Universidad Nacional de La Plata
Facultad de Informática

Un enfoque de metamodelado ágil utilizando técnicas de templating

Tesina de Licenciatura en Sistemas

Alumno: Alan Gabriel Garcia Camiña
Director: Gustavo Rossi

Agradecimientos

Quiero dedicar especialmente esta tesis a mi director Gustavo Rossi y a José Matías Rivero por brindarme la oportunidad de formar parte del LIFIA, acompañarme, guiarme, ayudarme a crecer profesionalmente y personalmente durante estos últimos años.

También agradecer a mi familia y a mi novia Camila por el apoyo durante mis estudios y a la Universidad Nacional de La Plata por ser pública, gratuita y estar continuamente ampliando las posibilidades para que todos puedan hacer una carrera universitaria.

Índice de contenidos

1. Capítulo 1: Introducción

1.1 Motivación

Con mucha frecuencia, los programadores se encuentran con ciertos escenarios durante el desarrollo de software. Estos escenarios, radican básicamente en la repetitividad con la que se presentan determinadas tareas.

El ciclo de vida de un producto de software es complejo y está conformado por tareas de diversa índole, algunas de las mismas no requieren del intelecto o capacidades específicas de miembros del equipo, los cuales pueden reusar o contar con el trabajo hecho por otros miembros expertos en determinadas áreas.

Si contáramos con una metodología y herramienta para generar código, el cual representa el reúso directo de conocimientos específicos de determinados miembros del equipo asociado a ese tipo de tareas, los tiempos en los cuales se puede llevar a cabo un proyecto se podrían reducir drásticamente, intentando de esta manera, revertir algo de aquella repetitividad que mencionamos.

Esta tesis consiste en el desarrollo de una metodología y una herramienta asociada, en el marco de desarrollo dirigido por modelos, en combinación con técnicas de templating[1], apuntado a la reducción de tiempos durante el ciclo de implementación de un producto de software.

Durante el desarrollo, desde el punto de vista puramente técnico, los programadores se encuentran con diversas tareas a resolver, entre ellas se pueden encontrar:

- Definición de interfaces de usuario.
- Gestión transaccional.
- Persistencia.
- Queries.
- Validación.
- Definición de tests automatizados.

Es común para un desarrollador que esté utilizando el paradigma de programación OO (Orientada a Objetos) tener métodos o fragmentos de código en diferentes proyectos que hagan técnicamente lo mismo, como los ABM (Altas Bajas Modificaciones) y les resulte tedioso e ineficaz en cuanto a tiempo la codificación manual cada vez que este sea necesario. A esto se le suma la alta probabilidad de cometer errores de sintaxis propios de una codificación manual.

Actualmente existen metodologías que se enfrentan a escenarios de esta índole, una de ellas es el Modelado Específico de Dominio (DSM por sus siglas en inglés, Domain Specific Modeling) [13].

El propósito de DSM es crear modelos, utilizando un lenguaje enfocado y especializado para la metodología. Dicho lenguaje es conocido como Lenguaje Específico de Dominio (DSL, por sus siglas en inglés Domain Specific Language). Se incluye en este marco la generación automática de código ejecutable directamente desde los modelos. En consecuencia las

[1] Creación de templates. Un template es un documento o archivo con un formato preestablecido, que se utiliza como punto de partida para una aplicación en particular por lo que el formato no tiene que volver a crear cada vez que se utiliza.

aplicaciones finales se conciben a partir de especificaciones de alto nivel liberando al desarrollador de las tareas de codificación y mantenimiento del código fuente, y se incrementa significativamente su productividad.

La metodología propone la formalización de modelos a través de su descripción gráfica o textual y un conjunto de reglas que lo caracterizan. Se puede lograr mayor eficiencia en esta derivación si la cantidad de aspectos considerados en el problema a tratar es finito, lo que conforma el "dominio específico" de los modelos.

La metodología DSM implica la ejecución de etapas estrictas, las cuales incluyen la creación de tres elementos principales, que serán la herramienta fundamental:

1. Un metamodelo formal inicial para crear el DSL, a través del cual se definen las soluciones.
2. Un generador que transforme los modelos a código ejecutable.
3. Un framework de dominio que sirva como base al código generado en el punto anterior, para que la transformación sea más sencilla y reusable.

Para crear un DSL es necesario un marco teórico brindado por un metamodelo. El metamodelado permite describir los elementos y restricciones de un modelo. Un DSL está compuesto por sintaxis abstracta, concreta y semántica.

Este lenguaje tiene un grado de abstracción que permite normalmente cubrir un dominio completo. Por ejemplo, un DSL generado de un metamodelo para teléfonos móviles puede permitir a los usuarios especificar abstracciones de alto nivel para la interfaz de usuario, así como abstracciones de bajo nivel para el almacenamiento de datos como números de teléfono o configuraciones.

Por otra parte, los objetivos principales de la metodología DSM comprenden:

1. Elevar el nivel de abstracción más allá del código fuente, mediante la especificación de la solución en términos de un lenguaje que utiliza directamente los conceptos y reglas del dominio específico sobre el que se trabaja.
2. Generar productos finales en un lenguaje de programación elegido u otra forma a partir de estas especificaciones de alto nivel utilizando técnicas de generación de código para evitar errores de codificación humanos.

Al trabajar sobre un dominio particular, se reduce la complejidad así como también reduce el número de defectos en los programas resultantes, mejorando la calidad y optimizando el tiempo de desarrollo.

En comparación con la codificación manual, en DSM se encuentran involucrados usuarios con diversos roles, entre ellos, expertos del dominio, esenciales para la metodología, que aportan su experiencia y conocimientos acerca de éste.

Si bien son muchos puntos a favor, actualmente el proceso de desarrollo de DSM posee ciertas limitaciones. Éstas radican principalmente, en que DSM posee etapas que son estrictas y linealmente interdependientes como: inicialmente definir un metamodelo formal para crear el DSL, luego definir generadores de código, el framework de dominio y por último, recién en este punto, poder testear los resultados obtenidos.

Debido a estas limitaciones, surge la idea de optimizar el actual DSM desde un punto de vista novedoso. Ésta idea consiste en utilizar técnicas de templating, las cuales conformen pequeñas soluciones estilo DSM iterativamente, realizando micros ciclos de detección de características no modularizables, para aumentar la productividad de partes puntuales del proceso de desarrollo, obteniendo como resultado, un metamodelo formal de una manera ágil y novedosa.

Esta forma de obtención del metamodelo, se podrá visualizar y testear al instante. Esto es una gran ventaja ya que a diferencia de DSM, necesariamente debemos estar en la etapa

final para poder hacerlo. Otra característica de este novedoso enfoque, es que inicialmente no se comenzará definiendo el metamodelo inicial, sino que el mismo se irá descubriendo incremental e iterativamente durante el proceso.

Durante esta investigación, se desarrolló y especificó este enfoque como una metodología de construcción denominada "ADSM" (*Agile Domain Specific Modeling*). La misma permite, incluyendo ventajas de una aproximación DSM y siguiendo con su filosofía "*don't repeat yourself*" (*No te repitas tú mismo*), evitar el problema de la repetitividad de código, que las tecnologías actuales no llegan a cubrir en forma precisa, y tener de manera más ágil, como resultado un metamodelo formal y código generado automáticamente para un dominio particular.

Para dar soporte a ADSM, en el marco de esta tesis, se implementó una herramienta, denominada "MYL" (*Make Your Language*), la cual permite aplicar esta metodología llevando a cabo la generación automática de código y toda la administración necesaria para ADSM. Esta metodología, junto a MYL, facilitará la ejecución de tareas rutinarias que se dan en el transcurso de un proyecto de software, principalmente en el área de implementación.

1.2 Objetivos

➢ Descubrir una solución a la problemática de repetición de código en la etapa de desarrollo en sistemas de software, procurando lograr una metodología ágil, en el marco de model-driven basada en técnicas de templating.

➢ Investigar y obtener una metodología eficaz, utilizable para quienes deseen generar código ágilmente, sin la necesidad de ser un experto en el dominio del problema.

➢ Implantar una herramienta que dé soporte a esta metodología de desarrollo ágil, siendo productiva para cualquier grupo de desarrolladores de software y lo más adaptativa posible.

➢ Relacionar conceptos de metodologías de desarrollo ágil con modelado a través de una nueva metodología, basándose en las existentes en el ambiente model-driven.

➢ Profundizar conceptos de modelado, templating y metamodelos para el desarrollo de sistemas de software.

➢ Ampliar el campo de uso de los modelos e incentivar el uso de los mismos para la generación automática de código.

➢ Reducir tiempos en la etapa de implementación de software e incentivar el templating como un recurso potente para la reusabilidad de código ya implementado.

1.3 Estructura de la tesina

La presente tesis se organizará de la siguiente manera:

Capítulo 1: En el primer capítulo se introduce el contexto sobre el cual se desarrolló esta tesis, se describe la motivación, se plantea el tema central y los objetivos propuestos.

Capítulo 2: En el segundo capítulo se presenta el marco teórico, donde se introducen diferentes definiciones relacionadas con el tema.

Capítulo 3: Se desarrolla la metodología investigada y desarrollada en el marco de esta tesis.

Capítulo 4: Se describe la herramienta desarrollada y conceptos relacionados con la misma.

Capítulo 5: Se presentan las estadísticas obtenidas de casos de estudios realizados y los análisis correspondientes.

Capítulo 6: En el capítulo seis, se realiza una conclusión en base al objetivo planteado y se detallan trabajos futuros a realizar.

2. Capítulo 2: Marco Teórico

2.1 Introducción

De acuerdo a lo expresado por Douglas C. Schmidt [1], durante las últimas cinco décadas, los investigadores y desarrolladores de software han estado creando abstracciones que les ayuden a desarrollar en términos de diseño y que los alejen de las complejidades que incumben los entornos de hardware –por ejemplo, la CPU, memoria, y los dispositivos de red.

Desde los primeros momentos de la informática, estas abstracciones incluían tanto el lenguaje como las plataformas tecnológicas. Por ejemplo, los primeros lenguajes de programación, como los ensambladores y Fortran, abstraían a sus desarrolladores de las complejidades de programar con lenguaje máquina.

Aunque estos primeros lenguajes y plataformas aumentaron su nivel de abstracción, tenían todavía un enfoque "orientado al hardware". En particular, ofrecían abstracciones del espacio solución, esto es, el dominio de las propias tecnologías de las computadoras, en vez de abstracciones del espacio problema que expresa conceptos aplicados al dominio. Esto se solucionó más tarde con un mayor nivel de abstracción mediante el paradigma de la ingeniería orientada a objetos y la programación estructurada. Muchos esfuerzos pasados han creado tecnologías que han aumentado el nivel de abstracción utilizado para desarrollar software.

En los años ochenta se comenzaron a utilizar las herramientas CASE [2] [32] (Computer-Aided Software Engineering), que estaban enfocadas en permitir a los ingenieros de software expresar sus diseños en representaciones gráficas tales como máquinas de estados, diagramas de estructura, diagramas de flujo de datos, etc. Un objetivo de las herramientas CASE era proporcionar más entornos gráficos que demanden menos complejidad que los lenguajes de programación ya que se conseguía un análisis mucho más intuitivo. Otro de los objetivos era la generación automática de código ejecutable, para reducir el esfuerzo manual de depurar el código fuente. Finalmente, también se perseguía que los programas proporcionasen portabilidad.

Aunque estas herramientas atrajeron una considerable atención en la investigación, no se adoptaron en la industria. Un problema que presentaban era que los lenguajes gráficos de los programas de las herramientas CASE no eran suficientemente expresivos para tener en cuenta las necesidades de las plataformas existentes, que eran sistemas operativos como por ejemplo MS-DOS, OS/2, Windows u otros sistemas operativos [32].

Dichas plataformas requerían soporte para características tan importantes como la escalabilidad, la tolerancia a fallos o la seguridad. Además, las herramientas CASE incluían entornos de ejecución propios, lo cual hacía mucho más difícil integrar el código que generaban con otros lenguajes software y plataformas tecnológicas.

Otro problema era el hecho de no soportar el desarrollo de software colaborativo, las herramientas CASE estaban limitadas a programas escritos por una persona o un equipo que serializase su acceso a los ficheros gestionados por la herramienta. Además no son muy versátiles hablando en términos de usabilidad, es decir, eran demasiado genéricas y muy poco configurables a la hora de hacerlas específicas. Estos problemas junto con la cantidad y complejidad del código generado, hizo complicado el desarrollo, la depuración y evolución de las herramientas CASE y de las aplicaciones creadas con ellas. Como resultado, tuvieron muy poco impacto comercial en el desarrollo software entre los ochenta y los noventa. Por otra parte, los avances en lenguajes y plataformas durante las dos últimas décadas han conseguido un mayor nivel de abstracción.

Gracias a esto, el desarrollo de aplicaciones resulta mucho más cómodo para los desarrolladores, reduciendo así uno de los principales problemas de las herramientas CASE (abstracción). Por ejemplo, los programadores hoy en día utilizamos lenguajes orientados a objetos, tales como C++, Java, Python, o C# en vez de Fortran o C. Este tipo de lenguajes, que están basados en el paradigma de la orientación a objetos, consiguen una mayor abstracción siendo más cercanos al mundo real, más legibles y más entendibles debido a que utilizan el concepto de "objeto", un elemento con propiedades que representa a los que nos rodean [3].Como causa de la madurez de la tercera generación de lenguajes y plataformas reutilizables, la mayoría de desarrolladores nos formamos en las universidades para abstraernos de las complejidades asociadas a la creación de aplicaciones utilizando la última tecnología.

Hoy en día existen nuevos problemas que afectan al desarrollo de software y que aumentan su complejidad, principalmente relacionados con requisitos no funcionales. La entrada de Internet como medio de trabajo y el hecho de que la mayoría de herramientas que se usan a diario utilizan la web como plataforma ha hecho que requisitos no funcionales como la distribución, la seguridad o privacidad cobren una gran relevancia. Con esto se observa que una vez cumplidos los requisitos funcionales, hay que ir más allá y, por tanto, se deben de tener en cuenta factores como el hecho de que se requiere la misma usabilidad de una herramienta para un niño que para un adulto.

Como solución a estas complejidades actuales inherentes al desarrollo de software, se tiende a realizar unos planos o guías, denominados *modelos*, que ayuden a que todos los ingenieros del software vayan en una misma dirección. Es en este contexto en el que aparece el propósito de desarrollar tecnologías en base a la Ingeniería Dirigida por Modelos (Mode-Driven Engineering, MDE) [4].

2.2 Conceptos básicos de Ingeniería dirigida por modelos

La Ingeniería dirigida por modelos (MDE, por sus siglas en inglés Model Driven Engineering)) [4] [26] es una metodología de desarrollo de software que se centra en la creación y explotación de modelos de dominio (es decir, representaciones abstractas de los conocimientos y actividades que rigen un dominio de aplicación particular). El uso de modelos para el desarrollo del software proporciona soluciones que son independientes de las tecnologías, y cuyo código fuente puede obtenerse mediante la generación automática de código. MDE pretende superar los problemas y limitaciones que tenían las herramientas CASE, soportando gran variedad de modos de representación de modelos (UML, esquemas relacionales ER, esquemas XML, etc.)

El enfoque MDE, además, tiene por objeto aumentar la productividad mediante la portabilidad de plataformas entre los sistemas (a través de la reutilización de modelos estandarizados), simplificando el proceso de diseño (a través de modelos de patrones de diseño que se repiten en el dominio de aplicación), y promoviendo la comunicación entre los individuos y equipos que trabajan en el sistema (a través de una estandarización de la terminología y las mejores prácticas utilizadas en el dominio de aplicación).

Un paradigma de modelado para el MDE se considera eficaz si los modelos tienen sentido desde el punto de vista de un usuario que está familiarizado con el dominio, y si ellos pueden servir como base para la implementación de sistemas. Los modelos son desarrollados

a través de una amplia comunicación entre los gerentes de producto, diseñadores, miembros del equipo de desarrollo e incluso los propios usuarios del dominio de la aplicación [5].
Algunas de las iniciativas MDE más conocidas son:

- La Arquitectura dirigida por modelos (MDA) del Object Management Group (OMG), que es una marca registrada de la OMG.
- El entorno de herramientas de modelado y programación Eclipse.

Historia

Las primeras herramientas para soportar MDE fueron las CASE desarrolladas en los ochenta. Compañías tales como Integrated Development Environments (IDE - StP), Higher Order Software (ahora Hamilton Technologies, Inc., HTI), Cadre Technologies, Bachman Information Systems, y Logic Works (BP-Win y ER-Win) fueron pioneras en el campo. Excepto por HTI de AXES Universal Systems Language (USL) y su automatización asociada, los CASE tenían el mismo problema que las actuales herramientas MDA/MDE tienen hoy: el modelo se desincroniza de la aplicación.

2.3 Desarrollo dirigido por modelos

El Desarrollo Dirigido por Modelos (MDD, por sus siglas en inglés, Model Driven Development) [6] es un paradigma de desarrollo software que está basado en modelos que utilizan técnicas de generación automática para obtener el producto de software. MDD basa el desarrollo de software en la construcción de modelos conceptuales que describen el sistema a desarrollar (estructura del sistema, comportamiento, estructura de datos, aspectos de presentación, etc.). Es parte de un modelo que conceptualiza el dominio del problema, y de manera sistemática este modelo se va transformando en otros modelos de niveles de abstracción más bajos, hasta generar implementaciones concretas.

Mientras más completa y precisa sea la especificación de las funcionalidades de un sistema en los modelos, mayor será el porcentaje del código fuente del sistema que se podrá generar automáticamente.

MDD se centraliza en dos ejes principales [38], por un lado hace énfasis en la especificación de los requisitos de un sistema y por el otro se centra en la implementación. Para esto MDD identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, denominados PIM (Platform Independent Model) y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, más conocidos como PSM (Platform Specific Model). Un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico. La transformación entre modelos constituye el motor del MDD y de esta manera los modelos pasan de ser entidades meramente contemplativas a ser entidades productivas.

Como se puede apreciar en la figura 1, y tal como lo menciona Jordi Cabot en su blog [34], MDD utiliza los modelos como el artefacto principal en un proceso de desarrollo.

Usualmente, en MDD la implementación es de forma (semi) generada automáticamente desde los modelos. MDA es una versión de MDD particular de la OMG y por esta razón se basa en la utilización de los estándares previstos por la OMG. Por lo tanto MDA puede considerarse como un subconjunto de MDD.

Por otro lado, MDE sería un superconjunto de MDD porque, como la E en MDE sugiere, MDE va más allá de las actividades de desarrollo puras y abarca otras tareas basadas en

modelos de un proceso de ingeniería de software completo (por ejemplo, el modelo basado en la evolución de la sistema o la ingeniería inversa basado en modelos de un sistema heredado). Todo esto se resume visualmente en la siguiente imagen (Figura 1).

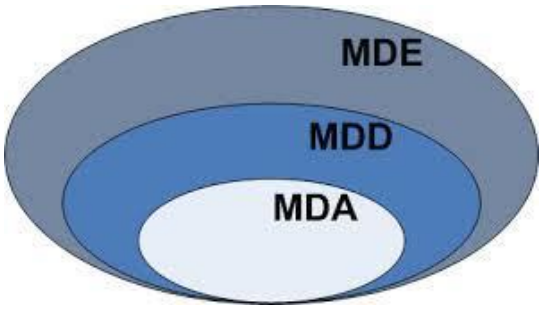


Figura 1 - Encapsulación de metodologías

MDA [7] [15] consigue abstraer el desarrollo software, de forma que un sistema software pueda adaptarse a diferentes tecnologías y lenguajes de programación. En MDA se define modelo como "la descripción o especificación de un sistema y su ambiente para cierto propósito y frecuentemente representado como una combinación de dibujos y texto" [33]. El texto puede ser o bien un lenguaje de modelado o un lenguaje natural.

MDD aplica lecciones aprendidas de los esfuerzos realizados con las herramientas CASE para desarrollar plataformas de más alto nivel y abstracciones del lenguaje, así como también hace posible la existencia de elementos gráficos para facilitar el modelado. Tener elementos gráficos que ayuden al desarrollador a realizar su tarea es muy importante ya que ayuda a la abstracción y además es intuitivo y visual.

2.4 Arquitectura dirigida por modelos

Teniendo en cuenta una descripción de la OMG [7] [15], se introduce el concepto de Arquitectura Dirigida por Modelos (MDA, por sus siglas en inglés Model Driven Architecture) de la siguiente manera:

"La misión de la OMG es ayudar a los usuarios de computadoras a resolver los problemas de integración, proporcionando especificaciones de interoperabilidad independientemente de los proveedores. MDA es el resultado de la OMG para solventar esta integración". La estrategia MDA, prevé un mundo donde los modelos juegan un papel más directo en la producción de software.

Esta estrategia fue impulsada por (MDE), la cual es más amplia en su alcance que MDA. Esta metodología, es un acercamiento al diseño de software propuesto por el "Object Management Group (OMG)" en el año 2001 [7]. MDA propone un marco de trabajo cuyo objetivo central es resolver el problema de coste que supone el cambio de tecnología en un sistema de software, así como su integración en la plataforma correspondiente.

La idea principal de MDA es usar modelos, de modo que las propiedades y características de los sistemas queden plasmadas de forma abstracta, y por tanto, los modelos no se vean afectados por los cambios tecnológicos y se facilite su mantenimiento. Esto es posible debido a la trazabilidad existente entre los modelos del diseño de la arquitectura del software.

2.4.1 Conceptos básicos de MDA

MDA [7] [15] se ha concebido para dar soporte a la ingeniería dirigida por modelos de los sistemas de software.

Proporciona una serie de guías o patrones expresadas como modelos. En este marco se proponen cuatro niveles de abstracción que componen la jerarquía o arquitectura de modelos.

Como se visualiza en la Figura 2, a mayor nivel, mayor grado de abstracción. Estos niveles, son CIM (Computation Independent Model), PIM (Platform[2]-Independent Model), PSM (Platform-Specific Model), y la Implementación de código (ISM) o aplicación final.

Figura 2 -Niveles de abstracción de MDA

CIM: El primer modelo de la jerarquía MDA siguiendo un criterio de más abstracto a menos abstracto es un modelo CIM. Éste modelo es una descripción de la lógica del negocio desde una perspectiva independiente de la computación. Es un modelo del dominio.

PIM: El segundo modelo que define MDA en su jerarquía es un modelo PIM. Éste es un modelo con un alto nivel de abstracción, una vista de un sistema desde el punto de vista independiente de la tecnología, pero teniendo en cuenta sus características de cómputo ya que se está pensando en el sistema software y no en el conocimiento del dominio, como en un modelo CIM. Siguiendo la arquitectura MDA, un modelo PIM es posible construirlo a partir de un CIM, aunque no es estrictamente necesario.

PSM: El tercer modelo que ocupa la arquitectura MDA, es un modelo PSM. Éste es un modelo específico de plataforma, concretamente de la plataforma tecnológica donde se ejecutará el sistema. Un modelo PSM se construye a partir de un PIM, es decir, un modelo PIM puede transformarse en uno o más PSM. Un PSM se encarga de especificar un sistema en términos de la plataforma en la que vaya a ejecutarse.

[2] Plataforma: Una plataforma es un conjunto de subsistemas y tecnologías que provee un conjunto de funcionalidades a través de interfaces y unos patrones específicos de uso, los cuales pueden ser empleados por cualquier aplicación sin que ésta tenga conocimientos de los detalles de cómo esta funcionalidad es implementada.

Implementación de código (ISM) o Aplicación final: El último modelo en la jerarquía MDA es la aplicación final y por tanto, el paso final en el desarrollo. La aplicación final se obtiene a partir de la transformación de un PSM a código, es decir, a un lenguaje de programación.

La Figura 3 muestra los cuatro niveles de abstracción MDA junto con una breve descripción de cada etapa. Dentro de esta jerarquía es posible establecer relaciones entre modelos, tanto verticales como horizontales. Las transformaciones horizontales son por ejemplo de PIM a PIM o de PSM a PSM, y las verticales son por ejemplo de PIM a PSM o de PSM a código.

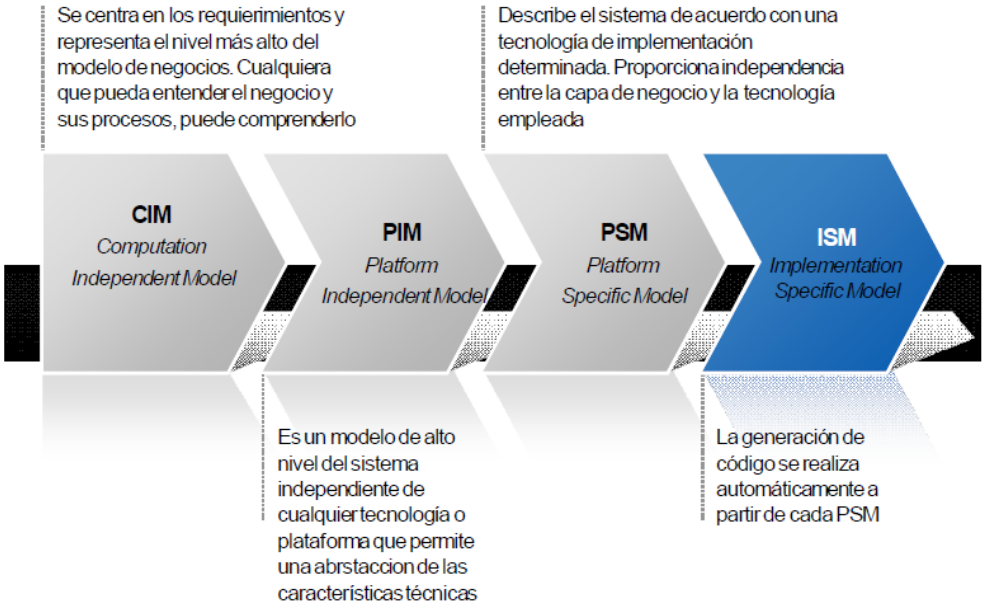


Figura 3 – Modelo MDA

Los PIM y PSM se pulen detalladamente antes de la transformación, idealmente éstos no deberían cambiar con la información que se recibe, por eso el patrón recibe el modelo como entrada. Sin embargo, lo más común es utilizar valores etiquetados, por lo que el patrón de transformación real es de PIM-marcado a PSM.

Según MDA, los modelos PIM y PSM deben expresarse como modelos UML y las transformaciones deben automatizarse lo máximo posible. Hay alternativas como la transformación directa, vía programación, el uso de patrones y el metamodelado.

Las versiones iniciales de MDA sirvieron como base de MDD, que generalizó MDA, y que define las transformaciones en el contexto del metamodelado. Cabe destacar que MDD no fija, al contrario que MDA, el número de niveles de transformación.

En MDD la forma más aceptada de definir modelos es a través de metamodelos, y las transformaciones a través de lenguajes de transformación de modelos. En cambio, en la propuesta original de MDA el metamodelado no era una condición necesaria. Las versiones posteriores de MDA incorporaron después ideas de MDD, que dieron lugar a Meta-Object Facility (MOF) y Query Views Transformations (QVT).

MDA es una materialización de MDD basada en las tecnologías y estándares de OMG. El estándar para el metamodelado es MOF y el lenguaje de transformación QVT [38].

2.5 Metamodelado

El metamodelado es un mecanismo que permite construir formalmente lenguajes de modelado, describe de manera formal los elementos de modelado, la sintaxis y la semántica de la notación que permite manipularlos. La ganancia de abstracción inducida por la construcción de un metamodelo facilita la identificación de eventuales incoherencias y favorece la generalización.

Los metamodelos según Atkinson [8] son uno de los pilares de la ingeniería dirigida por modelos (MDE). Sin embargo, cuando se trabaja con modelos se suelen encontrar dificultades en la definición de los metamodelos por la gran variedad de representaciones posibles para los mismos y la ausencia de manuales que guíen su definición.

Un metamodelo podría verse como un modelo donde se describen otros modelos. Una buena interpretación de lo que es un modelo, un metamodelo y un meta-metamodelo se encuentra en los estudios de García Díaz y Cueva Lovello (2010) [9]. Allí, un metamodelo se define como aquellas herramientas que permiten la creación de un modelo, es decir, la descripción de uno o varios elementos del dominio o mundo real; finalmente, el meta-metamodelo describe un lenguaje para crear metamodelos, generando un grado de abstracción supremamente alto, en el cual coinciden todos los modelos (Figura 4). En el marco de esta tesis, se tomará como sinónimo un metamodelo formal y un lenguaje DSL.

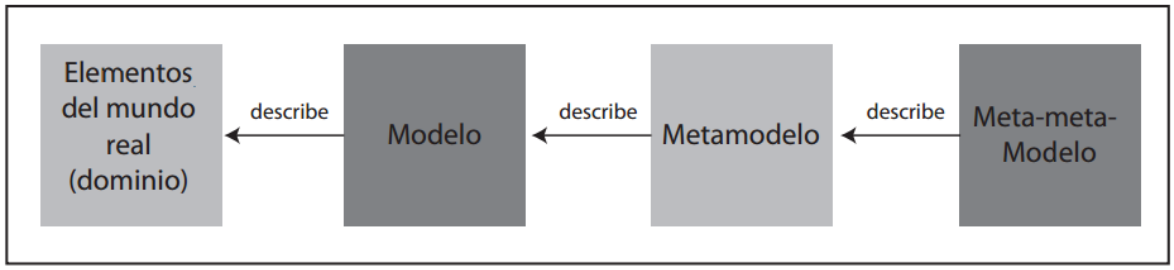


Figura 4 - Metamodelado

2.6 Automatización de los pasos de transformación

Las transformaciones en MDD son siempre ejecutadas por herramientas. Muchas herramientas son capaces de transformar PSM a código sin seguir el enfoque MDD ya que un PSM está muy próximo al código. Sin embargo, las transformaciones de CIM a PIM, y de PIM a PSM, resultan más complejas dentro del proceso de desarrollo software.

Una de las ventajas de MDD es que estas transformaciones se realizan de forma automática. Esto es debido a que en la etapa de desarrollo se invierte una gran cantidad de tiempo. Ahora, gracias a las herramientas que soportan MDD, como por ejemplo GMF (Graphical Modelling Framework)[3], EMF (Eclipse Modelling Framework)[4], GME (Generic

[3] Para más información visitar: http://eclipse.org/modeling/gmp/
[4] Para más información visitar: http://www.eclipse.org/modeling/emf/

Modelling Environment)[5], JetBrains[6], esta parte se realiza más fácilmente y se mejora el tiempo de desarrollo. Sin embargo, cabe destacar que no todas las herramientas son lo suficientemente sofisticadas como para proveer de transformaciones de PIM a PSM y de PSM a código.

2.7 Lenguajes de modelado en MDD

El Object Management Group (OMG) [10] ha definido un conjunto de lenguajes de modelado para, entre otros propósitos, dar soporte al enfoque MDD. Uno de los lenguajes más conocidos es UML, ya que su utilización está ampliamente extendida en el ámbito de la ingeniería del software, cabe destacar que esto no se logra a través de UML en estado puro, sino a través de UML Profiles.

Los lenguajes utilizados en MDD [38] [39] necesitan tener definiciones formales para que las herramientas sean capaces de transformar los modelos automáticamente escritos en esos lenguajes.

La Arquitectura 4 capas de Modelado (Figura 5) es la propuesta de OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. Los niveles definidos en esta arquitectura se denominan comúnmente: M3, M2, M1, M0:

- El nivel M3 (el meta-metamodelo) es el nivel más abstracto, donde se encuentra el MOF (Meta Object Facility) [11], que permite definir metamodelos concretos (como el del lenguaje UML), de esta forma, MOF asegura que las herramientas sean capaces de leer y escribir en todos los lenguajes estandarizados por el OMG, ya que todos están basados en MOF.
- El nivel M2 (el metamodelo), sus elementos son lenguajes de modelado, por ejemplo UML, SysMl. Los conceptos a este nivel podrían ser Clase, Atributo, Asociación.
- El nivel M1 (el modelo del sistema), sus elementos son modelos de datos, por ejemplo entidades como “Usuario”, “Teléfono”, atributos como “nombre”, relaciones entre estas entidades.
- El nivel M0 (instancias) modela al sistema real. Sus elementos son datos, por ejemplo el nombre de una persona, que vive en “Calle, N° XXX”

[5] Para más información visitar: http://www.isis.vanderbilt.edu/projects/gme/
[6] Para más información visitar: https://www.jetbrains.com/

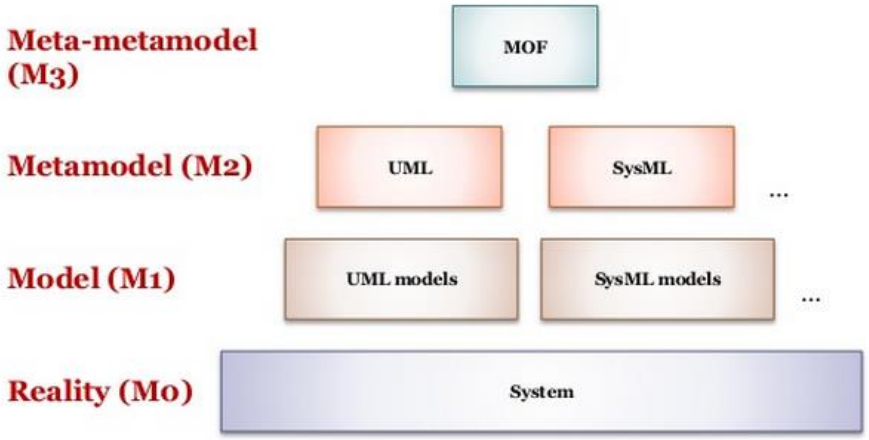


Figura 5 - Arquitectura de 4 capas

Las definiciones de transformaciones dentro de MDD se describen utilizando el lenguaje estándar, el cual se denomina QVT (Query/View/Transformations), que la OMG ha definido para escribir las transformaciones entre modelos.

2.7.1 UML

UML [12] es un lenguaje de modelado genérico que permite modelar el comportamiento de un sistema software independientemente de la plataforma en la que vaya a ser implementado. UML incluye diagramas para modelar el comportamiento, pero sin embargo se crean ambigüedades en el proceso de transformación y por lo tanto, no es posible generar automáticamente el código de la aplicación a partir de sus modelos.

2.7.2 Meta Object Facility (MOF)

La especificación del OMG de MOF [11] nos permite distinguir claramente entre tipos e instancias de una forma apropiada y elegante. Tiene como objetivo permitir la interoperación de las herramientas de metamodelado, y de las que dan soporte al proceso de desarrollo de software basado en MDD. MOF es una arquitectura de metamodelado que consta de cuatro capas las cuales conforman una especie de pirámide.

Las especificaciones de MOF proponen el meta-metamodelo MOF como el lenguaje abstracto para definir todo tipo de metamodelos, como UML.

2.8 Modelado específico de dominio

El Modelado Específico de Dominio (DSM, por sus siglas en inglés Domain Specific Modeling) [13], es una forma de automatizar la resolución de un determinado problema o proceso software cuando éste es reincidente. Cada uno de estos problemas o procesos tiene una parte común que puede ser resuelta de forma automática. Estos procesos o problemas

suelen compartir un dominio o familia que puede ser representado mediante un lenguaje específico para ellos.

DSM, está basada en modelos y se centra en el uso de los mismos como principales elementos en el proceso de desarrollo, lo que supone que es un tipo de desarrollo de software embebido en procesos MDD. DSM aumenta el nivel de abstracción por encima de los lenguajes de programación normales, especificando la solución directamente en un lenguaje que utiliza conceptos y reglas desde el dominio del problema.

La metodología propone comenzar el desarrollo con una definición formal de los conceptos y las reglas que caracterizarán a las representaciones de alto nivel (modelos), las cuales serán el artefacto principal de especificación del software. DSM para especificar estos modelos, utilizan lenguajes específicos de dominio (DSL) los cuales se detallarán más adelante. A partir de estos modelos específicos de dominio, el código puede entonces ser generado automáticamente utilizando generadores de código desarrollados y personalizados especialmente para tal fin por un equipo especializado dentro de los integrantes de un enfoque DSM.

Esta generación de código, es posible gracias a la cota que se pone en la cantidad de aspectos considerados del problema de software a tratar, la cual conforma el "dominio específico" de los modelos y promueve un incremento en la semántica de los conceptos y reglas utilizadas en las descripciones. Tal incremento permite la generación de código completa, es decir de un sistema completo, a partir de los modelos. El código puede ser producido en cualquier lenguaje de programación o para cualquier paradigma de programación.

DSM no requiere que todo el código sea generado desde modelos, pero sí busca que lo modelado por el desarrollador produzca código completo, sin la necesidad de ser posteriormente modificado. Dentro del marco MDD permite la creación automatizada de código fuente ejecutable extraída directamente de los modelos específicos del dominio. Esto supone que el código generado estará orientado a un determinado dominio y será el resultado del DSM. Con ello se consigue un mayor nivel de abstracción ocultando muchos de los aspectos de implementación.

El desarrollo de una plataforma DSM requiere una considerable inversión de recursos, la necesidad de usuarios con determinados roles como por ejemplo, desarrolladores experimentados y expertos del dominio que a menudo no poseen antecedentes en el desarrollo de software.

Liberando de la creación manual de código y del mantenimiento del código fuente, los DSMs pueden mejorar notablemente la productividad del desarrollador. La fiabilidad de la generación automática de código comparada con la generación de código manual reducirá también el número de defectos en los programas resultantes aumentando su calidad.

2.8.1 Metamodelos en DSM

Según lo expresado en [13], las herramientas tradicionales de modelado, como UML de hoy en día o las CASE en el pasado no proveían a los desarrolladores la posibilidad de adaptar los lenguajes de modelado y generadores. Estas herramientas están basadas en una arquitectura de "dos niveles": los diseños del sistema se almacenan en archivos o repositorios cuyo esquema es programado y compilado en una herramienta de modelado. La parte del "hard-coded" define qué tipos de modelos se pueden hacer y cómo pueden ser procesados. Lo más importante es que solo el proveedor de la herramienta puede modificar el lenguaje de modelado ya que él tiene acceso al código.

La tecnología basada en metamodelos elimina esta limitación y permite lenguajes de modelado más flexibles. Esto se logra agregando un nivel por encima del nivel de lenguajes de modelado, tal como se visualiza en la siguiente Figura 6[13]:

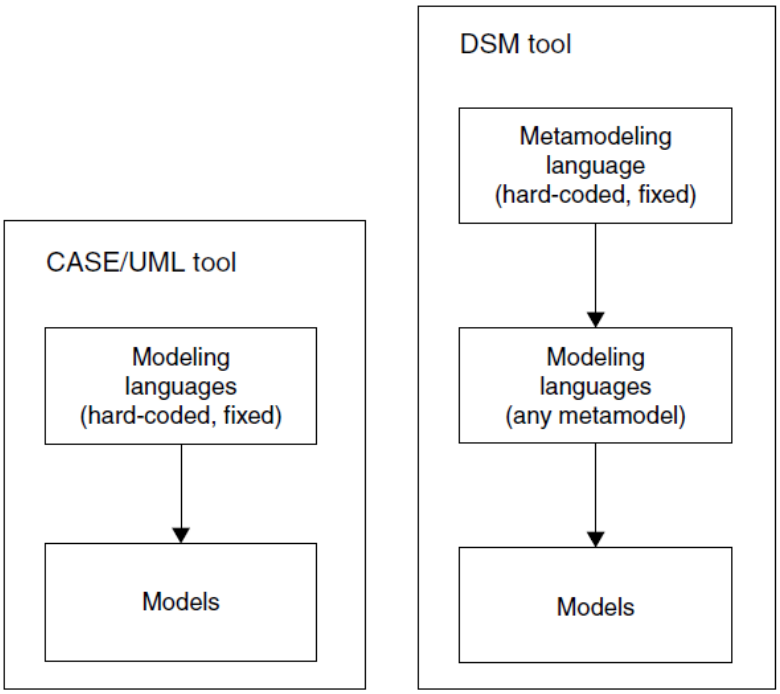


Figura 6 - Metamodelo en DSM

Las herramientas basadas en metamodelo siguen una arquitectura de "tres niveles". El nivel más bajo, el nivel de modelo, es similar al de las herramientas CASE. Incluye diseños de sistemas como modelos. El nivel del medio contiene un modelo del lenguaje, es decir un metamodelo. El metamodelo incluye los conceptos, reglas y notaciones de diagramas de un lenguaje dado. Por ejemplo un metamodelo puede especificar conceptos como un "caso de uso" y un "actor", cómo se relacionan y cómo están representados.

A diferencia de una herramienta CASE o UML, una basada en metamodelos permite al usuario acceder y modificar las especificaciones del lenguaje. Esto se logra por tener un tercer nivel, más alto que incluye el lenguaje de metamodelado para especificar lenguajes de modelado (DSL's). Este tercer nivel suele ser la parte "editable" de una herramienta basada en metamodelos.

Los tres niveles están estrechamente relacionados unos con otros: un modelo se basa en un metamodelo, que a su vez se basa en un lenguaje de metamodelado. Claramente, no es posible el modelado sin algún tipo de metamodelo. Esta estructura es similar a la dependencia que existe entre los objetos, clases y meta clases en algunos lenguajes de programación orientados a objetos.

2.8.2 Breve historia del Modelado Específico de Dominio

Los DSMs [13] surgen como solución a un conjunto de problemas. Es una aproximación al desarrollo software pero centrada en modelos que son aplicables a diferentes problemas o procesos de un mismo dominio.

Los lenguajes generados de los DSMs tienen grado de abstracción que permite normalmente cubrir un dominio completo. Por ejemplo, un lenguaje generado de un DSM para teléfonos móviles puede permitir a los usuarios especificar abstracciones de alto nivel para la interfaz de usuario, así como abstracciones de bajo nivel para el almacenaje de datos como números de teléfono o configuraciones.

Los beneficios de un proceso de desarrollo con DSM son bastante notables:

Se produce una reducción de la complejidad y un considerable incremento de la productividad (como resultado de la especificidad de dominio y de la automatización en la generación de código). La razón de esto es que los ingenieros se centran en las funcionalidades que ellos quieren desarrollar y no en laboriosas rutinas de implementación.

Se mejora la calidad. Al generarse el código de forma automática se evitan los riesgos de errores por descuidos, problemas de sintaxis y errores lógicos. En la codificación manual algunas actualizaciones del código suponen cambios en múltiples secciones del código; con DSM un simple cambio en el generador es a menudo suficiente para corregir todas las ocurrencias simultáneamente.

2.8.3 DSM y UML

En contraste con los DSLs, UML se utiliza para una gran variedad de propósitos en una amplia gama de dominios. Las primitivas ofrecidas por UML están relacionadas con la programación orientada a objetos siendo desconocida para el cliente mientras que los DSMs ofrecen primitivas cuya semántica es conocida por el cliente permitiendo una validación de los requisitos.

UML incluye varios mecanismos de extensión, uno de ellos es la creación de perfiles (profiles), que permite definir semántica específica y personalizar los modelos para diferentes dominios y plataformas específicas. Los perfiles UML utilizan estereotipos, valores etiquetados y restricciones para extender el alcance de UML para un dominio en particular. Quizá, el mejor ejemplo conocido de personalización de UML para un dominio específico es SysML, un lenguaje específico de dominio para ingeniería del software.

En contraste con otros procesos de desarrollo de software, la metodología DSM requiere, por parte de un grupo de desarrolladores con experiencia en el dominio particular, la creación de tres elementos principales que serán la herramienta fundamental del equipo que llevará a cabo la construcción del software:

- Un lenguaje de dominio específico formal a través del cual se definen las soluciones de modo concreto y sin ambigüedades,
- Un generador que transforma los modelos a código ejecutable
- Un framework de dominio que sirva como base al código generado en el punto anterior, para que la transformación sea más sencilla y reusable.

2.8.4 DSM y MDA

La OMG (Object Managment Group) [10], ha admitido tácitamente que la generación de código completo partiendo de UML no va a suceder y apunta hacia una arquitectura dirigida por modelos (MDA) [7] [15].

MDA comprende una transformación de un modelo UML en otro modelo UML, posiblemente varias veces, hasta llegar a una generación automática de código sustancial partiendo del modelo final. En MDA las transformaciones de modelos significan que durante cada etapa los desarrolladores extienden los modelos producidos de forma automática con más detalles. DSM tiene como objetivo generar código directamente de los modelos sin tener que modificar los modelos o código generado.

La diferencia entre MDA y DSM es muy visible en los procesos ágiles y especialmente en el mantenimiento, donde los cambios deben hacerse en los modelos creados anteriormente. La aproximación MDA conduce a resultados creados por asistentes: lotes de código (y modelos) que no fueron escritos por ellos mismos y que están a la espera de ser mantenidos. Estos asistentes en ciertas ocasiones pueden ser útiles, aumentando la productividad en el resultado, pero con el paso del tiempo se transforma en la creación de una masa de modelos desconocidos y código que necesita mantenimiento para mantener una imagen considerable. La forma MDA para manejar los cambios en el modelo, es utilizar el mismo lenguaje en todos los niveles y el uso de solo unos pocos conceptos, como una clase, eso reduce en forma natural el nivel de abstracción que se pueda utilizar en los modelos.

2.9 Lenguajes específicos del dominio

Como se mencionó anteriormente DSM requiere de un Lenguaje Específico de Dominio (DSL, por sus siglas en inglés Domain Specific Language) [17] para el diseño y especificación de los modelos específicos del dominio.

La creación de un lenguaje específico de dominio resulta ventajosa si el lenguaje permite expresar un tipo particular de problemas o soluciones de forma más clara que con los lenguajes preexistentes, y el tipo de problema en cuestión reincide frecuentemente.

La programación orientada a lenguajes considera la creación de lenguajes de propósito específico para expresar problemas como parte estándar de la resolución de los mismos. Un lenguaje específico de dominio se crea específicamente para solucionar problemas de un determinado dominio, un ejemplo de éstos puede ser EAGLE[7] (Easily Applicable Graphical Layout Editor), un software de modelado, creación y simulación de circuitos electrónicos. En contraste, los lenguajes de propósito general, se crean para solucionar problemas en diferentes dominios.

2.9.1 Definición de un Lenguaje Específico de Dominio

El concepto no es nuevo – lenguajes de propósito específico y todos los tipos de lenguajes de especificación/modelado han existido siempre, pero el término se ha hecho más

[7] EAGLE : http://www.cadsoft.de

popular debido a la tendencia del modelado específico de dominio [18]. En la Figura 7 se puede ver como el interés hacia los DSL's ha aumentado en los últimos años [16].

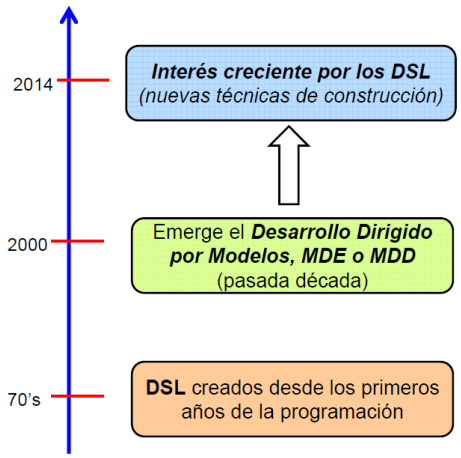


Figura 7 - Interés por DSLs

DSL, permite modelar soluciones para un determinado problema. Por ejemplo, para diseñar redes informáticas se necesitaría un lenguaje que definiera routers, hubs y switches entre otras.

Este enfoque de lenguaje, está atrayendo mucha atención en la ingeniería de software, así como en lenguajes de programación. El enfoque ha sido exitosamente aplicado en un número de áreas incluyendo sistemas bancarios, entornos gráficos, y la creación de redes.

Desde un punto de vista de ingeniería de software, un DSL se refiere a un lenguaje de modelado que ofrece notaciones y conceptos que pueden ser manipulados directamente por un experto del dominio para expresar una solución como un modelo [16]. Un propósito clave de un lenguaje de modelado es permitir comunicarse y compartir soluciones entre los expertos del dominio. Los modelos también pueden usarse para generar implementaciones, ya sea manualmente o automáticamente.

La Figura 8, visualiza el proceso de abstracción y automatización que ofrecen los lenguajes DSL, como se puede ver estos lenguajes están por encima de los lenguajes de programación, esto demuestra el alto grado de abstracción que logran, acercándose más a un usuario no necesariamente con formación en el ámbito del desarrollo de software.

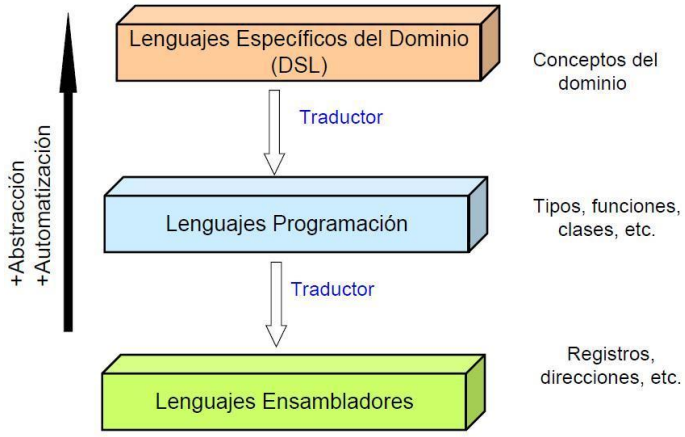


Figura 8 - Abstracción y automatización DSL

Para definir un lenguaje específico de dominio se necesita una herramienta que brinde soporte al DSL. Después se tendrán que definir los conceptos del lenguaje de modelado, así como las reglas de modelado, definiendo cómo usar los conceptos y conectarlos entre ellos. A continuación se deberán definir las metáforas gráficas, es decir, utilizar símbolos que se aproximen a los elementos del mundo real que representan. Los generadores de código específico de dominio aseguran que el código generado por cada desarrollador, siempre reúne con exactitud los requisitos establecidos por el experto de la organización desarrolladora, ya que el código se construye contemplando dichos requisitos [18].

2.9.2 Tipos de DSL

Los DSLs se pueden clasificar desde un punto de vista de la construcción del lenguaje en:

Internos: Utilizan un determinado lenguaje anfitrión para darle la apariencia de otro lenguaje concreto. Un ejemplo claro son lo que actualmente se conoce como Fluent Interfaces[8].

Externos: Tiene su propia sintaxis y es necesario un parser para poder procesarlos. Un ejemplo claro de DSL externo es SQL (Structured Query Language).

Desde el punto de vista del formato del lenguaje:

Textuales: La mayoría de los lenguajes informáticos son textuales y están formados por un conjunto ordenado de sentencias. Un ejemplo muy conocido de DSL textual es SQL utilizado para realizar consultas a una base de datos. Una forma de crear DSLs textuales es mediante la creación de una determinada gramática (por ejemplo utilizando EBNF - Extended Backus-Naur Form) y posteriormente crear o utilizar un parser para dicha gramática, que en etapas posteriores puede interpretar el DSL o generar código. El diseño de una máquina de estado a través de un lenguaje externo textual se demuestra en la Figura 9.

[8] Es una construcción orientada a objeto que define un comportamiento capaz de retransmitir el contexto de la instrucción de una llamada subsecuente.

Máquina de estado con DSL externo textual

```
state Inactivo {
    is_initial
    transitionTo Rastreando {
            event  objetivoEn(p)
            condition  amenaza
            action [t.addObjetivo(p)]
    }
    transitionTo Buscando {
            event ruido
}
}
state Buscando {
    transitionTo Rastreando { }
}
state Rastreando {
    transitionTo Acoplamiento {  event contactar }
}
```

Figura 9 - Maquina de estado con DSL externo textual

Gráficos: En los últimos años están ganando gran aceptación los lenguajes gráficos, podrían citarse como ejemplo UML. La creación de un lenguaje gráfico es similar a la de un lenguaje textual, la única diferencia es que en lugar de usar texto para representar los conceptos, se utilizan conectores y figuras simples. Una máquina de estados diseñada a través de un lenguaje externo gráfico se puede ver en la Figura 10.

Máquina de estado con DSL externo gráfico

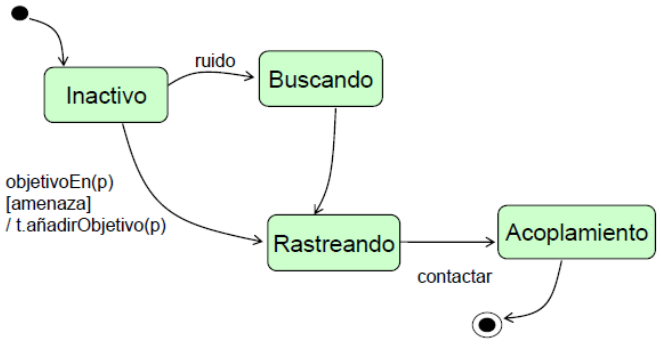


Figura 10 - máquina de estado con DSL externo gráfico

Desde el punto de vista del dominio del problema:

Horizontales: Los DSL horizontales son aquellos en los que el cliente que utilizará el lenguaje no pertenece a ningún dominio específico. Un ejemplo son los editores visuales de entornos de desarrollo que permiten generar interfaces de usuario automáticamente (por ejemplo Windows Forms de visual Studio).

Vertical: A diferencia de los DSL horizontales, el cliente que utilizará el lenguaje pertenece al mismo dominio que el lenguaje en sí. Por ejemplo un lenguaje de definición de encuestas, los usuarios finales serían los expertos en estadística encargados de definir dichas encuestas.

2.9.3 Partes de un DSL

Los lenguajes de dominio específico [34] son el elemento principal de cualquier solución de dominio específico, estructuralmente está compuesto por:

- Sintaxis abstracta: Define los diferentes elementos del lenguaje y las reglas que establecen cómo pueden ser combinados
- Sintaxis concreta: Define cómo los elementos del lenguaje aparecen en una notación visual o textual utilizable por los usuarios
- Semántica: Define el significado del lenguaje

La Figura 11 refleja esta composición.

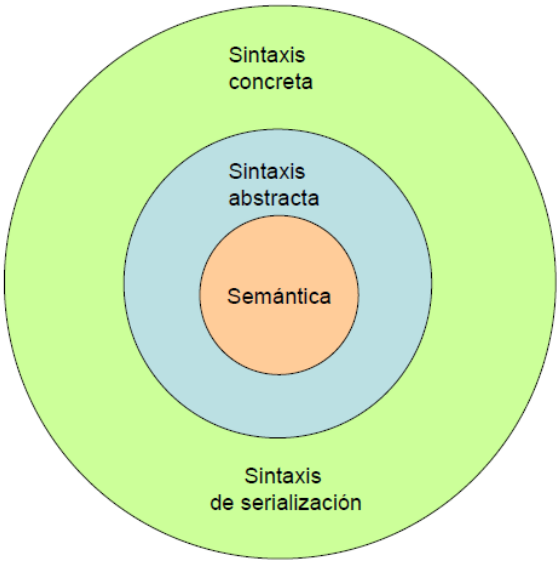


Figura 11 - Partes de un DSL

Resumiendo, la sintaxis abstracta de un lenguaje especifica su estructura, es decir, las construcciones, propiedades y conectores que pueda tener dicho lenguaje. La sintaxis concreta es necesaria para especificar la notación específica con la que los usuarios del lenguaje podrán utilizarlos. Idealmente cada concepto del dominio y del lenguaje se mapean a una representación en la notación específica. Es importante aclarar que una misma sintaxis abstracta podría tener diferentes sintaxis concretas.

2.10 Trabajos relacionados

Dada la aproximación de ADSM con DSM, como trabajos relacionados se encuentran todas las herramientas que están arraigadas a DSM, es decir herramientas que generan código a partir de los modelos. Mediante la generación de código completo directamente desde los modelos, el modelado específico de dominio utiliza lenguajes de modelado gráfico para construir rangos estrechos de aplicaciones más rápido que la codificación manual tradicional. Hasta la fecha, sin embargo, la única forma de hacerlo es desarrollando tanto el lenguaje de modelado y sus generadores de dominio específico.

Frameworks más conocidos y ampliamente utilizados en la actualidad son los desarrollados por Eclipse en su IDE para programadores Java:

1. El Eclipse Modeling Framework (EMF) que permite introducir datos deseados en su modelo, para luego generar editores basados en tablas y un esquema XMI para los modelos.
2. El Graphical Editor Framework (GEF) que proporciona funciones y clases útiles para especificar editores gráficos de datos de Eclipse

Aunque pueden ser utilizados en forma separada EMF y GEF, para construir una solución DSM se necesita, por supuesto el IDE Eclipse. La función principal de EMF es proporcionar una entrada de datos y un entorno de almacenamiento siguiendo el esquema que el usuario mismo provee.

EMF solo proporciona una parte de la solución DSM, almacenamiento de datos, archivos de propiedades, árboles, navegación basada en tablas y un framework para la generación de código. GEF, proporciona el soporte gráfico necesario para la construcción de un editor de diagramas en un nivel más alto de abstracción para el framework EMF, esto facilita la forma de modelar por su propiedad de graficar visualmente.

Otra de las herramientas que permiten aplicar una aproximación DSM es MetaEdit+, la cual está basada en todas las herramientas CASE. MetaEdit+ incluye propiedades genéricas de las herramientas CASE como el comportamiento de los objetos y relaciones, incluyendo un editor de diagramas, un buscador de objetos-gráficos y diálogos con propiedades. Los desarrolladores DSM solo tienen que especificar el lenguaje de modelado, por ejemplo crear un nuevo tipo de objeto, escogiendo un nombre y sus propiedades.

Además de las propiedades CASE incluidas por MEtaEdit+, también incluye propiedades de importación y exportación XML, una API para datos y control de acceso a las funciones MetaEdit+ junto con un generador de código genérico. El generador de código utiliza un lenguaje específico de dominio (DSL) que permite especificar cómo administrar sus modelos, este DSL tiene que ser especificado por el usuario MetaEdit+, esto hace que la definición de generadores de códigos sea un poco más sencilla pero sin quitar la precondición de tener previamente el DSL necesario para llegar a la generación de código.

Si bien estos frameworks brindan sus beneficios y las posibilidades de generar código automáticamente poseen las limitaciones de la metodología DSM, el hecho de tener que pasar por etapas estrictas previas a la generación de código hace que el usuario de estos frameworks necesite tener alguna experiencia previa en este tipo de aproximaciones.

ADSM replantea la filosofía de generación de código, un poco invirtiendo el proceso de creación del DSL, agilizando el proceso de generación de código para optimizar tiempo de implementación e impone la forma innovadora de ir descubriendo el DSL iterativa e incrementalmente a medida que un usuario sin experiencias previas, comienza a utilizar las sencillas técnicas de templating que ADSM ofrece.

3. Capítulo 3: Metodología Agile DSM

3.1 Introducción

Los desarrolladores de software, muy a menudo se encuentran con múltiples escenarios durante el desarrollo de diferentes proyectos, donde pequeñas repeticiones en el código al momento de la implementación van captando cierta atención. Muchos programadores, en multitudinarios proyectos se ven obligados a codificar una misma estructura de código con mínimas variantes entre ellas. Poniendo un ejemplo concreto, dentro de un desarrollo, al momento de desarrollar altas, bajas de diferentes datos, a través de bases de datos, comienzan a aparecer ciertas reincidencias en el código que conducen a desarrollar el mismo código, con pequeñas variantes.

Estudios avanzados sobre ciencia neuronal detallan la forma en que el cerebro humano trata con la incertidumbre buscando regularidades. Nuestras neuronas detectan patrones de forma natural, el cerebro humano no es más que un "banco de datos de experiencias previas" conformado para detectar ciertos patrones. La *pareidolia* es un fenómeno psicológico donde un estímulo vago y aleatorio es percibido por el cerebro como una forma reconocible lo que conduce a la detección de patrones [36].

Se pude detectar que las repeticiones de código en determinados proyectos similares conforman ciertos "patrones" repetitivos. A partir de estos patrones detectados se comenzó a investigar sobre la posibilidad de obtener generadores de código, templating, modelado y metodologías ágiles que hagan del desarrollo de software una tarea más ágil, eficiente, que no requiera de una suma considerable de recursos previos y la necesidad de familiarizarse con un enorme framework con todo lo que esto conlleva (aprender sintaxis complejas, instalar el framework, leer extensos tutoriales para su uso, etc.)

Si bien se sabe que la generación de código data desde la existencia de los primeros compiladores, hasta la aparición de los primeros generadores de código comercial; la generación de código era exclusividad de programas compiladores especializados. En tiempos más recientes la generación de código, gracias al avance de la ingeniería del software, se ha llevado a un nivel diferente, donde se encuentran programas generadores de pantallas, reportes y consultas, estas son herramientas de gran utilidad; pero se debe, en la mayoría de los casos, endeudarse económicamente para hacer uso de ellos.

Hoy en día se está avanzando y dando pasos firmes en el tema de generación de código y modelado, pero aún no se puede hablar de una herramienta de metamodelado ágil que cumpla con todos los requerimientos necesarios que permitan en un nivel de abstracción más alto, personalizarlos para un determinado dominio específico sin tener que pasar por un framework de scaffolding o el tradicional Domain Specific Modeling (DSM).

Es relevante destacar que existen diferentes tipos de generación de código [35]:

1. Templating: Se genera un armazón (o esbozo/plantilla) de código fuente no funcional para ser editado, con el que se evita tener que escribir la parte más repetitiva del código. Este tipo de generación es el adoptado por ADSM.
2. Parcial: Se genera código fuente que implementa parcialmente la funcionalidad requerida, pero que el programador usará como base para modificar, integrar y/o adaptar a sus necesidades.
3. Total: Se genera código fuente funcionalmente completo pero que no va a ser modificado por el programador, sino que si es necesario se vuelve a regenerar. Este tipo de generación es el utilizado por DSM.

El mismo autor de esta tesis fue testigo de que todos, o la mayoría de los programadores se encierran en el código que debe confeccionar, y no en la búsqueda de una herramienta que les pueda ayudar a generar parte de ese código.

Profundizando la investigación sobre este tema, se detectó que lo que más se aproximaba a lo deseado era la metodología "modelado específico del dominio" o DSM, la cual se tomó como base para desarrollar *ADSM*.

3.2 Filosofía DSM

DSM requiere de desarrolladores con experiencia para crear tres grandes componentes básicos:

1. Un metamodelo inicial para crear un lenguaje de modelado específico de dominio (DSL)
2. Un generador de código de dominio específico
3. Un framework de dominio

Con estos tres componentes, los usuarios finales (mayormente desarrolladores) solo tienen que enfocarse en crear modelos haciendo uso del lenguaje (DSL), desarrollado por personal específico, para lograr que se genere código automáticamente.

Los cambios producidos por DSM en una organización pueden parecer radicales, la metodología DSM pretende introducir al desarrollador la detección de tres simples prácticas [13]. Estas prácticas son:

- No te repitas tú mismo ("*don't repeat yourself*")
- Automatizar después de tres apariciones
- Las soluciones personalizadas encajan mejor que las genéricas

En DSM, los modelos se construyen usando conceptos que representan objetos del dominio de aplicación en lugar de conceptos de un lenguaje de programación determinado. El lenguaje de modelado sigue las abstracciones y semántica del dominio, permitiendo que los desarrolladores se perciban trabajando directamente con los conceptos del dominio.

Para estos fines, DSM utiliza lenguajes específicos de dominio (DSL) en combinación con generadores de código y frameworks de dominio, pasando de especificaciones de muy alto nivel a código resultante ejecutable.

El desarrollo de una plataforma DSM requiere una considerable inversión de recursos, la necesidad de usuarios con determinados roles como por ejemplo, desarrolladores experimentados y expertos del dominio que a menudo no poseen antecedentes en el desarrollo de software, más adelante en éste capítulo, se detallarán los roles de usuario dentro de la metodología DSM.

Mientras un lenguaje de modelado tradicional tiende a ser general, un lenguaje DSL para DSM resulta lo más específico posible, elevando el nivel de abstracción de los modelos. Mientras más específico sea el dominio, mayor es el beneficio en productividad de aplicar DSM. Para crear un DSL es necesario un marco teórico brindado por el metamodelado, el cual permite describir los elementos de un modelo para DSM. Para que los modelos sean lo más específicos posible, deben ser generados necesariamente por expertos en el dominio, es decir personal capacitado en el dominio es quien se encarga de crear, editar y borrar las especificaciones del sistema completo a través del modelo para aplicar la metodología en un proyecto determinado, y lograr de esta manera una generación de código completa de un sistema.

Un modelo es una representación de alguna realidad en la cual se expresa un requerimiento de una aplicación o sistema que se debe cumplir, usualmente un modelo simple, como puede ser un diagrama, representa una vista seleccionada de un sistema, programa o característica particular. Para lograr una especificación completa de un sistema, se necesitan varios modelos y conceptos de modelado para poder especificar cada aspecto del mismo.

Los modelos son formales, están respaldados por un generador de código, un framework de dominio y una plataforma subyacente de destino. Estos componentes proporcionan los trabajos necesarios para llevar a cabo la evolución de los modelos exactos y extremadamente trabajados por los expertos del dominio, para lograr una especificación necesaria y llegar al resultado de poseer un sistema completo a partir de la perspectiva de un modelador.

3.3 Roles de usuarios en DSM

En DSM se distinguen 2 grandes roles [13] dentro de la organización del desarrollo, los que crean aplicaciones con DSM y los que desarrollan la solución DSM. Esta separación no significa que las personas sean necesariamente diferentes. Por lo general los que desarrollan la solución DSM también lo están utilizando. Es crucial que los desarrolladores más experimentados hagan la solución DSM, especifiquen la automatización en términos de lenguajes, generadores y frameworks de dominio.
Dentro de DSM se pueden identificar los siguientes roles de usuarios:

- **Expertos del dominio:** Son personas que tienen conocimiento sobre el problema del dominio. Estos conocen la terminología, conceptos y reglas del dominio e incluso en ocasiones han creado el propio dominio. Los desarrolladores de aplicaciones similares en un pasado también son calificados expertos del dominio, la creación de una sola aplicación no suele ser suficiente para ser experto del dominio debido a no poseer suficiente experiencia para encontrar generalizaciones y abstracciones de más alto nivel.
- **Usuario:** Son los encargados de aplicar los lenguajes de modelado. Este grupo es normalmente el más grande entre los usuarios de DSM.
- **Desarrolladores de lenguajes:** Especifican el lenguaje de modelado. Se formalizan en un metamodelo, proporcionan manuales y ejemplos para su uso. Los desarrolladores de lenguajes trabajan cerca de los expertos del dominio y los usuarios DSM. Por lo general solo una o dos personas son responsables de la especificación del lenguaje, especialmente si se utilizan herramientas basadas en metamodelo para la creación de la solución DSM.
- **Ergonomista:** Un ergonomista puede ayudar a los desarrolladores del lenguaje a mejorar la usabilidad del lenguaje. Por ejemplo para la creación de una solución DSM para aplicaciones UI (Interfaz de usuario) o HMI (Interfaz hombre-máquina) puede ser relevante para el modelo poseer grandes coincidencias con el producto real. Este tipo de personal que definen los estilos de interfaz, apoyan a los desarrolladores del lenguaje. El papel del ergonomista es muy importante si el lenguaje es utilizado por un usuario sin conocimientos previo de programación, o si los usuarios abarcan varios continentes y culturas.
- **Desarrolladores del generador:** Estos usuarios especifican las transformaciones de modelos a código siguiendo los formatos y las implementaciones

de referencia dados por los arquitectos y desarrolladores del framework de dominio. A menudo los desarrolladores del generador son las mismas personas que definen el framework de dominio.

- **Desarrolladores del framework de dominio:** son por lo general desarrolladores experimentados y arquitectos de aplicaciones. Pueden proporcionar implementaciones de referencia y pueden especificar como se debe utilizar el entorno de destino.
- **Desarrolladores de herramientas:** Implementan los lenguajes de modelado y generadores de código. Dependiendo de las herramientas utilizadas, este grupo puede no ser necesario. En pocas ocasiones se cuenta con generadores automáticos y editores de modelado que proporcionen en forma automática la especificación del lenguaje y su generador.

3.4 Agile DSM

La metodología ADSM nace como una alternativa ágil al tradicional DSM, este enfoque permite, incluyendo ventajas de una aproximación DSM, el descubrimiento y creación de un metamodelo formal, al final de un proceso iterativo e incremental de templating, incluyéndose en este marco, la generación automática de código. De ésta forma, templetizando soluciones, evita el problema de la repetición de código tomando como principio una de las filosofías de DSM "*No te repitas tú mismo*" [13] y la capacidad humana de detectar patrones repetitivos en el código.

Este proceso incremental va conformando pequeñas soluciones con similitudes a una aproximación DSM, iterativamente para aumentar la productividad y optimizar partes puntuales del proceso de desarrollo. Como resultado del proceso incremental, al final, además de generar código automáticamente, se obtiene el metamodelo formal.

En el marco de esta tesis, se desarrolló una herramienta denominada Make Your Language que da soporte a la metodología ADSM, esta herramienta será detallada profundamente en el capítulo 4.

3.4.1 El proceso de ADSM

El proceso ADSM comprende etapas las cuales se dividen de la siguiente manera:

1. **Detección de fragmentos de código repetitivos (patrones)** capaces de ser abstraídos en un nivel más alto de expresión: Un desarrollador de software tiene el completo control sobre el código en el cual está desarrollando, esto le da la ventaja de ir detectando fragmentos o patrones que se repiten y que pueden ser abstraídos para no volver a reincidir lo mismo en un desarrollo que comprenda una estructura de código que solo difiere en mínimos aspectos.
2. **Introducción de anotaciones especiales para generar templates:** Una vez detectados estos patrones de código repetitivos, se debe elevar la abstracción (generalizar una solución) de los mismos para producir las plantillas (templates) que luego servirán para generar código y un metamodelo.
3. **Inferencia de modelo**: A partir del template abstraído en el punto anterior se infiere un modelo de ejemplo, en el marco de ésta tesis ADSM adopta JSON (JavaScript Object Notation) como formato para el modelo, el motivo de esta elección se describirá posteriormente; éste modelo será de gran utilidad ya que con él serán

renderizados los templates para le generación automática de código y una vez finalizado el proceso de ADSM se formalizará.

4. **Generación de código**, el cual puede testearse al instante y volver al paso 2 en caso de no estar conforme con lo generado: en este punto ya se tiene el template y el modelo en formato JSON, lo que habilita la renderización (generación de código) de los mismos para la obtención de código automáticamente.

Estas etapas se pueden resumir a 3 grandes pasos donde se producen las iteraciones necesarias y el desarrollo incremental para llegar al resultado conforme al usuario de ADSM. Estas etapas comprenden, editar un template, inferir y editar el modelo y generar código, gracias a la retroalimentación que se produce, de ser necesario se puede pasar del paso 3 (código ya generado) al paso 2 (edición del modelo) para obtener nuevos resultados al instante. La Figura 12 ilustra este proceso iterativo e incremental durante el proceso de ADSM.

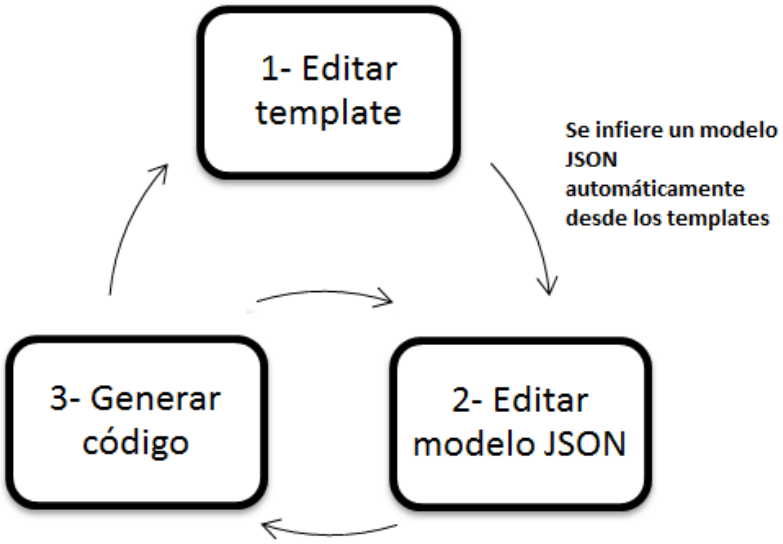


Figura 12 - Proceso ADSM

De esta manera se le da al desarrollador y usuario de la metodología posibilidad de ir testeando las generaciones de código prácticamente "on the fly" (sobre la marcha) logrando un feedback (retroalimentación) constante del proceso de metamodelado y generación de código. Agile DSM hace posible la inferencia de modelos y generación de código en forma automática, a través de la herramienta desarrollada en el marco de esta tesis, denominada Make Your Language (MYL), la cual se detallará en el capítulo 4.

Inicialmente, el usuario debe detectar en su código, un patrón repetitivo, es decir fragmentos de código que se ven reflejados con la misma lógica de programación, para llegar a esto, el usuario, debe realizar un detallado seguimiento del código de su proyecto y detectar estos fragmentos que se ven repetidos para elevar el nivel de abstracción y optimizar el desarrollo posterior de código similar. Existen 2 aproximaciones para la detección de patrones, las cuales se detallan más adelante en el presente capítulo.

Una vez detectados y abstraídos los patrones de código, se procede a la creación de la/las plantilla/s o template/s, donde será generalizada la estructura lógica del fragmento de código a automatizar a través de la sintaxis específica que se describirá más adelante.

Obteniendo los templates, el siguiente paso es generar los modelos. Los modelos de datos se describen en texto plano, es decir que son modelos textuales, se utiliza el formato JSON, en él, van reflejado los datos necesarios para lograr adecuadamente la generación del código. Esta generación se lleva a cabo realizando una renderización con el/los template/s anteriormente detectados obteniéndose un código fuente ejecutable.

En la Figura 14, se demuestra el proceso explicado hasta aquí, como se puede ver se parte del código inicial de algún proyecto existente, luego se detectan patrones de código que puedan ser elevados a un nivel más alto de abstracción, para luego templetizar, es decir, generar templates según preferencias y dominio de la solución; a partir de aquí se infiere o crea el modelo en formato JSON, para luego generar el código resultado. En este punto se puede observar el resultado al instante y en caso de detectar resultados de generación no adecuada se procede a realizar el feedback o retroalimentación si se considera necesario.

Este funcionamiento de ADSM en cada iteración permite ir detectando nuevos patrones, armar nuevos templates e incluso ir testeando y haciendo correcciones, dado a que al instante puede visualizarse la generación obtenida. En cada iteración se va enriqueciendo lo que conforma el modelo, que en una instancia final, formará parte del metamodelo en conjunto con reglas de validación que impondrá el mismo usuario, las cuales se detallarán más adelante.

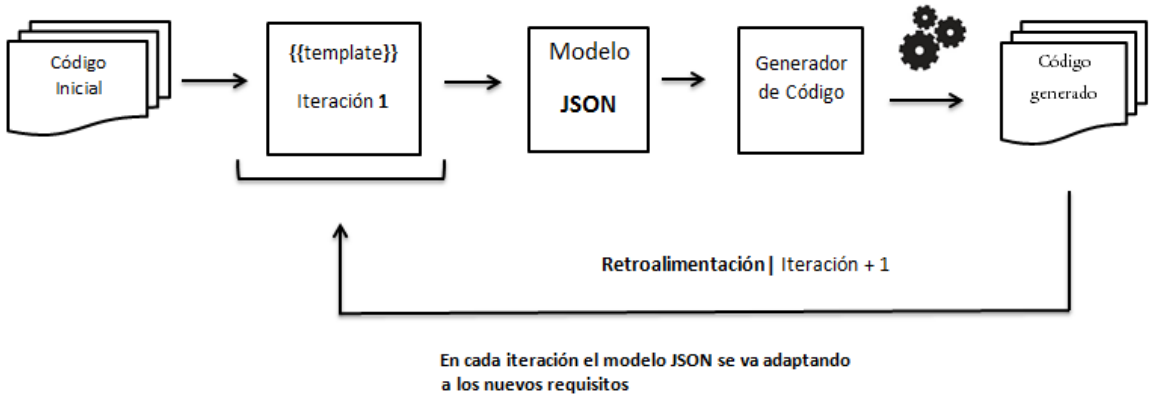


Figura 14 – Proceso ADSM

3.4.2 Escenarios posibles para la detección de patrones

Un patrón es un tipo de tema de sucesos u objetos recurrentes, como por ejemplo grecas[9], a veces referidos como ornamentos de un conjunto de objetos. Más abstractamente, podría definirse "patrón" como aquella serie de variables constantes, identificables dentro de un conjunto mayor de datos [37].

Estos elementos se repiten de una manera predecible. Puede ser una plantilla o modelo que puede usarse para generar objetos o partes de ellos, especialmente si los objetos

[9] Franjas donde se repite un mismo motivo

que se crean tienen lo suficiente en común para que se infiera la estructura del patrón fundamental, en cuyo caso, se dice que los objetos exhiben un único patrón.

En un ambiente de desarrollo es muy común detectar patrones reincidentes en el código compartido en múltiples archivos, donde las variantes entre un código y otro son muy sutiles y pequeñas [37]. Estas variantes pueden ser generalizadas para evitar la codificación repetitiva y optimizar el tiempo de desarrollo. Para llegar a esto es esencial detectar estos patrones.

Agile DSM admite 2 grandes aproximaciones para detectar patrones y generalizarlos en una plantilla (template). Como se visualiza en el siguiente gráfico, los patrones repetitivos pueden ser detectados en N archivos, generalizándolos en 1 template para luego a partir del template, llegar a la generación automática de N archivos resultado (Figura 13 - A). También se puede detectar un patrón repetitivo simplemente en un archivo determinado para luego templetizarlo y a partir de este generar N o 1 archivos automáticamente (Figura 13 - B).

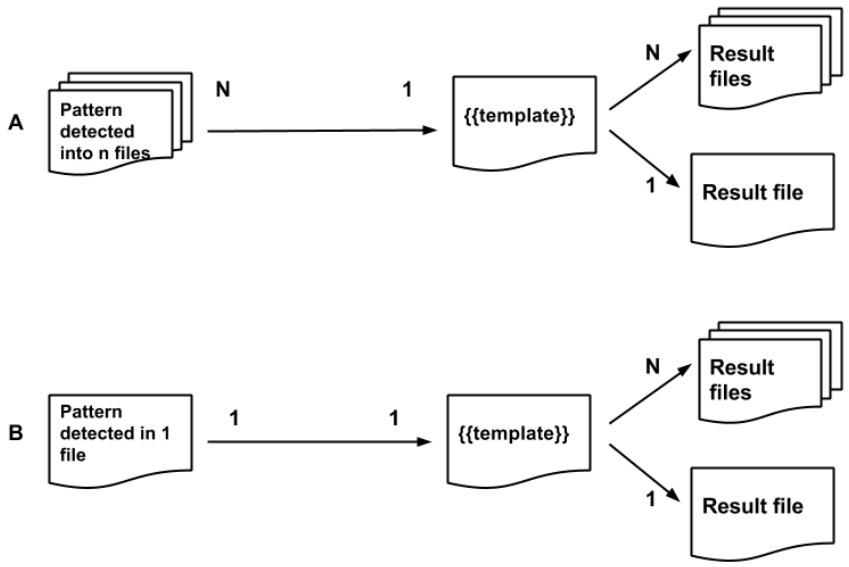


Figura 13 - Escenarios para la detección de patrones

Para la generación múltiple de archivos resultado, la herramienta MYL ofrece una sintaxis especial que consta de renombrar el nombre del archivo template, esta sintaxis se detallará en el próximo capítulo cuando se hable de sintaxis para generación múltiple de código.

3.4.3 JSON como modelos

ADSM adopta el formato JSON para la edición y creación de sus modelos de datos dado que utilizar JSON como formato para los modelos conduce a una simpleza y bajos recursos de aprendizaje para aplicarlo. Hoy en día JSON es ampliamente utilizado, es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML, cualquier desarrollador o usuario con mínimos conocimientos en programación puede comprender su sintaxis y semántica en unos pocos minutos. Estas características fueron tomadas en cuenta para que el entorno MYL utilice JSON como formato para los modelos.

3.4.4 Formalización del metamodelo

Como se mencionó, uno de los objetivos principales de ADSM, es descubrir un metamodelo formal incremental e iterativamente. ADSM hereda una de las ideas básicas de DSM, no utilizar conceptos de lenguajes de programación de propósito general, sino definir software utilizando modelos. Los conceptos y relaciones sobre los cuales pueden versar éstos modelos empleados, están definidos en un metamodelo.

Un lenguaje de modelado o DSL se define formalmente mediante un metamodelo, compuesto por sintaxis, restricciones y semántica donde se especifica su correcto empleo; por esta razón se necesita formalizar el metamodelo obtenido, para poder diseñar correctamente modelos JSON válidos según el dominio específico.

La Figura 15, intenta reflejar las iteraciones que llevan a la formalización del metamodelo en el proceso ADSM. Estas iteraciones se pueden subdividir en micro y major iterations, las micro iterations comprenden:

1. Generar los templates, que permiten obtener instancias de muestra de modelos JSON
2. Pueden ser utilizadas reglas de validación escritas en JavaScript, para ir formalizando los modelos en cuestión, obteniendo una parte del metamodelo
3. Finalmente se puede utilizar JSON Schema para validar la estructura de los modelos, esto es, validar sus conceptos y relaciones definidas

Las major iterations se producen en menor medida que las micro, y por lo general cuando son incorporados nuevos templates y patrones. En las major iterations sufre cambios el metamodelo formal obtenido en las micro iterations, sumándose nuevos patrones y templates.

Este proceso hace de la metodología ADSM su productividad desde un inicio, es decir que permita generar código y elevar la abstracción directamente desde los primeros momentos de su adopción e ir ampliando y ajustando el metamodelo.

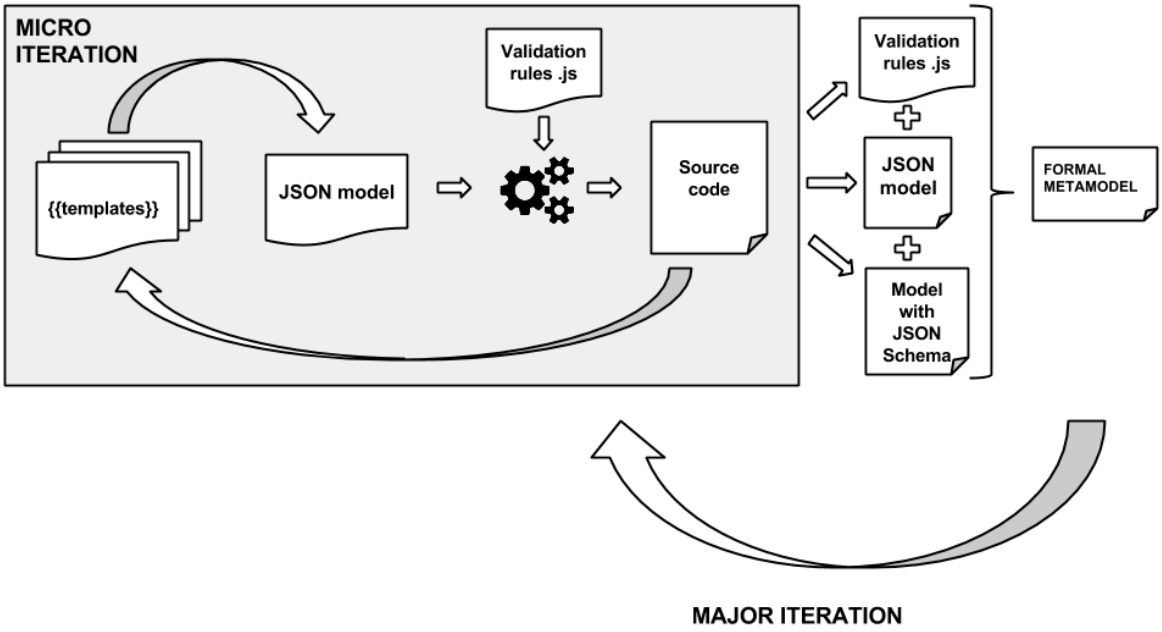


Figura 15 - Formalización de metamodelo

3.5 ADSM frente a DSM

Si bien son dos aproximaciones diferentes que se adecuan a diferentes proyectos y entornos de desarrollo, al fin y al cabo pretenden brindar agilidad en el plan del desarrollo de un sistema de software.

Como se mencionó anteriormente, DSM requiere una considerable inversión de recursos, la necesidad de desarrolladores experimentados y expertos del dominio que a menudo no poseen antecedentes en el desarrollo de software. En contraposición a esto en ADSM no es necesaria la presencia de desarrolladores expertos en el dominio, la característica de ADSM es que cualquier desarrollador puede aplicar por sí mismo la metodología sin necesidad de tener conocimientos extras en el dominio, solo basta con conocer la sintaxis de templating, JavaScript y JSON para lograr la automatización de partes puntuales de su proyecto de desarrollo.

En DSM se elevan los costos de aprender un nuevo lenguaje vs su limitada aplicabilidad, esto es, se necesita un considerable tiempo y esfuerzo para aprender un lenguaje que solo tiene finalidad en la herramienta DSM. En ADSM, si bien se necesita un esfuerzo de adaptación propio de toda nueva metodología, se incluyen lenguajes que se encuentran en el ámbito industrial, como JSON y JavaScript.

Otro costo de DSM se ve reflejado a la hora de diseñar, implementar y mantener un lenguaje específico de dominio, así como requieren las herramientas para desarrollar con él (IDE), ADSM ofrece un punto de vista diferente, en el cual el metamodelo se va descubriendo iterativamente y el mantenimiento del mismo es mínimo, no es necesario un entorno IDE lo que infiere un menor costo de configuración y mantenimiento del entorno

En DSM existe una baja oferta de expertos en el dominio y en un DSL particular, lo que tiende a elevar los costos laborales y retrasar el proceso de implementación en un desarrollo. Una de las precondiciones de DSM es que previamente se necesita tener el metamodelo para desarrollar el DSL que previamente servirá para crear los modelos necesarios.

Cuando hablamos de metamodelo en DSM, nos referimos al metamodelo formal necesario para construir el lenguaje DSL completo a generarse necesario para la administración de los modelos específicos del dominio. A diferencia de esto, ADSM trabaja con metamodelos un tanto más pequeños que se van incrementando y complejizando según las preferencias del usuario.

Con ADSM se genera solo el código que se necesita, se detecta un patrón, se templetiza, se infiere el modelo y se obtiene el código resultante de inmediato. Esta generación de código tan inmediata, desde una primer adopción de la metodología, sin necesidad de cumplir con etapas estrictas como en DSM, favorece el testing del código resultante, pudiéndose con ADSM testear al instante el resultado final y tener la posibilidad de realizar un feedback y generar nuevamente código si se desea. Otra ventaja de ADSM es que es usable desde el principio, a diferencia de una solución DSM la cual implica etapas estrictas para llegar al resultado final de la obtención de código.

Como se mencionó anteriormente DSM comprende una amplia variedad de usuarios con diferentes roles (expertos en el dominio, usuarios, desarrolladores de lenguajes, ergonomistas, desarrolladores de herramientas, frameworks de dominio y generadores de código). DSM no puede aplicarse con la ausencia de estos usuarios.

ADSM no necesita de esta amplia variedad de roles de usuarios, es decir, solo basta con la presencia de mínimamente un usuario el cual puede cumplir todos los roles. Esto se logra ya que ADSM apunta a un usuario con conocimientos de programación, pudiendo ser un desarrollador de software no muy experimentado, que sin embargo puede aplicar ADSM con tan solo tener conocimientos de templating, JSON y JavaScript.

En DSM el metamodelo abarca el 100% de los aspectos para modelar una solución completa y satisfacer todos los requerimientos del sistema en su completitud.

En ADSM el metamodelo se va descubriendo iterativamente y se va generando código parcialmente basándose en técnicas de templating, esto es una gran ventaja, ya que los testeos y cambios necesarios se van haciendo “on the fly” a diferencia de DSM habría que cumplir con etapas estrictas y hasta no tener un metamodelo formal para el lenguaje, los modelos, generadores de código y frameworks de dominio no se puede visualizar el resultado final de la generación automática.

A continuación se detalla un cuadro (Figura 16) comparativo entre ADSM y DSM donde se puede apreciar las claras diferencias de estas 2 aproximaciones.

ADSM	DSM
Cualquier usuario con mínimos conocimientos en programación puede aplicar ADSM sin necesidad de ser experto en el dominio. No es necesaria la experiencia de expertos ni de dominio ni de plataforma.	Implica la necesidad de expertos en el dominio que a menudo no poseen antecedentes en el desarrollo de software y desarrolladores experimentados en DSM.
El metamodelo es más reducido y se va descubriendo iterativamente a medida que se aplica el proceso ADSM.	Se precisa un metamodelo formal extenso, que solvente todos los aspectos del sistema previamente.
Generación de código desde el principio con testing incluido al instante.	Generación de código al final de las etapas de modelado, verificación y definiciones necesarias para el proceso DSM
Framework de dominio incorporado de forma natural	Necesidad de definir explícitamente un framework de dominio
Se genera el código que se necesita, proporciona un generador de código utilizando templating	Se genera todo el código de un sistema completo, proporciona un generador de código de tipo total
El metamodelo formal se obtiene al final del proceso ADSM	El metamodelo formal es necesario en principio para poder definir el DSL y aplicar DSM
Costo mínimo de aprendizaje del lenguaje de templating para poder aplicar ADSM a cualquier proyecto de software	Se elevan los costos de aprender un nuevo lenguaje vs su limitada aplicabilidad
El lenguaje específico de dominio se va descubriendo de manera casi transparente al usuario, no necesita un entorno IDE ni configuraciones iniciales para aplicar ADSM	Es muy costoso diseñar, implementar y mantener un lenguaje específico de dominio así como el entorno IDE necesario para DSM

Figura 16 - Comparativas de metodologías ADSM y DSM

3.6 ADSM frente a la codificación manual tradicional

La codificación manual tradicional, requiere una considerable inversión tiempo y aumenta la probabilidad de errores sintácticos. Codificar de manera tradicional código que puede ser automatizado es una pérdida de tiempo innecesaria. Reiteradas veces a los programadores les surgen escenarios donde nos vemos casi obligados codificar casi lo mismo con algunos cambios, muchas veces mínimos, para satisfacer una determinada tarea.

Una característica de ADSM es que busca automatizar de cierto modo esta brecha entre codificar nuevamente lo mismo o muy parecido, ofreciendo a cualquier desarrollador la

posibilidad de aplicar por sí mismo la metodología sin necesidad de tener conocimientos extras en el dominio ni en frameworks especializados que requieren tutoriales extensivos para su comprensión, con ADSM solo basta con conocer la ínfima sintaxis propia de la herramienta que da soporte a ADSM (MYL), JavaScript y JSON para lograr la automatización y generación de código.

Haciendo uso de ADSM, el código se genera de una manera casi transparente al usuario, agilizando mucho, por ejemplo, el desarrollo de pequeños prototipos o partes puntuales de código fuente. En el capítulo 5 se describirán casos de estudios aplicados en el marco de esta tesis donde se estudió la aplicabilidad de la metodología ADSM haciendo comparación con la tradicional forma de codificar manualmente, donde se refleja el grado de eficiencia ganado por ADSM.

4. Capítulo 4: Herramienta desarrollada

4.1 Introducción

En el siguiente capítulo se detalla el diseño y funcionamiento de la herramienta que brinda el soporte a la metodología ADSM, denominada Make Your Language (MYL). Esta herramienta consta de un entorno web desarrollado en Java que brinda la posibilidad de aplicar ADSM.

La herramienta permite administrar de forma clara y precisa los archivos para hacer un manejo de templetizado, esto es, generar plantillas (templates) de código, así como también permite desarrollar reglas de validación y el modelo JSON, para conformar metamodelos. Estas plantillas se crean incorporando anotaciones con una sintaxis determinada.

Una vez generadas las plantillas de código, la herramienta brinda la posibilidad de inferir automáticamente y al instante desde los templates, un modelo en formato JSON que posteriormente se utilizará para llevar a cabo una renderización y generar el código resultante. El usuario de la herramienta es libre de editar el/los modelo/s en formato JSON para poder hacer de él un lenguaje puro y exclusivo para su uso. La herramienta brinda la posibilidad de crear varios archivos a partir de un template y de introducir los datos deseables en los modelos JSON, una vez que el usuario creó su modelo.

Finalmente, la herramienta ofrece la posibilidad de aplicar al JSON reglas de validación y crear/editar un código de procesamiento previo. Las reglas de validación son escritas en lenguaje JavaScript, el mismo usuario es el responsable de poner las restricciones que a él le resulten relevantes para los modelos de datos que luego utilizará para generar su código fuente y serán parte del metamodelo final; esto es por ejemplo, poder validar que una dato se encuentre dentro de un rango o que un dato comience con una letra mayúscula.

El código de procesamiento previo, brinda al usuario la posibilidad de procesar el modelo previo a la renderización, esto es por ejemplo, editar, agregar o eliminar datos del modelo para lograr una generación óptima del código resultante. El usuario puede invocar una "renderización[10]" y de esta manera lograr el código fuente resultado.

4.2 Funcionamiento de la herramienta Make Your Language

4.2.1 Diseño de Interfaz de usuario

Para la creación de la Interfaz, se utilizó el framework de Twitter *Bootstrap*[11] el cual permite crear interfaces web con CSS y Javascript que adaptan la interfaz dependiendo del tamaño del dispositivo en el que se visualice de forma nativa.

[10] Proceso donde la plantilla de código anotada se completa a partir del modelo JSON
[11] En la sección de tecnologías utilizadas se describe en más detalle

Esta tecnología, automáticamente se adapta al tamaño de un navegador de PC o de una Tablet sin que el usuario tenga que hacer nada, esto se denomina diseño adaptativo o Responsive Design.

Además de todas las posibilidades que ofrece Bootstrap a la hora de crear interfaces web, los diseños creados con Bootstrap son simples, limpios e intuitivos, esto les da agilidad a la hora de cargar y al adaptarse a otros dispositivos.

El Framework trae varios elementos con estilos predefinidos fáciles de configurar: Botones, Menús desplegables, Formularios incluyendo todos sus elementos e integración jQuery para ofrecer ventanas y tooltips dinámicos, además posee una extensa documentación y sitios webs con ejemplos muy claros y fáciles de comprender, más adelante se ahondará acerca de este asombroso framework. A continuación se presentan las pantallas principales de Make Your Languaje junto con la descripción de su funcionamiento.

4.2.2 Pantalla principal

En un principio cuando se accede por primera vez a MYL, el usuario se encontrará con una pantalla como se muestra en la Figura 17, donde el sistema estará a la espera que se ingrese un proyecto en formato .zip para comenzar a operar, el usuario puede subir su proyecto en formato comprimido .zip presionando el botón *UploadZip* que se encuentra en la esquina superior derecha en color cian, una vez seleccionado el archivo .zip a subir y confirmado su selección, el sistema realizará la subida inmediata del proyecto y lo visualizará en forma de listado como se muestra en la Figura 18.

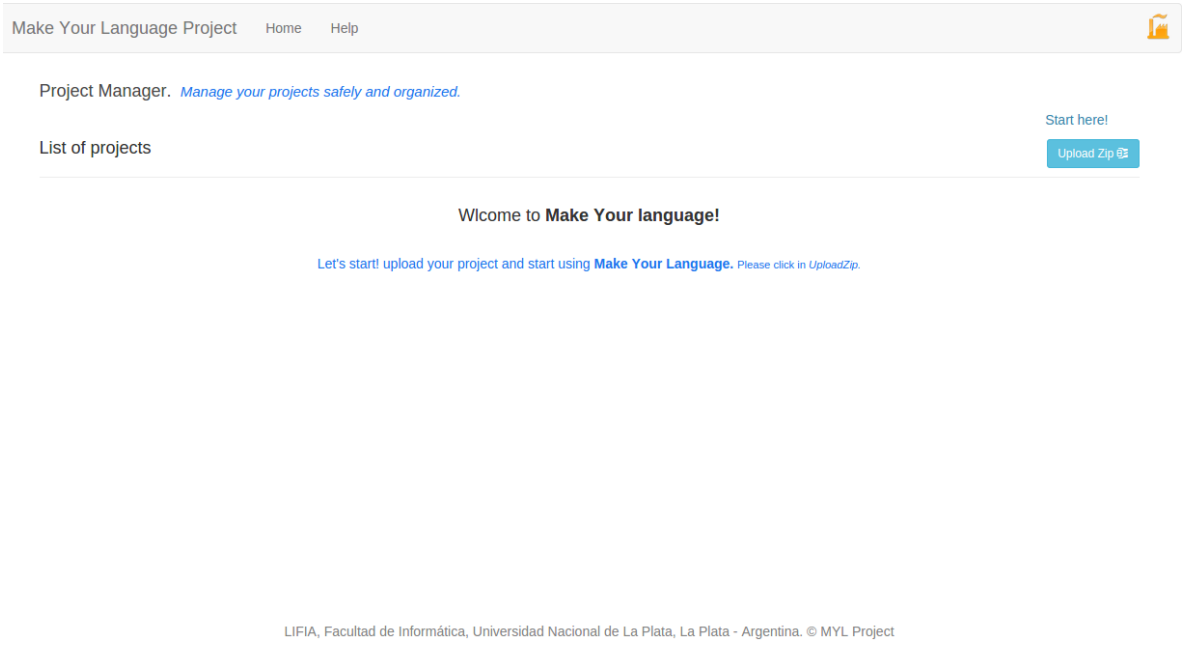


Figura 17 - Pantalla de inicio, ningún proyecto se cargó en el sistema

Con el proyecto cargado en el sistema, el usuario está listo para operar con él. En la pantalla se le presentarán múltiples opciones como acceder presionando sobre el nombre o en el icono que se encuentra más hacia la derecha, a la sección Factory (Figura 21, o a el historial de versiones (Figura 26) renderizadas presionando el botón que se encuentra más a la izquierda del botón de Factory

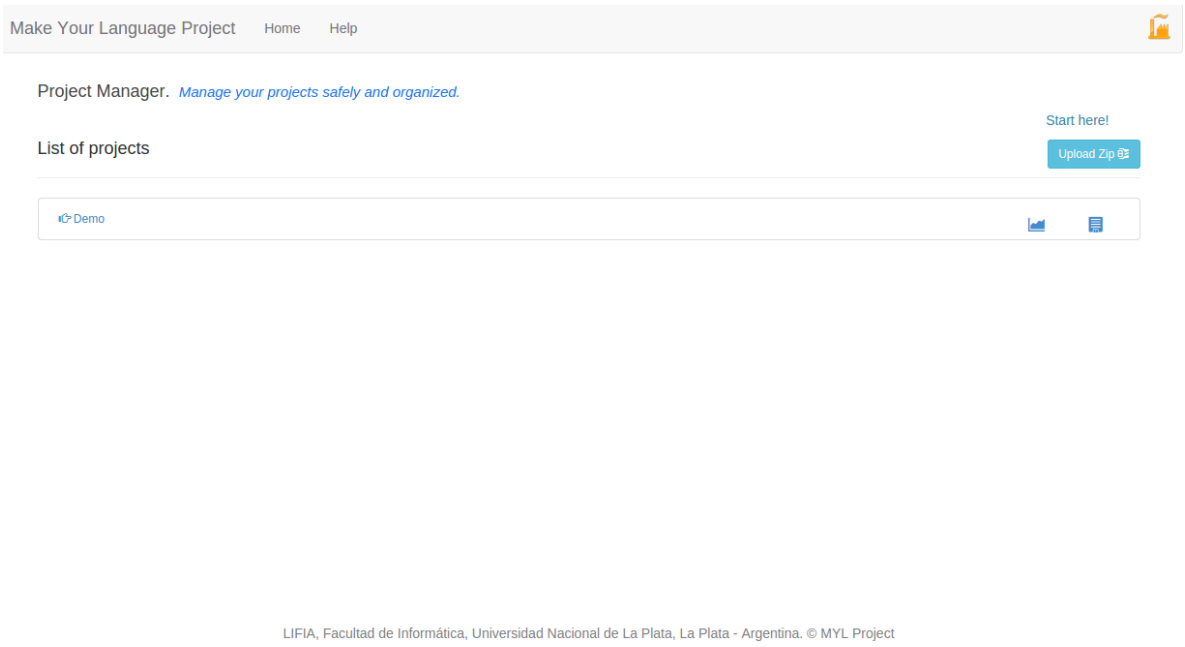


Figura 18-Pantalla de inicio, se cargó el proyecto Demo en el sistema

4.2.3 Factory o Fábrica

Una vez ingresado en la sección Factory, se le visualizará una pantalla como se muestra más adelante en la Figura 21. El usuario, podrá navegar por todos los archivos contenidos en el proyecto que se ha subido en formato .zip. Se visualizará un menú en forma de árbol en la esquina superior izquierda donde se le brindará al usuario la posibilidad de ver el contenido de cada archivo y la posibilidad de editar e incluso eliminar alguno de éstos.

Debajo del árbol de navegación de archivos, se encuentra el listado de los modelos en formato JSON, el usuario podrá acceder al contenido de estos archivos para editarlos a sus preferencias, así como también podrá eliminarlos si no le son de su utilidad.

4.2.3.1 Editor de código

La sección de edición de código en el Factory de MYL, cuenta con un editor donde se procesan los archivos del proyecto subido y se administran los templates, modelos, reglas de validación y código de procesamiento. Para el editor se optó por trabajar con un poderoso componente JavaScript que permite presentar un editor de código en el navegador, llamado CodeMirror. Figura 19.

```java
package model;

public class Contacto {

    private String nombre;
    private String apellido;
    private String numero;
    private String email;
    private long id;


    public Contacto(String name, String ap, String num, String mail, Integer idcontacto) {
        nombre = name;
        apellido = ap;
        email = mail;
        numero = num;
        id = idcontacto;
    }


    public Contacto() {
        // TODO Auto-generated constructor stub
    }


    public String getApellido() {
        return apellido;
    }
```

Figura 19 - Codemirror

4.2.3.2 Editor de archivos / recursos

Cuando se sube un proyecto en formato .zip, en esta sección del editor se visualiza una columna a la izquierda donde se ve el árbol de navegación de archivos del proyecto subido, para lograr esto se hizo uso de un plugin JavaScript llamado FancyTree[12], el cual se visualiza en la figura 20.

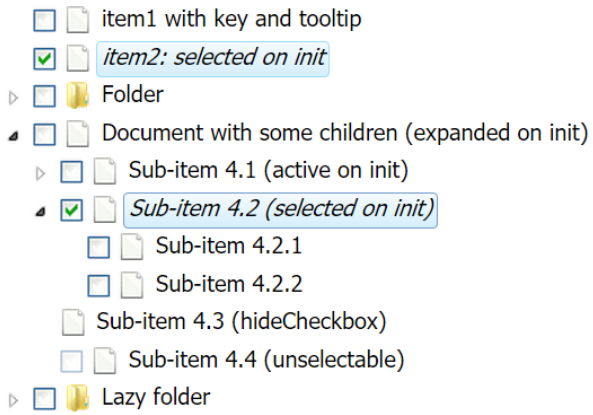


Figura 20 - FancyTree

[12] Mas informacion sobre Fancy Tree: https://github.com/mar10/fancytree

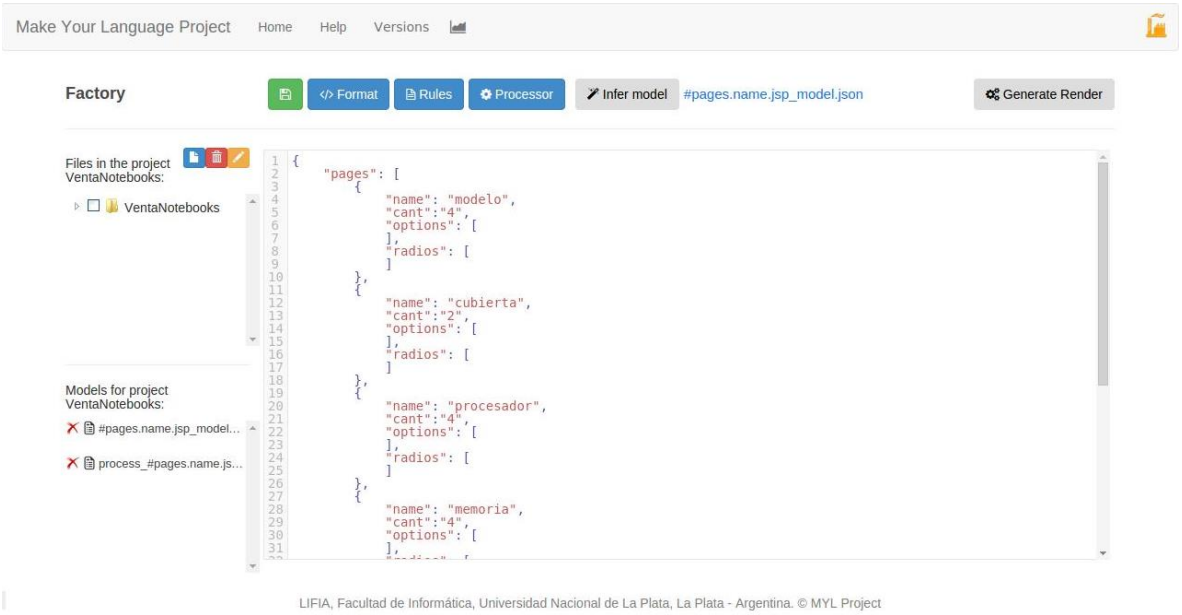


Figura 21 - Se ingresó a la sección Factory

4.2.4 Inferencia y creación de un modelo

MYL, ofrece una funcionalidad de inferencia de un modelo ejemplo automáticamente y de forma inmediata a partir de el/los templates anotados con la sintaxis de generación de plantillas descritas más adelante en el presente capítulo.

La inferencia de un modelo JSON es una opción a la hora de crear un modelo para el proyecto, MYL ofrece también, la posibilidad de agregar un archivo en blanco para construir un modelo, esto se detallará más adelante. Inferir un modelo ejemplo ayuda al usuario a orientarse en la creación del mismo y comprender cómo se estructura.

MYL infiere modelos para templates simples como también para los templates para generación múltiple. El procesamiento que realiza la herramienta para abordar esta tarea, consta de procesar el/los templates anotados en búsqueda de tags o etiquetas propias de la sintaxis para la generación de plantillas y armar un archivo JSON con un esqueleto inicial de un modelo. Un ejemplo de inferencia a partir de un simple template podría ser lo siguiente:

Template *cars.html*:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
  <HEAD>
    <TITLE>{{name}} classification</TITLE>
  </HEAD>
  <BODY>
    <H1>Welcome to {{upperCaseName}} classification</H1>
    <P>On this page you will find the results of competition</P>
    <P>Thanks for access!</P>
    <LABEL>List of best {{name}}:</LABEL>
      <table style="width:100%">
        <tr>
```

```
            <td>Pos</td>
            <td>Name</td>
            <td>Model</td>
            <td>Time</td>
          </tr>
          {{#list}}
          <tr>
            <td>{{pos}}</td>
            <td>{{description}}</td>
            <td>{{age}}</td>
          </tr>
         {{/list}}
        </table>
  </BODY>
</HTML>
```

El resultado de la inferencia arroja un modelo JSON (cars_model.json) como el siguiente:

```
{
    "name": "aValue",
    "upperCaseName": "aValue",
    "list": [
        {
            "pos": "aValue",
            "description": "aValue",
            "age": "aValue"
        }
    ]
}
```

Esta inferencia de un modelo ejemplo fue generada para obtener como resultado de a un archivo a la vez, es decir que permite ir generando un archivo por vez a diferencia de la inferencia de un modelo para generación múltiple que se detallará a continuación.

Un ejemplo de inferencia a partir de un template para generación múltiple podría ser el siguiente:

Template para generación múltiple (#pages.name.html):

```
<HTML>
  <HEAD>
    <TITLE>{{name}} classification</TITLE>
  </HEAD>
  <BODY>
    <H1>Welcome to {{upperCaseName}} classification</H1>
    <P>On this page you will find the results of competition</P>
    <P>Thanks for access!</P>
    <LABEL>List of best {{name}}:</LABEL>
      <table style="width:100%">
        <tr>
          <td>Pos</td>
          <td>Name</td>
          <td>Model</td>
          <td>Time</td>
```

```
          </tr>
          {{#list}}
           <tr>
            <td>{{pos}}</td>
            <td>{{description}}</td>
            <td>{{age}}</td>
           </tr>
         {{/list}}
         </table>
  </BODY>
</HTML>
```

El resultado de la inferencia arroja un modelo JSON (#**pages**.**name**.html_model.json) como el siguiente:

```
{
    "pages": [
        {
            "name": "aValue",
            "upperCaseName": "aValue",
            "list": [
                {
                    "pos": "aValue",
                    "description": "aValue",
                    "age": "aValue"
                }
            ]
        }
    ]
}
```

Es relevante destacar que se visualiza la presencia del atributo "***pages***" como un arreglo donde se puede incluir tantos elementos (nombrados con el valor del atributo ***name***) como archivos se deseen generar, esto se verá más en detalle cuando se describa la sintaxis para generación múltiple.

4.2.5 Reglas de Validación

La herramienta MYL cuenta con una sección donde es posible acceder a las reglas de validación escritas en lenguaje JavaScript, esto se logra accediendo al botón Rules, que se habilitará al presionar en cualquier archivo perteneciente a los modelos del proyecto.

La figura 23 visualiza la consola de administración de reglas de validación, como se puede ver a la izquierda se visualiza el modelo JSON al cual se le aplicará las reglas, a la derecha se encuentra el editor de código code Mirror para JavaScript y por debajo una consola para testear las reglas.

Las reglas de validación admiten la sintaxis completa de JavaScript, permitiendo un amplio manejo de los modelos para poder validar cada uno de los campos que el usuario considere relevante para su dominio específico. Un ejemplo sencillo de regla de validación podría ser lo siguiente:

```
for (i = 0; i < model.length; i++) {
if(model.prop != “Value”){
            errors.addError(“there have been problems!”)
      }
}
```

En este ejemplo sencillo se verifica que la propiedad “prop” del modelo “model” sea distinta de “Value”, en caso afirmativo agrega el mensaje de error “there have been problems!”, el cual será notificado en la situación donde se desee renderizar un modelo que cumpla con esta regla.

El usuario puede editar las reglas teniendo acceso a variables dentro del editor JavaScript. Estas variables, son model y errors, mediante model podrá tener acceso al modelo por el cual está accediendo y con errors podrá ingresar los errores que considere relevantes a visualizar cuando un error es encontrado al momento de generar una nueva renderización.

Contar con la posibilidad de validar el modelo a través de un código JavaScript es esencial para que llegado el punto final de iteraciones donde el modelo JSON ha alcanzado un grado de madurez deseado, estas reglas sirven para formalizar el metamodelo en conjunto con JSON Schema[13].

4.2.5.1 Edición y ejecución de reglas

Las reglas de validación pueden ser editadas por el propio usuario del sistema MYL. Cada regla de validación está asociada a un proyecto, es decir que cada proyecto posee un archivo de reglas de validación.

El entorno MYL ofrece en la sección reglas de validación (Figura 22), acceso a variables preseteadas para poder manipular el modelo y administrar excepciones o errores, estas son ***model*** y ***errors***. La variable *model* se encuentra preseteada con el modelo, es decir que el usuario puede acceder y navegar el modelo JSON a través de la misma. La variable *errors* se encuentra preseteada y corresponde a una lista de errores, donde el usuario podrá setear cada error/excepción que será notificada en caso de existir alguno, al momento previo a la renderización de un archivo, la variable *errors* admite el método *addError (String)* el cual será utilizado por el usuario para agregar un nuevo error/excepción a la lista de errores.

El sistema MYL, recorrerá esta lista de errores en búsqueda de los mismos al momento previo de renderizar el modelo con el/los template/s y visualizará las excepciones correspondientes en la lista (variable *errors*).

La sección de reglas de validación permite a su vez que el usuario chequee las reglas, estos es posible gracias a una consola que ese encuentra debajo donde el usuario puede interactuar con los resultados de su código, esta utilidad evita errores posteriores y ofrece un entorno mucho más amigable e intuitivo para el usuario final de MYL.

El guardado de las reglas de validación se irá realizando automáticamente mientras el usuario edita, esto es posible gracias a la función que ofrece el sistema de “autosave”, a medida que se van produciendo cambios en el archivo de reglas de validación se van salvando en el archivo en forma automática.

[13] JSON Schema describes your JSON data format http://json-schema.org/

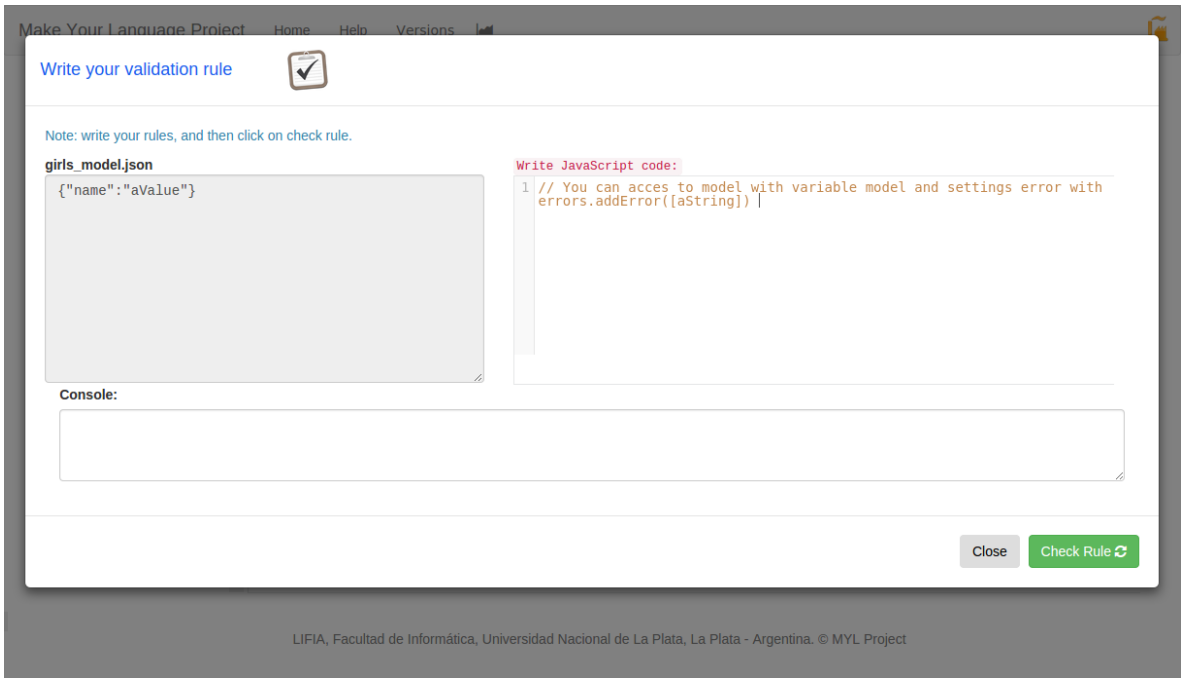


Figura 22 - Se ingresó a la sección de edición de las reglas de validación para el proyecto

4.2.6 Templating y procesamiento de modelos

Durante el contexto de esta tesis, las plantillas son utilizadas para abstraer un patrón de diseño en una o más plantillas, para generalizar una plantilla "base" que luego servirá para generar código a partir de las mismas.

Un ejemplo sencillo de plantilla podría ser el siguiente, más adelante se verá con más detalle el sistema de planillas adoptado por la herramienta de soporte.

```
<h1>{{headerTitle}}</h1>
{{#items}}
  <li><strong>{{name}}</strong></li>
  <li><a href="{{url}}">{{name}}</a></li>
{{/items}}
```

4.2.6.1 Template engine

Un template engine [19], también conocido como template processor o template parser, es un componente de software diseñado para combinar una o más plantillas (templates) con un modelo de datos para producir uno o más documentos como resultado, es decir es el encargado de procesar los templates y administrar las etiquetas contenidas en los mismos. En la siguiente Figura 23 se puede apreciar el funcionamiento de un template engine para llevar a cabo la renderización de archivos.

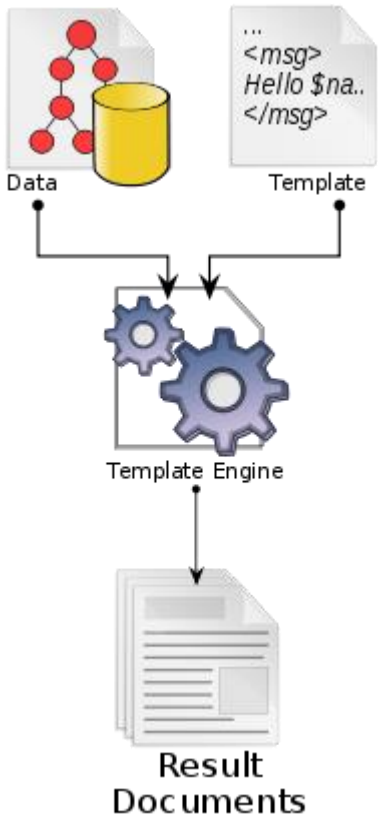


Figura 23 - Renderizado - template engine

MYL utiliza el template engine Mustache [21], Mustache está disponible para operar en una amplia gama de lenguajes como: Ruby, JavaScript, Python, Erlang, node.js, PHP, Perl, Perl6, Objective-C, Java, .NET, Android, C++, Go, Lua, ooc, ActionScript, ColdFusion, Scala, Clojure, Fantom, CoffeeScript, D, Haskell, XQuery, ASP, Io, Dart, Haxe, Delphi, Racket, Rust, C#, OCaml, Swift, y en Bash.

Mustache se puede utilizar para HTML, archivos de configuración, código fuente. Funciona mediante la ampliación de las etiquetas en una plantilla utilizando valores proporcionados en un hash o un objeto.

Los fundadores de Mustache lo mencionan como "menos-lógica" porque no existe lógica extra para administrar bucles ni cláusulas personalizables. En lugar de ello sólo hay etiquetas. Las etiquetas se reemplazan con un valor que haga match en un archivo JSON, si no se encuentra una llave que haga un match con alguna etiqueta, mustache ignora las etiquetas y no coloca nada en ellas.

4.2.6.2 Renderización de archivos

MYL ofrece al usuario la posibilidad de generar una nueva renderización del proyecto o archivo que desee, presionando en el botón Generate Render, donde se visualizará una pantalla como la que se demuestra en la Figura 24 correspondiente a la Fábrica de Renderizado.

La renderización consta en tomar el/los template/s anotados con la sintaxis para la creación de plantillas y procesarlos con un modelo JSON elegido. Más adelante cuando se explique la sintaxis para la creación de plantillas se detallará mejor el proceso de renderizado.

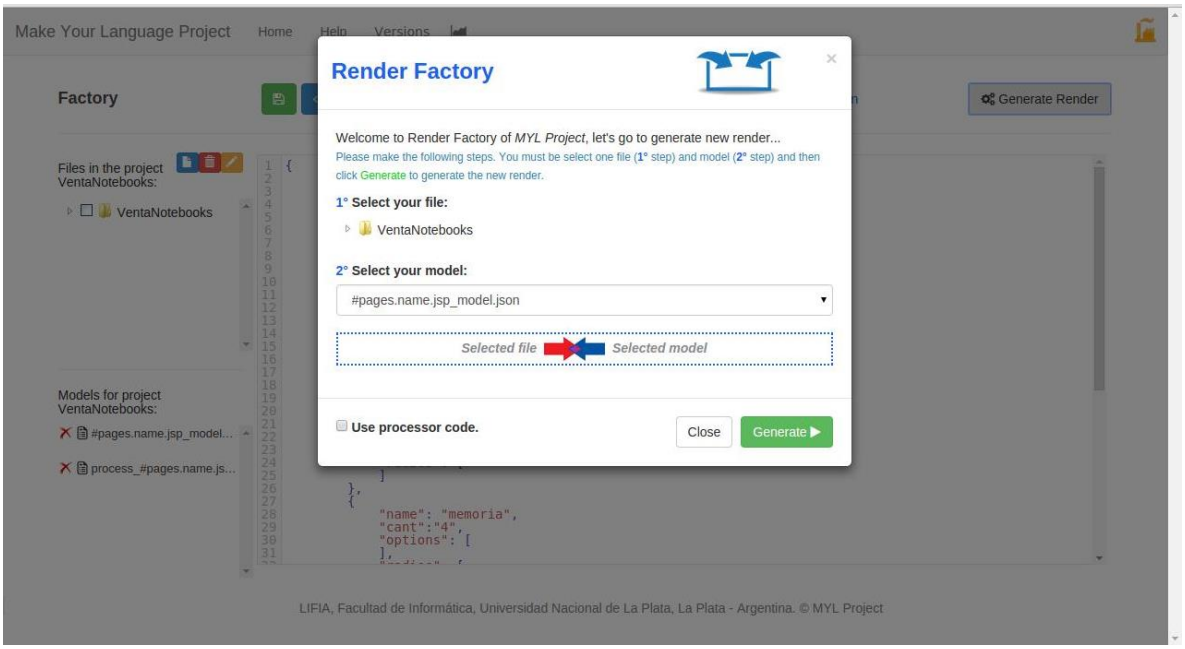


Figura 24 - Se ingresó a la sección Render Factory

Una vez renderizado el/los archivos se visualiza al usuario una notificación indicando que la renderización se ha efectuado satisfactoriamente (Figura 25) en caso de ausencia de algún error, en la notificación el usuario puede acceder desde el mismo diálogo a la sección history donde se visualizarán los resultados de la renderización realizada y las antiguas en caso que existan.

Dentro de MYL, el método `renderTemplate()`, se encarga de renderizar un modelo en formato JSON y un template.

```
public Object renderTemplate(String pathFile, String jsonStr) throws Exception {
      InputStream template = this.mylService.getContentsOfResource(1,pathFile);
      return new
MustacheTemplate(IOUtils.toString(template)).jsonRender(jsonStr);
}
```

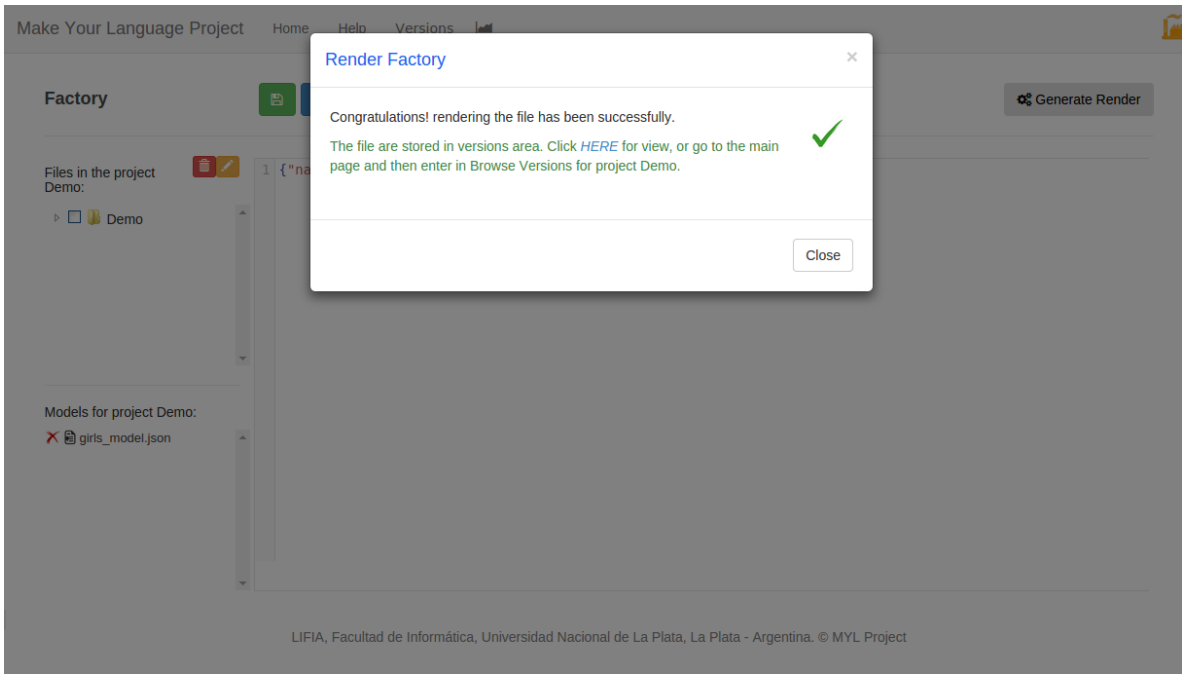


Figura 25 - Se realizó una nueva renderización y ha sido exitosa

4.2.7 Descarga e historial de archivos generados

En la sección History, el usuario podrá ver todos los archivos del proyecto que han sido renderizados y optar por descargar cada uno en forma individual o simplemente descargar toda la carpeta (en formato .zip) la cual contiene todas las renderizaciones y modelos realizados.

Para la descarga/exportación de archivos, el usuario debe hacer clic en el archivo deseado, el cual se visualizará en modo "solo lectura" el editor de texto de la derecha, una vez seleccionado el archivo presionando "Download" se le descargara el archivo propiamente seleccionado.

Si el usuario desea puede seleccionar la carpeta contenedora (nombrada con el nombre del proyecto junto con una subcadena "_MYL" al final), y luego presionar "Download" de esta manera descargara un archivo *.zip* donde están todos los archivos renderizados, modelo y códigos de validación y procesamiento declarados.

Como se puede visualizar en la Figura 26, se ha seleccionado el archivo girls.html el cual se despliega en el editor de su derecha para que el usuario pueda descargar oprimiendo el botón ubicado en la esquina superior derecha "Download".

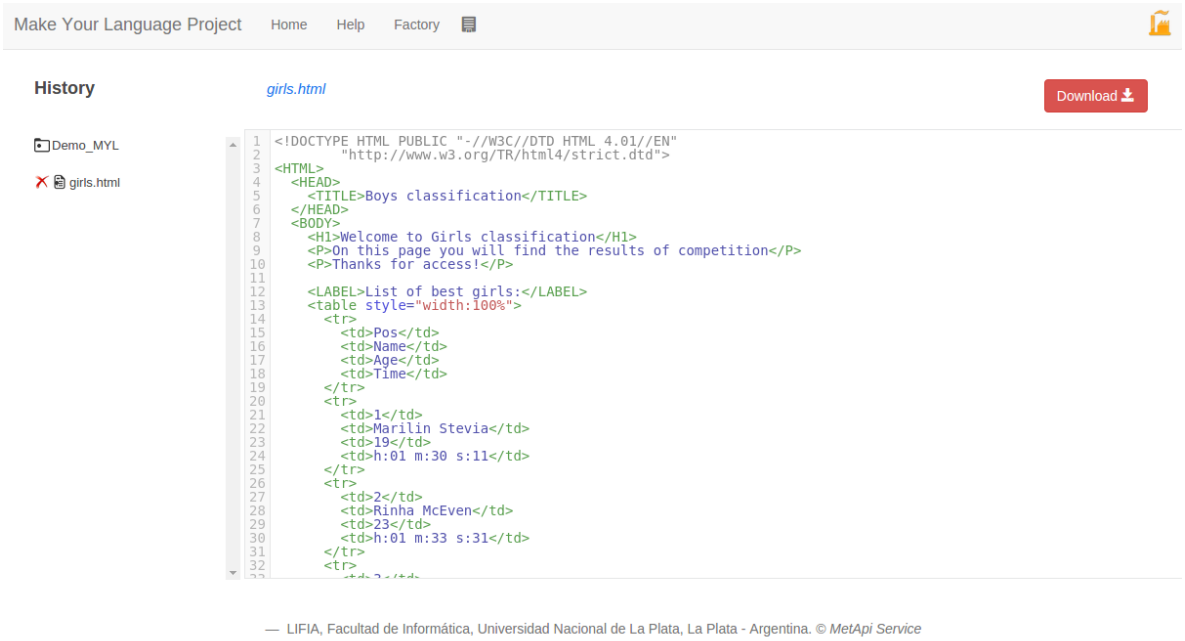


Figura 26 - Se ingresó a la sección History del proyecto Demo

4.2.8 Procesador de código

En la sección *processor* (procesador), Figura 27b, el sistema MYL ofrece un entorno, similar al de reglas de validación, para escribir código JavaScript, el cual se procesará previo a la renderización, este código es de gran utilidad debido a que permite que el usuario edite (procese) el modelo antes de la renderización, estos cambios que realice el processor, no afectarán al modelo inicial, es decir, se harán en runtime y solo el sistema podrá acceder a el modelo editado por el processor.

El usuario del sistema es el responsable de decidir si hacer uso de este código de procesamiento para su modelo al momento de renderizar. Se le presentará al usuario un check box (Figura 27a) donde se indicará si se debe utilizar o no el código para procesar el modelo previo a la renderización.

Al momento de renderizar, cuando se selecciona "*use processor code*", el sistema tomará como entrada el modelo JSON, aplicará el código de procesamiento en el mismo y enviará la salida al proceso de renderizado, una importante aclaración en este paso es mencionar que los cambios que realiza el código de procesamiento en el modelo entrante no serán reflejados en el modelo inicial, esto evita "ensuciar" el modelo JSON inicial y evita modelos estilo "verbose"[14].

De esta manera el usuario interactúa con un modelo simple y agrega en el todos los cambios que desee a través del código de procesamiento.

[14] Se denominan modelos verbose a aquellos modelos exageradamente descriptivos y extensos.

Figura 27 a - Seleccion de uso del código de procesamiento

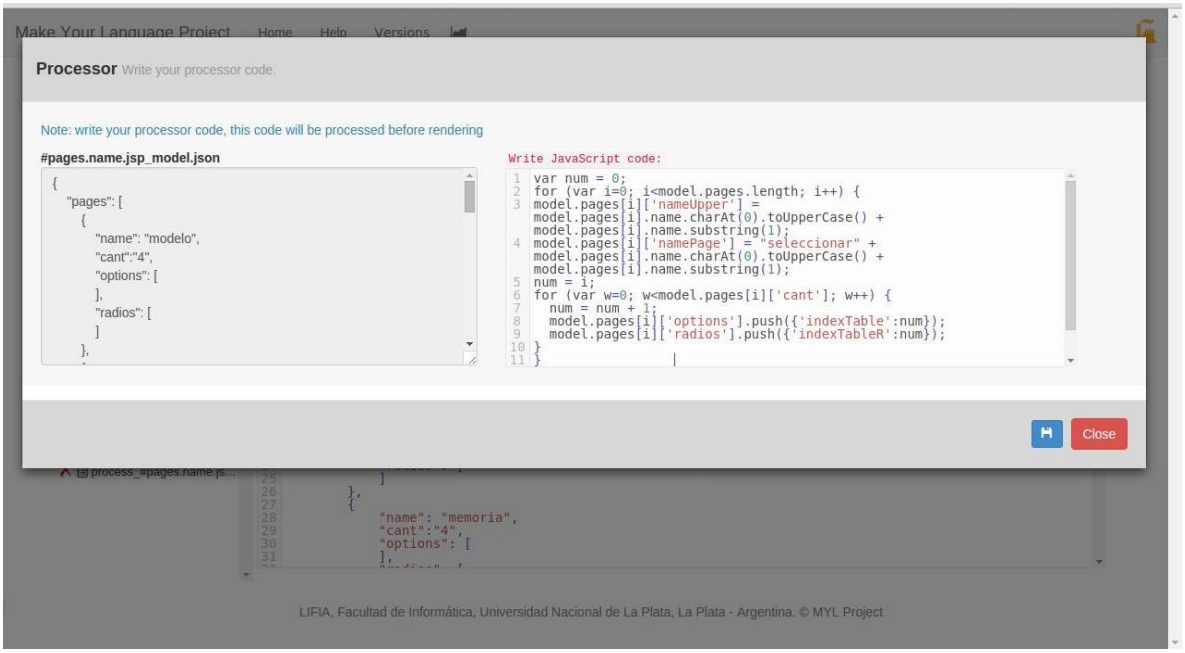


Figura 27b – Procesador de código

4.2.8.1 Edición y administración del código de procesamiento

El código de procesamiento es un archivo asignado a 1 proyecto, esto es, cada proyecto tiene asociado un código de procesamiento. El usuario puede editar el código según el/los template/s que vaya a renderizar. El entorno ofrece en la sección procesador de código, acceso a la variable pre-seteada para poder manipular el modelo, esta es ***model***.

La variable *model* se encuentra preseteada con el modelo, es decir que el usuario puede acceder y navegar el modelo JSON a través de la misma. El usuario podrá ir guardando

el código de procesamiento explícitamente oprimiendo el botón de save (botón azul con un icono de disquete), o se guardará automáticamente cuando el usuario finalice con la edición del código y presione Close, es decir el botón Close guarda los cambios producidos en el archivo de código de procesamiento.

4.2.9 Edición de nombre de archivo

MYL cuenta con la posibilidad de editar nombres de archivos, esta funcionalidad es exclusivamente para los archivos dentro del árbol de navegación de archivos. Esto es útil para la generación múltiple, ya que se puede editar el nombre de un archivo con la sintaxis necesaria para la generación múltiple. Figura 28.

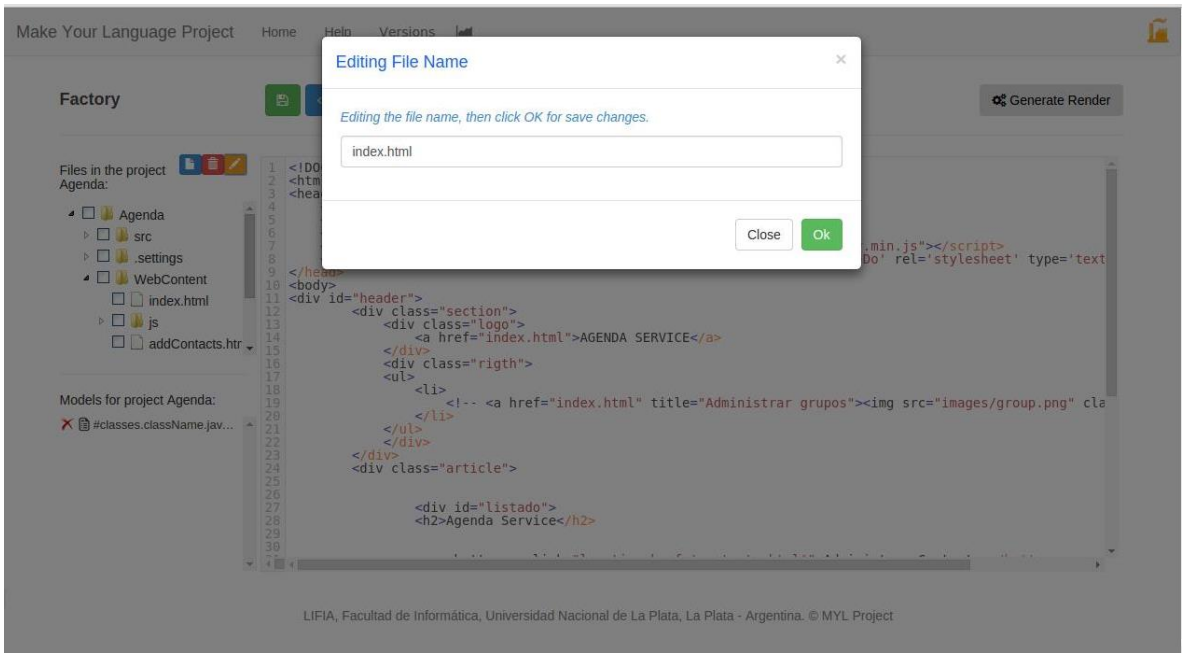


Figura 28 - Edición de nombre de archivo

4.2.10 Creación de un archivo

La herramienta cuenta con la funcionalidad de agregar un nuevo archivo, tanto en el sistema de archivos como en los modelos. Como refleja la figura el sistema despliega una pantalla donde se le da al usuario la posibilidad de optar por agregar el archivo en el filesystem del proyecto o agregarlo en la carpeta perteneciente a los modelos para el proyecto seleccionado, es decir un archivo JSON para redactar un modelo. Figura 29.

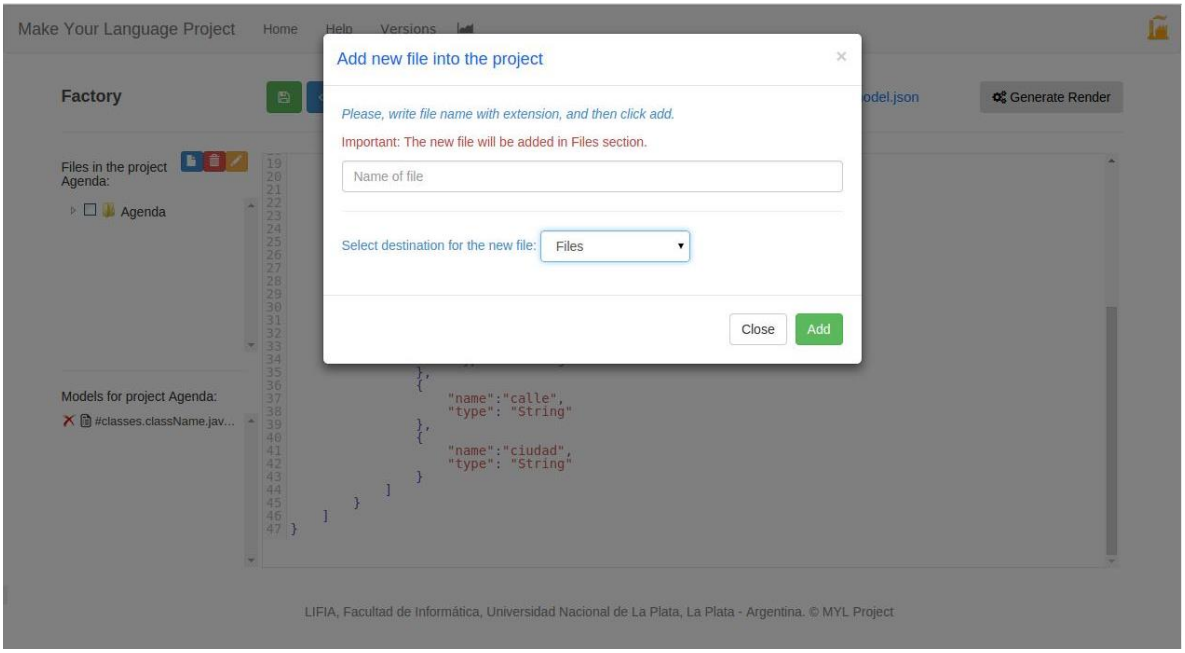


Figura 29 - Agregar un nuevo archivo

4.3 Sintaxis para la creación de templates

A continuación se describirán las sintaxis empleadas por la herramienta MYL para la creación de templates y la manipulación de los mismos.

Sintaxis para la creación de templates y renderización:

MYL, hace uso de Mustache, es el motor encargado de renderizar los templates anotados junto con los modelos escritos en JSON. Mustache ofrece una sintaxis muy simple y amigable para el usuario, administra etiquetas encerradas en doble llaves "{{" para abrir y "}}" para cerrar una etiqueta. Estas etiquetas, están admitidas por la herramienta MYL y especifican los datos que serán obtenidos del modelo JSON y serán incorporados al código resultado. Un ejemplo sencillo del uso de la sintaxis se puede ver en la renderización que se detalla continuación:

Template:

```
Hello {{name}}
You have just won {{value}} dollars!
```

Un modelo válido para el template descrito podría ser:

```
{
  "name": "Alan",
  "value": 10000
}
```

Como resultado de la renderización se obtiene:

```
Hello Alan
You have just won 10000 dollars!
```

MYL también admite “listas”, estas listas son etiquetas especiales heredadas de Mustache que difieren muy poco de las anteriormente descritas, solo se agrega el carácter “#” para la apertura y el carácter “/” para determinar el cierre de la lista. Estas etiquetas de lista permiten iterar sobre estas etiquetas y generar tantas instancias como llaves se encuentren en el modelo JSON, estas etiquetas poseen una sintaxis como la que se demuestra a continuación:

Template con una lista de etiquetas:

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
```

Un modelo JSON para este template podría ser:

```
{
  "repo": [
    { "name": "resque" },
    { "name": "hub" },
    { "name": "rip" }
  ]
}
```

El resultado de la renderización del template con el modelo JSON sería:

```
<b>resque</b>
<b>hub</b>
<b>rip</b>
```

Cabe destacar en este punto, que MYL posee la chance de decidir si una lista es tenida en cuenta al momento de la renderización o no, esto se determina con un atributo en el modelo JSON, donde se indica con valores Booleanos (true o false) si la lista será o no procesada al momento de renderizar. El atributo debe hacer match con el nombre de la etiqueta correspondiente a la lista sobre la cual se aplicará. Si el valor es “false” el template engine Mustache ignorará la existencia de la lista y pasará por alto la renderización de la misma, caso contrario (“true”) será tenida en cuenta durante el procesamiento. Un ejemplo sencillo se puede ver a continuación;

Template:

```
Shown.
{{#person}}
  Never shown!
{{/person}}
```

Modelo JSON para la renderización:

```
{
  "person": false
}
```

Resultado de la renderización:

```
Shown.
```

4.4 Sintaxis para la administración de templates y generación múltiple

Anteriormente se describió la sintaxis necesaria para la creación de las plantillas (templates), las etiquetas necesarias para editar el contenido interno de cada archivo al cual se le desee aplicar ADSM a través de la herramienta MYL. A continuación se describirá la sintaxis provista por MYL, desarrollada en el marco de este proyecto, para administrar la generación múltiple de archivos a partir de un template.

La generación múltiple permite a un usuario crear *N* instancias de código a partir de un mismo template, esto es de gran utilidad debido a que durante un proyecto se necesita implementar N archivos con una determinada estructura en la que solo difieren detalles menores que pueden ser abstraídos en una plantilla para luego hacer una renderización con un modelo JSON y generar N instancias en forma automática ahorrando un considerable tiempo de implementación. Para llevar a cabo la generación múltiple se debe renombrar la plantilla (template) con la sintaxis expresada en el diagrama (Figura 30) que se visualiza a continuación,

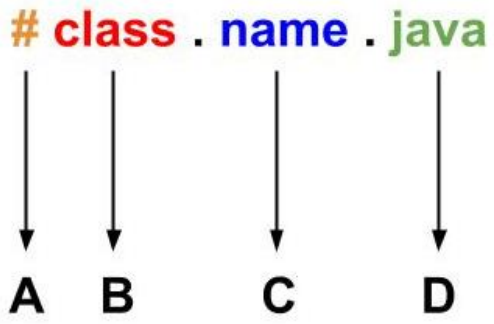


Figura 30 - Sintaxis para generación múltiple

Como se puede ver se trata de una cadena de palabras separadas por ".", cada sección que se visualiza posee una determinada función:

A. En el segmento **A**, debe ir el caracter "#", éste debe ser el primer elemento del nombre del archivo correspondiente a la plantilla. Indica a MYL que se generarán N instancias con el formato contenido en la plantilla.

B. El segmento **B**, está compuesto por un identificador, como mínimo de una letra, la cual indica a MYL que debe buscar en el modelo JSON un arreglo de N elementos nombrados con ese identificador.

C. El segmento **C** corresponde a el nombre de la llave que indica a MYL el nombre a colocar a los archivos resultantes de la renderización, en el modelo JSON MYL buscará dentro del arreglo "class" del segmento B, todos los elementos que hagan match con la llave "name" y generará por cada uno un archivo resultado con el nombre "name" y el contenido de la plantilla.

D. El segmento **D**, indica la extensión con la que serán generados los archivos resultados, en este caso de ejemplo se generarán N archivos con valores obtenidos del modelo JSON que hagan match con "name" y como extensión a estos archivos se les colocará "java".

Un ejemplo de generación múltiple haciendo uso de la sintaxis podría ser el siguiente:

Template `#classes.className.java`:

```
public class {{className}} {
	{{#vars}}
	private {{type}} {{name}};
	{{/vars}}

	public {{className}}({{#vars}} {{type}}
{{name}}{{#comma}},{{/comma}}{{/vars}}) {
		{{#vars}}
		this.{{name}} = {{name}};
		{{/vars}}
	}
}
```

El modelo para la generación múltiple seria: #classes.className.java_model.json

```
	{
	    "classes":[
	        {
	            "className": "Clase numero 1",
	            "vars": [
	                {
	                    "name":"Variable 1",
	                    "type": "Tipo variable 1"
	                },
	                {
	                    "name":"Variable 2",
	                    "type": "Tipo variable 2"
	                }
	            ]
	        },
	        {
	            "className": "Clase numero 2",
	            "vars": [
	                {
	                    "name":"Variable 1",
	                    "type": "Tipo variable 1"
	                },
	                ...
	            ]
```

```
        }
]
}
```

Como se puede apreciar se agrega un arreglo al modelo, el cual puede obtenerse infiriéndolo desde un template renombrado con la sintaxis para la generación múltiple o creándolo desde el inicio, como resultado MYL arrojara tantas clases como encuentre en el modelo cuya propiedad className está indicada.

4.5 Documentación herramienta MYL

4.5.1 Arquitectura de la herramienta MYL

La arquitectura de la herramienta MYL se encuentra establecida dentro del patrón MVC (Model View Controller). MVC separa la lógica de negocio (el modelo) y la presentación (la vista) por lo que se consigue un mantenimiento más sencillo de las aplicaciones.

El entorno MYL, fue desarrollado llevando a cabo la arquitectura MVC, a través del framework de aplicaciones Spring MVC. La arquitectura básica de Spring MVC está ilustrada en la siguiente Figura 31.

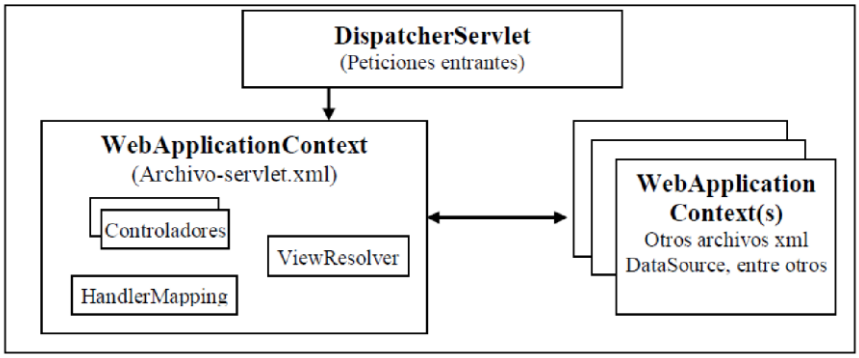


Figura 31 - Arquitectura básica Spring MVC

Ciclo de vida de un request en Spring MVC (figura 32)

1. El navegador manda un request y lo recibe un DispatcherServlet
2. Se debe escoger que Controller manejara el request, para esto el HandlerMapping mapea los diferentes patrones de URL hacia los controladores, y se le regresa al DispatcherServlet el Controller elegido.
3. El Contoller elegido toma el request y ejecuta la tarea
4. El Controller regresa un ModelAndView al DispatcherServlet
5. Si el ModelAndView contiene un nombre lógico de un View se tiene que utilizar un ViweResolver para buscar ese objeto View que representara el request modificado
6. Finalmente el DispatcherServlet despacha el request al View.

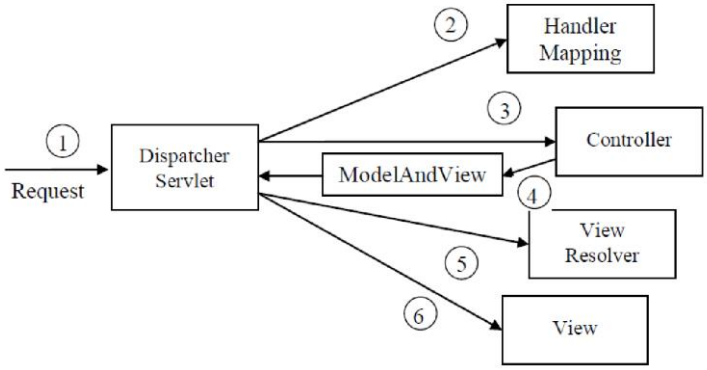


Figura 32 - Ciclo de vida de un request

En el dispatcher servlet es donde se procesan todas las peticiones entrantes (requests) para poder administrar cada pedido.

4.5.2 Diagrama de clases

El diagrama de clases de MYL (Figura 33), está compuesto por 6 clases, Project, ValidateRule ,ProcessCode, JSONModel, SimpleFile y RenderizedFile.

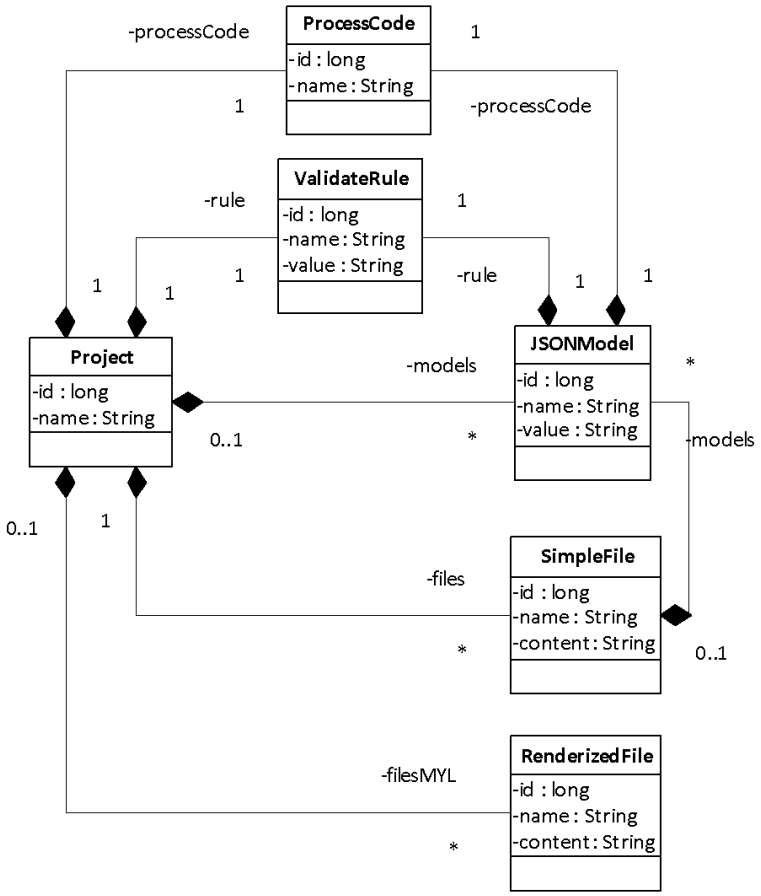


Figura 33 - Diagrama de clases

4.5.3 Diagramas de Casos de uso

A continuación se detallan los casos de uso de las funcionalidades de vital importancia para el sistema.

<table>
<tr><td>RF- 1</td><td colspan="2">Inferir un modelo a partir de un archivo</td></tr>
<tr><td>Versión</td><td colspan="2">1</td></tr>
<tr><td>Autores</td><td colspan="2">Usuario</td></tr>
<tr><td>Objetivos asociados</td><td colspan="2">Se infiere un modelo de ejemplo en formato JSON a partir de las anotaciones en el código fuente del archivo seleccionado.</td></tr>
<tr><td>Descripción</td><td colspan="2">El usuario presiona el botón “Infer model”, el sistema procesará el archivo seleccionado en busca de tags específicos de la sintaxis de la herramienta para crear un modelo de ejemplo en formato JSON</td></tr>
<tr><td>Precondición</td><td colspan="2">Se debe haber seleccionado un archivo del panel de navegación en forma de árbol.</td></tr>
<tr><td rowspan="6">Secuencia Normal</td><td>Paso</td><td>Acción</td></tr>
<tr><td>1</td><td>El usuario presiona el botón Infer model.</td></tr>
<tr><td>2</td><td>Si el usuario selecciona una carpeta del árbol de navegación de archivos, se realiza el caso de uso RF-2.</td></tr>
<tr><td>3</td><td>El sistema procesa el archivo seleccionado en busca de tags específicos de la herramienta para crear el modelo en formato JSON</td></tr>
<tr><td>4</td><td>El sistema crea el archivo correspondiente al modelo del archivo seleccionado</td></tr>
<tr><td>5</td><td>El sistema visualiza el modelo creado debajo del panel de navegación de archivos con el nombre del archivo agregando al final “_model.json”</td></tr>
<tr><td>Secuencia Anormal</td><td colspan="2">1- No se ha seleccionado un archivo del panel de navegación de archivos, el sistema informará un error.
2- El archivo a procesar no contiene tags especiales de la herramienta, el sistema devolverá un modelo ejemplo vacío donde incluirá las llaves {} correspondientes a cualquier archivo JSON sin contenido</td></tr>
<tr><td>Postcondicion</td><td colspan="2">El sistema creó un modelo para el archivo seleccionado con el nombre del archivo y agregándose al final “_model.json” y se visualiza debajo del panel de navegación de archivos.</td></tr>
<tr><td>Importancia</td><td colspan="2">Vital</td></tr>
<tr><td>Comentarios</td><td colspan="2">Inferir un modelo es la manera de crear un modelo, el sistema creará un modelo para el archivo seleccionado con el nombre del archivo y agregándose al final “_model.json”, una vez creado el modelo el usuario es libre de editar su contenido e incluso eliminar el modelo.</td></tr>
</table>

RF- 2	Inferir un modelo a partir de la raíz del proyecto	
Versión	1	
Autores	Usuario	
Objetivos asociados	Se infiere un modelo de ejemplo en formato JSON a partir de las anotaciones en el contenido de cada archivo contenido en todo el proyecto.	
Descripción	El usuario presiona el botón “Infer model”, el sistema procesará todos los archivos del proyecto en busca de tags específicos para generar un modelo general del proyecto a partir de la raíz para crear un modelo de ejemplo en formato JSON	
Precondición	Se debe seleccionar la carpeta raíz o cualquier carpeta del árbol de navegación de archivos.	
Secuencia Normal	Paso	Acción
	1	El actor presiona el botón Infer model
	2	El sistema procesará cada archivo soportado por la herramienta en busca de tags específicos para crear el modelo general en formato JSON
	3	El sistema crea el archivo correspondiente al modelo general del proyecto
	4	El sistema visualiza el modelo creado debajo del panel de navegación de archivos con el nombre del proyecto agregando al final “_model.json”
Secuencia Anormal	**1-** No se ha seleccionado una carpeta del panel de navegación de archivos, el sistema informará un error. **2-** El archivo a procesar no contiene tags especiales de la herramienta, el sistema devolverá un modelo ejemplo vacío donde incluirá las llaves {} correspondientes a cualquier archivo JSON sin contenido	
Postcondición	El sistema creó un modelo para el proyecto con el nombre del proyecto y agregándose al final “_model.json”	
Importancia	Vital	
Comentarios	Inferir un modelo es la manera de crear un modelo, el sistema creará un modelo para el proyecto con el nombre del proyecto y agregándose al final “_model.json”	

RF- 3	Generar una renderización de un archivo
Versión	1
Autores	Usuario
Objetivos asociados	Se genera una nueva renderización para el archivo seleccionado junto con el modelo seleccionado
Descripción	El usuario presiona el botón “Generate Render”, el sistema desplegará un diálogo donde se le da al usuario la posibilidad de elegir un archivo del panel de navegación de archivos en forma de árbol y un modelo en formato JSON de un campo desplegable para luego presionar el botón “Generate” y crear la nueva

	renderización.	
Precondición	Se debe seleccionar un archivo del árbol de navegación de archivos. Se debe seleccionar un modelo en formato JSON del campo desplegable de selección de modelo	
Secuencia Normal	Paso	Acción
	1	El actor presiona el botón Generate Render
	2	El sistema despliega un diálogo donde brindará la opción para elegir un archivo del árbol de navegación de archivos y un campo de selección para seleccionar un modelo en formato JSON existente.
	3	El usuario selecciona el archivo del menú de navegación de archivos en forma de árbol
	4	Si el usuario selecciona una carpeta del árbol de navegación de archivos, se realiza el caso de uso RF-4.
	5	El sistema visualiza el nombre del archivo seleccionado en color rojo en un recuadro ubicado en la parte inferior del diálogo
	6	El usuario selecciona un modelo en formato JSON del campo de selección
	7	El sistema visualiza el nombre del modelo seleccionado en color azul en un recuadro ubicado en la parte inferior del diálogo
	8	El usuario presiona Generate
	9	Si el nombre del archivo posee una sintaxis específica de la herramienta se realiza el caso de uso RF-5.
	10	El sistema procesa la información y genera un archivo renderizado con el nombre del archivo y respetándose la extensión del archivo original.
	11	El sistema almacena el archivo renderizado que se podrá ver en la sección History
	12	El sistema visualiza un diálogo de éxito
Secuencia Anormal	**1-** No se ha seleccionado un archivo del panel de navegación de archivos. **2-** No se ha seleccionado un modelo del campo de selección de modelos en formato JSON. **3-** El archivo a procesar no contiene tags especiales de la herramienta que se encuentren especificados en forma de llaves en el modelo JSON. El sistema ignorará la llave dejando en blanco el tag a completar a partir del modelo.	
Postcondición	El sistema creó un archivo de una nueva renderización y l almaceno en la sección History	
Importancia	Vital	
Comentarios	Se genera una renderización a partir de un archivo anotado con los tags especiales de la herramienta y un modelo en formato	

	JSON con valores deseables a incorporar en el archivo de resultado final.

RF- 4	Generar una renderización del proyecto	
Versión	1	
Autores	Usuario	
Objetivos asociados	Se genera una nueva renderización para el proyecto junto con el modelo seleccionado	
Descripción	El usuario presiona el botón "Generate Render", el sistema desplegará un diálogo donde se le da al usuario la posibilidad de elegir un archivo del panel de navegación de archivos en forma de árbol y un modelo en formato JSON de un campo desplegable para luego presionar el botón "Generate" y crear la nueva renderización.	
Precondición	Se debe seleccionar la carpeta raíz o cualquier carpeta del árbol de navegación de archivos. Se debe seleccionar un modelo en formato JSON del campo desplegable de selección de modelo	
Secuencia Normal	Paso	Acción
	1	El actor presiona el botón Generate Render
	2	El sistema despliega un diálogo donde brindará la opción para elegir una carpeta del árbol de navegación de archivos y un campo de selección para seleccionar un modelo en formato JSON existente.
	3	El usuario selecciona la carpeta del menú de navegación de archivos en forma de árbol
	4	El sistema visualiza el nombre de la carpeta seleccionado en color rojo en un recuadro ubicado en la parte inferior del diálogo
	5	El usuario selecciona un modelo en formato JSON del campo de selección
	7	El sistema visualiza el nombre del modelo seleccionado en color azul en un recuadro ubicado en la parte inferior del diálogo
	8	El usuario presiona Generate
	9	Si el nombre del archivo posee una sintaxis específica de la herramienta se realiza el caso de uso RF-5.
	10	El sistema procesa la información y genera un archivo renderizado con el nombre del proyecto.
	11	El sistema almacena el archivo renderizado que se podrá ver en la sección History
	12	El sistema visualiza un diálogo de éxito
Secuencia Anormal	**1-** No se ha seleccionado una carpeta del panel de navegación de archivos. **2-** No se ha seleccionado un modelo del campo de selección de modelos en formato JSON.	

	3- Ningún archivo del proyecto a procesar contiene tags especiales de la herramienta que se encuentren especificados en forma de llaves en el modelo JSON. El sistema ignorará la llave dejando en blanco el tag a completar a partir del modelo.
Postcondición	El sistema creó un archivo de una nueva renderización y se almaceno en la sección History
Importancia	Vital
Comentarios	Se genera una renderización del proyecto.

RF- 5	[RF-3 | RF-4] <<extends>> Procesar sintaxis especial de la herramienta	
Versión	1	
Autores		
Objetivos asociados	Se procesa la sintaxis especial de la herramienta	
Descripción	El sistema procesa la sintaxis especial	
Precondición	Se debe editar el nombre del archivo o carpeta con la sintaxis especial de la herramienta	
Secuencia Normal	Paso	Acción
	1	El sistema procesa el archivo en busca de tags para crear un modelo en formato JSON
Secuencia Anormal	1- No se encuentra la sintaxis especial en formato correcto	
Postcondición	Se generó un modelo con un arreglo dentro donde se especifican los tags encontrados en el archivo	
Importancia	Importante	
Comentarios		

4.6 Código fuente

El código fuente de MYL se encuentra centralizado en el sistema de control de revisiones Git como un módulo de Metapi y se puede acceder a través de https://goo.gl/Db0uUA .

4.7 Instrumentos y tecnologías utilizadas

Durante el desarrollo de la herramienta se utilizaron múltiples tecnologías y frameworks que hicieron posible el resultado final. Es relevante mencionar las características principales de algunas de ellas, ya que fueron su motivo de elección para abordar este proyecto.

Dentro de las tecnologías más importantes se encuentran:

Maven: Durante todo el desarrollo de la herramienta MYL se utilizó Maven de la organización Apache. Maven significa acumulación de conocimiento y es de utilidad en la gestión de proyectos de software basados en el lenguaje Java. Se basa en el concepto de Modelo de Objetos (POM), puede administrar proyectos, reportes y documentación. Es de fácil extensión mediante el uso de plugins. Maven fue de utilidad para:

- Manejo de artefactos y dependencias mediante el uso de repositorios.
- Ejecución de tests automáticos.
- Garantizar la cobertura de los tests.
- Garantizar la uniformidad y documentación del código fuente mediante la definición de políticas.
- Búsqueda de defectos comunes.
- Generar informes sobre todos los puntos anteriores.

Java™ SE Development Kit 8, Update 5 (JDK 8u5):

Como lenguaje de programación, he elegido al lenguaje Java. No hay demasiado para explicar sobre el mismo, es ampliamente utilizado por toda la comunidad.

Se escogió porque se buscaba un lenguaje que cumpla con los siguientes requisitos:

- Que el uso del lenguaje sea gratuito y preferentemente, con librerías de código abierto.
- Lograr que el software sea multiplataforma, que pudiese correr tanto en Windows como en Linux. Esto se logra gracias a la máquina virtual de java sin mayores problemas.
- Contar con un IDE gratuito para utilizar durante el desarrollo. Hay varios IDE para desarrollar en java. Los más utilizados son NetBeans y Eclipse, ambos son gratuitos y cuentan con numerosos plugins que podremos utilizar.
- Contar con herramientas gratuitas para soportar testing unitario y mocking. Hay numerosas librerías para java de índole gratuito.
- Contar con herramientas, también gratuitas, para dar soporte a la integración contínua. Para esto también existen muchas herramientas con plugins gratuitos que servirán a distintos fines.

El lenguaje Java cumple con todos los requisitos enumerados, por lo que pensamos que es el ideal para ser utilizado en este proyecto. No se puede hablar de otra alternativa, porque ninguna conocida cumple con todos los puntos anteriores.

SpringMVC (Release 3.2.5):

Un framework para el desarrollo de aplicaciones y contenedor de inversión de control, de código abierto para la plataforma Java.

Spring brinda un MVC (Model View Controller) para web bastante flexible y altamente configurable, pero esta flexibilidad no le quita sencillez, ya que se pueden desarrollar aplicaciones sencillas sin tener que configurar muchas opciones. Para esto se puede utilizar muchas tecnologías ya que Spring brinda soporte para JSP, Struts, Velocity, entre otros. El Web MVC de Spring presenta algunas similitudes con otros frameworks para web que existen en el mercado, pero son algunas características que lo vuelven único:

- Spring hace una clara división entre controladores, modelos de JavaBeans y vistas

- El MVC de Spring está basado en interfaces y e bastante flexible
- Provee interceptores (interceptors) al igual que controladores.
- Spring no obliga a utilizar JSP como una tecnología View también se puede utilizar otras.

- Los Controladores son configurados de la misma manera que los demás objetos en Spring, a través de IoC.
- Los web tiers son más sencillos de probar que en otros frameworks
- El web tiers se vuelve una pequeña capa delgada que se encuentra encima de la capa business objects.

Jquery:

jQuery es una biblioteca liviana de JavaScript, pensada para interactuar con los elementos de una web por medio del DOM (Document Object Model - Modelo en Objetos para la representación de

Documentos es una interfaz de programación de aplicaciones para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos con lenguajes como HTML, XML, etc). Lo que la hace tan especial es su sencillez y su reducido tamaño. Esta librería permite enriquecer estéticamente una página web.

jQuery es software libre y de código abierto, posee un doble licenciamiento bajo la Licencia MIT y la Licencia Pública General de GNU v2, jQuery, al igual que otras bibliotecas, ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código, es decir, con las funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio.

Características de JQuery:

- jQuery es software libre y de código abierto, permitiendo su uso en proyectos libres y privados.
- jQuery ofrece una serie de funcionalidades basadas en JavaScript, permitiendo funciones que logran grandes resultados en menos tiempo y espacio.
- Permite la selección de elementos DOM.
- Permite la interactividad y modificaciones del árbol DOM, incluyendo soporte para CSS 1-3 y un plugin básico de XPath.
- Maneja eventos.
- Manipulación de la hoja de estilos CSS.
- Permite la generación de efectos y animaciones.
- Compatible con los navegadores Mozilla Firefox 2.0+, Internet Explorer 6+, Safari 3+, Opera 10.6+ y Google Chrome 8+.

Jersey:

Java API for RESTful Web Services es una API del lenguaje de programación Java que proporciona soporte en la creación de servicios web de acuerdo con el estilo arquitectónico Representational State Transfer (REST). JAX-RS usa anotaciones, introducidas en Java SE 5, para simplificar el desarrollo y despliegue de los clientes y puntos finales de los servicios web.

CSS3: Hojas de Estilo en Cascada (Cascading Style Sheets); es un mecanismo para dar estilo a documentos HTML y XML, que consiste en reglas simples a través de las cuales se establece cómo se va a mostrar un documento en la pantalla, o cómo se va a imprimir, o incluso cómo va

a ser pronunciada la información presente en ese documento a través de un dispositivo de lectura.

Esta forma de descripción de estilos ofrece a los desarrolladores control total sobre el estilo y formato de sus documentos separando contenido y presentación.

JavaScript: Es un lenguaje de programación basado en objetos con muchas posibilidades, utilizado para crear pequeños programas que luego son insertados en una página web y en programas más grandes.

Este lenguaje posee varias características, entre ellas podemos mencionar que es un lenguaje basado en acciones que posee menos restricciones. Además, es un lenguaje que utiliza Windows y sistemas X-Windows, gran parte de la programación en este lenguaje está centrada en describir objetos y escribir funciones que respondan a los movimientos del mouse, aperturas, utilización de teclas, cargas de página, utilizando DOM2 dinámico. Técnicamente, es un lenguaje de programación interpretado, porque no es necesario compilar los programas para ejecutarlos, ya que sus fuentes se ejecutan en todos los navegadores sin necesidad de procesos intermedios.

- DOM: Modelo de Objeto Documento, para la interacción y manipulación dinámica de la presentación, que surgió de la necesidad de procesar manipular el contenido de los archivos XML. También Se utiliza para crear y diseñar especificaciones y etiquetas HTML de una manera más rápida y eficiente.
- XHTML: Lenguaje de marcado de hipertexto extensible, similar a HTML, con características que lo hacen más robusto, básicamente es HTML expresado como XML válido.

Ajax (Tecnología):

Es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (rich internet applications). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano.

Twitter Bootstrap:

Es una colección de herramientas de software libre para la creación de sitios y aplicaciones web dinámicas y con altos estándares de calidad. Contiene plantillas de diseño basadas en HTML5 y CSS 3 con tipografías, formularios, botones, gráficos, barras de navegación y demás componentes de interfaz, así como extensiones opcionales de JavaScript.

También se utilizaron durante el desarrollo de la herramienta, el entorno IDE EclipseIDE, Jetty Server y el repositorio de archivos GitHub a través de BitBucket.

5. Capítulo 5: Casos de estudio, estadísticas y análisis

5.1 Introducción

Durante éste capítulo se describirán los casos de estudios que fueron llevados a cabo durante la etapa de desarrollo de esta tesis. Se optó por realizar casos de estudios demostrativos, donde se pueda apreciar la optimización de tiempo haciendo comparación del uso de la metodología tradicional de codificación manual y la metodología desarrollada en el marco de esta tesis ADSM.

5.2 Casos de estudio

Durante el proyecto de tesis, se llevaron a cabo casos de estudios, donde formaron parte desarrollos codificados manualmente y luego se realizó el mismo desarrollo utilizando la metodología ADSM a través de la herramienta *Make Your Language*.

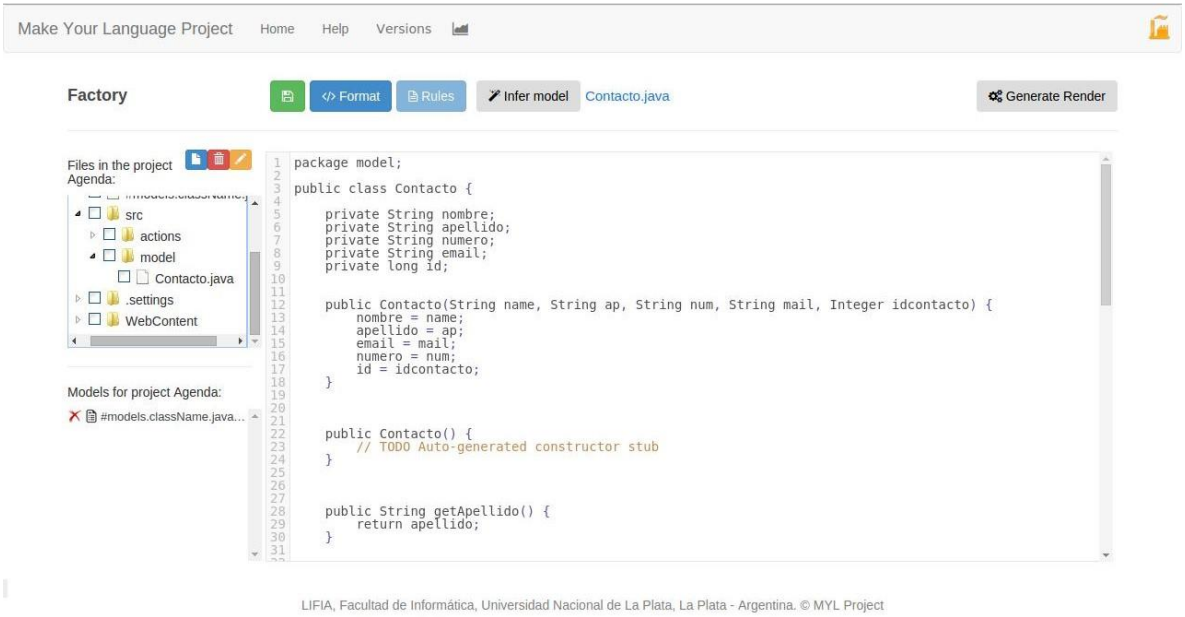
Se tomaron desarrollos de proyectos implementados durante la carrera de Licenciatura en diferentes cátedras.

Para llevar a cabo la medición del tiempo se utilizó la aplicación "cronómetro-online", disponible en: http://cronometro-online.chronme.com.

El inicio de medición comenzó desarrollando código utilizando la metodología tradicional de codificar manualmente y se fueron cronometrando los tiempos transcurridos. Una vez finalizada la codificación manual de los archivos, se procedió a desarrollar los mismos archivos aplicando la metodología ADSM, haciendo uso de la herramienta Make Your Language.

5.2.1 Caso de estudio N° 1 - Agenda

El primer caso de estudio que se consideró fue un proyecto llamado Agenda. Agenda es un simple proyecto desarrollado en lenguaje Java que consta de administración básica de contactos, como se puede ver en la imagen, posee un modelo inicial muy sencillo compuesto solo por la clase Contacto, el cual se extenderá a través de ADSM en varias iteraciones.

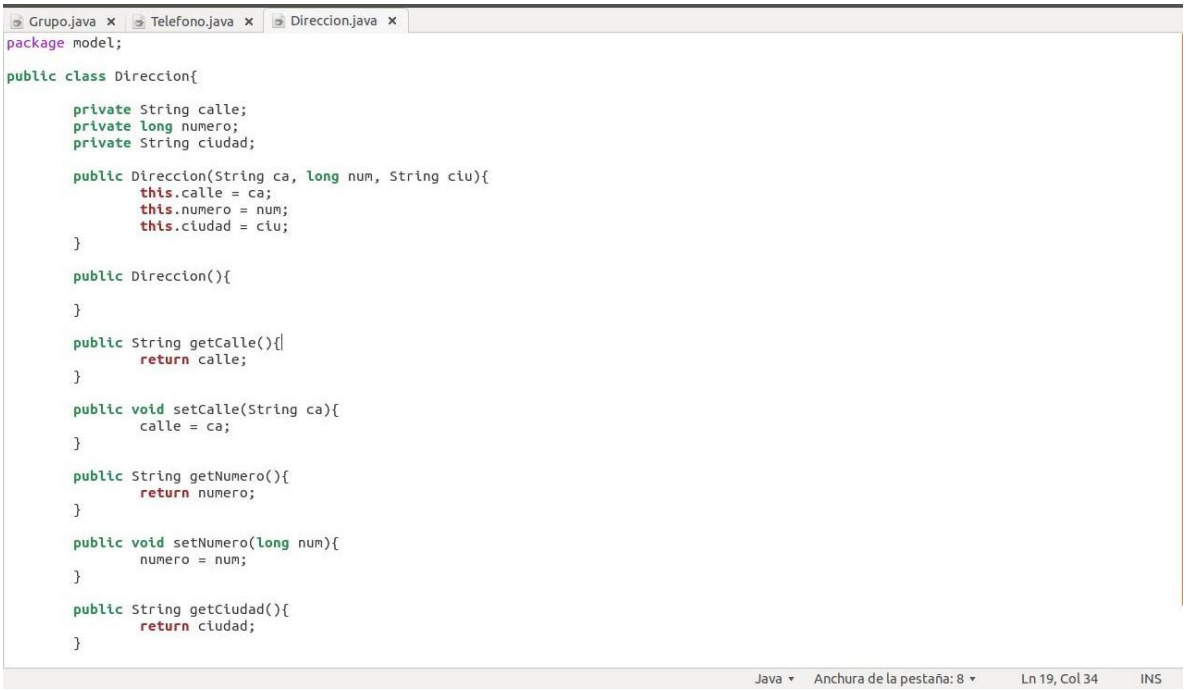


Se pretende extender el modelo de datos de Agenda agregando las clases Grupo, Teléfono y Dirección.

Adición de la clase Grupo, Teléfono y Dirección codificada manualmente:

1 – Para agregar la clase Grupo, Teléfono y Dirección al proyecto Agenda se comenzó diseñando e implementando las clases en el package llamado “model” en Java.
Tiempo insumido en la codificación: 17 minutos.

En el resultado de la codificación manual, se registraron 3 archivos codificados manualmente.
Código resultante de la codificación manual:



Adición de la clase Grupo, Teléfono y Dirección codificada con ADSM en MYL:

1 – Para agregar la clase Grupo, Teléfono y Dirección al proyecto Agenda utilizando la metodología Agile DSM a través de Make Your Language, se comenzó detectando patrones repetitivos entre la clase existente "Contacto" y las codificadas manualmente "Grupo", "Teléfono" y "Dirección" generándose el template metamodelo con nombre "#models.className.java" para la generación automática de clases del modelo.

```
package model;

public class Contacto {

	private String nombre;
	private String apellido;
	private String numero;
	private String email;
	private long id;

	public Contacto(String nombre, String apellido, String numero, String
email, Integer id) {
		nombre = nombre;
		apellido = apellido;
		email = email;
		numero = numero;
		id = id;
	}

	public Contacto() {

	}
	public String getApellido() {
		return apellido;
	}

	public void setApellido(String apellido) {
		this.apellido = apellido;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public long getId() {
		return id;
	}

	public void setId(long id) {
		this.id = id;
```

```
	}

	public String getNumero() {
		return numero;
	}

	public void setNumero(String numero) {
		this.numero = numero;
	}
…
// clase Grupo, Telefono y Direccion
```

Una vez detectado el patrón repetitivo en el código, se procede a la creación del template para abstraer este patrón detectado y automatizar el proceso de desarrollo de las mismas clases haciendo uso de ADSM.

El template se guardó utilizando la sintaxis propia de MYL #classes.className.java para generación múltiple, esto es necesario ya que a partir del template se generarán las clases Grupo, Telefono y Direccion.

Template generado:

```
package model;

import model.{{className}};
import java.util.*;

public class {{className}} {

	{{#vars}}
	private {{type}} {{name}};
	{{/vars}}

	public {{className}}({{#vars}} {{type}}
{{name}}{{#comma}},{{/comma}}{{/vars}}) {
		{{#vars}}
		this.{{name}} = {{name}};
		{{/vars}}
	}

	{{#vars}}
	public {{type}} get{{nameUpper}}() {
		return {{name}};
	}
	{{/vars}}

	{{#vars}}
	public void set{{nameUpper}}({{type}} {{name}}) {
		this.{{name}} = {{name}};
	}
	{{/vars}}

}
```

Tiempo insumido en detectar el patrón de código y generación del template metamodelo: 3 minutos

2 – Una vez detectado el patrón y creado el template se procede con la creación del modelo en formato JSON haciendo uso de la funcionalidad de inferir automáticamente el modelo necesario para la generación automática de las clases “Grupo”, “Teléfono” y “Dirección”.

Modelo: #classes.className.java_model.json

```
{
    "classes": [
        {
            "className": "Grupo",
            "vars": [
                {"name":"nombre","type": "String"},
                {"name":"contactos","type": "Set<Contacto>"}
            ]
        },
        {
            "className": "Telefono",
            "vars": [
                 {"name":"numero","type": "String"},
                {"name":"cmpania","type": "String"}
            ]
        },
        {
            "className": "Direccion",
            "vars": [
                {"name":"numero","type": "String"},
                {"name":"calle","type": "String"},
                {"name":"ciudad","type": "String"
                }
            ]
        }
    ]
}
```

Tiempo insumido en generar y completar el modelo: 2 minutos

3 – Luego se escribió el código de procesamiento para procesar el modelo previo a la renderización, en este código lo que se hace es recorrer el modelo JSON y setear una propiedad nueva “nameUpper” con el valor que se encuentra en el atributo name cambiando la primer letra para llevarla a mayúscula. También se verifica si un método posee más de 1 parámetro y se setea la coma correspondiente en el área de declaración de parámetros de los métodos setter, el código de procesamiento es:

```
for(var i=0; i<model.classes.length; i++) {
 for (var w=0; w<model.classes[i].vars.length; w++) {
       model.classes[i].vars[w]['nameUpper'] =
model.classes[i].vars[w].name.charAt(0).toUpperCase() +
model.classes[i].vars[w].name.substring(1);
       if(model.classes[i].vars.length > 1 && w< (model.classes[i].vars.length -
1)){
        model.classes[i].vars[w]['comma'] = true;
   }
 }
}
```

Tiempo insumido en realizar el código de procesamiento: 3 minutos

4 – El último paso fue la generación, teniendo el template metamodelo y el modelo en formato JSON, se procedió a la generación de la clase “Grupo”, “Teléfono” y “Dirección” en forma automática.

Tiempo insumido en generar las clases “Grupo”, “Teléfono” y “Dirección”: inmediata

Código resultante de la generación automática:

```
package model;
import model.Contacto;
import java.util.*;

public class Direccion {

       private String numero;
       private String calle;
       private String ciudad;

       public Direccion( String numero, String calle, String ciudad) {
           this.numero = numero;
           this.calle = calle;
           this.ciudad = ciudad;
       }

       public String getNumero() {
           return numero;
       }
       public String getCalle() {
           return calle;
       }
       public String getCiudad() {
           return ciudad;
       }

       public void setNumero(String numero) {
              this.numero = numero;
       }
       public void setCalle(String calle) {
```

```
            this.calle = calle;
        }
        public void setCiudad(String ciudad) {
            this.ciudad = ciudad;
        }

}
…
// Clases Grupo y Telefono
```

Análisis y estadísticas

Con respecto al caso de estudio se pudo observar que el tiempo de codificación de las clases “Grupo”, “Teléfono” y “Dirección” en forma manual y tradicional fue de 17 minutos.

Realizando el desarrollo de las clases haciendo uso de la metodología ADSM a través de la herramienta Make Your Language, llevo el tiempo en detectar y diseñar el template metamodelo sumado al tiempo insumido al ingresar datos en el modelo JSON, esto es:

Tiempo insumido en detectar el patrón de código para generar el template: 3 minutos

Tiempo insumido en generar y completar el modelo: 2 minutos

Tiempo insumido en crear el código de procesamiento: 3 minutos

Un total de tiempo de: 00:08

Cuadro de estadísticas

Codificación tradicional en forma manual:

Tarea	Tiempo invertido
Codificar clase Grupo, Teléfono y Dirección	17 minutos

Codificación haciendo uso de ADSM a través de MYL:

Implementación realizada	Tiempo insumido en minutos
Detectar patrón y generar template	3 minutos
Generar y completar modelo JSON	2 minutos
Crear código de procesamiento para el modelo	3 minutos
Generar código resultado	inmediato
Total	8 minutos

Comparando las 2 metodologías empleadas se puede visualizar que al utilizar Agile DSM se optimiza un considerable tiempo ya que la generación automática en permite obtener los mismos resultados que la codificación manual en un tiempo mucho inferior a la metodología tradicional de codificación manual.

5.2.2 Caso de estudio N° 2 - Tesinas

El siguiente caso de estudio, se trata de un entorno web que permite la administración de tesinas para una determinada facultad, permitiendo a un usuario del sistema acceder, visualizar, publicar, editar y compartir tesinas a través de redes sociales, entre otras funcionalidades.

Está desarrollado en lenguaje Java haciendo uso de Hibernate con clases DAO para persistencia a través de JPA (Java Persistence Api) con annotations. JPA es una capa que aísla la persistencia del ORM Hibernate utilizado, de tal manera que si un día se cambia de ORM con cambiar la configuración sería suficiente y no haría falta tocar el código de la capa de persistencia, un concepto parecido al que implementa Hibernate con los lenguajes de las diferentes bases de datos. JPA, tiene un lenguaje de consultas objeto relacional el JPL (muy similar a HQL) y se pueden mapear las tablas utilizando annotations en las propias clases.

El modelo de este sistema desarrollado, está conformado por las clases Carrera, Tesis y Usuario.

El sistema permite realizar búsquedas para facilitar la navegación de un usuario. Actualmente solo admite búsquedas por Titulo y por Carrera. Durante este caso de estudio se pretende realizar un refactoring a nivel de controlador expandiendo el modelo agregando las clases: Tags, Director, Area y Autor junto con sus respectivas configuraciones para lograr persistencia a través de DAOs en Hibernate con anotaciones e incorporar la lógica necesaria en Java para admitir búsquedas adicionales por: Area, Director, Autor y Tags.

Etapa 1: "Codificación tradicional en forma manual"

Se comenzó expandiendo el modelo haciendo uso de la metodología tradicional de codificación manual a través del entorno de desarrollo Eclipse.

Implementación de Clase Director con la metodología tradicional codificando manualmente:

```
package clases;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Director implements Serializable{

	private static final long serialVersionUID = 1L;
	@Id @GeneratedValue
```

```
	private Integer id;
	private String nombre;
	private String apellido;

	@OneToMany
	private List<Tesis> tesinas;

	public Director() {
	}

	public Director(String nombre,String apellido,List<Tesis> tesinas){
		super();
		this.nombre = nombre;
		this.apellido = apellido;
		this.tesinas = tesinas;
	}

	@Column(name = "nombre")
	public String getNombre() {
		return nombre;
	}

	@Column(name = "apellido")
	public String getApellido() {
		return apellido;
	}

	@Column(name = "tesinas")
	public List<Tesis> getTesinas() {
		return tesinas;
	}

	public void setNombre(String nombre) {
	      this.nombre = nombre;
	}

	public void setApellido(String apellido) {
	      this.apellido = apellido;
	}

	public void setTesinas(List<Tesis> tesinas) {
	      this.tesinas = tesinas;
	}

	public String toString(){
		return this.nombre;
	}
}
...
// Clases Tags, Autor y Area
```

(Por una cuestión de simplicidad, en este documento se visualiza el resultado de 1 clase para no abundar, se entiende que los tiempos tomados son respecto a los códigos correspondientes a Director, Tags, Area y Autor)

El tiempo insumido durante la codificación manual de las clases Tags, Director, Area y Autor junto con las anotaciones necesarias para persistir a través de Hibernate fue de: 31 minutos.

Una vez expandido el modelo se pasó a implementar la lógica necesaria en Java para admitir búsquedas por Area, Director, Autor y Tags. Para esto se comenzó con el desarrollo de las interfaces DAO, los DAO implementan el mecanismo de acceso requerido para trabajar con la fuente de datos. Esta fuente de datos puede ser un almacenamiento persistente como una RDMBS, un servicio externo como un intercambio B2B, un repositorio LDAP, o un servicio de negocios al que se accede mediante CORBA Internet Inter-ORB Protocol (IIOP) o sockets de bajo nivel. Los componentes de negocio que tratan con el DAO utilizan un interface simple expuesto por el DAO para sus clientes. El DAO oculta completamente los detalles de implementación de la fuente de datos a sus clientes. Como el interface expuesto por el DAO no cambia cuando cambia la implementación de la fuente de datos subyacente, este patrón permite al DAO adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio. Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos.

También se generarán las clases Hibernate necesarias para persistir las nuevas clases agregadas al modelo a través de la herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java, Hibernate.

Director Hibernate:

```
package clasesDAO;
import java.util.ArrayList;
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import utils.Conexion;
import clases.Director;

public class DirectorHibernate implements DirectorDAO {

	@Override
	public Director getDirectorByName(String nombre) {
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
		etx.begin();
		ArrayList<Director>lista_directores=(ArrayList<Director>)(em.create
	Query("select A from Director A where (A.nombre =
	\'"+nombre+"\')").getResultList());
		etx.rollback();
		em.close();
		if(lista_directores.size() != 0){
			Director area= lista_directores.get(0);
			return area;
		}
		return null;
	}

	@Override
	public Director getDirectorById(Integer id) {
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
```

```
            etx.begin();
            ArrayList<Director>lista_directores=(ArrayList<Director>)(em.create
      Query("select A from Director A where (A.id =
      \'"+id+"\')").getResultList());
            etx.rollback();
            em.close();
            if(lista_directores.size() != 0){
                  Director area= lista_directores.get(0);
                  return area;
            }
            return null;
      }

      @Override
      public ArrayList<Director> getDirectors() {
            EntityManager em = Conexion.getConexion();
            EntityTransaction etx = em.getTransaction();
            etx.begin();
            ArrayList<Director>lista_directores=(ArrayList<Director>)(em.create
      Query("select A from Director A").getResultList());
            etx.rollback();
            em.close();
            return lista_directores;
      }

      @Override
      public void saveDirector(Director a) {
            EntityManager em = Conexion.getConexion();
            EntityTransaction etx = em.getTransaction();
            etx.begin();
            em.persist(a);
            etx.commit();
            em.close();
      }
}
...
//Clases AreaHibernate,TagsHibernate y AutorHibernate
```

Director DAO Interface:

```
package clasesDAO;
import java.util.ArrayList;
import clases.Director;

public interface DirectorDAO {

      public  ArrayList<Director> getDirectors();

      public Director getDirectorById(Integer id);

      public Director getDirectorByName(String nombre);

      public void saveDirector(Director a);

}
```

```
…
//Interfaces AreaDAO, AutorDAO y TagsDAO
```

El tiempo insumido durante la codificación manual de las clases TagsHibernate, DirectorHibernate, AreaHibernate y AutorHibernate fue de: 22 minutos**.**

El tiempo insumido durante la codificacion manual de las las interfaces TagsDAO, DirectorDAO, AreaDAO, AutorDAO fue de: 18 minutos.

Teniendo implementadas las clases e interfaces necesarias, se desarrolló la lógica necesaria para hacer las búsquedas correspondientes a Autor, Area, Tags y Director. Se implementaron las clases necesarias para hacer las búsquedas correspondientes en la base de datos y setear los elementos a la sesión para concretar la acción en las JSPs.

Búsqueda por director:

```
package actions;

import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpSession;
import org.apache.struts2.ServletActionContext;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Namespace;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;
import clases.Director;
import clasesDAO.DirectorDAO;
import clasesDAO.FactoryDAO;
import com.opensymphony.xwork2.ActionSupport;

@Namespace("/")
@Action(value="BuscarTesisPorDirector")
@Results ({
@Result(name="success", location="/busquedaPorDirector.jsp"),
@Result (name="input", location="/busquedaPorDirector.jsp")})
public class BuscarTesisPorDirectorAction extends ActionSupport {

	private static final long serialVersionUID = 1L;

	public String execute() {
		DirectorDAO directorDAO = FactoryDAO.getDirectorDAO();
		List<Director> listaDirectores = directorDAO.getDirectors();
		Iterator<Director> ite2 = listaDirectores.iterator();
		Map<Integer, Director> directores = new HashMap<Integer ,
	Director>();
		while (ite2.hasNext()){
			Director actual = ite2.next();
			directores.put(actual.getId(), actual);
		}
		HttpSession session =
ServletActionContext.getRequest().getSession();
```

```
            session.setAttribute("directores", directores);
            return SUCCESS;
      }

}
…
//Clases busquedaPorArea, busquedaPorAutor, busquedaPorTags
```

El tiempo insumido en implementar las clases: busquedaPorAutor.java, busquedaPorArea.java, busquedaPorTags.java y busquedaPorDirector.java fue de: 29 minutos

Teniendo en cuenta las estadísticas recolectadas durante la primera etapa se puede visualizar lo siguiente:

Implementación realizada	**Tiempo insumido en Minutos**
Implementación de las clases Area, Autor, Director y Tags	31
Implementación de las clases AreaDAO, AutorDAO, DirectorDAO y TagsDAO	18
Implementación de las clases AreaHibernate, AutorHibernate, DierctorHibernate y TagsHibernate	22
Implementación de las clases busquedaPorAutor, busquedaPorArea, busquedaPorTags y busquedaPorDirector	29
Tiempo total insumido durante la Etapa 1	1 hora 40 minutos

Etapa 2: "Codificación utilizando ADSM a través de MYL"

Durante el transcurso de la codificación de las clases se fueron detectando patrones que se hacían repetitivos. Esto dio pie a utilizar la metodología ADSM para hacer el mismo trabajo pero ahora generando automáticamente el código haciendo uso de la herramienta Make Your Language (MYL).

Para hacer esto, se comenzó con abstraer en templates (plantillas) estos patrones detectados, inferir modelos, completarlos junto con códigos de procesamientos para lograr una generación óptima y generar el código resultado.

Se comenzó con la generación de las clases para el modelo:

Generación de clases del modelo:

Comenzando por la generación de las clases para expandir el modelo, se importó el proyecto en el entorno MYL, luego se editó el template agregando sus anotaciones y cambiándole el nombre original por: #classes.nameFile.java, ésto se hizo para poder generar

múltiples archivos a partir del template, y sacar el provecho que MYL ofrece. El contenido del template fue el siguiente:

Template: `#classes.nameFile.java`

```
package clases;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class {{nameFile}} implements Serializable{

	private static final long serialVersionUID = 1L;
	@Id @GeneratedValue
	private Integer id;
	{{#vars}}
	{{association}}
	private {{type}} {{name}};
	{{/vars}}

	public {{nameFile}}() {
	}

	public
{{nameFile}}({{#vars}}{{type}}{{name}}{{#comma}},{{/comma}}{{/vars}}) {
		super();
		{{#vars}}
		this.{{name}} = {{name}};
		{{/vars}}
	}

	{{#vars}}
	@Column(name = "{{name}}")
	public {{type}} get{{nameUpper}}() {
		return {{name}};
	}
	{{/vars}}

	{{#vars}}
	public void set{{nameUpper}}({{type}} {{name}}) {
		 this.{{name}} = {{name}};
	}
	{{/vars}}

	public String toString(){
		return this.{{nameToString}};
	}
}
```

Tiempo insumido en generar el template: 5 minutos.

Luego se generó el modelo correspondiente para generar las clases Area, Autor, Director y Tags, cabe destacar que en este paso la herramienta MYL provee la funcionalidad de inferencia del modelo inicial, con lo que solo basta completarlo con los datos deseados. El modelo quedó como se visualiza a continuación.

Cabe destacar que a partir del modelo se generarán todas las clases necesarias para las búsquedas, clasesDAO, clasesHibernate y clases del modelo.

Modelo:

```
{
    "classes":
     [
        {"nameFile": "Area",
            "vars": [
                {"name": "nombre","type": "String"},
                {"name": "tesinas","type": "List<Tesis>"}
             ]
        },
        {"nameFile": "Director",
            "vars": [
                {"name": "nombre","type": "String"},
                {"name": "apellido","type": "String"},
                {"name": "tesinas","type": "List<Tesis>"}
             ]
        },
        {"nameFile": "Autor",
            "vars": [
                {"name": "nombre","type": "String"},
                {"name": "apellido","type": "String"}
             ]
        },
        {"nameFile": "Tags",
            "vars": [
                {"name": "nombre","type": "String"}
             ]
        }
    ]
}
```

Tiempo insumido en completar el modelo: 4 minutos

Luego se diseñó el código de procesamiento para el modelo creado, este código fue necesario para editar el modelo previamente a la renderizazcion, lo que hace el código es recorrer el modelo JSON, seteando la propiedad “nameToString” con el valor del atributo name para satisfacer el método toString(), luego setea la propiedad nameUpper donde se coloca el valor del nombre cambiando el primer caracter a mayúscula, se hace el control para las comas en la sección de parámetros de los métodos constructores y setters y se agrega la anotación propia de JPA @OneToMany para el caso donde una valiable sea un “List”. El código para el modelo previamente creado se puede visualizar a continuación:

```
for (var i=0; i<model.classes.length; i++) {
       model.classes[i]['nameToString'] = model.classes[i].vars[0].name;
       for (var w=0; w<model.classes[i].vars.length; w++) {
       model.classes[i].vars[w]['nameUpper'] =
       model.classes[i].vars[w].name.charAt(0).toUpperCase() +
       model.classes[i].vars[w].name.substring(1);
           if(model.classes[i].vars.length > 1 &&
w<(model.classes[i].vars.length-1)){
               model.classes[i].vars[w]['comma'] = true;
            }
          if(model.classes[i].vars[w].type.indexOf("List") > -1){
               model.classes[i].vars[w]['association'] = "@OneToMany";
          }
       }
}
```

Tiempo insumido en realizar el código de procesamiento: 6 minutos.

Una vez teniendo el template, el modelo junto con el código de procesamiento se procedió a la generación automática de las clases, el tiempo insumido en generar las clases fue de 0 minutos gracias a la automatización provista por MYL.

Resultado:

```
package clases;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Director implements Serializable{

       private static final long serialVersionUID = 1L;
       @Id @GeneratedValue
       private Integer id;
       private String nombre;
       private String apellido;

       @OneToMany
       private List<Tesis> tesinas;

       public Director() {
       }

       public Director(String nombre,String apellido,List<Tesis> tesinas){
              super();
              this.nombre = nombre;
              this.apellido = apellido;
              this.tesinas = tesinas;
       }
```

```
	@Column(name = "nombre")
	public String getNombre() {
		return nombre;
	}

	@Column(name = "apellido")
	public String getApellido() {
		return apellido;
	}

	@Column(name = "tesinas")
	public List<Tesis> getTesinas() {
		return tesinas;
	}

	public void setNombre(String nombre) {
	     this.nombre = nombre;
	}

	public void setApellido(String apellido) {
		this.apellido = apellido;
	}

	public void setTesinas(List<Tesis> tesinas) {
		this.tesinas = tesinas;
	}

	public String toString(){
		return this.nombre;
	}
}

...
// Clases Tags, Autor y Area
```

(Por una cuestión de simplicidad, en este documento se visualiza el resultado de 1 clase para no abundar, se entiende que la generación otorga los códigos correspondientes a Director, Tags, Area y Autor)

Generación de clases necesarias para persistir a través de Hibernate (Clases DAO):

Template: `#classes.nameFileDAO.java`

```
package clasesDAO;

import java.util.ArrayList;

import clases.{{nameFile}};

	public interface {{nameFile}}DAO {

	public ArrayList<{{nameFile}}> get{{nameFile}}s();
```

```
	public Autor get{{nameFile}}ById(Integer id);

	public Autor get{{nameFile}}ByName(String nombre);

	public void save{{nameFile}}({{nameFile}} a);

}
```

Tiempo insumido en crear el template: 2 minutos

Se utilizó el modelo inicialmente planteado para la renderización.

Código de procesamiento para el modelo: Este código es necesario para recorrer el modelo y setear la propiedad nameFileDAO, la cual será el nombre del archivo resultado.

```
for (var i=0; i<model.classes.length; i++) {
	 model.classes[i]['nameFileDAO'] = model.classes[i].nameFile + "DAO";
}
```

Tiempo insumido en crear el código de procesamiento para el modelo: 2 minutos

Resultado:

```
package clasesDAO;

import java.util.ArrayList;

import clases.Director;

public interface DirectorDAO {

	public ArrayList<Director> getDirectors();

	public Autor getDirectorById(Integer id);

	public Autor getDirectorByName(String nombre);

	public void saveDirector(Director a);

}
...
// Clases TagsDAO, AutorDAO y AreaDAO
```

Tiempo insumido para generar el código resultado: inmediato

Generación de clases necesarias para persistir a través de Hibernate (Clases Hibernate):

Template: #classes.nameFileHibernate.java

```
package clasesDAO;

import java.util.ArrayList;
```

```
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import utils.Conexion;
import clases.{{nameFile}};

public class {{nameFile}}Hibernate implements {{nameFile}}DAO {

	@Override
	public {{nameFile}} get{{nameFile}}ByName(String nombre) {
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
		etx.begin();
		ArrayList<{{nameFile}}>lista_{{nameFile}}=(ArrayList<{{nameFile}}>)
	(em.createQuery("select T from {{nameFile}} T where (T.nombre =
	\'"+nombre+"\')").getResultList());
		etx.rollback();
		em.close();
		if(lista_{{nameFile}}.size() != 0){
			{{nameFile}} {{nameLower}} = lista_{{nameFile}}.get(0);
			return {{nameLower}};
		}
		return null;
	}

	@Override
	public Tags get{{nameFile}}ById(Integer id){
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
		etx.begin();
		ArrayList<{{nameFile}}>lista_{{nameFile}}=(ArrayList<{{nameFile}}>)
	(em.createQuery("select T from {{nameFile}} T where (T.id =
	\'"+id+"\')").getResultList());
		etx.rollback();
		em.close();
		if(lista_{{nameFile}}.size() != 0){
			{{nameFile}} {{nameLower}} = lista_{{nameFile}}.get(0);
			return {{nameLower}};
		}
		return null;
	}

	@Override
	public ArrayList<{{nameFile}}> get{{nameFile}}() {
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
		etx.begin();
		ArrayList<{{nameFile}}>lista_{{nameFile}}=(ArrayList<{{nameFile}}>)
	(em.createQuery("select T from {{nameFile}} T").getResultList());
		etx.rollback();
		em.close();
		return lista_{{nameFile}};
	}

	@Override
	public void save{{nameFile}}({{nameFile}} {{nameLower}}) {
```

```
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
		etx.begin();
		em.persist({{nameLower}});
		etx.commit();
		em.close();
	}
}
```

Tiempo insumido en crear el template: 5 minutos

Código de procesamiento para el modelo: Este código recorre el modelo agregando la propiedad “nameFileHibernate” la cual se corresponderá con el nombre del archivo resultado de la renderización, así como también se setea en el modelo la propiedad “nameLower”, con el valor del atributo name y la característica que es toda la palabra en minúsculas, esto es necesario para poder lograr una generacion óptima, prolija y 100 % compilable.

```
for (var i=0; i<model.classes.length; i++) {
model.classes[i]['nameFileHibernate'] = model.classes[i].nameFile + "Hibernate";
 model.classes[i]['nameLower']=model.classes[i].nameFile.charAt(0).toLowerCase()
+ model.classes[i].nameFile.substring(1);
}
```

Tiempo insumido en crear el código de procesamiento: 3 minutos

Resultado:

Clase director Hibernate:

```
package clasesDAO;

import java.util.ArrayList;
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import utils.Conexion;
import clases.Director;

public class DirectorHibernate implements DirectorDAO {

	@Override
	public Director getDirectorByName(String nombre) {
		EntityManager em = Conexion.getConexion();
		EntityTransaction etx = em.getTransaction();
		etx.begin();
		ArrayList<Director>lista_Director =
	(ArrayList<Director>)(em.createQuery("select T from  Director T where
	(T.nombre = \'"+nombre+"\')").getResultList());
		etx.rollback();
		em.close();
		if(lista_Director.size() != 0){
			Director director = lista_Director.get(0);
			return director;
		}
```

```
			return null;
		}

		@Override
		public Tags getDirectorById(Integer id){
			EntityManager em = Conexion.getConexion();
			EntityTransaction etx = em.getTransaction();
			etx.begin();
			ArrayList<Director>lista_Director =
		(ArrayList<Director>)(em.createQuery("select T from  Director T where
		(T.id = \'"+id+"\')").getResultList());
			etx.rollback();
			em.close();
			if(lista_Director.size() != 0){
				Director director = lista_Director.get(0);
				return director;
			}
			return null;
		}

		@Override
		public ArrayList<Director> getDirector() {
			EntityManager em = Conexion.getConexion();
			EntityTransaction etx = em.getTransaction();
			etx.begin();
			ArrayList<Director>lista_Director =
		(ArrayList<Director>)(em.createQuery("select T from Director
		T").getResultList());
			etx.rollback();
			em.close();
			return lista_Director;
		}


		@Override
		public void saveDirector(Director director) {
			EntityManager em = Conexion.getConexion();
			EntityTransaction etx = em.getTransaction();
			etx.begin();
			em.persist(director);
			etx.commit();
			em.close();
		}
}
...
// Clases TagsHibernate, AutorHibernate y AreaHibernate
```

Generación de las clases para las búsquedas:

Template: #classes.nameFileSearches.java

```
package actions;

import java.util.HashMap;
import java.util.Iterator;
```

```
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpSession;
import org.apache.struts2.ServletActionContext;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Namespace;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;
import clases.{{nameFile}};
import clasesDAO.{{nameFile}}DAO;
import clasesDAO.FactoryDAO;
import com.opensymphony.xwork2.ActionSupport;

@Namespace("/")
@Action(value="BuscarTesisPor{{nameFile}}")
@Results ({
@Result(name="success", location="/busquedaPor{{nameFile}}.jsp"),
@Result (name="input", location="/busquedaPor{{nameFile}}.jsp")})
public class BuscarTesisPor{{nameFile}}Action extends ActionSupport {

	private static final long serialVersionUID = 1L;

	public String execute() {
		{{nameFile}}DAO {{nameLower}}DAO = FactoryDAO.get{{nameFile}}DAO();
		List<{{nameFile}}> lista{{nameFile}} =
{{nameLower}}DAO.get{{nameFile}}s();
		Iterator<{{nameFile}}> ite2 = lista{{nameFile}}.iterator();
		Map<Integer, {{nameFile}}> {{nameLower}}s = new HashMap<Integer,
{{nameFile}}>();
		while (ite2.hasNext()){
			{{nameFile}} actual = ite2.next();
			{{nameLower}}s.put(actual.getId(), actual);
		}
		HttpSession session =
ServletActionContext.getRequest().getSession();
		session.setAttribute("{{nameLower}}s", {{nameLower}}s);
		return SUCCESS;
	}
}
```

Tiempo insumido en crear el template: 5 minutos

Processor Code:

Este código de procesamiento se encarga de recorrer el modelo introduciendo la propiedad 'nameFileSearchs' con el valor de “nameFile” + “Search”, esto se hace para que la herramienta genere cada archivo con el nombre [nameFile]Search.java. Tambien se introduce en el modelo la propiedad “nameLower” con el valor de “nameFile”con su primer letra en minúsculas.

```
for (var i=0; i<model.classes.length; i++) {
model.classes[i]['nameFileSearchs'] = model.classes[i].nameFile + "Search";
```

```
model.classes[i]['nameLower'] =
model.classes[i].nameFile.charAt(0).toLowerCase() +
model.classes[i].nameFile.substring(1);
}
```

Tiempo insumido en crear el código de procesamiento para el modelo: 4 minutos

Resultado:

```
package actions;

import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpSession;
import org.apache.struts2.ServletActionContext;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Namespace;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;
import clases.Director;
import clasesDAO.DirectorDAO;
import clasesDAO.FactoryDAO;
import com.opensymphony.xwork2.ActionSupport;

@Namespace("/")
@Action(value="BuscarTesisPorDirector")
@Results({
@Result(name="success", location="/busquedaPorDirector.jsp"),
@Result (name="input", location="/busquedaPorDirector.jsp")})

public class BuscarTesisPorDirectorAction extends ActionSupport {

private static final long serialVersionUID = 1L;

public String execute() {
       DirectorDAO directorDAO = FactoryDAO.getDirectorDAO();
       List<Director> listaDirector = directorDAO.getDirectors();
       Iterator<Director> ite2 = listaDirector.iterator();
       Map<Integer, Director> directors = new HashMap<Integer , Director>();
       while (ite2.hasNext()){
              Director actual = ite2.next();
              directors.put(actual.getId(), actual);
       }
       HttpSession session = ServletActionContext.getRequest().getSession();
       session.setAttribute("directors", directors);
       return SUCCESS;
       }
}
...
// Clases busquedaPorTags, busquedaPorAutor y busquedaPorArea
```

Teniendo en cuenta las estadísticas recolectadas durante la segunda etapa se puede visualizar lo siguiente:

Implementación realizada	**Tiempo insumido en Minutos**
Implementación de las clases Area, Autor, Director y Tags	15
Implementación de las clases AreaDAO, AutorDAO, DirectorDAO y TagsDAO	4
Implementación de las clases AreaHibernate, AutorHibernate, DierctorHibernate y TagsHibernate	8
Implementación de las clases busquedaPorAutor, busquedaPorArea, busquedaPorTags y busquedaPorDirector	9
Tiempo total insumido durante la Etapa 2	36 minutos

Análisis y estadísticas

Como se puede apreciar en los cuadros de tiempos presentados anteriormente, utilizando ADSM se optimiza considerablemente el tiempo de implementación, esto es debido a la generación automática y el código simple que pretende su utilización. Realizando una codificación manual tradicional, se insume mucho tiempo en detalles de sintaxis, se implementa lógica repetitiva que lleva a introducir errores humanos. El diagrama de barras que se visualiza a continuación, representa estos tiempos insumidos durante este caso de estudio aplicando la metodología tradicional y luego ADSM.

Como se puede ver, en este caso de estudio, se optimizan 64 minutos (1:04 hs.) de implementación, esto es, codificando tradicionalmente en forma manual se insumieron un total de 100 minutos (1:40 hs.) aproximadamente, codificando con ADSM a través de MYL se insumió un tiempo de 36 minutos (0:36 hs.).

6. Capítulo 6: Conclusiones y trabajos futuros

6.1 Conclusiones

Luego de haber trabajado durante mucho tiempo en el desarrollo de este proyecto, se ha observado la gran utilidad que provee la generación automática de código, la abstracción de soluciones a través de un proceso de detección de patrones y templating, para llegar a la obtención de un metamodelo de una manera novedosa y ágil.

Si bien la idea principal de este trabajo está enfocada en la generación de código para lenguajes de programación y el descubrimiento de un metamodelo formal, cabe destacar que puede ser utilizado para generar otro tipo de salida, como puede ser la documentación técnica en diversos formatos: html, pdf, odt, etc. o cualquier otro uso donde exista la posibilidad de parametrizar la salida esperada a través de técnicas de templating.

A lo largo de ésta tesis, se cumplió el objetivo de investigar y construir una herramienta, que provee el desarrollo dirigido por modelos, brindando generación de código automática, atacando el problema principal de la reincidencia de código en los proyectos de software y las limitaciones del Modelado Específico de Dominio (DSM) de tener que cumplir con etapas estrictas necesarias para su uso, como la previa definición de un matamodelo formal como elemento principal.

Se ha realizado un estudio en profundidad de la ingeniería dirigida por modelos (MDE), del desarrollo dirigido por modelos (MDD) y de las dos corrientes más consolidadas para la aplicación de MDD, la arquitectura dirigida por modelos (MDA), el modelado específico de dominio (DSM) para llegar la definición de la metodología Agile DSM (ADSM). En este contexto se han analizado los términos modelo, metamodelo, modelos específicos de dominio y lenguajes específicos de dominio, así como las relaciones existentes entre ellos, ampliando el campo de conocimiento pretendido desde el inicio. Asimismo, se cumplió el objetivo de implementar la herramienta (Make Your Language, MYL) en lenguaje Java con un diseño responsive adaptable a cualquier plataforma.

Este proyecto fue pensado para poder ser extendido y adaptado a los diversos usos y lenguajes de programación, sin restringir el tipo de salida generado, ofreciendo la máxima adaptación que el usuario demande. Su gran ventaja radica en la utilización de software libre y de lenguajes ampliamente conocidos y extremadamente documentados para poder garantizar el uso de todas las características que la metodología ADSM a través de MYL ofrecen.

Si bien muchos IDEs modernos ya cuentan con características de generación de código, el hecho de utilizar en todo el proceso la misma herramienta brinda consistencia e integración, además de brindar la capacidad de poder extenderse y adaptarse al modelo de trabajo de una empresa. Los beneficios que atrae utilizar ADSM han sido demostrados en los casos de estudios donde se pudo cumplir con el objetivo de optimizar el tiempo de desarrollo, reduciendo el tiempo de implementación, sin la necesidad de hacer uso de las etapas estrictas que pretende un tradicional DSM, obteniendo como resultado un metamodelo formal de una manera novedosa y ágil.

6.2 Trabajo futuro

Si bien la herramienta MYL fue diseñada e implementada para cubrir la mayor de las expectativas, algunos aspectos quedan a optimizar y añadir a futuro.

Actualmente la herramienta MYL, no cuenta con la posibilidad de formalizar un modelo JSON a través de un JSON Schema, esta funcionalidad se debe hacer en el entorno propio de JSON Schema. Una de las futuras extensiones de MYL es, añadir esta funcionalidad dentro del mismo entorno MYL, es decir, que el usuario al finalizar con su proceso de ADSM, pueda a través de una UI para JSON Schema dentro de MYL y sus reglas de validación definidas en JavaScript, formalizar su modelo.

Por otra parte, MYL brinda la posibilidad de importar un proyecto en formato .zip, se pretende en un futuro implementar la funcionalidad de importar un JSON en formato .json una vez estando en la sección fábrica o factory. Esto será de utilidad ya que otorgará un grado de portabilidad de los modelos mucho más eficiente, e incentivará compartir modelos diseñados entre distintos desarrolladores.

En un futuro queda pendiente proveer una capa de servicios REST y plugins para IDEs de modo de integrarlos nativamente a la plataforma para ofrecer endpoints utilizables por algún cliente estilo web service, como también extender la solución para que soporte otros lenguajes interpretados como por ejemplo Groovy.

Otras características pensadas para extender el uso de la metodología es la integración con plugins de maven para disparar la generación de código y la ejecución de las aplicaciones generadas en un solo paso y brindar estrategias de despliegues automáticos estilo Git del código generado.

Actualmente, se está trabajando en un experimento para una publicación donde se investigan múltiples herramientas de scaffolding y se harán pruebas de generación de un módulo específico haciendo uso de las mismas comparando con ADSM haciendo uso de MYL.

Por último se efectuarían mejoras en la accesibilidad de algunos aspectos, cabe destacar que la capa de la Vista (presentación) de MYL actualmente está diseñada con Bootstrap 3, lo cual brinda un diseño Responsive, (adaptativo) para poder hacer uso de la herramienta en cualquier dispositivo.

7. Bibliografía

[1] Douglas, C. Schmidt (2006). Vanderbilt University, Disponible en: http://ftp.icm.edu.pl/packages/ace/ACE/PDF/GEI.pdf

[2] Grimán, A. y Mendoza, Luis E. SISTEMAS DE INFORMACIÓN III LABORATORIO CONTENIDO: CONSIDERACIONES GENERALES SOBRE LAS HERRAMIENTAS CASE - Disponible en: http://prof.usb.ve/lmendoza/Documentos/PS-6117%20(Laboratorio)/Transparencias%20PS6117%20(Lab)%20HC.pdf

[3] Notas de clase de la cátedra Orientación a Objetos 1 (2011) - Facultad de Informática UNLP

[4] Fondement, F. y Silaghi R. (2004). Defining Model Driven Engineering Processes, Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML). Lisbon, Portugal.

[5] Kent, S. (2002) Model-driven engineering (MDE). Disponible en: https://kar.kent.ac.uk/13657

[6] Beydeda, S., Book, M., & Gruhn V. (2005), Model-Driven Software Development, Springer.

[7] Kleppe A., Warmer J., Bast W. (2004). MDA EXPLAINED The Model Driven Architecture: Practice and Promise, Addison Wesley, Object Technology Series, Grady Booch, Ivar Jacobson, and James Rumbaugh.

[8] Atkinson, C. and Kühne, T. (2002). Rearchitecting the UML Infrastructure ACM journal "Transactions on Modeling and Computer Simulation", Vol. 12, No. 4.

[9] García Díaz, V. & Cueva Lovelle, J. (2010). Ingeniería Dirigida por Modelos. Oviedo.

[10] Oficial web site OMG. Disponible en: http://www.omg.org/

[11] Meta-Object Facility (MOF). Disponible en: http://www.omg.org/mof/

[12] The UML 2.0 Specification, Object Management Group. Disponible en: http://www.uml.org/#UML2.0

[13] Steven Kelly y Juha-Pekka Tolvanen. (2008). Domain-Specific Modeling: Enabling Full Code Generation.

[14] The Unified Modeling Language Website, Object Management Group (OMG), http://www.uml.org/

[15] Model Driven Architecture (MDA) FAQ, Object Management Group. Disponible en: http://www.omg.org/mda/faq_mda.htm#whatismda

[16] Debasish, G. (2011) DSLs in ACTION. Manning Publications

[17] Domain specific development with Visual Studio DSL Tools. Disponible en: http://www.domainspecificdevelopment.com

[18] Developing DSLs using combinators. A design pattern - Disponible en: https://fedcsis.org/proceedings/2009/PTI/pliks/8.pdf

[19] Template processor. Disponible en: http://en.wikipedia.org/wiki/Template_processor

[20] JSON. Disponible en: json.org/ y http://es.wikipedia.org/wiki/JSON

[21] Mustache template. Disponible en: https://mustache.github.io/

[22] Van Deursen, A. (1997). Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study. In Proceedings of Smalltalk and Java in Industry and Academia (STJA'97), pages35–39. Ilmenau Technical University.

[23] S. Beydeda and V. Gruhn (2005). Model-Driven Software Development. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[24] Graeme, S. y Graham, W. (2004). Data Modeling Essentials, Third Edition

[25] The Art of Agile Development. Disponible en: https://poetiosity.files.wordpress.com/2011/04/art_of_agile_development.pdf

[26] Stuart, K. Model Driven Engineering. University of Kent, Canterbury, UK. Disponible en: http://www.cs.ukc.ac.uk

[27] When and how to develop domain-specific languages. Disponible en: http://www.researchgate.net/profile/Marjan_Mernik/publication/200040446_When_and_How_to_Develop_Domain-Specific_Languages/links/02bfe50f4c6709f12e000000.pdf

[28] Clark, T., Sammut, P. y Willans, J. (2008). APPLIED METAMODELLING A FOUNDATION FOR LANGUAGE DRIVEN DEVELOPMEN SECOND EDITION. Disponible en: https://eprints.mdx.ac.uk/6060/1/Clark-Applied_Metamodelling_%28Second_Edition%29[1].pdf

[29] Language Workbenches: The Killer-App for Domain Specific Languages? Disponible en: http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf

[30] OMG Architecture Board ORMSC. (2001) Model driven architecture (MDA). OMG document number ormsc/2001-07-01. Disponible en: www.omg.org

[31] S. W. Ambler, Agile Modeling (AM) Home Page: Effective Practices for Modeling and Documentation. Disponible en: http://www.agilemodeling.com

[32] C.A.S.E. Technology. Disponible en: https://www.utdallas.edu/~chung/SYSM6309/RE_chapters/Chapter%206.pdf

[33] Lic. Favre, L. (2008) Tesis grado de Magíster en Ingeniería de Software. COMPONENTES MDA PARA PATRONES DE DISEÑO - Página 5. Disponible en:

http://postgrado.info.unlp.edu.ar/Carreras/Magisters/Ingenieria_de_Software/Tesis/Martinez_Liliana.pdf

[34] Clarifying concepts: MBE vs MDE vs MDD vs MDA, Disponible en: http://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/

[35] Arquitecturas Reflexivas y Generación de Código, Disponible en: http://webtaller.com/maletin/articulos/arquitecturas-reflexivas-generacion-codigo-2.php

[36] Viaje al universo neuronal - fundación española para la ciencia y tecnología, Disponible en: http://www.oei.es/salactsi/unidad_didactica.pdf

[37] Christopher, A., Ishikawa, S. y Silverstein, M. (1977). A Pattern Language: Towns, Buildings, Construction. Disponible en: https://books.google.com.ar/books/about/A_Pattern_Language.html?id=hwAHmktpk5IC&redir_esc=y

[38] MDA Guide. (2003) v1.0.1, omg/03-06-01, Disponible en: http://www.omg.org

[39] Giandini, Roxana S. y Pons, Claudia F. Un lenguaje para Transformación de Modelos basado en MOF y OCL. Universidad Nacional de La Plata, Facultad de Informática LIFIA - Laboratorio de Investigación y Formación en Informática Avanzada. La Plata, Argentina.

- Thomas, D. A. y Barry, B. M. (2003) 'Model-Driven Development: the Case for Domain-Oriented Programming, Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 2–7.

- Pérez, F. y De Lara, J. HACIA LA DEFINICION DE LENGUAJES ESPECIFICOS DE DOMINIO CON SINTAXIS GRAFICA Y TEXTUAL. Departamento de Ingeniería Informática Escuela Politécnica Superior Universidad Autónoma de Madrid Ciudad Universitaria de Cantoblanco, Madrid.