



UNIVERSIDAD NACIONAL DE COLOMBIA

**HERRAMIENTA PARA LA GENERACIÓN
AUTOMÁTICA DEL CÓDIGO FUENTE PARA
APLICACIONES CON ARQUITECTURA MODELO
VISTA CONTROLADOR (MVC) BAJO
DESARROLLO DIRIGIDO POR MODELOS
TEXTUALES (MDD)**

DEIVIS DE JESUS MARTINEZ ACOSTA

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2014

HERRAMIENTA PARA LA GENERACIÓN AUTOMÁTICA DEL CÓDIGO FUENTE PARA APLICACIONES CON ARQUITECTURA MODELO VISTA CONTROLADOR (MVC) BAJO DESARROLLO DIRIGIDO POR MODELOS TEXTUALES (MDD)

DEIVIS DE JESUS MARTINEZ ACOSTA

Trabajo de investigación presentado como requisito parcial para optar al título de:

MSc. en Ingeniería de Sistemas y Computación

Director:

MSc. Henry Roberto Umaña Acosta

Línea de Investigación:

Ingeniería de Software

Universidad Nacional de Colombia

Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial

Bogotá, Colombia

2014

*A Dios, a mi esposa Yulibeth, a mi hijo David,
y mi familia que siempre me han apoyado en
mis decisiones.*

Agradecimientos

A la Facultad de Ingeniería de la Universidad Nacional de Colombia por llevar este programa hasta la ciudad de Valledupar, a sus docentes por su aporte académico e investigativo a mi formación.

Al docente Henry Roberto Umaña director de este proyecto, por su motivación para trabajar en esta línea de investigación, su orientación, apoyo y sugerencias para la elaboración de este trabajo.

Al fondo de tecnologías de la información y las comunicaciones del Ministerio de Tecnologías y Comunicaciones de Colombia (Min TIC) que ha financiado este proyecto.

Al compañero Jhon Cruz, por su apoyo inicial en la parte técnica, a mis siempre compañeros Wilman y Danier por su persistencia y motivación a terminar este proyecto.

Resumen

Esta investigación se centra en facilitar el proceso de desarrollo de software, partiendo de la abstracción del problema para construir modelos que permitan la representación de una solución general. Se emplea el desarrollo dirigido por modelos para la elaboración de un lenguaje de dominio específico y las plantillas para la generación de código, tomando como base una implementación de referencia.

Se desarrolla una herramienta encargada de proporcionar un entorno de trabajo al desarrollador, y la generación de código fuente basado en un meta modelo, contribuyendo a la construcción de aplicaciones en la optimización de la productividad sobre el equipo que elabora software, asegurando aspectos principales como son la calidad, mantenibilidad y reutilización de elementos.

La generación de código en forma automática bajo la arquitectura modelo vista controlador proporciona un mantenimiento factible de las aplicaciones construidas y la facilidad para la distribución de elementos dentro de un equipo de desarrollo de software.

Palabras clave: Desarrollo dirigido por modelos (MDD), lenguajes de dominio específico (DSL), generación automática de código, modelo vista controlador (MVC), ingeniería de software, abstracción, transformaciones de modelos.

Abstract

This research focus on facilitate the software development process, based on the abstraction of the problem in order to build models that allow the representation of a general solution. It is used the Model-Driven Development for the development of a domain specific language and the templates for source code generation, based on a reference implementation.

It is developed a tool that provides a framework to the developer, and the source code generation based on a meta-model, contributing to the applications development in optimizing the productivity of the development team, ensuring main aspects such as quality, maintainability, and elements reuse.

The automatic code generation on the Model-View-Controller architecture provides a feasible maintenance of applications built an ease of elements distribution in a software development team.

Keywords: Model Driven Development (MDD), Domain Specific Language (DSL), Generate Automatic Code, Model View Controller (MVC), Software Engineering, Abstraction, Model Transformation.

Contenido

	Pág.
Resumen	V
Lista de figuras	IX
Lista de tablas	XI
Lista de Símbolos y abreviaturas	XII
1. INTRODUCCIÓN	13
1.1 Objetivos.....	14
1.1.1 Objetivo General	14
1.1.2 Objetivos Específicos	14
1.2 Etapas del Proyecto.....	14
1.2.1 Etapa I	15
1.2.2 Etapa II	15
1.2.3 Etapa III	16
2. Capítulo 2.....	17
2.1 Conceptos principales y trabajos relacionados	17
2.1.1 Abstracción y Modelos	17
2.1.2 MDD y sus etapas dentro del desarrollo de software	18
2.1.3 Metas principales de MDD	18
2.1.4 Metodologías basadas en MDD	20
2.1.4.1 Descripción general del enfoque metodológico MDA	20
2.1.4.2 Descripción general del enfoque metodológico DSM	22
2.2 Comparativo entre las dos metodologías marcadas que trabajan MDD	23
2.3 Trabajos relacionados.....	24
2.3.1 WebML	26
2.3.2 WebDSL.....	27
2.3.3 WebRatio	27
2.4 Características diferenciales de la herramienta a construir	28
2.5 Metodología seleccionada dentro de MDD	30
2.6 Ventajas de trabajar con MDD	31
2.6.1 Lenguaje Específico del Dominio (Domain Specific Language DSL) ...	31
2.7 Arquitectura Modelo Vista Controlador (MVC)	33
2.8 Entorno integrado de desarrollo Eclipse.....	34
2.9 Modelado textual (Xtext)	34
2.10 Conclusiones del capítulo	35

3. Capítulo 3	37
3.1	
Selección del tema para la implementación de referencia.....	37
3.2 Implementación de referencia	40
3.3 Metodología a seguir en la elaboración de la herramienta.....	40
3.4 Análisis del dominio del problema y selección de artefactos a generar	41
3.5 Identificación de patrones dentro de la implementación de referencia	41
3.6 Características de la herramienta y creación de la arquitectura de la solución.....	42
3.7 Diseño e implementación del Lenguaje de Dominio Específico	43
3.8 Plantillas de generación de código automático	46
3.9 Pruebas sobre el DSL construido	49
3.10 Conclusiones del capítulo.....	51
4. Capítulo 3	53
4.1 Selección de pruebas.....	53
4.1.1 Caso I (Aplicación básica, un solo componente).....	54
4.1.2 Caso II (Aplicación con dos o más componentes).....	54
4.1.3 Caso III (Aplicación con dos o más componentes y relaciones).....	55
4.2 Características de las pruebas	57
4.3 Generación de Archivos	57
4.3.1 Archivos PHP	59
4.3.2 Archivos JAVA.....	60
4.3.3 Archivos JSP	61
4.3.4 Archivos HTML	62
4.3.5 Archivos XML	63
4.3.6 Archivos JRXML	65
4.3.7 Archivos SQL.....	65
4.3.8 Archivos CCS y JS	66
4.3.9 Análisis del resultado de las pruebas.....	68
5. Conclusiones y Análisis de Resultados	71
5.1 Conclusiones.....	71
5.2 Análisis de Resultados	72
5.3 Metas Alcanzadas	74
5.4 Contribuciones del trabajo	76
5.5 Trabajo Futuro.....	76
A. Anexo: Aplicación de referencia	77
B. Anexo:Código fuente de la aplicación de referencia	79
Bibliografía	85

Lista de figuras

	Pág.
Figura 1: Actividades de la Etapa I	15
Figura 2: Actividades de la Etapa II	16
Figura 3: Proceso de abstracción en modelos	18
Figura 4: Pasos MDA en el proceso de desarrollo de software (Stahl & Völter, 2006), (Cañadas, Palma, & Túnez, 2011), (Ceri & Daniel, 2003).....	20
Figura 5: Enfoque gráfico de la arquitectura MVC.....	33
Figura 6: Proceso de desarrollo con MDD (Lin, Zhang, & Gray, 2004), (Moreira & Rossi, 2004) Vs proceso de desarrollo tradicional (Cañadas, Palma, & Túnez, 2011), (Torres, Muñoz, & Pelechano, 2005).	36
Figura 7: Diagrama de clases para la implementación de referencia	38
Figura 8: Modelo relacional de la implementación de referencia	38
Figura 9: Diseño de las interfaces de usuario	39
Figura 10: Metodología para la elaboración de la herramienta	40
Figura 11: Esquema de los archivos de la implementación de referencia	41
Figura 12: Solución planteada informalmente	43
Figura 13: Meta modelo formalmente definido	44
Figura 14: Código de gramática en la herramienta.....	45
Figura 15: Listado de plantillas para la generación del código de forma automática ...	47
Figura 16: Código fuente de las plantillas para la generación del artefactos de forma automática	48
Figura 17: Código fuente de validaciones	49
Figura 18: Pruebas sobre el DSL	51
Figura 19: Caso de prueba I.....	54
Figura 20: Caso de prueba II.....	55
Figura 21: Código del caso de prueba II.....	55
Figura 22: Caso de prueba III.....	56
Figura 23: Caso de prueba III Código en el DSL	56
Figura 24: Directorios y paquetes generados	58
Figura 25: Estructura de archivos PHP	59
Figura 26: Código de archivos PHP	60
Figura 27: Estructura de archivos java	60
Figura 28: Estructura de archivos java	61
Figura 29: Código fuente JSP	62
Figura 30: Esquema de Vistas en HTML	63
Figura 31: Vista en HTML	63
Figura 32: Archivo XML.....	64
Figura 33: Vista en la herramienta de diseño	64
Figura 34: Archivo de reporte	65

Figura 35:	Script de bases de datos.....	66
Figura 36:	Prueba de script de bases de datos.....	66
Figura 37:	Archivo de estilos.....	67
Figura 38:	Archivo de validaciones	67

Lista de tablas

	Pág.
Tabla 1: Principales metas de MDD (Stahl & Völter, 2006).....	19
Tabla 2: Principales características de la metodología MDA y DSM dentro del desarrollo dirigido por modelos (Barceló, 2010), (Lin, Zhang, & Gray, 2004).....	23
Tabla 3: Listado de productos más relevantes bajo MDD que se encuentran operativos hasta el año 2009, 2010 y aún persisten (Barceló, 2010), (Rizvi & Hassan, 2009).	24
Tabla 4: Características diferenciales de la herramienta a elaborar	29
Tabla 5: Listado de las características relevantes para la selección de la metodología	30
Tabla 6. Herramientas usadas en la implementación de referencia	39
Tabla 7: Características de las pruebas.....	57
Tabla 8: Requisitos que debe cumplir la herramienta	68
Tabla 9: Análisis de la cantidad de líneas de código generadas frente a las digitadas	73

Lista de Símbolos y abreviaturas

Abreviatura	Descripción extendida
AST	Abstract Syntax Tree
CSS	Cascade Style Sheet
DSL	Domain Specific Language
DSM	Domain Specific Modeling
EBNF	Unified Modeling Language
GPL	General Purpose Language
HTML	HyperText Markup Language
JS	Java Script
JSP	Java Server Pages
MDA	Model Driven Architecture
MDD	Model Driven Development
MVC	Model View Controller
OMG	Object Management Group
UML	Extended Backus–Naur Form
XML	eXtensible Markup Language

1. INTRODUCCIÓN

Desde el inicio de la computación el hombre se ha interesado por desarrollar herramientas de software que faciliten el trabajo que se realiza en las máquinas de computo, es así como cada día se emprenden investigaciones que buscan lograr este objetivo; la investigación que se presenta a continuación involucra características de la ingeniería de software tales como: generación de código automático, desarrollo de aplicaciones a través de modelos, trabajo sobre lenguajes de dominio específico, modelo vista controlador, entre otras. Basado en estas características se busca la construcción de una herramienta capaz de producir sistemas informáticos automáticamente con alto porcentaje de completitud y los aspectos de código fuente que se deban implementar de forma manual sean mínimos. La solución se pretende implementar con el uso de un lenguaje de dominio específico que permita al usuario un ambiente de desarrollo reducido y validado, donde se disminuyen los errores, logrando evitar la complejidad de la escritura de código (Ceri & Daniel, Extending WebML for modeling multi-channel context-aware web applications, 2003), (Cabot & Gómez, 2008), esta disminución se refleja en los artefactos generados que pueden ser socializados a nivel gerencial y con los diferentes roles que se presentan en un equipo de elaboración de software.

Según lo descrito anteriormente se busca la facilidad para el desarrollador con el uso de un lenguaje de dominio específico, evitando estructuras complejas de un lenguaje de propósito general, al igual se aumenta la calidad de las aplicaciones evitando sentencias o instrucciones fuera del dominio (Ceri & Fraternali, 2002) (Barceló, 2010), esta meta se logra con el uso de una herramienta DSL capaz de generar código automático para la implementación de aplicaciones software.

Este trabajo propone aumentar la productividad, calidad, modularidad y mantenibilidad de las aplicaciones construidas en un equipo de desarrollo de software; para lograr este propósito se plantea el interrogante: cómo una herramienta de generación de código contribuye favorablemente en el proceso de construcción de software y cómo construir

una herramienta MDD sobre metodologías DSM para generación de código con la arquitectura MVC. Para conseguir la respuesta a estos interrogantes se plantean los siguientes objetivos.

1.1 Objetivos

1.1.1 Objetivo General

Construir una herramienta para la generación automática del código fuente para aplicaciones con arquitectura modelo vista controlador bajo desarrollo dirigido por modelos textuales.

1.1.2 Objetivos Específicos

- A. Comparar las metodologías que usan MDD para implementar las mejores prácticas de ingeniería de software para la obtención de aplicaciones funcionales.
- B. Construir un DSL, que permita la generación automática de aplicaciones java con arquitectura MVC a través de MDD usando Xtext para los modelos textuales.
- C. Evaluar la herramienta demostrando funcionalidad con una prueba de concepto.

Para alcanzar los objetivos planteados y los aspectos intrínsecos a ellos, se plantea desarrollar tres etapas enmarcadas en los objetivos planteados, implementados en el orden presentado a continuación.

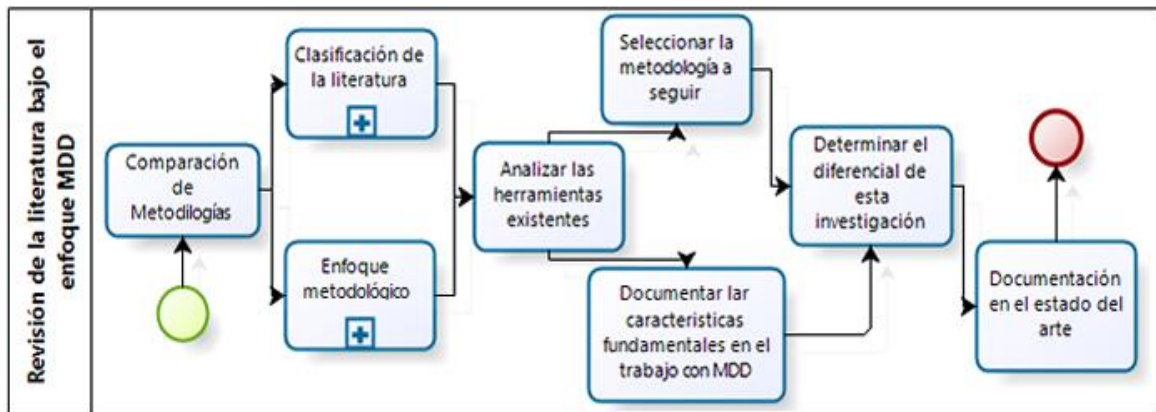
1.2 Etapas del Proyecto

Las etapas de este proyecto describen puntualmente las actividades que se desarrollan para alcanzar las metas propuestas.

1.2.1 Etapa I

La primera etapa pretende comparar las metodologías basadas en MDD, en la que se realiza una revisión de la literatura científica que trata sobre el tema; esta actividad nos permite seleccionar el enfoque de trabajo investigativo a seguir y determinar las mejores prácticas de ingeniería de software que se toman como base para la implementación proyectada, también se proporcionan los elementos necesarios para el desarrollo de las siguientes etapas, así como la elaboración del estado del arte usado como guía. Las actividades de esta etapa son presentadas en la Figura 1 (Barceló, 2010).

Figura 1: Actividades de la Etapa I



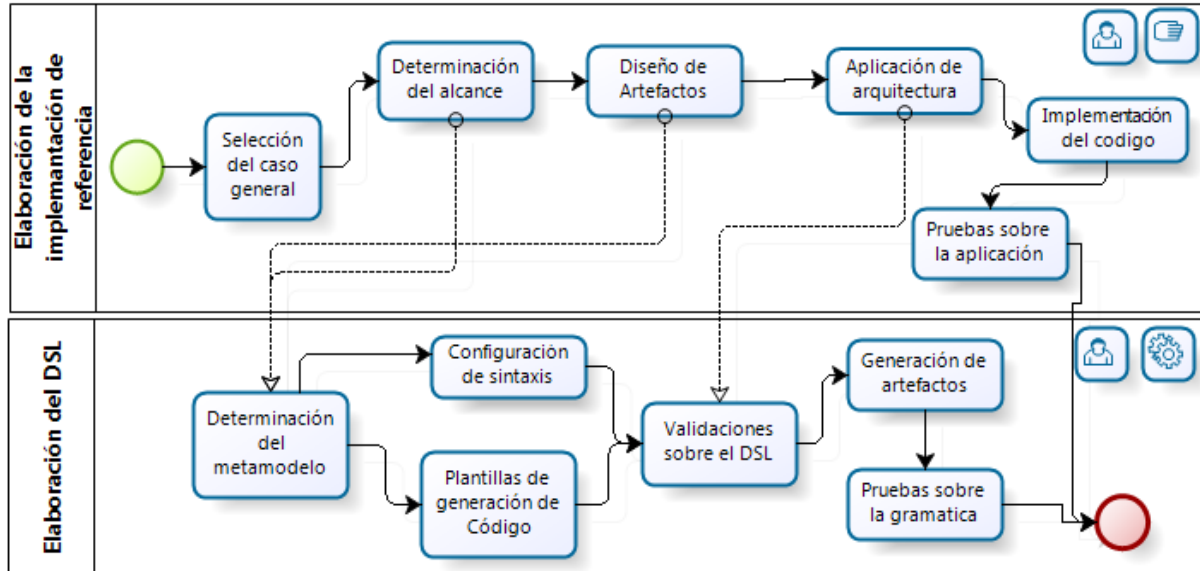
La Figura 1 destaca las actividades seguidas para la ejecución de la primera etapa del proyecto.

1.2.2 Etapa II

En la segunda etapa se construye una herramienta lenguaje de dominio específico para la generación de aplicaciones con arquitectura modelo vista controlador, tomando como punto de entrada un modelo de base de datos relacional. Se realiza una implementación de referencia probando su funcionamiento, determinando el meta-modelo informalmente, se procede al diseño del lenguaje de dominio específico con su sintaxis definida y basado en las pruebas realizadas sobre la implementación de referencia, se concretiza en Xtext con sus respectivas validaciones y se crean las plantillas para la generación de código automáticamente basado en los modelos textuales definidos para aplicaciones con

arquitectura MVC (Efttinge & Völter, 2006). Esta serie de actividades las encontramos descritas en la Figura 2.

Figura 2: Actividades de la Etapa II



1.2.3 Etapa III

Se finaliza con las pruebas sobre la herramienta construida en la segunda etapa, cuyas pruebas conducen a la verificación de la funcionalidad y aplicabilidad del producto generado mediante la iteración de todo el proceso; tomando como base la prueba de concepto.

2. Capítulo 1

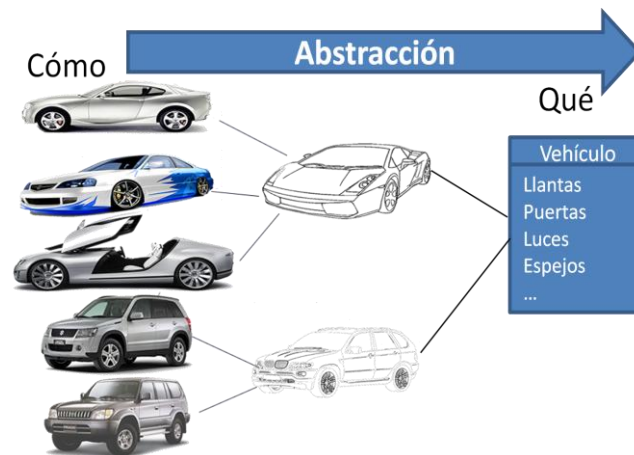
Este capítulo hace una descripción y comparación de las metodologías de trabajo inmersas en MDD como lo son MDA y DSM, se describen características representativas comunes y específicas de ellas, al igual que una ilustración sobre las herramientas que permitirán la construcción de nuestra solución para alcanzar los objetivos propuestos. Se presentan también los trabajos relacionados con esta investigación, el diferencial de este trabajo frente a los existentes, el por qué hacer esta investigación y aplicación de tecnologías.

2.1 Conceptos principales y trabajos relacionados

A continuación se presentan los principales conceptos dentro del desarrollo del trabajo.

2.1.1 Abstracción y Modelos

La abstracción es una característica preponderante en el desarrollo de software que proporciona la posibilidad de aislar los elementos de un contexto determinado para enfocar el propósito de donde se extrae, restando importancia en este punto a la forma de cómo hacerlo e invertir los esfuerzos en el qué hacer (Barceló, 2010). Esta conceptualización se encuentra ligada al proceso implementado con la generación de modelos que se enfocan en aplicar una capa de abstracción superior que generaliza una solución aceptable dentro de un problema específico planteado. En la Figura 3 se muestra de forma gráfica este proceso de modelado elevando el nivel de abstracción de una solución puntual y tomando características conceptuales.

Figura 3: Proceso de abstracción en modelos

2.1.2 MDD y sus etapas dentro del desarrollo de software

El desarrollo de software dirigido por modelos (MDD), consiste en la producción de artefactos software a partir de una representación a escala de una solución general, realizándolo a través de la abstracción que se lleva implícito en el concepto de modelos y meta modelos, permitiendo ocultar características a cada nivel de abstracción superior para que la solución de un problema sea más sencillo a los actores; cuando más complicado es el problema surge la necesidad de producir más niveles de abstracción y al momento de la concretización de los mismos bajar cada uno de estos niveles para llegar a la generación de un producto terminado, que en este caso corresponde al código fuente; en este entorno se encuentran metas a ser cumplidas por MDD, de las cuales se seleccionan a continuación las pertinentes a este trabajo (Stahl & Völter, 2006), (Pastor, España, Panach, & Aquino, 2008).

2.1.3 Metas principales de MDD

En la Tabla 1 se encuentran listadas las principales metas de MDD.

Tabla 1: Principales metas de MDD (Stahl & Völter, 2006)

Metas	Descripción
Velocidad en el desarrollo	Esta se plantea a través de la generación de código de forma automática lo que reduce el tiempo en desarrollar una aplicación.
Calidad del software	Los componentes son generados de forma automática a partir de modelos formales lo que disminuye los errores humanos.
Evitar la redundancia	Se evita la implementación de un artefacto varias veces debido a que son generados y distribuidos para las implementaciones específicas.
Manejabilidad de los cambios tecnológicos	El meta modelo y la herramienta siempre persisten, lo que cambian son las plantillas para la generación de código a una tecnología específica.
Reusabilidad	El código fuente que se genera y las herramientas pueden ser reutilizados para la construcción de varias aplicaciones.
Gestión de la complejidad con abstracción	Se aplica la abstracción para que las soluciones obedezcan a meta modelos sencillos.
Entorno productivo	Las aplicaciones son digitadas en la herramienta de generación de código fuente solamente, es un solo punto de entrada.
Interoperabilidad	Los meta modelos no obedecen a la tecnología sino a la lógica de negocio.
Portabilidad	Los artefactos se crean independientes a las plataformas por lo tanto pueden ser portados de forma sencilla.

De las metas listadas anteriormente en este proyecto toma como referencia la velocidad en el desarrollo, la calidad del software, gestión de la complejidad y el entorno productivo. Dentro de las buenas prácticas en la ingeniería de software se planta la utilización del principio de abstracción, permitir que el computador haga el trabajo y escribir programas

para las personas y no para las computadoras; en la aplicación de estas características dentro de MDD se pueden tomar dos rutas marcadas como metodologías que se describen a continuación.

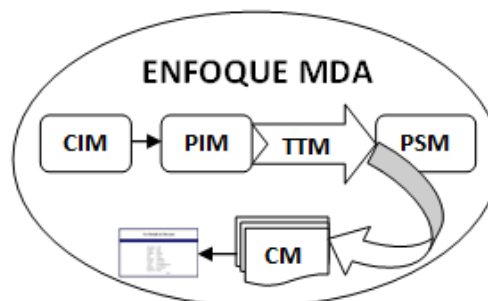
2.1.4 Metodologías basadas en MDD

En la actualidad los trabajos del desarrollo dirigido por modelos siguen dos enfoques de trabajo denominados metodologías, conocidas como Arquitectura Conducida por Modelos (MDA) y el Modelado sobre Dominios Específicos (DSM). La primera MDA, es una iniciativa liderada por la Object Management Group (OMG) que busca definir estándares que permitan la utilización de modelos para conducir el proceso de desarrollo de software en todas sus etapas y su principal característica es la producción de artefactos basados en abstracción presentada gráficamente partiendo de elementos formales descritos con diagramas UML entre otros diagramas (Barceló, 2010), (Cañadas, Palma, & Túnez, 2011). DSM trabaja los conceptos de dominio basados en la utilización de modelos que hagan la representación de uno o varios dominios descritos en una misma organización, tecnología o arquitectura (Barceló, 2010), (Igor, Gordana, Branko, & Maja, 2010), (Fraternali & Paolini, 2000).

2.1.4.1 Descripción general del enfoque metodológico MDA

En la arquitectura dirigida por modelos es necesario tener en cuenta algunas etapas evolutivas o de iteraciones básicas que se muestran en la Figura 4. (Cañadas, Palma, & Túnez, 2011), (Ceri & Daniel, 2003).

Figura 4: Pasos MDA en el proceso de desarrollo de software (Stahl & Völter, 2006), (Cañadas, Palma, & Túnez, 2011), (Ceri & Daniel, 2003)



Modelo Independiente de la Computación (CIM): Éste modelo expresa las necesidades del negocio de una manera general y poco técnica donde se presenta el conocimiento de la problemática que tiene el usuario final o dueño del producto (Cañadas, Palma, & Túnez, 2011), (Rode & Rosson, 2003).

Modelo Independiente de la Plataforma (PIM): En este punto el nivel de abstracción es alto, se constituye como la base para la implementación de una aplicación, debe estar construido de manera que defina la solución con una semántica precisa y formal, llevando la abstracción a tal punto que puedan ser implementados en varias plataformas (Kraus, Knapp, & Koch, 2007), (Ceri & Fraternali, 2002).

Tecnologías de transformación de modelos (TTM): Para pasar de un modelo a otro, es necesario la utilización de una tecnología capaz de convertirlo en un nuevo modelo, con un nivel de abstracción mayor o menor según el caso, al igual que el paso al código fuente; ésta labor la realizan las TTM dando al enfoque MDD la trazabilidad que necesita para bajar y subir en nivel abstracción en las soluciones planteadas (Cañadas, Palma, & Túnez, 2011).

Modelo Específico de la Plataforma (PSM): Las tareas técnicas en el manejo de lenguajes de programación: posibles máquinas virtuales, sistemas operativos, hardware e interconexiones entre otros se ven involucradas en estos modelos (Cañadas, Palma, & Túnez, 2011), (Kraus, Knapp, & Koch, 2007), ellos pueden superar en número a los PIM, pero cada uno debe estar basado en por lo menos un PIM y trabajar de forma colaborativa para dar una solución completa.

Modelo del código (CM): Consiste en el resultado del ejercicio de la transformación de modelos más abstractos hasta llegar a la ausencia de modelos, es conocido también como el código generado (Cañadas, Palma, & Túnez, 2011).

En la actualidad no se ha llegado al grado de madurez de los modelos para descartar la importancia del código fuente (Kraus, Knapp, & Koch, 2007), es por ello que se encuentra el proceso de depuración de posibles errores; aunque teóricamente la producción de éste código es de forma automática, debe presentar una compilación para llegar a un producto funcional que no debió necesitar la intervención humana para corregir, modificar

o complementar el código fuente (Cañadas, Palma, & Túnez, 2011), (Kraus, Knapp, & Koch, 2007).

Las características descritas anteriormente pertenecen a la primera metodología MDA, aunque alguna de ellas sean aplicables intrínsecamente también en la metodología DSM la cual concretiza su trabajo basado directamente en una especificación del dominio.

2.1.4.2 Descripción general del enfoque metodológico DSM

El trabajo con DSM está enfocado en elevar en uno o varios niveles de abstracción una solución, presentando un meta modelo que describe las necesidades globales. El modelo con las respectivas transformaciones a través de las plantillas es capaz de generar artefactos de forma automática (Stahl & Völter, 2006). Puntualmente DSM se basa en la implementación de tres pasos que son:

- *Identificación y construcción de un meta modelo para el dominio específico:* se realiza el estudio del dominio y se plantean los posibles casos generalizados, se describe la gramática del lenguaje y los elementos necesarios para obtener la generalidad de aplicaciones del mismo estilo que compartan el punto de entrada.
- *Construcción de las plantillas de generación de código:* Estas plantillas permiten obtener el código para las plataformas elegidas según la información suministrada como punto de entrada en la herramienta.
- *Elaboración y configuración de las validaciones para el lenguaje a generar:* Estas son las encargadas de asegurar que lo que se escribe en el lenguaje de dominio específico construido obedece a la gramática y a las reglas del negocio configuradas. Veamos la descripción de las características más relevantes de cada enfoque, de donde se obtiene referencia para la selección de uno de ellos.

2.2 Comparativo entre las dos metodologías marcadas que trabajan MDD

A continuación se presenta en la Tabla 2 las características relevantes de las dos metodologías enunciadas.

Tabla 2: Principales características de la metodología MDA y DSM dentro del desarrollo dirigido por modelos (Barceló, 2010), (Lin, Zhang, & Gray, 2004).

Características	Metodología	
	MDA	DSM
Promotor	OMG	No hay promotor
Propósito Principal	Portabilidad, interoperabilidad y reusabilidad de los modelos	Mejorar la productividad de los desarrolladores mediante el trabajo en un dominio de problema conocido
Enfoque	Diferentes niveles de abstracción hasta generar el código	Un modelo para la aplicación y otro para la arquitectura que se complementan para generar el código
Basado en estándares	Si (UML, MOF, CWM, QVT)	No
Independencia de plataforma	Si, mediante un modelo independiente de plataforma (PIM)	Parcial, aunque el modelado es específico de una plataforma, se pueden construir generadores para varias plataformas
Dirigido a un dominio de problema particular	No, enfoque generalista	Si
Herramientas disponibles	Si (disponibilidad alta)	Si (disponibilidad baja)
Costo de implementación	Medio, hay que dominar UML y construir las	Alto, es necesario definir el lenguaje de modelado y los

	transformaciones	generadores
Tipos de proyectos idóneos	Integración de aplicaciones o sistemas	Dominio del problema se estable con solución arquitectónica bien definida

2.3 Trabajos relacionados

Después de conocer las características principales de la metodología DSM podemos describir su potencial y poco uso en la actualidad (Deursen, Klint, & Visser, 2000), (Fraternali & Paolini, 2000), para hacer más clara esta afirmación podemos analizar la Tabla 3, que muestra los productos más relevantes que se encuentran operando en la actualidad bajo MDD, en este análisis podemos notar rápidamente que los trabajos para DSM son pocos comparados con los trabajos guiados por MDA.

Tabla 3: Listado de productos más relevantes bajo MDD que se encuentran operativos hasta el año 2009, 2010 y aún persisten (Barceló, 2010), (Rizvi & Hassan, 2009).

Metodología	Producto	Fabricante	Año	País
MDA	Acceleo	Obeo	2006	Francia
MDA	Ameos	Aonix	2007	USA
MDA	AndroMDA	Open Source	2003	USA
MDA	ArcStyler	Interactive Objects	2004	USA
MDA	AtoM3	Open Source	2004	USA
MDA	Codagen Architect	Manyeta	2005	USA
MDA	CodeGenie	Domain Solutions	2004	USA
MDA	Constructor MDARAD	i3 Desing	2005	USA
MDA	iQgen	innoQ	2002	Alemania
MDA	MCC	InferData	2003	USA
MDA	MDE	Open Source	2003	USA

MDA	MDWorkbench	Sodius	2003	USA
MDA	Mia Studio	Mia-Software	2007	Francia
MDA	ModFact	Open Source	2003	Francia
MDA	Objecteering/UML	Objecteering Software	2004	Francia
MDA	OlivaNOVA	CARE Technologies	2005	España
MDA	OpenMDX	Open Source	2004	USA
MDA	OptimalJ	Compuware	2003	USA
MDA	Rational Software Architect	IBM	2002	USA
MDA	Select Solution for MDA	Select Business Solutions	2002	USA
MDA	Together	Borland	2006	USA
MDA	UCL MDA Tools	Open Source	2003	USA
MDA	XFMosaic	Xactium	2003	USA
DSM	DSL Tools	Microsoft	2005	USA
DSM	Eclipsed EMF	Open Source	2006	USA
DSM	MetaEdit+	MetaCase	2000	USA
DSM	Mia DSL	Mia-Software	2009	Francia
DSM	Sculptor	Open Source	2006	USA
MDA/DSM	openArchitecturaWare	Open Source	2009	USA

Además de las herramientas descritas en la Tabla 3 (Barceló, 2010), (Rizvi & Hassan, 2009) también encontramos tres herramientas desarrolladas recientemente como lo son WebML, WebRatio y WebDSL que basan sus trabajos en la producción de código para aplicaciones web, estas herramientas aparecen en versiones estables a partir del año 2003, estableciendo características robustas y de liderato en los aspectos de desarrollo de software basado en modelos para aplicaciones con características web, que tienen como punto de entrada una base de datos, (Ceri & Daniel, 2003), (Murugesan &

Deshpande, 2002). A continuación se describe cada una de ellas e identificando el diferencial de esta herramienta.

2.3.1 WebML

Es una herramienta de generación automática de código y aplicaciones web, estas aplicaciones se generan a través de transformaciones que tienen por punto de entrada un modelo entidad relación o un diagrama de clases estandarizado por UML, se enfoca en el análisis de la estructura funcional de una aplicación web trabajando con los enlaces de navegación y sus características que permitan moverse de un página a otra, también involucra en su trabajo un enfoque de punto de entrada el modelado de negocios representado en diagramas BPMN que describan la lógica que va a ser generada en las diferentes aplicaciones que se deseen construir con esta herramienta, se basa en un esquema de trabajo marcado sobre la arquitectura dirigida por modelos, sus elementos son característicos para trabajar codificación en un entorno de trabajo integral buscando la generación del 100% de una aplicación funcional (Brambilla, 2010), (Brambilla, Comai, Fraternali, & Matera, 2013), su interés se centra en proporcionar aplicaciones funcionales sin la intervención de un programador; los elementos generados obedecen al lenguaje java. El código producido por la herramienta no está direccionado por una arquitectura específica. La herramienta se basa en el modelo de base de datos y el modelo de hipertextos. Se describen cada uno de estos modelos (Pérez, Irazábal, Pons, & Giandini, 2012):

- *Modelo de datos:* Esta herramienta tiene como base el modelo de una base de datos relacional que consta de dos elementos principales como lo son: las entidades representadas en rectángulos con su respectivo nombre y una lista de atributos y las relaciones representadas con líneas que proporcionan características de trabajo relacionado entre la entidades (Margaria, Winkler, & Kubczak, 2005).
- *Modelo de Hipertextos:* Este modelo es el eje central de la herramienta, representa las salidas, lo que la herramienta produce como instancia principal, se determinan las vistas y enlaces entre estas, el número de páginas y en general los elementos correspondientes a lo que verá y utilizará el usuario final de las aplicaciones que se generen con la herramienta. Es tan importante el enfoque de la operatividad usuario aplicación que se estudia bajo el modelo los enlaces contextuales, los enlaces de

transporte, los selectores y operadores que se pueden presentar en la funcionalidad generada para la web (Margaria, Winkler, & Kubczak, 2005), (Brambilla, 2010), (Brambilla, Comai, Fraternali, & Matera, 2013).

2.3.2 WebDSL

Esta herramienta constituye un lenguaje de dominio específico que soporta la generación de aplicaciones software web dinámicas, plantea en sus expectativas la capacidad para producir aplicaciones con una lógica de negocio definida la cual se puede modelar por medio de los artefactos de entrada que constituyen un modelo entidad relación para bases de datos, plantea en su esquema general reglas de validación en la funcionalidad y definición de páginas navegables, además de la interacción entre estas a través de enlaces, provee los pasos para la generación de código por medio de transformaciones a modelos (Schmidt, 2006). La generación de código de esta herramienta obedece al lenguaje de programación java, donde sus elementos se encuentran directamente contruidos para funcionar y operar con los usuarios finales de las aplicaciones generadas; la depuración de estas aplicaciones son de carácter individual y bajo el esquema de trabajo de la misma herramienta, no siguiendo una arquitectura de software se hace dispendioso el entendimiento de los elementos generados y puede incurrir en un trabajo tedioso el mantenimiento y/o actualización de las soluciones generadas (Brambilla, 2010), (Brambilla, Comai, Fraternali, & Matera, 2013).

2.3.3 WebRatio

Es una herramienta de uso comercial en su funcionalidad global y para el manejo de los desarrolladores de software, no obstante también presenta un uso libre en línea, donde se pueden generar aplicaciones web funcionales y a la vez tenerlas publicadas para trabajar. Al igual que WebML esta herramienta no sigue ninguna arquitectura de software para la generación de código fuente, aunque actualmente se soporta el lenguaje WebML dentro de WebRatio, su interés principal no está enfocado en la producción de artefactos dentro de un proceso de desarrollo de software, sino que va directamente a las aplicaciones funcionales. En la versión libre el código fuente es generado solamente en el lenguaje de programación de propósito general java (Brambilla, Ceri, & Fraternali,

2012). Actualmente se constituye como una plataforma que tiene presencia en varios países, promoviendo su uso y la adopción como marco de trabajo en las empresas que construyen software. Está construido a partir de modelos que se basan en la presentación de un esquema de negocio con BPMN y como punto de entrada un esquema entidad relación; aunque está desarrollada bajo el esquema de la arquitectura conducida por modelos, también permite que sus elementos de entrada sean construidos de forma textual y gráfica. Esta herramienta hace uso del estándar XML para soportar los modelos generados, sus transformaciones están descritas en clases java, y se encuentra como un conjunto de herramientas pequeñas que se pueden integrar al IDE Eclipse (Margaria, Winkler, & Kubczak, 2005), (Brambilla, 2010), (Brambilla, Comai, Fraternali, & Matera, 2013).

2.4 Características diferenciales de la herramienta a construir

En el medio informático de hoy existen varias herramientas capaces de generar código de aplicaciones a partir de especificaciones, ya sea partiendo de características de bases de datos, de características de entidades a nivel general, diagramas de clases, diagramas BPMN u otro aspecto para guiar un desarrollo de software, pero estas herramientas son especializadas en sus conceptualizaciones y los artefactos a generar en algunas ocasiones se enfocan a la aplicación en entornos definidos de productos clasificados como web, de escritorio, de servicios, entre otros. En el caso puntual de esta investigación el enfoque es modular y separado, proporcionando al desarrollador y a los gerentes de desarrollo de software herramientas de distribución de tecnologías y esquematización de las mismas dentro de los roles que se designan en un equipo de producción de software. La herramienta proporcionada en este proyecto cuenta con características puntuales y que pueden complementar las existentes en el mercado. En la Tabla 4 se presenta una comparación de las características relevantes de la herramienta construida (Margaria, Winkler, & Kubczak, 2005), (Brambilla, 2010), (Brambilla, Comai, Fraternali, & Matera, 2013).

Tabla 4: Características diferenciales de la herramienta

Característica	Web ML	Web DSL	Web Ratio	DSL propuesto
Generación de código para aplicaciones Web	X	X	X	X
Punto de entrada: modelo entidad relación	X	X	X	X
Generación para varios lenguajes				X
Generación de modelos para visualizar con otras herramientas				X
Generación de la navegación de páginas	X	X	X	
Código generado bajo la arquitectura de software				X
Generación de plantilla de reportes				X
Uso de varios puntos de entrada	X	X	X	
Enfoque de uso para desarrolladores y académico				X
Generación de componentes por separado			X	X

Los puntos diferenciales se enmarcan en que el código sea generado en el lenguaje java y en el lenguaje PHP, lo que permite una opción académica en el sentido que se puede comparar el funcionamiento y rendimiento de una aplicación en diferentes lenguajes; en el ámbito industrial no todas las empresas tienen sus software desarrollados con la tecnología java. Otro punto diferencial lo constituye el trabajo con la arquitectura MVC debido a que este permite la mejor distribución y codificación de los elementos de un software, al igual proporcionar componentes para la generación de reportes y esquemas de bases de datos visuales para ser gestionados desde otra herramienta (Margaria, Winkler, & Kubczak, 2005), (Brambilla, Comai, Fraternali, & Matera, 2013).

2.5 Metodología seleccionada dentro de MDD

En el presente trabajo de investigación se encaminan los esfuerzos para la generación automática de código mediante la metodología DSM dentro del desarrollo dirigido por modelos. Esta selección se hace basada en las características relevantes descritas en la Tabla 5.

Tabla 5: Listado de las características relevantes para la selección de la metodología

Característica	Justificación	Basado en
Mejorar la productividad de los desarrolladores mediante el trabajo en un dominio de problema conocido	Apoyo a la industria del software, y a la apropiación de los conocimientos actuales	La Tabla 2
Dominio del problema se estable con solución arquitectónica bien definida	Se evitan errores en la codificación y se presenta un lenguaje limitado y estructurado	La Tabla 2
Fortalecer el desarrollo de herramientas que trabajan bajo la metodología DSM	Se desarrolla una nueva herramienta en esta metodología	La Tabla 3
Se pueden aplicar esquemas para llegar a usuarios expertos, aprendices y solo con conocimientos del dominio	Se hace más estrecha la relación software – experto en el negocio proporcionando soluciones más especializadas	(Fraternali & Paolini, 2000) (Frankel, 2004)

De las características presentes en la Tabla 5 se enfoca este trabajo en un lenguaje de dominio específico, para lo cual se describen las bondades presentes al desarrollar software bajo MDD en concordancia a la metodología DSM:

2.6 Ventajas de trabajar con MDD

El no estar direccionado inicialmente a usar una plataforma, arquitectura de hardware y ninguna tecnología específica, hace de MDD un horizonte atractivo al desarrollador. La complejidad en las aplicaciones actuales hacen del trabajo del desarrollador y de la ingeniería de software un campo de mucha investigación, para alcanzar a suplir estos inconvenientes y producir a la vez software con calidad se han tomado rumbos distintos para encontrar la solución, y todos han llegado a un punto común que es la elevación en el nivel de abstracción de los problemas para las soluciones informáticas, lo que redundará para proyectos actuales en la aplicación de MDD para la producción de sistemas informáticos con mantenibilidad, portabilidad y escalabilidad, lo que refleja calidad de software. A continuación se presentan los aspectos relevantes en la ingeniería del software bajo la metodología DSM (Igor, Gordana, Branko, & Maja, 2010), (Stahl & Völter, 2006).

2.6.1 Lenguaje Específico del Dominio (Domain Specific Language DSL)

Como en todas disciplinas de la ingeniería en la de software existen enfoques generales para la solución de muchos problemas y enfoques para solución de problemas específicos o cierta línea de problemas. Cuando nacieron los lenguajes de programación como C, Fortran, entre otros, llegaban a resolver problemas específicos, pero se han convertido como los más nuevos Java, Scala, entre otros, en lenguajes de propósito general. A partir de estos casos surge la necesidad de implementar nuevos lenguajes que obedezcan a dominios definidos, lo que permite mayor eficiencia en la producción de una aplicación (Stahl & Völter, 2006), (Igor, Gordana, Branko, & Maja, 2010), para lo cual MDD toma a los DSLs como parte de sus implementaciones (Deursen, Klint, & Visser, 2000), (Kapitsaki, Kateros, Prezerakos, & Venieris, 2009).

Trabajar los DSL enmarcados en MDD brinda como resultado ventajas para los desarrolladores de software, de las cuales se presentan las más relevantes a continuación:

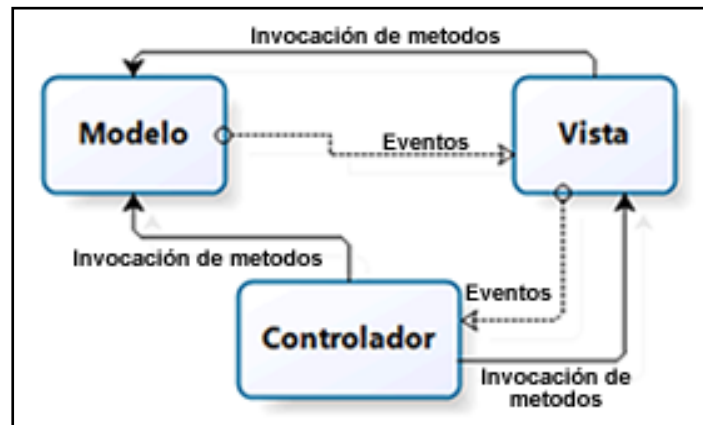
- La mayoría de los problemas en la industria del software están dados en la comunicación existente entre el experto en el negocio y los encargados de las implementaciones técnicas; este proceso se lleva de forma manual donde se pueden presentar errores, si se usa un DSL que implemente la abstracción adecuada y el uso de los términos propios del dominio puede directamente interactuar con el experto del negocio.
- Cuando se trabaja con DSL estamos garantizando la portabilidad, mantenibilidad y reusabilidad de aplicaciones, lo que permite terminar con el caos que causan los avances y cambios en la tecnología que obligan a las aplicaciones a ser actualizadas o migradas generando altos costos en sistemas (Igor, Gordana, Branko, & Maja, 2010), (Deursen, Klint, & Visser, 2000), los cuales se ven reflejados en tiempo y dinero.
- El nivel de abstracción en MDD es alto lo que permite solucionar el problema una vez en el modelo más abstracto, y luego a partir de transformaciones se produce de forma automática el código, con mayor productividad del equipo que provee las soluciones software (Barceló, 2010), (Deursen, Klint, & Visser, 2000).
- Trabajar con un DSL brinda seguridad para que el desarrollador realice sus tareas estando limitado a usar las características de su dominio y no tiene que manejar estructuras complejas de un lenguaje de propósito general, aumentando así la calidad de las aplicaciones debido a que no es validado un posible error incluyendo sentencias o especificaciones no permitidas (Barceló, 2010), (Igor, Gordana, Branko, & Maja, 2010).

Después de conocer las características relevantes del trabajo con MDD, las metodologías que lo implementan y en especial DSM, es pertinente estudiar la arquitectura en la cual se basa este trabajo, de la cual se hace la presentación a continuación.

2.7 Arquitectura Modelo Vista Controlador (MVC)

En el desarrollo de software es muy común usar una arquitectura para la elaboración de aplicaciones, una de las usadas y aceptadas por la comunidad académica e industrial es el que se presenta en la Figura 5.

Figura 5: Enfoque gráfico de la arquitectura MVC



Cuando se hace desarrollo de software se pueden identificar tres áreas en las cuales el usuario final de la aplicación hará su implementación, aunque no lo perciba mientras realiza sus tareas (Rizvi & Hassan, 2009), los cuales se describen en el MVC. El dominio de la aplicación se denota por el modelo que hace una representación directa del conocimiento propio del negocio, así también se perciben los estados de la aplicación para la interacción con los usuarios finales llamadas vistas y el encargado de comunicar e interactuar entre las vistas y los modelos conocido como controlador (Rizvi & Hassan, 2009), (Deacon, 2009), (Gupta & Govil, 2010). Trabajar con esta arquitectura brinda características que aportan facilidad para el trabajo con paradigmas tales como orientado a objetos. A continuación se describen las características básicas de MVC.

- Al igual que la filosofía de MDD, la arquitectura MVC busca una independencia de la lógica del negocio, la presentación y diseño de aplicaciones (Rizvi & Hassan, 2009), dando una facilidad para el mantenimiento y la escalabilidad.
- El controlador permite la interacción entre las vistas y el modelo, manteniendo la aplicación funcional aun cuando se hagan cambio en uno de ellos, solo con el paso

de notificaciones las representaciones se ejecutan, permitiendo reutilizar componentes de la aplicación (Deacon, 2009).

- Facilita el trabajo en equipo, hace el código entendible entre el grupo de desarrolladores que usan la arquitectura (Gupta & Govil, 2010).
- Las pruebas en las aplicaciones que siguen la arquitectura MVC son más fáciles de realizar y automatizar (Rizvi & Hassan, 2009), (Gupta & Govil, 2010).

Conociendo las características de la arquitectura de desarrollo de software seleccionada que se complementa de forma eficiente con las características de DSM, se pasa a describir los entornos de trabajo posibles.

2.8 Entorno integrado de desarrollo Eclipse

Al conceptualizar sobre Eclipse hay que mencionar que además de ser un IDE que brinda un entorno donde el programador puede hacer desarrollo de software en un lenguaje como java o C++, también se constituye en un marco de trabajo donde se integran nuevos lenguajes que se van desarrollando, debido a que provee una infraestructura para crear IDEs, esto lo convierte en un fuerte candidato para la generación de lenguajes de programación. Este IDE es el usado en esta investigación para realizar la implementación del DSL.

2.9 Modelado textual (Xtext)

De los dos aspectos principales que enmarcan el trabajo con MDD, tenemos la construcción de los modelos y la verificación que estos funcionen correctamente, para esta verificación contamos con una herramienta como marco de trabajo llamada Xtext que es la evolución del proyecto openArchitectureWare relacionado en la Tabla 4 (Willink, 2010), (Efftinge & Völter, 2006). Xtext es un marco que nos permite desarrollar DSL, estos nuevos lenguajes que se producen contienen características que enriquecen el trabajo del desarrollador de aplicaciones, tales como: Tomando la notación EBNF como base para proporcionar un analizador y un editor para leer y escribir sintaxis textual, con vista y completado de código así como manejador de error (Efftinge & Völter, 2006).

Con Xtext el trabajo se hace con alto nivel de aplicabilidad de la ingeniería de software dado que se puede hacer separación del analizador de sintaxis y la vinculación de referencias cruzadas, el trabajo consiste en la descripción de la gramática definida en una sintaxis textual el meta-modelo, y luego producir las vinculaciones que se requieren para la aplicación. Se pasa a la generación del editor que se puede entregar al desarrollador, donde realizará las implementaciones necesarias, lo que se denomina como el DSL construido, en donde se demuestra la funcionalidad en la producción del código a través de modelos (Efttinge & Völter, 2006), (Willink, 2010).

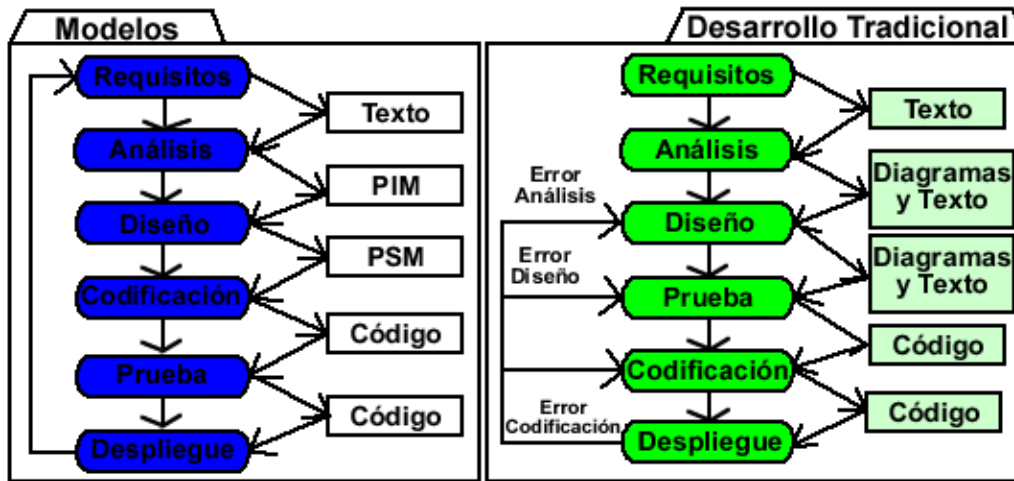
2.10 Conclusiones del capítulo

Este capítulo nos ilustra sobre cómo se ha trabajado el tema MDD a nivel general y cómo se puede hacer desarrollos bajo esta tecnología. Se describe la metodología DSM para la construcción de DSL capaces de proporcionar facilidades a los desarrolladores; enfatiza sobre las ventajas y como la aplicación de una arquitectura como lo es MVC, puede ser implementada de forma fácil en software que se basen en MDD.

Una solución software a nivel industrial puede ser dirigida por el negocio sin dependencia de la tecnología (Kumar & Mohan, 2012). Tal postura se centra en el proceso ocurrido al pasar del modelo independiente de la computación a modelos independientes de la plataforma (Cañadas, Palma, & Túnez, 2011).

Para el desarrollador actual el MDD se convierte en una herramienta de gran ayuda porque permite mayor facilidad al hacer mantenimiento de su código, poder replantear las soluciones desde el punto de vista del negocio y no por la dependencia de una tecnología, le permite hacer la abstracción necesaria para resolver los problemas y el aprovechamiento de características tales como reusabilidad de los artefactos producidos, portabilidad de su software e interoperabilidad de los artefactos generados (Figura 6).

Figura 6: Proceso de desarrollo con MDD (Lin, Zhang, & Gray, 2004), (Moreira & Rossi, 2004) Vs proceso de desarrollo tradicional (Cañadas, Palma, & Túnez, 2011), (Torres, Muñoz, & Pelechano, 2005).



En la Figura 6, se aprecia la equivalencia comparativa entre dos tendencias en el desarrollo de software, se puede apreciar que en el desarrollo tradicional se tienen varias etapas para la depuración de errores debido a que se presenta la intervención humana y se ve un desgaste en la corrección de los artefactos generados en cada fase, mientras que en el desarrollo dirigido por modelos la corrección de errores se hace una sola vez en el nivel de análisis y entendimiento del negocio donde se construye el modelo y luego los artefactos subsiguientes son generados con alto grado de calidad, aumentando la productividad en el proceso de construcción de software.

3. Capítulo 2

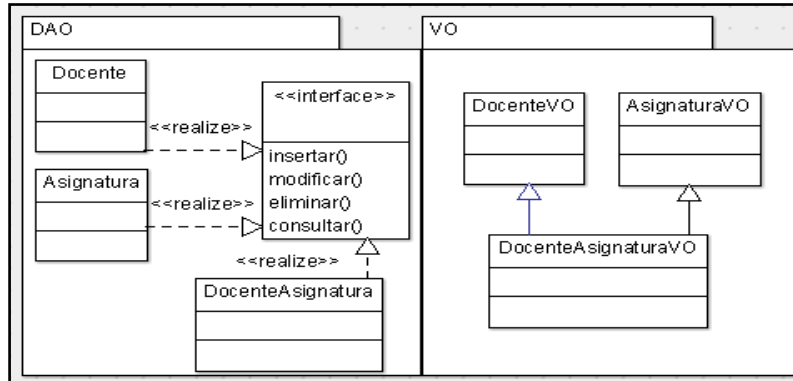
Implementación de la Herramienta de Desarrollo De Software

Después de estudiar los conceptos básicos del trabajo con MDD, se procede a realizar la implementación de la teoría en un caso de estudio específico, conocido como implementación de referencia. Con el desarrollo de esta herramienta se busca una aproximación a la generalidad de aplicaciones que se puedan modelar y realizar una producción del código fuente de forma automática.

3.1 Selección del tema para la implementación de referencia

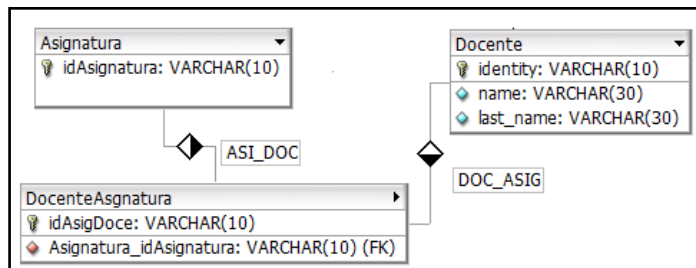
El tema seleccionado para la implementación de referencia hace mención a un caso de estudio donde se necesita crear una aplicación para el registro de docentes y asignaturas, al igual que la asignación de cada docente a la asignatura que desarrolla; el tema y caso de estudio puntual son seleccionados porque constituyen un esquema general que puede ser entendido y seguido fácilmente, lo que permite que esta investigación sea base para nuevos trabajos. Se procede a la elaboración del diagrama de clases como se muestra en la Figura 7, donde se especifican los componentes del software a construir.

Figura 7: Diagrama de clases para la implementación de referencia



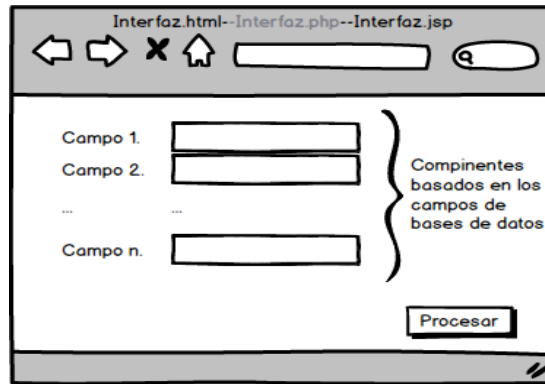
Se procede a la construcción del modelo relacional de la base de datos donde se van a almacenar cada uno de los involucrados en la solución planeada, también se especifican las propiedades que van a ser capturadas a la hora de ejecución de la aplicación funcional como se muestra en la Figura 8.

Figura 8: Modelo relacional de la implementación de referencia



Se realiza el diseño de las interfaces de usuario y se procede a concretizar cada una de ellas, al igual que los otros artefactos necesarios para desarrollar la aplicación, estas son presentadas en la Figura 9.

Figura 9: Diseño de las interfaces de usuario



Estas interfaces al igual que la codificación de la aplicación pueden ser implementados en código puro (natural), o se pueden usar entornos integrados de desarrollo (IDE) para la facilidad del programador, en nuestro caso describiremos las herramientas utilizadas en la implementación de referencia en la Tabla 6.

Tabla 6. Herramientas usadas en la implementación de referencia

Logo	Herramienta	Uso	Justificación
	ArgoUML 0.38	Creación de diagramas de clases	Uso libre, para construir modelos y esquemas de aplicaciones académicas
	DBDesigner 4.0	Diseño del modelo entidad relación y producción de script para la creación de bases de datos	Uso libre, permite la generación de código SQL para MySQL
	NetBeans 8.0	Desarrollo de las interfaces, y las clases necesarias tanto en JAVA, como en PHP	Uso libre, robusta, uso popular en la academia
	Ireport 5.0	Para diseñar las plantillas de reportes	Uso libre, robusta, popular en la industria y la academia

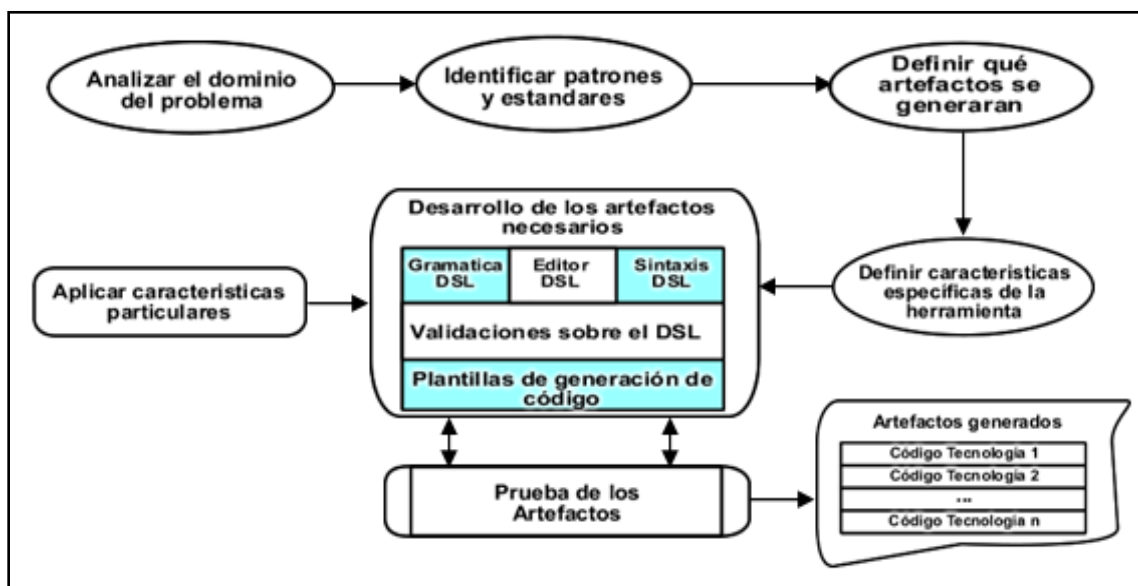
3.2 Implementación de referencia

Caso de trabajo: generación de plantillas para aplicaciones, partiendo de la base de datos. El desarrollo sobre tecnología web y la particularidad de multiplataforma nos lleva a pensar en una solución que proporcione un lenguaje de dominio específico que tenga plantillas de código para desarrollo de aplicaciones de forma ágil; en ello también se tiene en cuenta la interacción entre personas y el diseño visual, para lo cual se enfoca la aplicación DBDesigner 4.0, tomada como base para realizar el prototipo de implementación de referencia. En esta herramienta se puede producir una representación gráfica de los objetos de bases de datos para servidores MySQL.

3.3 Metodología a seguir en la elaboración de la herramienta

Para iniciar con la herramienta debemos describir la metodología a utilizar basada en DSM de MDD. El procedimiento de elaboración de la herramienta hace uso de la metodología DSM la cual se presenta en forma específica para esta investigación como se muestra en la Figura 10, la cual se sigue paso a paso en el desarrollo.

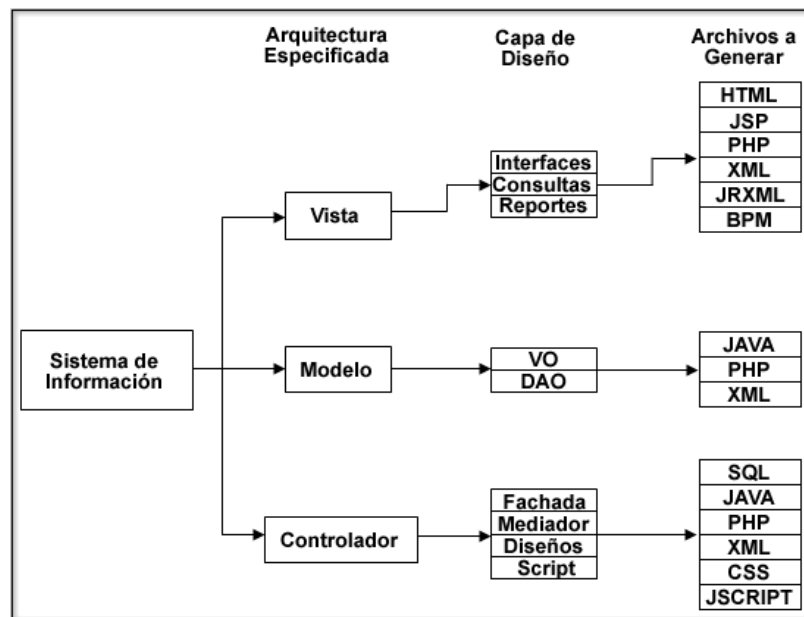
Figura 10: Metodología para la elaboración de la herramienta



3.4 Análisis del dominio del problema y selección de artefactos a generar

Para la implementación de referencia se pretenden generar archivos que corresponden a las características que se pueden modelar como una generalización del problema, para el cual se construye un meta modelo común donde cambien aspectos muy específicos para cada lógica de negocio. Se ilustra en la Figura 11.

Figura 11: Esquema de los archivos de la implementación de referencia



3.5 Identificación de patrones dentro de la implementación de referencia

Esta identificación se realiza basada en lo que indica la metodología DSM, e identificamos el código esquemático repetitivo, el genérico y el individual.

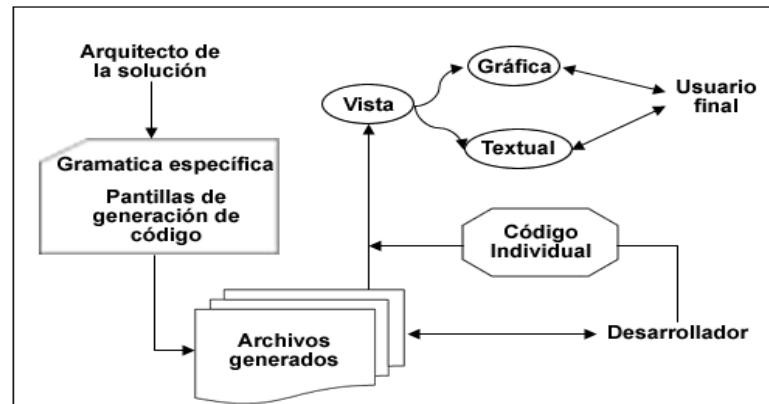
- El código esquemático se puede generar a partir de una serie de elementos bien definidos, en la implementación de referencia lo encontramos en los campos y

atributos de las vistas, clases, tablas, columnas entre otras, por ejemplo todas las vistas tienen campos y se definen de forma general, lo que la diferencia son los atributos definidos por el usuario final al momento de darle su función. Ver Anexo B.

- El código genérico es el que se identifica como base para la construcción de un archivo específico y no cambia siempre es el mismo para cualquier implementación, por ejemplo para incluir código JAVA en JSP siempre se usa `<% //código %>` y para PHP `<?PHP //código ?>`.
- El código individual es aquel que no puede ser generado fácilmente, en su mayor porcentaje está ligado a la lógica de negocio que implementa cada aplicación y es el que debe ser incluido por un programador para lograr la completitud del código necesario en la ejecución correcta de una aplicación generada, en este caso como ejemplo se detallan las configuraciones de conexión a servidores de bases de datos y en los reportes específicos necesarios en la funcionalidad.

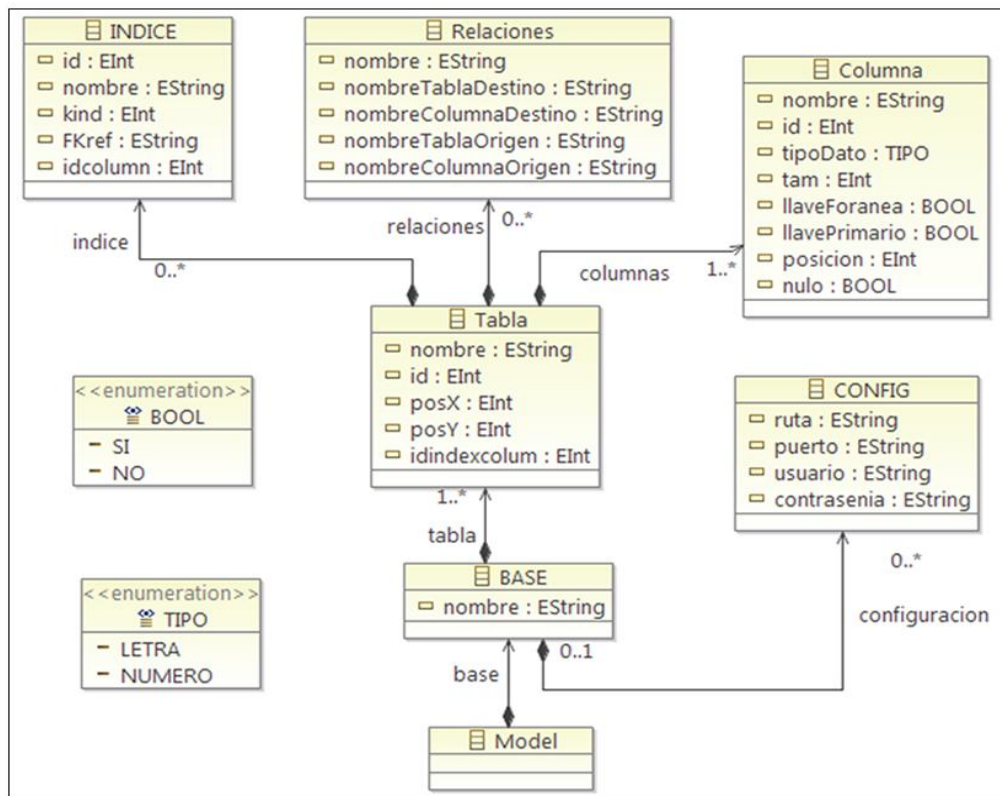
3.6 Características de la herramienta y creación de la arquitectura de la solución

En este punto ya se han identificado los artefactos susceptibles de ser generados y el porcentaje posible de generación de ellos, ahora se pasa a la forma de hacerlo, ésta forma se conoce como la determinación del meta modelo informalmente, el cual plantea la arquitectura a seguir a grandes rasgos para la elaboración de la herramienta de generación de código, en la Figura 12 se muestran las capas necesarias para esta solución; estas capas y pasos no necesariamente deben ser idénticas a la solución final del meta modelo concreto, pero si una aproximación válida basada en el análisis de la implementación de referencia y lo que se desea lograr con el DSL.

Figura 12: Solución planteada informalmente

3.7 Diseño e implementación del Lenguaje de Dominio Específico

Al determinar el meta modelo informal y la clasificación del código de la implementación de referencia, se procede a construir la herramienta, para lo cual se utiliza Xtext en la definición del meta modelo que se muestra en la Figura 13, para usarlo se toma como base el entorno de desarrollo integrado Eclipse Juno, en el cual se pueden implementar DSL en Xtext (Willink, 2010).

Figura 13: Meta modelo formalmente definido

Esta definición nos permite tener una gramática definida que seguirá la herramienta, el usuario del DSL puede crear un modelo que puede contener cero(0) o una(1) base de datos, este es el punto de partida para que la herramienta pueda producir artefactos, al incluir una base de datos debe gestionar por lo menos una tabla que a su vez esta tenga por lo menos una columna. A partir de esta definición básica se pueden generar los artefactos para aplicaciones funcionales, pero la herramienta brinda también la posibilidad de agregar relación entre entidades de bases de datos, índices e involucra tipos de datos definidos por la gramática la cual se puede ver de forma compacta en la Figura 14, nos muestra el listado específico de la gramática construida con el meta modelo.

Para este meta modelo que obedece a una gramática también puede ser construido y editado de forma textual que en ocasiones es más utilizado debido a las reglas o características del DSL que se pueden configurar, como lo son el uso de valores

terminales, la limitación a ciertas opciones y la comunicación o línea de interacción entre los elementos, una vista gráfica del código implementado se muestra en la Figura 14.

Figura 14: Código de gramática en la herramienta

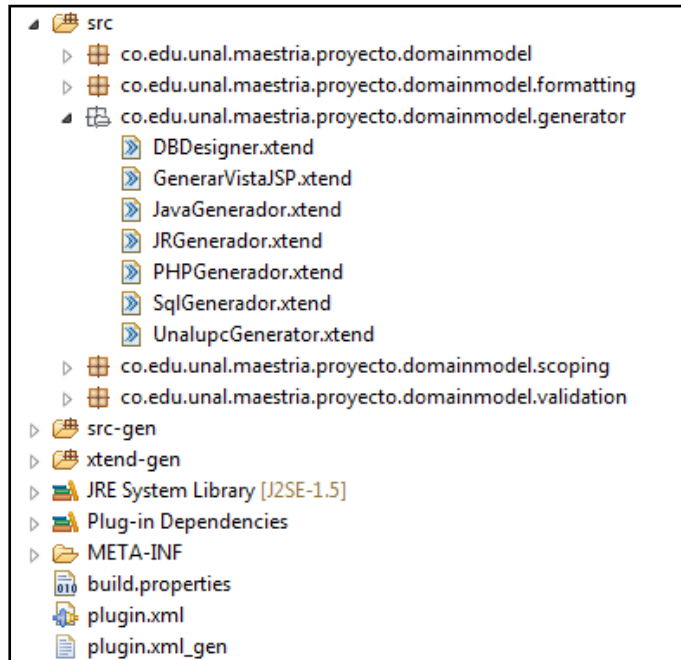
```
Model: base = BASE;
BASE:
  'base' nombre=ID '{'
    (tabla+=Tabla)+
    (configuracion+=CONFIG)?
  '}'
;
Tabla:
  'table' nombre=ID '{'
    ('id' id=INT)?
    ('pos_x' posX=INT)?
    ('pos_y' posY=INT)?
    ('index_colum' idindexcolum=INT)?
    (columnas+=Columna)+
    (relaciones+=Relaciones)*
    (indice+=INDICE)*
  '}'
;
```

La construcción de la herramienta en este punto puede generar los archivos de flujo de trabajo para el DSL, y al correr una instancia de la esta es capaz de reconocer la gramática definida, de limitarnos a trabajar solo con los elementos definidos siempre y cuando en el proyecto sobre en lenguaje específico se defina un archivo del mismo tipo que hemos construido; aunque se pueda pasar a pruebas aún faltan componentes que le dan forma a lo que se propone lograr, estos generarían el código fuente para aplicaciones, basado en lo que el usuario digite en el entorno de trabajo implementado. También es necesario realizar las respectivas validaciones sobre la escritura en el DSL para minimizar los errores que pueda cometer un usuario en determinado momento; un punto importante aunque no determinante es poder configurar que nuestra sintaxis se presente de forma agradable y que funcione como apoyo a la escritura sobre el DSL; para esto es necesario construir y configurar nuestro propio esquema, con colores y estilos para las diferentes partes del código escrito, por ejemplo que los números tengan un color diferente a los valores alfabéticos. A continuación se presenta cómo fueron elaborados los artefactos y qué contribución tienen en la herramienta.

3.8 Plantillas de generación de código automático

En esta apartado del DSL se busca poder generar el mismo archivo XML que se construye en la implementación de referencia y además el código en Java para los modelos identificados en nuestro trabajo como VO, la vista en HTML y el controlador para los lenguajes propuestos. Estas plantillas analizan el código digitado por el usuario y hacen la producción del código necesario para ser interpretado y visualizado en las herramientas donde fue elaborada la implementación de referencia, además que este lenguaje a partir de una base de datos haga la producción automática del código SQL para MySQL y PostgreSQL como servidores de bases de datos, también la producción automática de las clases entidades y de las plantillas para los DAO y VO en JAVA de las tablas, al igual que las plantillas JSP de una aplicación software a partir de la base de datos. Esta herramienta también provee la separación de artefactos y estructuración de ellos en la arquitectura MVC, la separación tanto de tecnologías como de implementaciones, permitiendo la producción de código en paralelo de artefactos para PHP, JavaScript, CSS y plantilla de reportes en JRXML, así como el código necesario para la construcción del modelo de negocio.

Estas plantillas están construidas bajo el concepto de la metodología DSM donde se plasman el código genérico, el código individual y el código esquemático. En la Figura 15 se muestra el listado de plantillas construidas en un lenguaje llamado Xtend que permite tomar en tiempo real las características digitadas en un DSL y plasmarlas como se necesiten en el archivo específico.

Figura 15: Listado de plantillas para la generación del código de forma automática

En la Figura 16 se muestra una porción de código fuente en Xtend que es capaz de producir de forma automática el código SQL para la base de datos en MySQL a partir de los datos digitados por el usuario en el DSL. Estas plantillas son determinantes para la generación de aplicaciones, son las encargadas de traducir de la gramática escrita a los archivos propuestos a generar.

Cada componente o tecnología a generar puede hacer uso de una plantilla de este tipo, pero también pueden ser recopilados en un archivo único como el arquitecto de la herramienta considere conveniente según el dominio que se modela. El código implementado en Xtend, es diferente al que se confecciona en Xtext y se debe tener cuidado para no confundirse en el uso de los dos al mismo tiempo.

Figura 16: Código fuente de las plantillas para la generación del artefactos de forma automática

```
class SqlGenerador {
    def MySQL (BASE base)'''
CREATE DATABASE «base.nombre»;
USE «base.nombre»;
«FOR tabla:base.tabla»
CREATE TABLE «tabla.nombre» (
«FOR columna:tabla.columnas»
«columna.nombre»«IF columna.tipoDato==TIPO»
«ENDFOR»
«FOR columna:tabla.columnas»
«IF columna.llavePrimario==BOOL::SI»
,PRIMARY KEY(«columna.nombre»)
«ELSE»
«IF columna.llaveForanea==BOOL::SI»
,FOREIGN KEY(«columna.nombre»)
REFERENCES «FOR relacion:tabla.relaciones»«IF
ON DELETE NO ACTION
ON UPDATE NO ACTION
«ENDIF»
«ENDIF»
«ENDFOR»
);
«ENDFOR»
'''
    def PostgreSQL (BASE base)'''
}
```

Hasta este punto se tiene el DSL construido y generando código fuente para diferentes tecnologías, se tienen las limitaciones propias de uso que corresponde solo a la gramática definida. En este punto el desarrollador usuario no tendrá espacio a incluir código inválido como lo puede hacer en un lenguaje de propósito general, pero aún se pueden cometer errores que la herramienta no está validando. No de carácter sintáctico sino de carácter semántico, por ejemplo un usuario puede definir dos o más entidades de base de datos con el mismo nombre, o dos atributos de una entidad con el mismo nombre o una relación a entidades no existentes, es por ello la necesidad de hacer este tipo de validaciones que van ligadas a la lógica del negocio y le dan a nuestra herramienta robustez al momento de entrar en producción. Para lograr estas características debemos trabajar sobre otro lenguaje de verificación, contenido dentro de java y que tiene unos parámetros definidos y etiquetas validas de configuración como lo es la etiqueta @check que permite al DSL en tiempo real hacer las validaciones particulares que se configuren, para ilustración en la Figura 14 se definen dos métodos de validación, uno de alerta o preventivo y uno de error los cuales se reflejaran instantáneamente se cometan dentro de la herramienta de generación de código.

Figura 17: Código fuente de validaciones

```
@Check
public void verificarLetraInicial(Tabla table) {
    if (!Character.isUpperCase(table.getNombre().charAt(0))) {
        warning("Se sugiere que el nombre de las entidades inicie por una mayuscula", null);
    }
}

@Check
public void verificarNombreUnicoTabla(Tabla table) {
    Model modelo;
    EObject object = EcoreUtil.getRootContainer(table);
    if (!(object instanceof Model))
        return;
    modelo = (Model) object;
    for (Tabla tabla : modelo.getBase().getTabla()) {
        if (tabla == null || tabla == table)
            continue;
        if (table.getNombre().equals(tabla.getNombre())) {
            error("Duplicación de nombre : " + table.getNombre() + "[" + table.getId() + "]",
                null);
            return;
        }
    }
}
```

Para tener una herramienta con características robustas, es conveniente construir los archivos de configuración de colores para la sintaxis que se va digitando, pero lo cual se crea en el proyecto de interfaz de usuario un archivo java que contiene las configuraciones de color y tipo de fuente que se usará para determinados atributos que se digiten en el DSL.

3.9 Pruebas sobre el DSL construido

Para realizar las respectivas pruebas sobre la herramienta desarrollada se den seguir tres pasos fundamentalmente.

Paso 1: Verificar que los archivos construidos no tienen ningún tipo de error.

Paso 2: Generar los artefactos del DSL, este paso se ejecuta, pero el encargado de hacer el proceso de fondo es el IDE Eclipse donde se concretiza la implementación.

Paso 3: Desplegar una nueva instancia del IDE para iniciar la programación sobre la herramienta construida.

Seguido los pasos propuestos en su respectivo orden se está frente a la nueva herramienta creada, que proporciona una interfaz para digitar el código que se toma como base en la construcción de las aplicaciones recordando que debe cumplir con la gramática definida, en la Figura 18 se presenta el lenguaje construido y la edición de un archivo para el cual se crea una base de datos, dentro de ella una tabla y sobre la tabla una columna. Se puede verificar que los elementos soportados en el lenguaje son solo los especificados en la gramática y que el editor textual toma el formato configurado de la sintaxis con las fuentes y colores definidos para tal fin.

En la Figura 18 se presentan las características configuradas del lenguaje construido donde se hay errores y advertencias por el uso inadecuado de la herramienta, estos mensajes son los configurados manual y específicamente para esta herramienta, y las validaciones de uso general que toma un DSL del orden de anotación de los elementos y la validación frente a la gramática especificada. Con la Figura 18 se puede verificar las pruebas sobre la herramienta y la operatividad de esta; para las pruebas de funcionalidad y generación de los artefactos propuestos se hace la ampliación y explicación respectiva en el capítulo 4.

Figura 18: Pruebas sobre el DSL

```

base instituto{
  table Docente {
    columna identidad{
      id 1
      tipoDato letras
      largo 150
      llaveForanea no
      llavePrimaria si
      nulo no
    }
  }
}

```

```

base instituto{
  table docente {
    columna identidad{
      id 1
      tipoDato letras
      largo 150
      llaveForanea no
      llavePrimaria si
      nulo no
    }
  }
}

```

id 2

id 3

id 4

id 5

id 6

id 7

id 8

id 9

Se sugiere que el nombre de las entidades inicie por una mayuscula
Press 'F2' for focus

3.10 Conclusiones del capítulo

Este capítulo proporciona el esquema general para la construcción de un lenguaje de dominio específico capaz de producir los elementos necesarios en la generación de artefactos reutilizables en la construcción de software. Se identifican los puntos principales para el éxito en la creación de una herramienta de producción de código fuente como lo son:

- El punto de entrada de una herramienta para la generación de código determina el meta modelo y proporciona las características iniciales para la construcción de una gramática definida.
- Una implementación de referencia bien definida e implementada es fundamental para la construcción de una herramienta de generación de código fuente.
- La validación del lenguaje construido permite asegurar la calidad con que se producen los elementos.
- Las plantillas de generación de código deben obedecer a la implementación de referencia, según los elementos clasificados como esquemático, genéricos e individuales.

En la creación de una herramienta de generación de código se hace una representación abstracta de un problema que obedece a la generalización, y sus componentes trabajan para dar características particulares a cada aplicación generada a partir de la especificación realizada por medio del punto de entrada.

Se ha creado un DSL que permite la codificación determinada por una gramática definida sobre archivos del tipo unalupc, cuyo tipo fue definido específicamente para este lenguaje y todos los archivos de este tipo en el DSL deben obedecer las características configuradas para él, al igual que producir los elementos definidos en las plantillas de generación de código.

4. Capítulo 3

EVALUACIÓN DE LA HERRAMIENTA

Para la elaboración de las pruebas sobre la herramienta, se tiene en cuenta tres casos que van aumentando el grado de complejidad del tema propuesto y a la vez permiten hacer una evaluación objetiva, debido al cambio de tipo de aplicación que se genera en cada caso particular. A continuación se presentan los criterios para la selección de las pruebas.

4.1 Selección de pruebas

En este punto se tiene en cuenta la complejidad de las pruebas y la versatilidad de las mismas, se inicia con el caso de menor grado de complejidad y a medida que se pasa a otra prueba se aumenta la complejidad. Es de anotar que las pruebas seleccionadas deben ser aplicadas en el DSL construido.

Las pruebas presentadas para la evaluación de la herramienta no son pruebas convencionales sobre un software desarrollado, son pruebas de generación de código y artefactos que deben ser validados bajo una arquitectura y una metodología, este código además debe ser validado en la herramienta; también debe compararse puntualmente con el código de la implementación de referencia. Por ello que se presentan los siguientes tres casos de pruebas.

4.1.1 Caso I (Aplicación básica, un solo componente)

Para este caso se plantea un solo componente, el cual se implementa en la herramienta y obedece a la creación de una entidad vehículo con los atributos básicos, su representación gráfica está en la Figura 19 en el costado izquierdo, se escoge este ejemplo debido a que es sencillo para el inicio de las pruebas, y los atributos son mínimos, pero a partir de esta representación se pueden generar artefactos con la completitud de los elementos necesarios para la funcionalidad y otros con alto porcentaje de completitud.

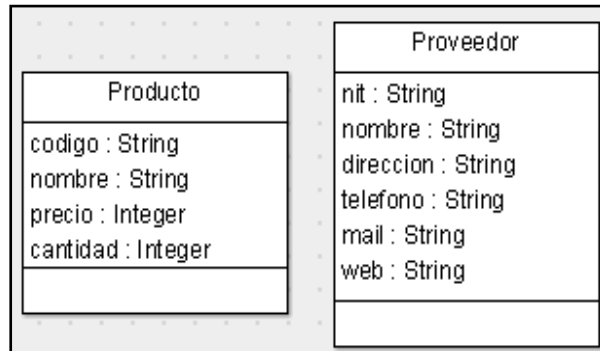
Figura 19: Caso de prueba I

Producto	Ensambladora.unalupc
color : String	base Ensambladora {
placa : String	table Vehiculo {
modelo : Integer	columna modelo { tipoDato numero largo 4 }
	columna color { tipoDato letras largo 30 }
	columna placa { tipoDato letras largo 10 llavePrimaria si }
	}
	}

Para esta representación pasamos a la implementación en la herramienta DSL construida, se debe tener en cuenta la gramática definida debido a que será validada por el editor textual. En la Figura 19 en el costado derecho se muestra el código digitado sobre el DSL en un archivo del tipo especificado para la herramienta.

4.1.2 Caso II (Aplicación con dos o más componentes)

El caso de prueba dos plantea la creación de una aplicación para la gestión de productos y proveedores, esta especificación permite analizar desde el ángulo de varios elementos representados dentro del proyecto, donde estos tienen atributos diferentes, y no hay relación entre ellos; la representación del caso de prueba se encuentra descrito en la Figura 20 donde se verifican los dos elementos que se desean tener en cuenta para esta aplicación.

Figura 20: Caso de prueba II

Plasmar este caso de prueba en la herramienta nos permite identificar la pluralidad de la herramienta y cómo la gramática reflejada en el meta modelo permite agregar varios elementos, esto se nota en la codificación digitada dentro de la herramienta DSL y se puede verificar en la Figura 21.

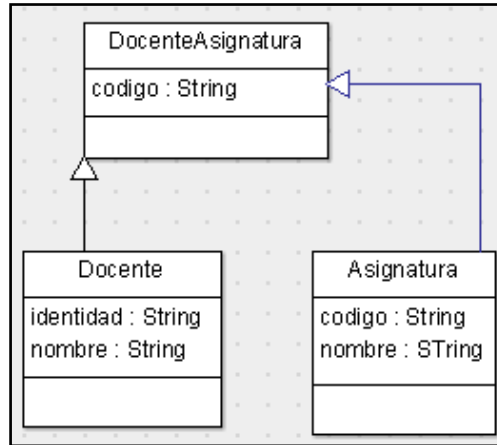
Figura 21: Código del caso de prueba II

```

*Almacen.unalupc
base Almacen {
  table Producto {
    columna codigo {tipoDato letras largo 20 }
    columna detalle {tipoDato letras largo 100 }
    columna precio {tipoDato numero largo 10 }
    columna cantidad {tipoDato numero largo 6 }
  }
  table Proveedor {
    columna nit {tipoDato letras largo 20 }
    columna detalle {tipoDato letras largo 100 }
    columna direccion {tipoDato letras largo 100 }
    columna telefono {tipoDato letras largo 15 }
    columna mail {tipoDato letras largo 100 }
    columna web {tipoDato letras largo 100 }
  }
}
  
```

4.1.3 Caso III (Aplicación con dos o más componentes y relaciones)

El tercer caso permite hacer una aplicación similar a la implementación de referencia tomada como base para la construcción del DSL que se está probando; la aplicación del caso de prueba tiene más de dos componentes y estos están relacionados entre sí permitiendo mayor grado de complejidad de elaboración en la aplicación, en la Figura 22 podemos observar de forma gráfica los elementos que deben estar en la creación de la aplicación del caso de prueba presentado.

Figura 22: Caso de prueba III

Esta aplicación implementada en el DSL construido, tiene una apariencia más compleja que da la relación entre elementos, esto lo podemos observar en la Figura 23 que proporciona una vista de lo que debe hacer el programador al momento de utilizar la herramienta, limitado solo a cumplir la gramática correspondiente, digitando los elementos de tipo entidades y sus atributos definidos de forma concreta y correcta.

Figura 23: Caso de prueba III Código en el DSL

```

Colegio.unalupc
base escuela {
  table Docente {
    columna identidad { tipoDato letras largo 20 llavePrimaria si }
    columna nombres { tipoDato letras largo 80 }
    columna proesion { tipoDato letras largo 100 }
    columna telefono { tipoDato letras largo 20 }
    relaciones DOC_ASI {
      nombreTablaDestino 'DocenteAsignatura' nombreColumnaDestino 'docente_id'
      nombreTablaOrigen 'Docente' nombreColumnaOrigen 'identidad'
    }
  }
  table Asignatura {
    columna codigo { tipoDato letras largo 20 llavePrimaria si }
    columna detalle { tipoDato letras largo 100 }
    columna codigo { tipoDato letras largo 20 }
    relaciones ASI_DOC {
      nombreTablaDestino 'DocenteAsignatura' nombreColumnaDestino 'adignatura_id'
      nombreTablaOrigen 'Asignatura' nombreColumnaOrigen 'codigo'
    }
  }
  table DocenteAsignatura {
    columna codigo { tipoDato letras largo 10 llavePrimaria si }
    columna docente_id { tipoDato letras largo 20 llaveForanea si }
    columna adignatura_id { tipoDato letras largo 20 llaveForanea si }
  }
}

```


4.2 Características de las pruebas

Las aplicaciones a generar con la herramienta DSL están especificada con las características descritas en la Tabla 7, cuyos elementos obedecen a la utilización de la gramática especificada según el meta modelo construido.

Tabla 7: Características de las pruebas

No.	Característica
1	Los elementos deben cumplir la gramática del meta modelo
2	La construcción sintáctica debe ser correcta
3	La implementación de referencia es la guía de pruebas
4	Cada caso de prueba debe obedecer a una aplicación diferente
5	Las pruebas son válidas si obedecen a un lógica similar a la descrita en la implementación de referencia
6	La aplicación de pruebas es sobre el DSL y los artefactos generados
7	Las pruebas pueden ser repetidas
8	Cada caso de prueba debe incluir por lo menos un elemento para generar archivos de cada tipo

4.3 Generación de Archivos

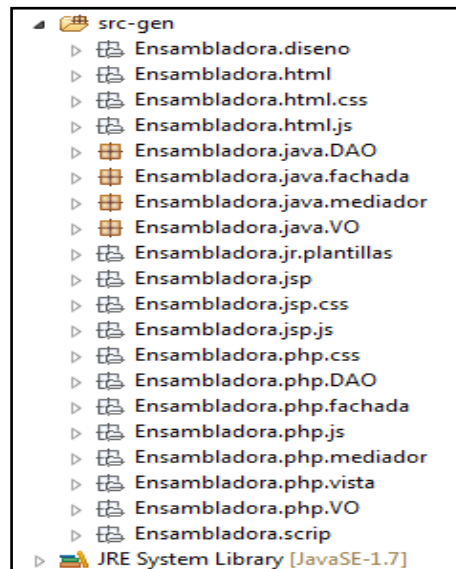
Para la generación de los archivos se debe hacer la digitación de los elementos básicos en el DSL que es el encargado de producir los resultados solicitados basándose en las plantillas para la generación de código, estas plantillas están definidas de forma tal que permiten a la herramienta la generación de los diferentes artefactos involucrados en la construcción de una aplicación.

Es de anotar que en las Figuras 19, 21, y 23, se presentan las líneas del código fuente de cada caso de prueba digitado en el DSL, esta estructura y contenido del archivo digitada es tomada como base para la generación de cada grupo de elementos presente en un caso de prueba en particular.

Los tipos de archivos generados se describen a continuación, destacando las características que tiene cada uno dentro de la aplicación a construir, el cómo se clasifican dentro de la arquitectura adoptada y en qué punto interviene el programador. Los elementos a generar están distribuidos en la estructuración de archivos y paquetes presentados en la Figura 24.

Los archivos descritos a continuación son los generados por el primer caso de prueba únicamente, los que se generan para un caso de prueba más complicado son presentados en el Anexo B, y en la comparación puntual frente a la implementación de referencia.

Figura 24: Directorios y paquetes generados



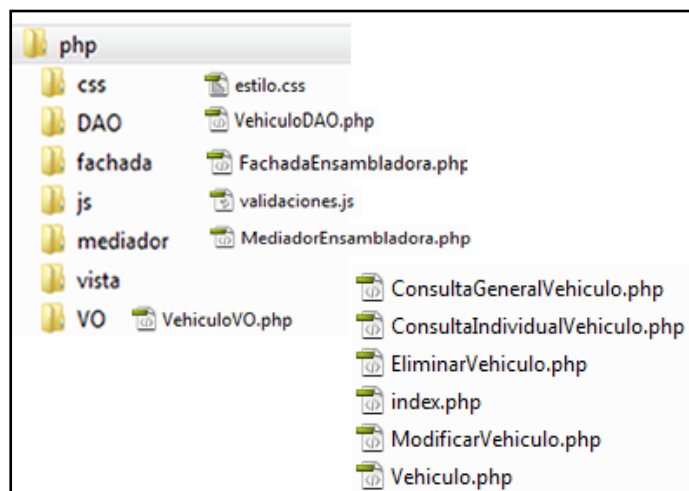
Estos archivos y las carpetas están organizados por tipos, donde se almacena la codificación que ha generado la herramienta. Analizando la gráfica de la estructura de archivos construida por el IDE Eclipse de los archivos generados, a nivel general se nota que no se tienen errores, por lo menos en la formación de los archivos, y que ellos tienen una estructura estandarizada y aceptada, si existieran errores el paquete donde está el archivo erróneo tendría una marca roja de error validada directamente por el IDE donde están soportados los archivos.

Los archivos generados deben ser verificados específicamente, con ello se puede comprobar la funcionalidad de la herramienta. La descripción en esta sección no se hace sobre cada uno de los archivo sino de sobre tipos de archivos donde se toma uno con las características relevantes que se desean validar, entendiendo este archivo como la representación de la generalidad de los de su tipo, y en el caso específico donde sea relevante mostrar al menos dos archivos del mismo tipo se mostrará y se documentará al respecto.

4.3.1 Archivos PHP

Los archivos de tipo PHP constituyen en su generalidad un apartado para un lenguaje determinado, estos son soportados por un servidor de aplicaciones común y tanto las clases que conforman los modelos, las clases del controlador y las vistas se construyen en un archivo de este tipo, no se diferencia entre tipos de archivos, la estructura de carpetas para PHP se presenta en la Figura 25 donde se pueden apreciar los elementos. Cada carpeta contiene uno o varios elementos que a su vez son archivos estructurados bajo el tipo PHP.

Figura 25: Estructura de archivos PHP



Hasta el momento se tiene una clasificación de subcarpetas pertenecientes al tipo de archivos, se describen los elementos de la carpeta DAO y los del paquete VO, ya que estos tienen una codificación y estructura diferente debido a que permiten la interacción de los elemento que se generan, también uno del paquete vista para apreciar la funcionalidad de los archivos generados y como estos se mostrarían al usuario final.

El código puntual para una de las clases generadas en PHP se presenta en la Figura 26 donde se encuentran las líneas de código que no se implementan manualmente, en estas dos clases se puede apreciar según la prueba y la comparación con la implementación de referencia que el código es generado en su totalidad.

Figura 26: Código de archivos PHP

```

VehiculoVO.php
<?php
class VehiculoVO{
function __construct(){
    private $modelo;
    public function getmodelo(){
        return $this->modelo;
    }
    public function setmodelo($modelo){
        $this->modelo=$modelo;
    }
    private $color;
    public function getcolor(){
        return $this->color;
    }
    public function setcolor($color){
        $this->color=$color;
    }
    private $placa;
    public function getplaca(){
        return $this->placa;
    }
    public function setplaca($placa){
        $this->placa=$placa;
    }
}
?>

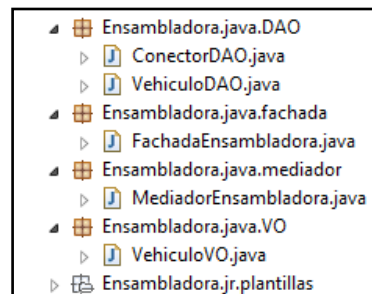
VehiculoDAO.php
<?php
class VehiculoDAO {
function __construct(){
    public insertarVehiculo($Vehiculo){
        return true;
    }
    public modificarVehiculo($Vehiculo){
        return true;
    }
    public eliminarVehiculo($Vehiculo){
        return true;
    }
    public consultaGeneralVehiculo(){
        $consulta="SELECT modelo,color,placa FROM Vehiculo";
        return null;
    }
    public consultaIndividualVehiculo($Vehiculo){
        return null;
    }
}
?>

```

4.3.2 Archivos JAVA

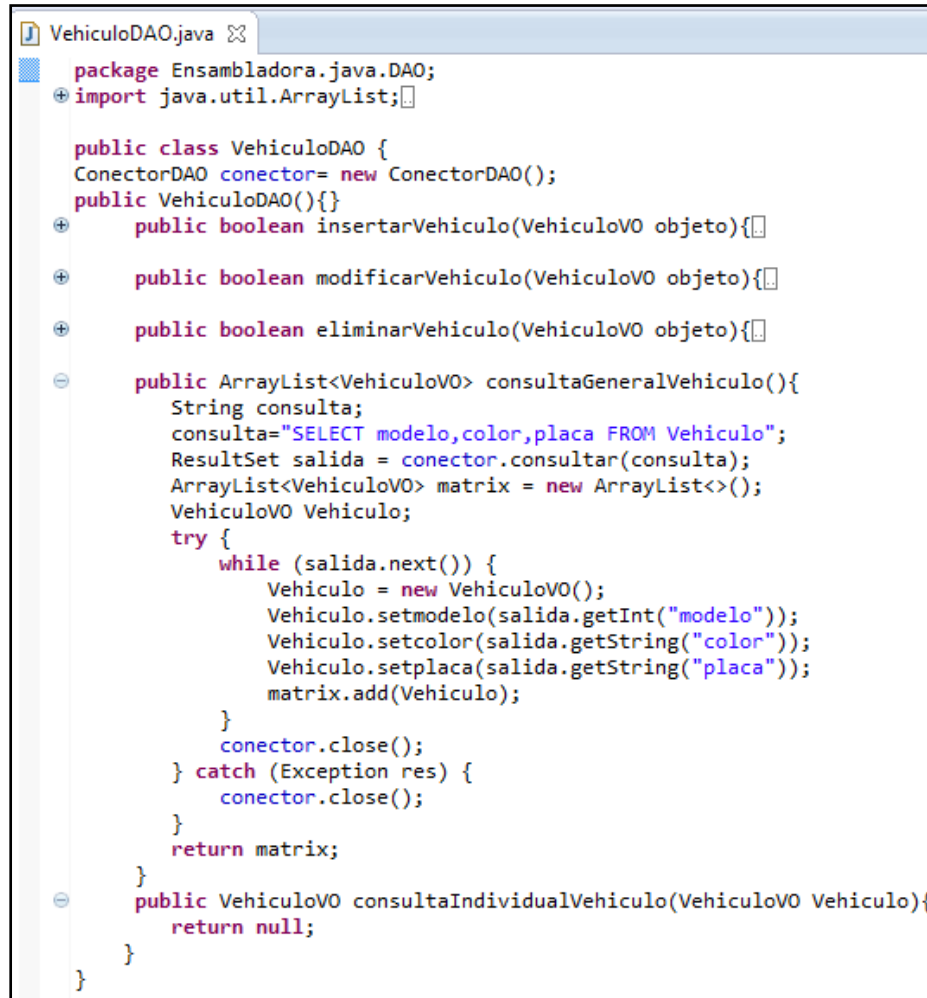
Los archivos java generados tienen una estructura básica de elementos de clases se presentan en la Figura 27 y corresponden a la totalidad de los archivos con esta especificación.

Figura 27: Estructura de archivos java



En la Figura 28 encontramos el código java de modelo a base de datos la clase encargada de hacer las operaciones con base de datos, de la cual se denota la ayuda en la producción de código fuente funcional por parte de la herramienta.

Figura 28: Estructura de archivos java



```
package Ensambladora.java.DAO;
import java.util.ArrayList;

public class VehiculoDAO {
    ConectorDAO conector= new ConectorDAO();
    public VehiculoDAO(){

    }
    public boolean insertarVehiculo(VehiculoVO objeto){

    }
    public boolean modificarVehiculo(VehiculoVO objeto){

    }
    public boolean eliminarVehiculo(VehiculoVO objeto){

    }
    public ArrayList<VehiculoVO> consultaGeneralVehiculo(){
        String consulta;
        consulta="SELECT modelo,color,placa FROM Vehiculo";
        ResultSet salida = conector.consultar(consulta);
        ArrayList<VehiculoVO> matrix = new ArrayList<>();
        VehiculoVO Vehiculo;
        try {
            while (salida.next()) {
                Vehiculo = new VehiculoVO();
                Vehiculo.setModelo(salida.getInt("modelo"));
                Vehiculo.setColor(salida.getString("color"));
                Vehiculo.setplaca(salida.getString("placa"));
                matrix.add(Vehiculo);
            }
            conector.close();
        } catch (Exception res) {
            conector.close();
        }
        return matrix;
    }
    public VehiculoVO consultaIndividualVehiculo(VehiculoVO Vehiculo){
        return null;
    }
}
```

Los archivos java que se generan en la herramienta constituyen la base de uno de los propósitos de esta investigación, estos archivos tienen la particularidad de ser usados en capas según la arquitectura MVC, permitiendo que las vistas estén en varios formatos.

4.3.3 Archivos JSP

Los archivos JSP hacen la especificación de las vistas para un publicador de páginas o servidor de aplicaciones web, en el caso de prueba se generan la completitud de archivos necesarios para hacer una implementación, lo que se puede apreciar en la

Figura 29, la cantidad de archivos generados para JSP es la misma para los archivos HTML, y para los archivos de las vistas PHP.

Figura 29: Código fuente JSP

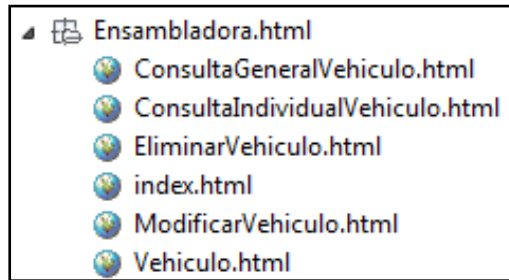
```
ConsultaGeneralVehiculo.jsp
<%@page import="java.util.ArrayList"%>
<%@page import="VO.VehiculoVO"%>
<%@page import="fachada.FachadaEnsambladora"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<%
FachadaVehiculo fachadaEnsambladora = new FachadaEnsambladora();
ArrayList<VehiculoVO> listaVehiculo = fachadaEnsambladora.consultaGeneralVehiculo<>();
%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Menú Principal --- Ensambladora</title>
    <link rel="stylesheet" type="text/css" media="screen" href="css/estilo.css" />
    <script type="text/javascript" src="js/validaciones.js"></script>

  </head>
  <body>
    <form name="form_docente" method="get">
      <table width="774" height="350" border="5" align="center" bordercolor="#000000">
        <tr height="40">
          <td width="752" height="78">
            <label>
              <h1 align="center">CONSULTAR VEHICULO</h1>
            </label>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

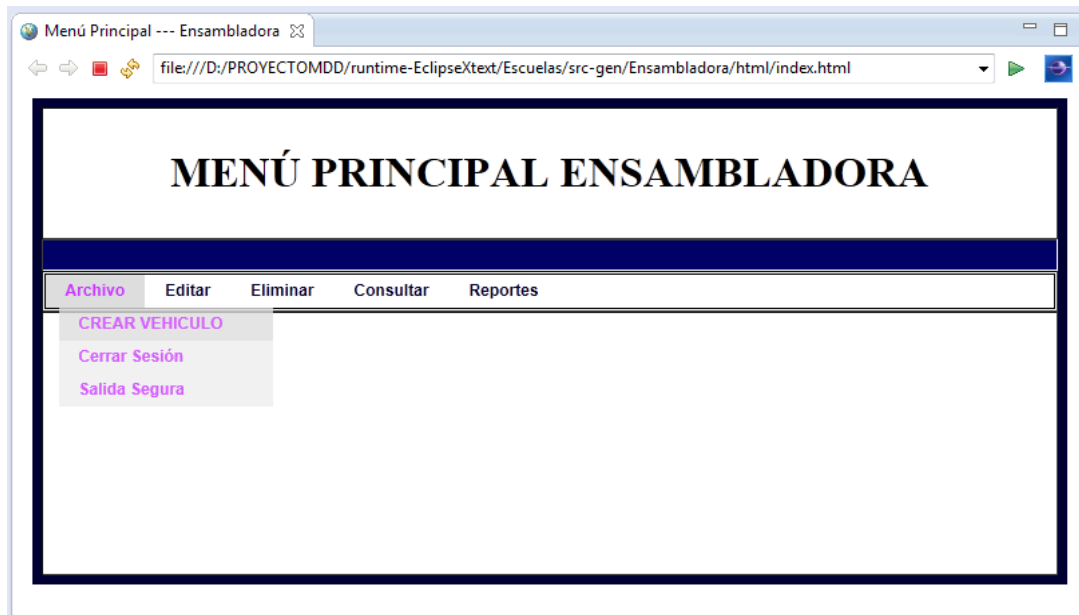
Estos archivos tienen interacción con las clases que representan el modelo y controlador, para servir como interfaces de usuario dentro del lenguaje java. Cabe anotar que son generados de forma equivalente a los que se tienen en la implementación de referencia véase Anexo A.

4.3.4 Archivos HTML

Los archivos HTML constituyen una parte de las vistas dentro de la aplicación. Son generados para apoyar la organización dentro de un equipo de trabajo, donde existe un rol específico para la construcción y el diseño de las vistas, la estructura de los archivos se muestra en la Figura 30.

Figura 30: Esquema de Vistas en HTML

Estos archivos, tienen la particularidad de establecer dos formas de analizarlos, una es gráfica como se muestra en la Figura 31, desde un navegador web que nos deja ver cómo está distribuida la presentación de la aplicación generada, la otra forma está constituida por el código en formato HTML que interpreta el navegador. Esta segunda forma es similar a la presentada en las vistas JSP y las PHP, debido a manejan el punto de interacción con el usuario final de las aplicaciones.

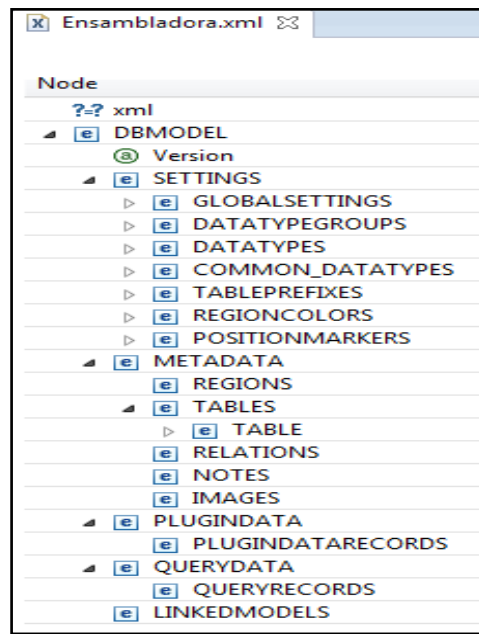
Figura 31: Vista en HTML

4.3.5 Archivos XML

Los archivos XML son específicamente construidos para las plantillas de diseño gráfico de las entidades, las cuales en determinado escenario son base para que el equipo de desarrollo de software, permitiendo asignarlos a los arquitectos de bases de datos y en

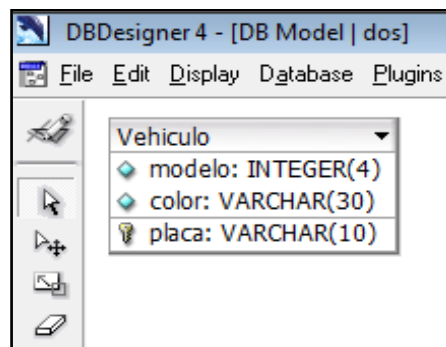
su defecto a los administradores de bases de datos. En la Figura 32 se muestra el esquema jerárquico de este archivo.

Figura 32: Archivo XML



La vista de los elementos que componen el archivo XML, son difíciles de entender y en algunos momentos confusa para el personal de bases de datos, este archivo se puede gestionar desde la herramienta DBDesigner que permite una vista gráfica como se muestra en la Figura 33.

Figura 33: Vista en la herramienta de diseño



Podemos notar que la herramienta no solo genera archivos para los lenguajes de programación específicos, sino que además proporciona una serie de archivos que pueden ser ejecutados desde otras herramientas y se constituyen valiosos en un entorno

de trabajo en equipo, ya sea en el ámbito académico o en el campo de la industria del software.

4.3.6 Archivos JRXML

Este tipo de archivo son elementos de valor agregado a la herramienta construida, permiten la plantilla global para la gestión de reportes en las aplicaciones generadas, son específicas para trabajar con JasperReport que es un administrador de servicios para los reportes, creado bajo licencia de uso libre sobre el lenguaje de programación java. En la Figura 34 se muestra una parte del código generado para la plantilla de reporte.

Figura 34: Archivo de reporte

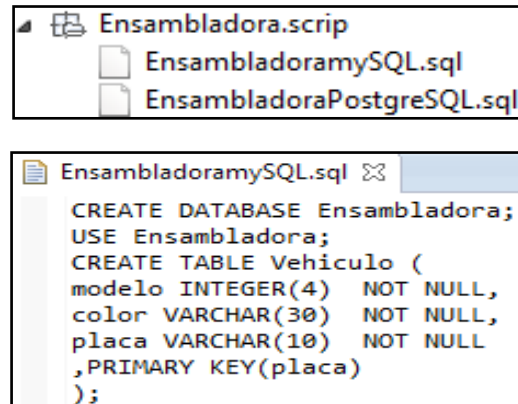


```
<?xml version="1.0" encoding="UTF-8"?>
<jasperReport xmlns="http://jasperreports.sourceforge
  <background>
    <band splitType="Stretch"/>
  </background>
  <title>
    <band height="79" splitType="Stretch"/>
  </title>
  <pageHeader>
    <band height="35" splitType="Stretch"/>
  </pageHeader>
  <columnHeader>
    <band height="61" splitType="Stretch"/>
  </columnHeader>
  <detail>
    <band height="125" splitType="Stretch"/>
  </detail>
  <columnFooter>
    <band height="45" splitType="Stretch"/>
  </columnFooter>
  <pageFooter>
    <band height="54" splitType="Stretch"/>
  </pageFooter>
  <summary>
    <band height="42" splitType="Stretch"/>
  </summary>
```

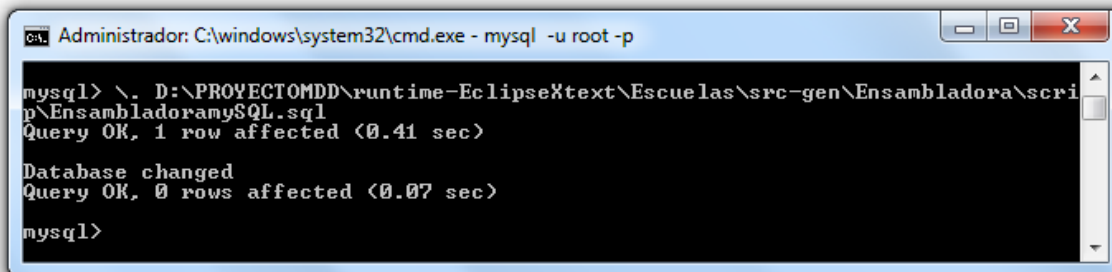
Esta plantilla de reporte puede ser compilada y ejecutada directamente desde un programa en java, pero también existen herramientas para la gestión de estas plantillas, como lo es ireport, usado para la implementación de referencia.

4.3.7 Archivos SQL

Los archivos de este tipo son generados para interactuar directamente con los servidores de bases de datos, estos pueden cargarse directamente en el servidor determinado para dicha tarea, por ejemplo se tiene en la Figura 35 la generación de dos archivos uno para MySQL y otro para PostgreSQL respectivamente estos archivos en la misma figura están descritos de forma textual lo que interpreta el servidor de bases de datos correspondiente.

Figura 35: Script de bases de datos

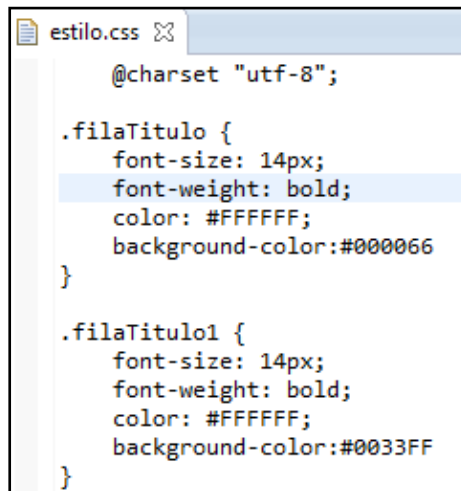
La funcionalidad de estos archivos puede ser verificada al subirlos al servidor de bases de datos, el cual valida si cumplen con las características para subir correctamente. En la figura 36 se puede verificar que el archivo generado sube de forma correcta al servidor de bases de datos.

Figura 36: Prueba de script de bases de datos

Esta forma de probar los archivos SQL generados fue utilizada para subir los tres casos de pruebas descritos al inicio de este capítulo, y se hace la verificación de los elementos que se crean dentro de la base de datos diseñada y estructurada automáticamente a partir de lo que se digite dentro del DSL.

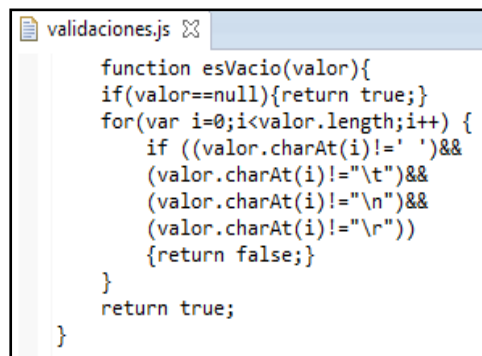
4.3.8 Archivos CCS y JS

Los archivos CSS representan la parte de diseño de interfaz de usuario, permiten hacer configuraciones de los elementos visuales sobre una página de presentación, para la prueba sobre el DSL; si no se digita ningún valor para los colores, tipos de letras entre otros, se establecen los valores por defecto como se muestra en la Figura 37.

Figura 37: Archivo de estilos

```
estilo.css ✕  
  
@charset "utf-8";  
  
.filaTitulo {  
    font-size: 14px;  
    font-weight: bold;  
    color: #FFFFFF;  
    background-color: #000066  
}  
  
.filaTitulo1 {  
    font-size: 14px;  
    font-weight: bold;  
    color: #FFFFFF;  
    background-color: #0033FF  
}
```

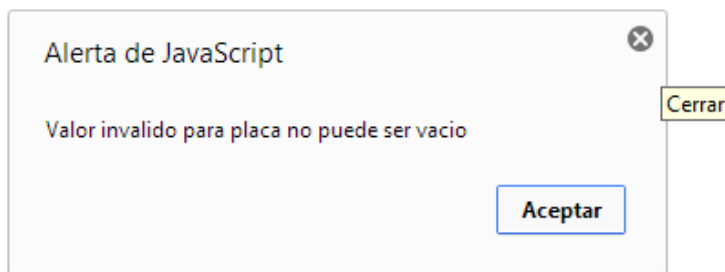
Los archivos de tipo JS establecen la facilidad para realizar las validaciones en tiempo de ejecución de los datos que se intentan gestionar, constituyen la base para el trabajo con la tecnología AJAX que permite la actualización de datos dentro de una página en particular sin tener que refrescar nuevamente todo el código HTML en el navegador. La validación presentada en la Figura 38 corresponde a una de las especificaciones de la lógica de negocio debido a que en el caso de las llaves que se digiten al momento de capturarlas no deben ser vacíos, al igual que los campos requeridos con obligatoriedad.

Figura 38: Archivo de validaciones

```
validaciones.js ✕  
  
function esVacio(valor){  
    if(valor==null){return true;}  
    for(var i=0;i<valor.length;i++) {  
        if ((valor.charAt(i)!=' ')&&  
            (valor.charAt(i)!='\t')&&  
            (valor.charAt(i)!='\n')&&  
            (valor.charAt(i)!='\r'))  
            {return false;}  
    }  
    return true;  
}
```

La funcionalidad de este archivo se prueba en las vistas donde son invocados y sirven para presentar mensajes del tipo que se muestra en la Figura 39.

Figura 39. Prueba de validaciones



La Figura 39 corresponde a uno de los mensajes de validación realizados sobre la información que se propone gestionar.

Para la verificación de los archivos generados se hacen dos procedimientos de verificación, el primero es completar los archivos para tener la aplicación funcional y el segundo reemplazar los archivos de la implementación de referencia por los archivos generados y verificar la funcionalidad de estos.

4.3.9 Análisis del resultado de las pruebas

Las pruebas aplicadas a la herramienta DSL implementada se analizan frente a la tabla 8 de requisitos considerada como una lista de chequeo del funcionamiento de las aplicaciones:

Tabla 8: Requisitos que debe cumplir la herramienta

No.	Requisito
1	Permite que sean digitado datos de entrada
2	Ofrece los mensajes de validación mínimos mencionados
3	Genera la totalidad de archivos de la implementación de referencia
4	Genera archivos con la completitud de líneas de código que la implementación de referencia
5	Genera archivos que deban ser complementados con código individual
6	Los archivos generados tienen el tipo y la estructura válida frente a la

	implementación de referencia
7	Los archivos generados no incluyen errores
8	Los archivos generados en su totalidad son fácilmente verificables
9	El esfuerzo para completar la aplicación funcional es menor al requerido para la implementación de referencia total
10	Los archivos son fácilmente tratables con una estructura escalable

El primer punto fue verificado en la Figura 18, Figura 20 y Figura 22 que ilustran como se puede digitar datos de entrada sobre la herramienta, para aplicaciones muy sencillas y para las más robustas.

El segundo punto se verifica frente a la Figura 17, donde se muestran los mensajes de errores que presenta el DSL cuando no se cumple la gramática definida en el meta modelo o los mensajes de advertencia cuando lo digitado no cumple con las reglas de validación configuradas.

El punto 3 y 4 hacen referencia al número de archivos generados que se constituyen a la misma cantidad de archivos utilizados en la implementación de referencia, y aun genera otros archivos que complementan los resultados tales como los archivos de configuración, el script para otro servidor de bases de datos y las plantillas HTML, esto lo podemos verificar en la Figura 29, Figura 30 y Figura 33, también se verifican aquellos archivos que deben ser completados para la funcionalidad total de la herramienta.

El desarrollador que tome los elementos generados tiene fácil acceso a ellos para hacer la respectiva digitación del código individual que se requiere.

Los archivos generados tienen la estructura idéntica a los archivos de la implementación de referencia, y los tipos corresponden en la totalidad.

Los errores sobre los archivos son claramente verificados desde los script que funcionan correctamente sobre el servidor de bases de datos hasta los archivos para el diseño de interfaz que muestran los colores predeterminados, las clases java generadas son validadas automáticamente en eclipse como se puede apreciar en la Figura 15.

La verificación de los archivos se realizó en cada uno de los pasos presentados en este capítulo, en donde se revela la facilidad para ser verificados.

Los esfuerzos realizados para que la herramienta generada funcione correctamente son mínimos, debido a que la mayor proporción de código fuente fue generada y se proporciona un archivo léame donde se dan las directrices del código específico que se debe aplicar para tener una aplicación totalmente funcional.

Ningún archivo está cerrado, en ejecutable u otro tipo que no permita su edición, y están disponibles para darle la escalabilidad pertinente.

5. Conclusiones y Análisis de Resultados

5.1 Conclusiones

El trabajo con MDD hace un aporte importante a la ingeniería de software, sus principios pueden ser aplicados en casos concretos dando como resultado soluciones generales. Se evidencia la producción de archivos para intercambiar tecnologías y hacer desarrollos que cada vez sean más independientes del desarrollador, entendibles y modificables por otros programadores, llegando en algunas ocasiones a ser entendidos por el experto en el negocio.

A partir de la apropiación de los conceptos y experiencias descritas en el proyecto se genera código para varias tecnologías, servidores y arquitecturas, presentando una alternativa muy fuerte para el futuro del desarrollo de software; su enfoque de trabajo indicado por la implementación de DSLs y la generación automática de código a partir de la lógica de negocio es cada vez más atractiva para la construcción de aplicaciones.

A nivel general se destaca el ahorro en el esfuerzo humano y en los costos de elaboración de un proyecto software. Al igual en el ámbito académico se puede apoyar el trabajo docente – estudiante con el uso de esta herramienta que le facilite la mayoría de elementos a trabajar y el enfoque esté dado en la forma de idear nuevas alternativas y soluciones a problemas complejos de investigación en el área.

En el proceso de afianzamiento de los conceptos y el análisis de otras herramientas que aportan características similares a la ingeniería de software se identifican dos grandes puntos de referencia: es necesario que se trabaje con MDD en la industria y la academia a nivel general en el país, no deteniéndonos solo en las regiones principales.

El paso lógico a seguir en la evolución de la ingeniería de software está marcado por el desarrollo conducido por modelos, por lo tanto es ahora el momento de hacer una apropiación del tema.

Se identifica como la herramienta construida proporciona un lenguaje de dominio específico. Este en relación al uso la curva de aprendizaje aumenta para programadores, porque aprender un nuevo lenguaje es de mayor inversión de tiempo que aplicar uno conocido, pero la proporción disminuye al momento que se adopta la herramienta en el tiempo y se integran programadores que inician en este campo, la herramienta posee una gramática sencilla y fácil de usar.

El nuevo lenguaje de programación construido permite al programador ahorro de tiempo en sus implementaciones, contribuye en la calidad de código con sus autocompletados y gramática definida no da espacios para que el programador se equivoque en la digitación de código.

5.2 Análisis de Resultados

Los resultados obtenidos se enfocan en primera instancia resaltando las metas de MDD que se cumplen en este proyecto (Stahl & Völter, 2006).

Velocidad en el desarrollo de software: Esta promesa de MDD la vemos cumplida, en la generación de las aplicaciones a punto que el código individualmente digitado es mínimo frente al código generado, para soportar esta afirmación se hace el análisis del número de líneas de código digitadas en la implementación de referencia frente al número de líneas de código generadas por la herramienta, este análisis se presenta en la Tabla 9.

Tabla 9: Análisis de la cantidad de líneas de código generadas frente a las digitadas

Descripción	Digitadas	Generadas	Diferencia	Porcentaje(%) generado
Caso de prueba I	124	107	17	86,2
Caso de prueba II	325	298	27	91,7
Caso de prueba III	897	812	85	90,5

Podemos notar que la generación de código para las aplicaciones con la herramienta construida esta alrededor del 90%, lo cual es un porcentaje alto que permite demostrar que un desarrollador no invierte tiempo en construir esta cantidad de código, lo que indica que demoraría menos en elaborar una aplicación con la herramienta construida en este proyecto que hacerlo de forma manual o con otras herramientas que generen componentes por separado.

La calidad del código software: Esta meta se obtiene en el momento de la generación del código fuente que está estandarizado y no tiene intervención humana, lo que connota un alto grado de calidad. Las validaciones dentro de la herramienta evitan que el punto de entrada tenga posibles errores que se puedan digitar por el usuario del DSL.

Gestión de la complejidad: Esta meta se cumple en que la complejidad de las aplicaciones está modelada directamente en la herramienta, las características que no están ligadas a la lógica de negocio no se deben trabajar, de esa parte se encarga el DSL, interpretando y generando a las tecnologías que se requieren. En este caso puntual se genera código para java y PHP, y también archivos para manejar con otros entornos y/o tecnologías.

Entorno productivo: Esta promesa la vemos cumplida en la comparación del trabajo desarrollado para construir una aplicación de referencia tanto de la forma tradicional como basados en MDD. De la forma tradicional se desarrolla un modelo de base de datos se implementa en una herramienta visual se produce el código y se carga en el servidor, se crean las clases para los controladores y modelos, las vistas, los archivos de validación, los reportes los archivos de configuración entre otros, se prueban los

artefactos y luego se hace el mismo para la otra tecnología o lenguaje de programación, en algunos casos sobre herramientas diferentes. El segundo caso con MDD solo hay cuatro pasos de menos labor que cada paso de la forma anterior: primero se digita el modelo en el DSL y se guarda lo cual produce el código automáticamente, segundo se suben los script a los servidores, tercero se implementa el código individual y se prueba. El soporte en la Tabla 9 hace referencia a la disminución del tiempo para desarrollar una aplicación y también permite evidenciar el aumento en la productividad debido a que el generar alrededor del 90% de las líneas de código es un ahorro de esfuerzo para el programador.

5.3 Metas Alcanzadas

En este apartado se hace una relación directa entre los objetivos específicos planteados para este proyecto y el cumplimiento de cada uno de ellos dentro del desarrollo del trabajo presentado. Se resaltan las características significativas y diferenciales dentro de cada meta alcanzada. A continuación se presentan los objetivos en el orden que fueron planteados.

Primer objetivo: *Comparar las metodologías que usan MDD para implementar las mejores prácticas de ingeniería de software para la obtención de aplicaciones funcionales.*

Este objetivo se cumple en su totalidad y se encuentra descrito en el desarrollo del segundo y tercer capítulo donde se realiza un análisis comparativo entre MDA y DSM, describiendo sus fortalezas y debilidades, al igual que los lineamientos metodológicos que cada uno de estos enfoques tiene, se analizan y comparan herramientas elaboradas con cada uno de los enfoques y se implementa una herramienta basada en DSM.

Las mejores prácticas de ingeniería de software resaltadas en este estudio se describen en el capítulo dos y son aplicadas en el capítulo tres, el principio de abstracción está inmerso en el trabajo con modelos y meta modelos donde se realiza la abstracción de

una solución general y esta a su vez está representada en una gramática definida en un meta modelo. El principio de permitir que el computador realice el trabajo está descrito con gran claridad en la herramienta construida debido a que se genera el código para las aplicaciones, son tareas realizadas por el computador internamente a través de transformaciones. Hacer aplicaciones para humanos y no para maquina se implementa en el momento de la generación de interfaces de usuario de las aplicaciones generadas, y que los meta modelos cada vez más abstractos están cerca de ser entendidos por las personas que conocen el negocio sin la saturación de los tecnicismos.

Segundo Objetivo: Construir un DSL, que permita la generación automática de aplicaciones java con arquitectura MVC a través de MDD usando Xtext para los modelos textuales.

Esta meta se alcanza al construir completamente una herramienta capaz de generar aplicaciones java, y aun aplicaciones para otro lenguaje de programación como lo es PHP, en el capítulo dos se describen las características de la herramienta que genera los artefactos bajo la arquitectura MVC, y la implementación de la gramática perteneciente al DSL, además la herramienta cuenta con validaciones personalizadas y plantillas para generación de código para otras herramientas, demostrando que se puede generar código para varias tecnologías a partir de un mismo punto de entrada en el meta modelo construido.

Tercer Objetivo: Evaluar la herramienta demostrando funcionalidad con una prueba de concepto.

Este objetivo se cumple y se encuentra descrito en el capítulo cuarto donde se plantea una prueba de concepto bajo tres casos posibles con diferente grado de complejidad, se implementa cada caso dando como resultado que la herramienta es capaz de generar aplicaciones para el lenguaje, la especificación propuesta y para otras como valor agregado. Los archivos generados en las aplicaciones fueron validados frente a la implementación de referencia y la funcionalidad de cada uno de ellos, se analizaron de forma individual y en conjunto para garantizar la aplicabilidad del DSL construido.

5.4 Contribuciones del trabajo

Producción de código de forma automática, segura, robusta y con buenas prácticas de ingeniería de software, que proporciona optimización y productividad al momento de desarrollar software.

Se produce un estado del arte actualizado con respecto al tema MDD, el trabajo con DSL y la arquitectura Modelo Vista Controlador.

Se presenta una ponencia en el congreso andino de computación en el año 2013, y aceptación de un artículo para su publicación en la revista Ventana Informática edición No. 30 de la Universidad de Manizales.

5.5 Trabajo Futuro

A corto plazo la herramienta puede ser potencializada para la generación de código para marcos de trabajos, otros lenguajes de programación, otro tipo de bases de datos. Por ejemplo para Groovy, JSF, Mongo DB entre otras.

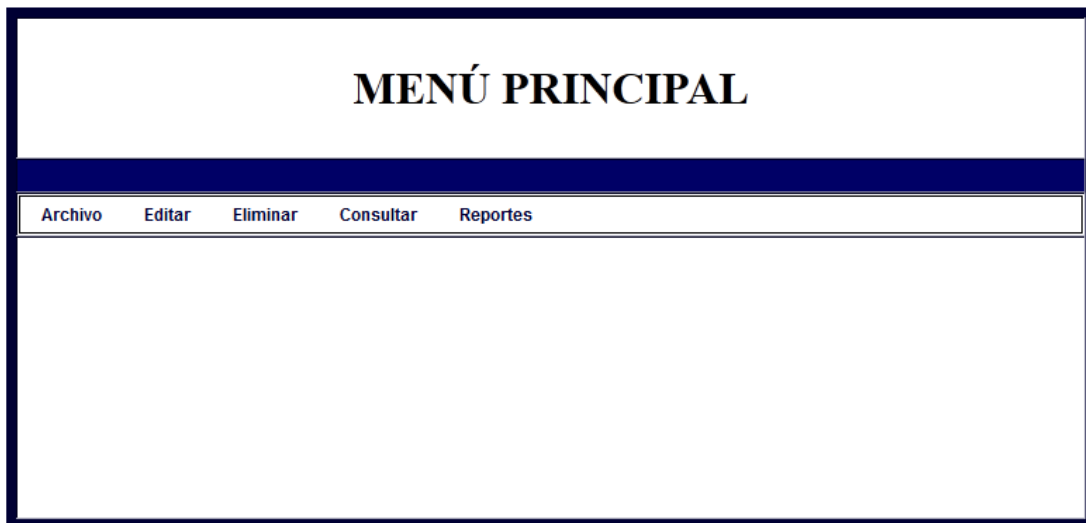
La investigación abre el camino para que estos enfoques de trabajo se implementen en las diferentes regiones del país, específicamente la costa norte.

Implementación de aplicaciones más complejas validadas frente al desarrollo realizado por una muestra importante de programadores, para afianzar el valor agregado de la herramienta. Poner la herramienta al servicio de la academia para su aplicación funcional y validarla para el nivel industrial.

A mediano plazo la herramienta puede ser tomada como base e implementar artefactos que permitan la fechada del desarrollo de software a gran escala, cambiando de interfaces y lenguajes, pasando por software para móviles, web, de escritorios e incluso para dominios de investigación puntuales, como bioinformática y equipos especiales de medicina.

A. Anexo: Aplicación de referencia

Se presenta una parte de las vistas que proporciona la implementación de referencia.



Registrar Docente

Documento :
Nombres :
Apellidos :
Sexo :
Direccion :
Telefono :
e-mail :
Profesion :
Estado Civil :

[Volver](#)

[Guardar](#)

Ver Detalle de Docente

Documento : 12388
Nombres : Deivis
Apellidos : Martinez
Sexo : Acosta
Direccion : Valledupar
Telefono : 3103234512
e-mail : deivis@hotmail.com
Profesion : Ing. Sistemas
Estado Civil : Casado

[Volver](#)

B. Anexo: Código fuente de la aplicación de referencia

Parte del código fuente en la implementación de referencia codificado manualmente.

```
ConectorDAO conector;  
public DocenteDAO() {...}  
public boolean insertarDocente(DocenteVO docente) {...}  
public boolean modificarDocente(DocenteVO docente) {...}  
public ArrayList<DocenteVO> consultaGeneral() {  
    conector.conectar();  
    ResultSet salida = conector.consultar("SELECT identidad, nombres  
ArrayList<DocenteVO> matrix = new ArrayList();  
    DocenteVO docente;  
    try {  
        while (salida.next()) {  
            docente = new DocenteVO();  
            docente.setApellidos(salida.getString("apellidos"));  
            docente.setDireccion(salida.getString("direccion"));  
            docente.setDocumento(salida.getString("identidad"));  
            docente.setEmail(salida.getString("email"));  
            docente.setEstadocivil(salida.getString("estadocivil"));  
            docente.setNombres(salida.getString("nombres"));  
            docente.setProfesion(salida.getString("profesion"));  
            docente.setSexo(salida.getString("sexo"));  
            docente.setTelefono(salida.getString("telefono"));  
            matrix.add(docente);  
        }  
        conector.close();  
    } catch (Exception res) {  
        conector.close();  
    }  
    return matrix;  
}
```

```
import Mediator.MediadorDocente;
import VO.DocenteVO;
import java.util.ArrayList;
public class FachadaDocente {
    MediatorDocente mediador;
    public FachadaDocente () {
        mediador = new MediatorDocente ();
    }

    public boolean insertarDocente (DocenteVO docente) {
        return mediador.insertarDocente (docente);
    }

    public boolean modificarDocente (DocenteVO docente) {
        return mediador.modificarDocente (docente);
    }

    public ArrayList<DocenteVO> consultaGeneral () {
        return mediador.consultaGeneral ();
    }

    public boolean eliminarDocente (DocenteVO docente) {
        return mediador.eliminarDocente (docente);
    }

    public DocenteVO consultaIndividual (DocenteVO docente) {
        return mediador.consultaIndividual (docente);
    }
}
```


Gramática de la herramienta DSL construida

```
Unalupc.xtext
grammar co.edu.unal.maestria.proyecto.domainmodel.Unalupc
with org.eclipse.xtext.common.Terminals

generate unalupc "http://www.edu.co/unal/maestria/proyecto/domainmodel/Unalupc"

Model: base = BASE;

BASE:
    'base' nombre=ID '{'
        (tabla+=Tabla)+
        (configuracion+=CONFIG)?
    '}';

Tabla:[]
INDICE: []
Relaciones:[]
Columna:[]
enum BOOL:[]
enum TIPO:[]
CONFIG: []
```

Plantilla de Validación del código digitado en el DSL

```
UnalupcJavaValidator.java
package co.edu.unal.maestria.proyecto.domainmodel.validation;

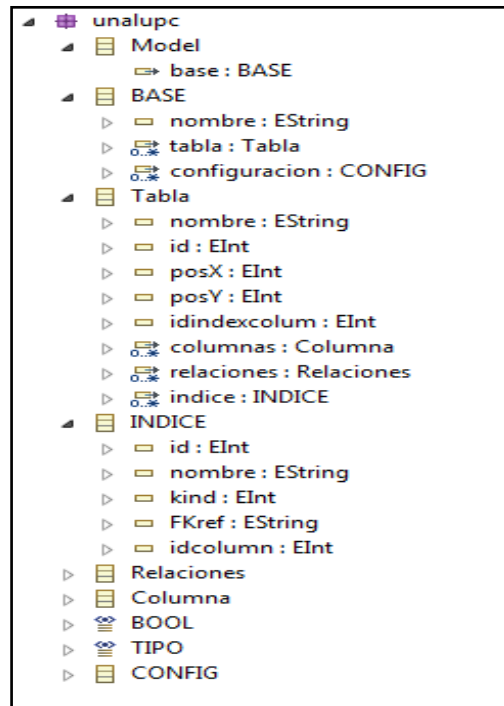
import org.eclipse.xtext.validation.Check;

public class UnalupcJavaValidator extends AbstractUnalupcJavaValidator {

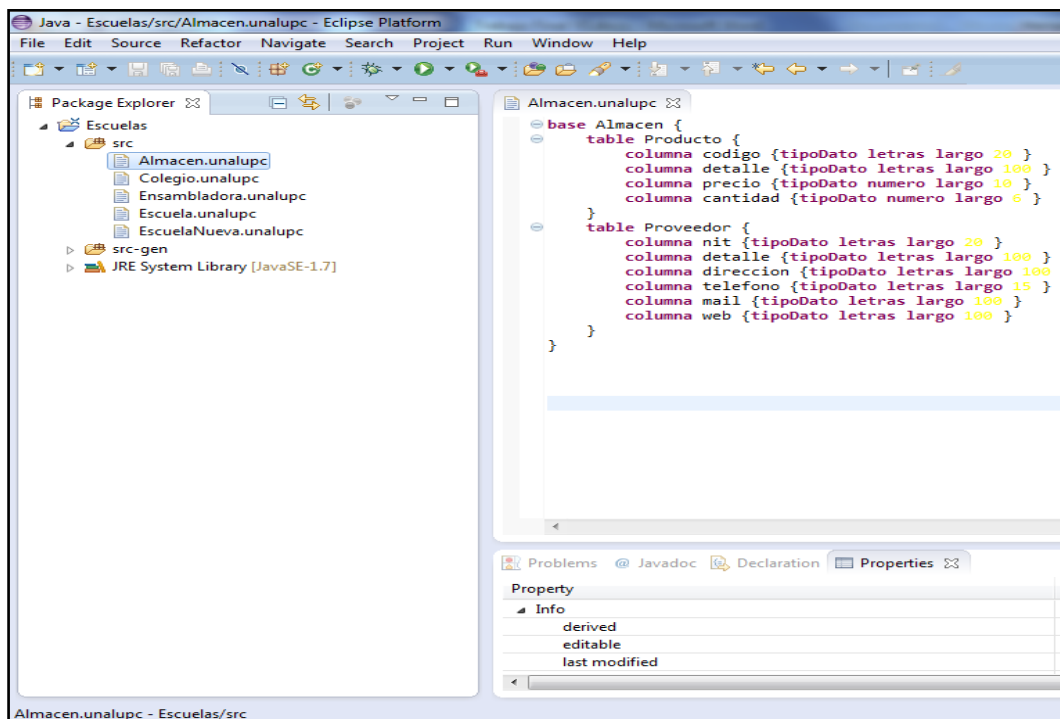
    @Check
    public void verificarLetraInicial(Tabla table) {
        if (!Character.isUpperCase(table.getNombre().charAt(0))) {
            warning("Se sugiere que el nombre de las entidades inicie por una mayuscula", null);
        }
    }

    @Check
    public void verificarNombreUnicoTabla(Tabla table) {
        Model modelo;
        EObject object = EcoreUtil.getRootContainer(table);
        if (!(object instanceof Model))
            return;
        modelo = (Model) object;
        for (Tabla tabla : modelo.getBase().getTabla()) {
            if (tabla == null || tabla == table)
                continue;
            if (table.getNombre().equals(tabla.getNombre())) {
                error("Duplicación de nombre : " + table.getNombre() + "[" + table.getId()+ "]",
                    null);
                return;
            }
        }
    }
}
```

Estructura del lenguaje de dominio específico construido



Entorno de trabajo del lenguaje construido



Implementación de referencia en el DSL

```
Colegio.unalupc
base escuela {
  table Docente {
    columna identidad { tipoDato letras largo 20 llavePrimaria si }
    columna nombres { tipoDato letras largo 80 }
    columna proesion { tipoDato letras largo 100 }
    columna telefono { tipoDato letras largo 20 }
    relaciones DOC_ASI {
      nombreTablaDestino 'DocenteAsignatura' nombreColumnaDestino 'docente_id'
      nombreTablaOrigen 'Docente' nombreColumnaOrigen 'identidad'
    }
  }
  table Asignatura {
    columna codigo { tipoDato letras largo 20 llavePrimaria si }
    columna detalle { tipoDato letras largo 100 }
    columna codigo { tipoDato letras largo 20 }
    relaciones ASI_DOC {
      nombreTablaDestino 'DocenteAsignatura' nombreColumnaDestino 'adignatura_id'
      nombreTablaOrigen 'Asignatura' nombreColumnaOrigen 'codigo'
    }
  }
  table DocenteAsignatura {
    columna codigo { tipoDato letras largo 10 llavePrimaria si }
    columna docente_id { tipoDato letras largo 20 llaveForanea si }
    columna adignatura_id { tipoDato letras largo 20 llaveForanea si }
  }
}
```


Bibliografía

Barceló, F. R. (2010). Metodologies de Desenvolupament Dirigides per Models. Retrieved from <http://openaccess.uoc.edu/webapps/o2/handle/10609/547>.

Brambilla, M., Comai, S., Fraternali, P., & Matera, M. (2013). DESIGNING WEB APPLICATIONS WITH. *Dipartimento di Elettronica e Informazione, Politecnico di Milano, Pizza L. da Vinci 32*.

Cabot, J., & Gómez, C. (2008). A Catalogue of Refactorings for Navigation Models. *Eighth International Conference on Web Engineering*, 75–85.

Cañadas, J., Palma, J., & Túnez, S. (2011). Defining the semantics of rule-based Web applications through model-driven development. *International Journal of Applied Mathematics and Computer Science*, 41–55.

Ceri, S., & Fraternali, P. (2002). Conceptual Modeling of Data-Intensive Web Applications. *IEEE INTERNET COMPUTING*, 20–30.

Ceri, S., Daniel, F., & Matera, M. (2003). Extending WebML for modeling multi-channel context-aware web applications. *Web Information Systems*.

Deacon, J. (2009). Model-view-controller (mvc) architecture. <http://www.jdl.co.uk/briefings/MVC.pdf>, (Mvc), 1–6.

Deursen, A. V., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*. Retrieved from <http://138.26.64.7/courses/cs593/spring2010/DSLAnnotatedBib.pdf>.

Efftinge, S., & Völter, M. (2006). oAW xText: A framework for textual DSLs. Workshop on Modeling Symposium at Eclipse Retrieved from http://eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf.

Frankel, D. S. (2004). Model-Driven Software Development Executive Overview. 1–13.

- Fraternali, P., & Paolini, P. (2000). Model-driven development of Web applications: the AutoWeb system. *ACM Transactions on Information Systems (TOIS...)*, 28(4), 323–382. Retrieved from <http://dl.acm.org/citation.cfm?id=358110> , 323–382.
- Gupta, P., & Govil, M. (2010). MVC Design Pattern for the multi framework distributed applications using XML, spring and struts framework. *International Journal*, 02(04), 1047–1051. Retrieved from <http://www.enggjournals.com/ijcse/doc/IJCSE10-02-04-4> , 1047–1051.
- Igor, D., Gordana, M., Branko, P., & Maja, T. (2010). Computer Science and Information Systems, 7(3), 409–440. doi:10.2298/CSIS090203002D. A domain-specific language for defining static structure of database applications. , 409–440.
- Kapitsaki, G. M., Kateros, D. a., Prezerakos, G. N., & Venieris, I. S. (2009). Model-driven development of composite context-aware web applications. *Information and Software Technology*, 51(8), 1244–1260. doi:10.1016/j.infsof.2009.03.002 , 1244–1260.
- Kraus, A., Knapp, A., & Koch, N. (2007). Model-driven generation of web applications in UWE. *Model-Driven Web Engineering (MDWE 2007 ...)*, (1st 016004). Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.9883&rep=rep1&type=pdf&e> .
- Kumar, M. P., & Mohan, K. (2012). Enhanced Framework to Evaluate Performance & Functionality of Web Applications. ... Applications. 44(15). Retrieved from <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Enhanced+Framework+to+Evaluate> , 24–27.
- Lin, Y., Zhang, J., & Gray, J. (2004). Model comparison: A key challenge for transformation testing and version control in model driven software development. ... for Model-Driven Software Development. Retrieved from <http://gray.cs.ua.edu/pubs/oopsla-2> .
- Moreira, A., & Rossi, G. (2004). UML and Model Engineering. V(2) , 1–5.
- Murugesan, S., & Deshpande, Y. (2002). Meeting the challenges of Web application development: the web engineering approach. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 687–688. doi:10.1109/ICSE.2002.1008030 , 687–688.
- Pastor, O., España, S., Panach, J., & Aquino, N. (2008). Model-driven development. *Informatik-Spektrum*. Retrieved from <http://www.springerlink.com/index/e55q55086j62lh58.pdf> .

Rizvi, D., & Hassan, S. (2009). Achieving Loose Coupling Between Different Components of Model-View-Controller For Web Based Application. *Proceedings of the 3rd National Conference* , 1–4.

Rode, J., & Rosson, M. (2003). Programming at runtime: Requirements and paradigms for nonprogrammer web application development. *IEEE Symposium on Human-Centric ... Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1260198 .*

Schmidt, D. (2006). Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY* , 25–31.

Stahl, T. (., & Völter, M. (2006). Model-driven software development. *Retrieved from http://www.st.informatik.tu-darmstadt.de/pages/lectures/se/ws06-07/design/lecture/12_modelDrivenSoftwareDevelopment-1x1.pdf .*

Torres, V., Muñoz, J., & Pelechano, V. (2005). A model driven method for the integration of Web applications. ... Congress. *LA-WEB ... Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1592355 .*

Willink, E. (2010). MODELS 2010 Workshop on OCL and Textual Modelling: Re-engineering Eclipse MDT/OCL for Xtext. *modeling-languages.com*, 1–15. *Retrieved from http://modeling-languages.com/events/OCLWorkshop2010/submissions/ocl10_submission_5.pdf .*