] ASCEND [Online]. Available: http://ascend4.org/Main_Page

[26] AIMMS [Online]. Available: http://www.aimms.com/

Alejandro Rodas Vásquez. Profesor catedrático Programa Ingeniería de Sistemas y Computación – Universidad Tecnológica de Pereira. Es Ingeniero de Sistemas y Telecomunicaciones – Universidad Católica de Pereira, MSc. en Ingeniería de Sistemas y Computación - Universidad Tecnológica de Pereira. Sus áreas de actuación son el Desarrollo Web, Desarrollo de Sistemas Expertos, Arquitectura de Software, Ingeniería de Software y Usabilidad en Lenguajes Específicos de Dominio Interno.

Jorge Iván Ríos Patiño. Profesor Titular del Programa Ing. Sistemas y Computación – Universidad Tecnológica de Pereira. Es Ingeniería Industrial – Universidad Tecnológica de Pereira, MSc Informática e Ingeniera del Conocimiento – Universidad Politécnica de Madrid y PhD (c) Informática– Universidad Politécnica de Madrid. Es director de la Maestría en Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira desde junio de 2009. Sus áreas de actuación son la Inteligencia Artificial, Ciencias de la Computación y de la Información.

Guillermo Roberto Solarte Martínez, Profesor Asociado, Transitorio del Programa Ing. Sistemas y Computación.
Doctor en Informática de la Universidad Pontificia de Salamanca con sede Madrid España Suficiencia investigativa, D .E. A Universidad Pontificia de Salamanca con sede Madrid España, Magister en Investigación de Operativa y Estadística de la Universidad Tecnológica de Pereira Risaralda e Ingeniero de Sistemas Grupo de Investigación En Inteligencia Artificial, Semillero de Investigación GNTO Grupo de Nuevas técnicas de búsqueda y de optimización.