Universidad Autónoma de Madrid

Escuela Politécnica Superior
Máster I²TIC

Trabajo de Fin de Máster

Descripción de las actividades de una propuesta de Metodología de Desarrollo de Software Dirigida por Modelos

---

Autor: Santiago Jácome
[santiago.jacome@estudiante.uam.es]

Tutor: Juan de Lara
[juan.delara@uam.es]

Madrid, Septiembre de 2013

Contenido

Indice de Figuras

Indice de Tablas

Resumen

A lo largo de la pasada década, la Ingeniería de Software Dirigida por Modelos (MDE, del inglés Model-Driven Engineering) ha surgido como un nuevo paso en el camino hacia la verdadera industrialización de la producción de software. Tras el éxito de la tecnología orientada a objetos, el uso sistemático de modelos se presenta ahora como la forma apropiada para conseguir programar con un nivel más alto de abstracción, proveer mayor facilidad en el desarrollo, aumentar la calidad de los productos desarrollados y sobre todo aumentar el nivel de automatización. Sin embargo, para lograrlo se necesita crear una serie de artefactos y la definición de procesos que apoyen la labor MDE. Para lo cual existen dos tendencias, la una apoyada por el OMG (Object Management Group) a través de la Arquitectura Dirigida por Modelos (MDA, del inglés Model-Driven Architecture) la cual se basa en la utilización de UML (Unified Modeling Language) como lenguaje de modelado, la otra tendencia apoyada principalmente por la comunidad investigadora a través de la utilización de Lenguajes de Dominio Específico (DSLs, del inglés Domain-Specific Language).

Al hablar de desarrollo de software dirigido por modelos en base a DSLs se deben cubrir dos etapas, la primera la creación de un conjunto de artefactos MDE, principalmente DSLs, mecanismos de transformación de modelos, generadores de código; y una segunda etapa donde personas no necesariamente con formación en informática (usuario final) utilizando los artefactos MDE creados, puedan desarrollar aplicaciones para apoyar sus labores profesionales de una manera rápida y simple, debido al alto nivel de abstracción permitido por la tecnología.

El presente proyecto plantea la creación de una propuesta de una metodología de desarrollo de software dirigida por modelos con DSLs, debido a que se ha detectado que a pesar de haberse logrado crear mecanismos de desarrollo de soluciones MDE a través de varios proyectos de investigación, poco o nada se ha realizado con la intención de establecer un proceso formal de desarrollo que integre las diferentes propuestas existentes utilizando este enfoque. Situación que puede resultar útil a aquellas personas y empresas que hayan escuchado hablar de MDE y estén interesadas en adoptar este enfoque de desarrollo de software. La metodología propuesta se encuentra estructurada tomando en cuenta criterios de metodologías de desarrollo de software tradicional y ágil, análisis de dominio, reutilización, y desarrollo por el usuario final.

Palabras Clave: Ingeniería de Software Dirigida por Modelos, Metodología de Desarrollo de Software Dirigida por Modelos, Metodología Dirigida por Modelos, Lenguaje de Dominio Específico, Análisis de Dominio, Desarrollo por el Usuario Final, Ingeniería de Software de Usuario Final, DSL, MDE, MDM1.

Capítulo 1

Introducción al Proyecto

1.1 Antecedentes

Se considera que el proceso de desarrollo de software ha venido evolucionando desde la aparición del primer ordenador. En sus inicios los únicos que desarrollaban programas computacionales eran los mismos creadores de aquellos grandes equipos, lo hacían a través de modificar la circuitería. Posteriormente aparecieron los lenguajes de programación como Ensamblador, Fortran, Cobol, Basic, C, C++; que lograron hacer relativamente más fácil la tarea de crear programas computacionales.

Sin embargo, es a partir de la aparición de la Ingeniería de Software que se logró introducir cierto formalismo a la creación de software. Es así que primero aparecieron los lenguajes de programación o enfoques de creación software y cuando éstos se consideraban confiables, aparecían las correspondientes metodologías de desarrollo. Esta afirmación se la puede realizar debido a que cuando se desarrollaron los lenguajes de programación estructurados como Pascal y C, aparecieron posteriormente las metodologías estructuradas u orientadas a procesos que tomaban como base los *procesos.* De igual manera cuando se desarrolló completamente el paradigma orientado a objetos y los lenguajes de programación de soporte como C++, aparecieron posteriormente las metodologías orientadas a objetos, que tomaban como base a los *objetos.*

Desde hace algunos años han aparecido nuevos enfoques de creación de software para el manejo más eficiente y adecuado de la cada vez mayor complejidad de los sistemas. Uno de esos enfoques toma en cuenta la representación abstracta del sistema en base a los modelos que los representa [97]. La Ingeniería de Software Dirigida por Modelos ha tomado en cuenta este criterio, siendo principalmente la comunidad investigadora la que se encuentra trabajando para proveer de madurez a este enfoque. Dentro de sus postulados está el de proveer mayor facilidad, productividad y calidad al proceso de desarrollo de software [62].

Para apoyar esta labor se han desarrollado varias herramientas que dan soporte al nuevo enfoque, pero aún no se ha trabajo lo suficiente en aspectos de metodologías formales de desarrollo, modelos de ejecución y simulación, testing y validación, técnicas de aseguramiento de calidad, administración de proyectos, desarrollo por el usuario final con herramientas MDE. Por lo cual se puede considerar que todavía no se ha alcanzado la suficiente madurez para ser considerado un proceso formal de desarrollo, que sea conocido y utilizado por la mayoría de la comunidad de desarrollo de software.

1.2 Motivación

En el entorno de la ingeniería de software, la modelización tiene una rica tradición que se remonta a los primeros días de la programación. Las innovaciones más recientes se han centrado en las notaciones y herramientas que permiten a los desarrolladores expresar diversos puntos de vista del sistema, de vital valor sobre todo para los arquitectos y desarrolladores de software. Los elementos expresados en los modelos deben ser mapeados en el código de un lenguaje de programación que será compilado en una determinada plataforma.

Los modelos proporcionan abstracciones de un sistema, permitiendo a los ingenieros razonar acerca de éste, ignorando detalles superfluos mientras se centran en los relevantes. Los modelos se utilizan de muchas maneras, como para tratar de "v*isualizar*" *el* sistema antes de construirlo debido a que los modelos constituyen un precursor del sistema a construir, para realizar variaciones en él de tal manera que permita determinar su comportamiento situación que no se podría realizar en el sistema real para predecir cualidades del sistema, y fundamentalmente para comunicar las características clave del sistema a las distintas partes interesadas.

Con la finalidad de aportar la iniciativa MDE como un nuevo paradigma en la creación y evolución del desarrollo de software, el presente trabajo estructura una propuesta de una metodología de desarrollo de software con enfoque MDE basado DSLs para proyectos de baja a media complejidad, donde los artefactos MDE sean creados por personal informático y la aplicación pueda ser desarrollada y desplegada por el usuario final (personal no necesariamente informático) a partir de los artefactos MDE creados. De tal manera que se podría hablar de una *Metodología Orientada a Modelos* o *Metodología Dirigida por Modelos*.

1.3 Objetivo General

El objetivo del presente trabajo es realizar una descripción general de las actividades que conforman una propuesta de Metodología de Desarrollo de Software dirigida por Modelos con Lenguajes de Dominio Específico.

Objetivos Específicos

- Elaborar el marco teórico que permita formular la propuesta, en base a la revisión de MDE, DSLs, metodologías de desarrollo de software tradicional y ágil, análisis de dominio, y desarrollo por el usuario final.
- Diseñar la estructura de las actividades que conforman la metodología a proponer, tomando en consideración las particularidades de MDE en base a DSL y el tipo de aplicaciones a desarrollar (de baja a mediana complejidad).
- Realizar la descripción general de las actividades de la metodología.

Consideración del alcance del proyecto

Debido al limitado número de horas para el desarrollo del Trabajo de Fin de Máster (300 horas), la propuesta de la metodología de desarrollo de software cubrirá únicamente el establecimiento de la estructura y descripción de cada una de la actividades, sin llegar a la proceso de evaluación de la metodología en la práctica, pudiendo éste ser desarrollado como trabajo futuro.

1.4 Metodología

Existen cuatro técnicas empíricas que pueden utilizarse en la realización de una investigación: observación, experimentos formales, estudios en el caso industrial, evaluaciones comparativas en base a la literatura del área (investigación exploratoria). Para fines de este trabajo se selecciona la evaluación de la literatura del área en base al análisis de varios artículos y libros técnicos del área, y la experiencia de un primer contacto con la tecnología en la construcción de una solución MDE donde se utilizaron varias de las herramientas MDE para dar solución a un problema real.

1.5 Estructura de la Memoria

La memoria se encuentra organizada en siete capítulos.

- El Capítulo 1 consta de una breve introducción del proyecto, donde se establece los antecedentes, motivación, objetivos y alcance del proyecto, y la metodología utilizada.
- El capítulo 2 presenta el marco teórico cubierto para la realización del proyecto, en él se presentan varios aspectos de MDE, como tendencias de desarrollo, artefactos MDE con DSLs, descripción del espacio tecnológico donde se describen varios entornos de desarrollo que se

pueden emplear, y técnicas de diseño de DSLs. Este capítulo también cubre aspectos de Análisis de Dominio como mecanismo para comprender y representar el dominio de la aplicación, análisis de las particularidades de las metodologías de desarrollo de software tradicional y ágil, finalmente se hace una revisión de aspectos de desarrollo de software por el usuario final.

- El Capítulo 3 hace referencia al impacto de la utilización de MDE en la empresa, en base a la revisión de artículos del área. Los ámbitos cubiertos son: experiencias con MDE, aspectos de calidad en MDE, artefactos de soporte al proceso de desarrollo con MDE y procesos de desarrollo de MDE en base a DSL. Se finaliza con un análisis personal de los artículos revisados.
- En el Capítulo 4 se establece las directrices que debe adoptar la metodología a proponer.
- En el Capítulo 5 se estructura la arquitectura de la metodología y se describen las actividades que se deben desarrollar.
- En el Capítulo 6 se propone un proceso de desarrollo de aplicaciones ligeras con artefactos MDE por parte del usuario final.
- El Capítulo 7 consta de las conclusiones obtenidas a partir del trabajo realizado, incluyendo líneas de investigación y trabajos futuros a realizar.

Capítulo 2

Marco Teórico

Este capítulo permite cubrir los aspectos más relevantes de MDE, DSLs, análisis de dominio, metodologías de desarrollo de software tradicional y ágil, y desarrollo por el usuario final. Debido a que la metodología a proponer se fundamenta en el desarrollo de software con enfoque MDE en base a DSL, para lo cual se consideran las principales recomendaciones de las metodologías tradicionales y ágiles. El análisis de dominio se emplea para determinar las características más relevantes de un dominio de sistemas lo que permitirá delimitar el área de influencia del DSL. Mientras que el desarrollo por el usuario final permitirá determinar el comportamiento y las actividades de los usuarios desarrolladores de aplicaciones con los artefactos MDE.

2.1 Ingeniería de Software Dirigida por Modelos

2.1.1 Fundamentos de MDE

Actualmente el desarrollo de software utiliza un gran abanico de modelos (documentación) que se los emplea para representan las diferentes vistas de la arquitectura de un sistema. Se pueden tener modelos para representar los elementos constitutivos de un sistema, desde elementos básicos como datos, procesos, eventos, interfaces, hasta modelos más completos que permitan representar la distribución de los datos y la funcionalidad a través de toda la red de datos. Se puede elegir construir modelos con cierto grado de libertad, como utilizar cajas dibujadas a mano sobre papel [61], o con cierto grado de sofisticación como utilizar complejos diagramas UML producidos con herramienta de modelado automático. Una vez elaborado el código del sistema informático, el código es sometido a un proceso de mantenimiento continuo y los modelos que sirvieron para conseguirlo prácticamente quedan obsoletos y sin reflejar la realidad cambiante del software

MDE es un nuevo paradigma que conecta más estrechamente el modelo a la aplicación, el modelo no sólo encapsula el diseño de la aplicación, sino que se lo utiliza para generar la implementación del código. El enfoque MDE ha sido propuesto con la finalidad de proveer técnicas y herramientas para tratar con modelos de forma automática en el proceso de desarrollo de software. El criterio fundamental de MDE es la utilización de la abstracción que permite construir modelos para representar el sistema a desarrollar y la posibilidad de utilización de estos modelos para la generación automática de código. Un enfoque de ingeniería dirigida por modelos tiene que especificar los lenguajes de modelado, los modelos, las traducciones entre los modelos, los lenguajes utilizados para el efecto, y el proceso utilizado para coordinar la construcción y la evolución de los modelos; por lo que para garantizar los beneficios de la utilización de modelos para desarrollar software, se requiere el apoyo de herramientas de gran alcance [44].

2.1.2 Tendencias de desarrollo en MDE

Actualmente existen dos tendencias de MDE. La una mediante el empleo de los principios de MDA propuesta por el OMG y la otra mediante la utilización de DSLs que lo llamaremos enfoque MDE con DSL. Con respecto a la primera tendencia, cuatro principios subyacen en la opinión del OMG de MDA [4]:

- Los modelos expresados en una notación bien definida son la piedra angular para entender las soluciones de los sistemas a escala empresarial.
- La construcción de sistemas puede organizarse en torno a un conjunto de modelos mediante la imposición de una serie de transformaciones entre modelos, organizados en un marco arquitectónico de capas y transformaciones.
- Un soporte formal para describir los modelos en un conjunto de metamodelos facilita la integración significativa y transformación entre modelos, y es la base para la automatización a través de herramientas.
- La aceptación y adopción generalizada de este enfoque basado en modelos requiere estándares de la industria para proporcionar transparencia a los consumidores, y fomentar la competencia entre los proveedores.

Para apoyar estos principios, OMG ha definido un conjunto específico de capas y transformaciones que proporcionan un marco conceptual y el vocabulario para MDA [57]. En particular el OMG identifica tres tipos de modelos: Modelo Independiente de la Computación (CIM, Computación Independent Model), Modelo Independiente de la Plataforma (PIM, Platform Independent Model), Modelo Específico de la Plataforma (PSM, Platform Specific Model). La Figura 2.1 ilustra el enfoque de la MDA.

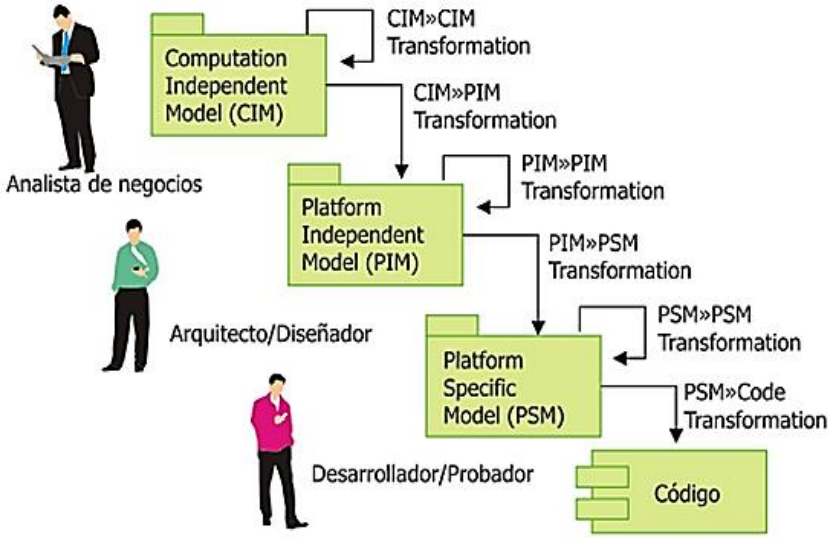


Figura 2.1: Enfoque MDA [74]

Para MDA una "plataforma" sólo tiene sentido en relación con un punto de vista particular, en otras palabras, el PIM de una persona es el PSM de otra persona. Por otro lado, cada uno de los niveles de modelos tiene un propósito específico en el desarrollo de un sistema, para lo cual cada nivel maneja su propio conjunto de herramientas para conseguirlo, como se señala en la Figura 2.2:

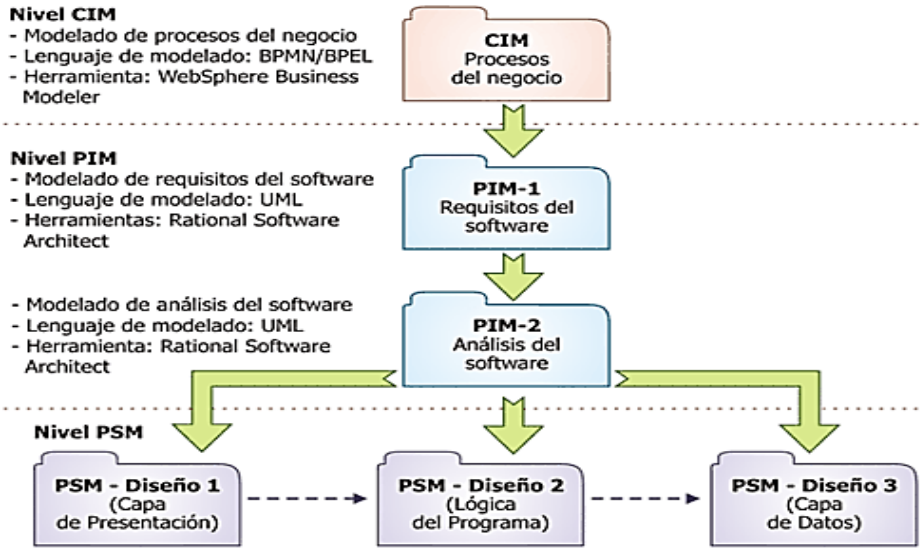


Figura 2.2: Herramientas utilizadas en cada uno de los niveles de MDA [74]

Mientras que con respecto a la segunda tendencia, MDE pretende sobrepasar los marcos de MDA debido a que en la actualidad los sistemas son cada vez más complejos y la utilización de UML puede ya no ser suficiente para modelar toda la complejidad del software para dominios especializados, debido a que puede resultar muy genérico tal como un Lenguaje de Programación de Propósito General (GPL,

General-Purpose Programming Language), por otro lado los mecanismos de adaptación de UML (perfiles) son a veces demasiado "pesados" y van muy ligados a herramientas concretas. Por ello para proyectos específicos, en un área concreta, es más productivo usar un lenguaje que tenga abstracciones más orientadas a ese dominio, como un DSL. Un DSL sitúa a los modelos como el centro del desarrollo de software, pero plantea el problema de encontrar lenguajes de modelización que contengan la suficiente semántica para expresar de manera formal niveles altos de abstracción en cada etapa del proceso del ciclo de vida. La ventaja de los DSLs es que proporcionan primitivas útiles en un dominio, que de otra manera habría que expresarlas en un GPL. Un DSL es un lenguaje desarrollado para hacer frente a la necesidad de un determinado dominio, por ejemplo en los ámbitos de seguros, salud, transporte, o en aspectos concretos del sistema como, datos, presentación, lógica de negocio o flujo de trabajo. La idea es tener un lenguaje con conceptos que se encuentren centrados en un dominio específico, esto permite a los lenguajes de alto nivel mejorar la productividad del desarrollador y la comunicación con los expertos de dominio; en muchos casos incluso es posible dejar que los expertos de dominio utilicen los DSLs para desarrollar sus propias aplicaciones [37].

2.1.3 Descripción de los principales artefactos del enfoque MDE con DSL

Para automatizar el desarrollo de aplicaciones con un enfoque MDE se requiere construir varios productos o artefactos.

2.1.3.1 Metamodelo

Un metamodelo define la sintaxis abstracta de un DSL. El metamodelo es un modelo que describe las características de un conjunto de modelos, es decir el metamodelo establece la estructura del modelo [29]. A su vez, dado que un metamodelo es también un modelo, un metamodelo es expresado también en un lenguaje que se denomina “lenguaje de metamodelado” o “lenguaje de definición de modelos”. Un lenguaje gráfico de definición de modelos, como podría ser UML, proporciona una idea general e intuitiva, generalmente fácil de comprender por parte del usuario, de los conceptos del dominio que se está modelando (ya sean conceptos del mundo real, componentes software, etc.) y de las relaciones entre ellos. Sin embargo, estos lenguajes no son lo suficientemente expresivos como para definir toda la información relevante de este dominio de referencia, por ello el uso de lenguajes gráficos de modelado se complementan con el uso de otros lenguajes, normalmente textuales aunque basados en un formalismo lógico, que permite definir la información precisa y no ambigua [29]. El lenguaje OCL, acrónimo de Object Constraint Language es el más conocido de ellos, propuesto por el OMG [68]. OCL juega un papel importante en la creación de DSLs.

En este punto se podría plantear con qué lenguaje se define el lenguaje de metamodelado y así sucesivamente. Dado que el lenguaje del metamodelo también tiene su sintaxis abstracta y concreta, la cuestión es cómo definir el metamodelo de esta sintaxis abstracta, que realmente sería un meta-metamodelo (un modelo de un metamodelo, o lo que es lo mismo, un modelo de un modelo de un modelo). A esta cuestión se la puede responder con la noción de Arquitectura de Cuatro Niveles usada

tradicionalmente para establecer la relación entre modelos y metamodelos [6]. En ella se establecen los siguientes niveles:

- **M0. Nivel de datos del usuario o del mundo real:** caracteriza los objetos del mundo real que son manipulados por el software. El término "objeto" es utilizado en un amplio sentido para significar también "procesos", "conceptos", "estados", etc.
- **M1. Nivel de modelo:** caracteriza a los modelos que representan los datos del nivel M0.
- **M2. Nivel del metamodelo:** caracteriza a metamodelos que describen los modelos del nivel M1.
- **M3. Nivel de metametamodelo:** caracteriza a los metametamodelos que describen los metamodelos del nivel M2.

De acuerdo a esta arquitectura, un metametamodelo se define con los mismos conceptos que son definidos (definición circular), como se muestra en la Figura 2.3 [29].

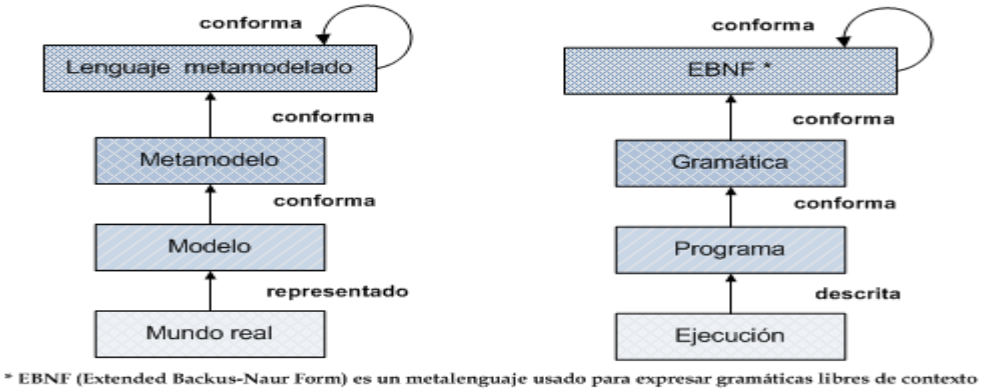


Figura 2.3: Arquitectura de cuatro niveles para metamodelos y gramáticas

2.1.3.2 DSL

Una vez establecida la sintaxis abstracta expresada por lo general en un diagrama de clases, se tiene que establecer la sintaxis concreta del DSL, la sintaxis concreta realiza una representación de la sintaxis abstracta (metamodelo), ya sea gráficamente (por ejemplo, un circuito eléctrico), textualmente (por ejemplo, una consulta SQL), o una combinación de ambos. Se debe señalar que también un DSL se define por su semántica, es decir el significado del modelo [96].

Los editores gráficos o textuales se construyen a partir del metamodelo. Es recomendable utilizar elementos textuales y gráficos con los cuales estén familiarizan los usuarios finales, de esta manera se evitaría utilizar elementos técnicos que son comunes para los ingenieros de software con habilidades especializadas en MDE, y que puede ser difícil para los usuarios finales [16]. En el caso de construir un editor gráfico, es habitual que el diseñador defina una serie de modelos con información sobre 1) los elementos gráficos que se mostrarán en los diagramas; 2) la paleta de herramientas del editor; y 3) La relación entre los dos modelo anteriores y el metamodelo de partida, así este último modelo define una relación unívoca (mapping) entre cada elemento del metamodelo (sintaxis abstracta) y la herramienta

de la paleta con la que añadirá dicho elemento al diagrama [29]. La definición completa de un lenguaje de modelado necesariamente incluye cuatro aspectos, señalados en la Tabla 2.1.

Tabla 2.1: Los cuatro aspectos de la definición de un lenguaje de modelado [29]

Semántica	**Interpretación** (correspondencia semántica)	Relación de correspondencia con la realidad representada en el modelo.
	Derivación (teoría deductiva)	Relación con otros modelos derivables por medio de reglas deductivas de transformación.
Sintaxis	**Concreta** (notación textual/gráfica)	**Aspecto:** conjunto de símbolos textuales/gráficos que se utilizan en los diagramas.
	Abstracta (metamodelo) (estructura lógica)	**Estructura lógica:** reglas que especifican las expresiones bien formadas de las características del modelo.

2.1.3.3 Transformación de modelos

Cualquier propuesta MDE implica trabajar con varios modelos relacionados entre sí, lo cual requiere mucho esfuerzo para abordar la complejidad que entrañan las diferentes tareas relacionadas con la gestión de modelos, como refinarlos, validarlos, refactorizarlos, etc. De hecho uno de los principales retos que deben abordarse para potencializar el papel de los modelos en la ingeniería del software es automatizar dichas tareas para reducir la complejidad asociada, y la forma de automatizar no es otra cosa que desarrollar transformaciones de modelos que las soporten [29].

La literatura de MDE señala muchas definiciones de transformación de modelos, entre ellas:

- En la guía de referencia para MDA [67], se dice que “la transformación de modelos es el proceso de convertir un modelo en otro del mismo sistema”.
- Por su parte, Kleppe en [45] define una transformación de modelos como la generación automática del modelo de salida a partir de un modelo de entrada, el cual es conforme a la definición de transformación.
- Tratt en [95] proporciona otra definición: “una transformación de modelos es un programa que muta un modelo en otro, en otras palabras, algo parecido a un compilador”

Así por ejemplo, una vez que se tenga creado el DSL gráfico o textual, el usuario de éste artefacto, podrá diseñar un modelo expresado en el DSL creado para dar solución a su problema, por tanto el siguiente paso es ser capaces de utilizar el modelo diseñado por el usuario para desarrollar software en un determinado GPL, por lo que en este caso, los generadores de código constituyen un tipo de transformadores de modelos. Para ello se parte del modelo de alto nivel expresado por el DSL que constituye la especificación del sistema requerido, a un proceso de refinamientos sucesivos en modelos con menor nivel de abstracción que combinan dicha especificación con las características técnicas de la plataforma tecnológica elegida, hasta que su nivel de detalle permita utilizar esos modelos como entradas como por ejemplo para generar código que implementa el sistema [29].

Por ello puede entenderse a la transformación de modelos como la manipulación de modelos para cubrir diferentes niveles de abstracción de una aplicación software [65]. Se puede realizar transformaciones Modelo-a-Modelo (M2M), Modelo-a-Texto (M2T), o también transformaciones "in-place" que cambian un modelo, reestructurándolo u optimizándolo. Para ello se requiere un modelo fuente y modelo destino, la transformación se la realiza en base a la definición de reglas de transformación. Por ejemplo un modelo de clases puede ser transformado a una base de datos relacional con código de nivel de acceso ODBC o JDBC O ProC [50].

2.1.3.4 Generadores de código

El paradigma MDE permite la automatización del desarrollo de software mediante la aplicación de transformaciones que se aplican sobre los modelos de alto nivel que describen un sistema. Un tipo particular de transformaciones son las transformaciones Modelo-a-Texto, también conocidas como generadores de código. Estos lenguajes comparten habitualmente una serie de características comunes, como son, cómo describir las reglas y plantillas de generación, cómo iterar sobre el modelo, la consistencia incremental cuando se modifica el código generado a mano o el formateo del texto generado. En esencia, el generador de código basado en modelos interpreta el modelo a la luz de las decisiones del diseño adecuado y estrategias arquitectónicas para generar código en una plataforma tecnológica específica [50].

2.1.3.5 Mecanismos de análisis de modelos

El análisis de modelos permite determinar si los modelos que se han construido son correctos, no sólo desde un punto de vista estructural, sino también funcional. Al ser los modelos de más alto nivel de abstracción que el código, resulta más fácil analizarlos a través de un proceso de validación y verificación.

2.1.4 Espacio Tecnológico de MDE

A continuación se describen los principales entornos de desarrollo y herramientas utilizadas en el ámbito MDE que podrían ser utilizadas para construir una solución basada en MDE:

Tabla 2.2: Entorno de desarrollo y herramientas utilizadas en MDE

Entorno de Desarrollo	Plataformas de Metamodelado (Sintaxis Abstracta)
Arquitectura OMG	MOF
Arquitectura Eclipse/EMF	Ecore/OCL
MetaCase/MetaEdit+	GOPRR
Microsoft DSL Tools	Domain Model
Proyecto TCS	KM3
Propietario	MetaDepth

Editores (Sintaxis Concreta)	Transformación de Modelos	Lenguajes de Generación de Código
Editores Gráficos	ATL	Acceleo
GMF	QVT	JET
EuGENia	RubyTL	MOFScript
Graphiti		XPand
Editores Textuales		Mof2Text
Xtext		
EMFText		
TCS		

2.1.4.1 Entornos de Desarrollo

Arquitectura de metamodelado de OMG

Fue establecida por el OMG [68] en la definición de UML y ha habido varias versiones (ahora UML 2). UML es un lenguaje visual de modelado de software cuyo metamodelo está definido con el lenguaje de metamodelos MOF (Meta Object Facility). Dado que MOF y UML comparten todas las construcciones relacionadas con el modelado de clases, se organizó la definición del metamodelo de UML en dos partes (esto es dos paquetes) con el fin de reutilizar elementos: InfraStructure y SuperStructure.

El paquete InfraStructure incluye el paquete Core con todos los elementos comunes a UML y MOF (clases, propiedades, generalización) junto con otro paquete que define un mecanismo de extensión de cualquier lenguaje definido con MOF que es denominado “Perfiles” (Profiles), este mecanismo fue ideado para conseguir que UML fuese aplicable a cualquier dominio, no solo al software [28]. Esta idea no ha tenido el éxito esperado y en la actualidad existe un consenso sobre la conveniencia de crear DSLs en vez de crear perfiles UML [53]. El paquete SuperStructure, además de completar la especificación relacionada con el modelo de clases, define el resto de elementos y notación de UML (casos de uso, máquinas de estados, etc.).

Arquitectura de metamodelado de Eclipse/EMF

Modeling Project es el nombre del proyecto Eclipse destinado a ofrecer soporte a la construcción de herramientas y soluciones MDE. Todos estos proyectos se articulan en torno a EMF (Eclipse Modeling

Framework) que proporciona los mecanismos básicos para manejar metamodelos, esto es, implementa lo que se puede denominar "una arquitectura de metamodelado" [29].

MetaEdit+

El Workbench MetaEdit+ de la empresa MetaCase [59], proporciona un conjunto de herramientas para diseñar un lenguaje de modelado. Para lo cual plantea tres pasos: 1) definir los conceptos y reglas del lenguaje gráficamente basadas en formas (form-based), 2) dibujar la notación con un editor de símbolos o importar elementos gráficos existentes, y 3) hacer los generadores para producir el código requerido, configuraciones, análisis, probar los datos, etc. MetaEdit+ utiliza GOPRR como arquitectura de metamodelado.

Microsoft DSL Tools

DSL Tools está diseñada para la creación de lenguajes de dominio específico en el universo Microsoft. DSL Tools se instala en el entorno de desarrollo integrado Visual Studio (VS IDE) y necesita como prerrequisito el kit de extensión SDK que proporciona Microsoft para su IDE. La implementación de DSL con la herramienta de metamodelado DSL Tools consiste en la definición de una sintaxis abstracta que modele todos los elementos del lenguaje en base a la cual se define una sintaxis concreta. No obstante Microsoft no utiliza esta denominación para estas sintaxis, sino que utiliza los conceptos Domain Model y Designer, lo que traducido sería Modelo de Dominio y Diseñador, para referirse a las sintaxis abstracta y concreta respectivamente.

Proyecto TCS

El proyecto de Lenguajes TCS [94] es un proyecto de Eclipse (por lo general con naturaleza ATL), abarca: un lenguaje independiente de metamodelado KM3 que permite definir la sintaxis abstracta del lenguaje, un modelo TCS que define la sintaxis concreta del lenguaje, opcionalmente este proyecto también contiene: restricciones (por ejemplo OCL, como parte de una transformación ATL) y un compilador (por ejemplo, ATL VM Code Generator para ATL).

2.1.4.2 Plataformas de Metamodelado

Para representar un metamodelo es necesario un lenguaje que proporcione un conjunto de construcciones destinado a este fin, entre ellos se tienen:

MOF

MOF es el lenguaje de metamodelado de la OMG en la definición de UML. MOF en particular incluye la parte destinada al modelado de clases, junto con algunas facilidades relacionadas con el soporte de la reflexión y el manejo de identificadores. Debido a que se entendió que MOF era muy complicado y que

en la mayoría de ocasiones era suficiente con una pequeña parte de lo que ofrecía se definió el lenguaje EMOF (Essential MOF) [29].

Ecore

Es el más utilizado en la actualidad, el cual es parte de la arquitectura de metamodelado de EMF [37] que proporciona Eclipse para crear entornos y herramientas destinadas a aplicar MDE. Ecore fue creado como implementación de MOF de la OMG (MOF, 2005). Sin embargo, Ecore fue diseñado como un subconjunto de MOF con las construcciones básicas para construir metamodelos. Se entendió que MOF era muy complicado y que en la mayoría de ocasiones era suficiente con una pequeña parte de lo que ofrecía [29]. Ecore proporciona los conceptos orientados a objetos para la creación de metamodelos: clases, atributos, agregación, referencias y generalización, excluye las asociaciones. En un metamodelo en ocasiones se tiene que incluir reglas que determinan cuando un modelo está bien formado, esto se logra con OCL (Object Constraint Language), el cual ha sido propuesto por el OMG (OMG, 2012), y recientemente adoptado por el estándar ISO/IEC (ISO, 2012), OCL es un lenguaje de definición de expresiones, a partir de las cuales pueden definirse condiciones que un modelo debe satisfacer.

MetaDepth

La característica principal de MetaDepth es su soporte para un número arbitrario de meta-niveles ontológicos. Esto hace que MetaDepth sea especialmente útil para definir lenguajes multi-nivel. Es decir, en lugar de adherirse a la típica arquitectura de metamodelado de 3 capas (modelo/metamodelo /metametamodelo), los diseñadores pueden definir cualquier número de niveles de metamodelado. Un ejemplo es la relación entre los objetos y clases en UML. Uno puede pensar en diagramas de objetos como instancias de los diagramas de clases. Por lo tanto los diagramas de clase+objetos pueden ser considerados como un lenguaje multi-nivel, y se pueden definir en MetaDepth a través de un meta-modelo único [22].

KM3

KM3 (Kernel MetaMetaModel) proporciona una solución relativamente simple para definir un DDMM (Domain Definition MetaModel) de un DSL. El lenguaje KM3 pretende ser un lenguaje de definición de metamodelos textual ligero que permite una fácil creación y modificación de metamodelos [39]. Los metamodelos expresadas en KM3 presentan propiedades adecuadas para ser comprendidos de forma relativamente fácil. El formalismo es lo suficientemente rico como para apoyar la información esencial. Los metamodelos expresados en KM3 pueden ser fácilmente convertidos a/o desde otras notaciones como Emfatic o XMI.

2.1.4.3 Sintaxis Concreta

Eclipse Graphical Modeling Framework (GMF)

GMF es un plug-in de Eclipse que da soporte al desarrollo de editores gráficos de modelos a partir de metamodelos. Para construir estos editores, GMF precisa que el diseñador defina una serie de modelos con información sobre: 1) los elementos gráficos que se mostrarán en los diagramas, 2) la paleta de herramientas del editor, y 3) la relación entre los dos modelos anteriores y el metamodelo de partida.

EuGENia

Es una herramienta desarrollada en el contexto de la familia de lenguajes Epsilon [29]. Su finalidad es desarrollar editores gráficos de modelos basados en GMF. Dada la complejidad del uso de los modelos GMF, EuGENia trata de facilitar a los usuarios, especialmente a los menos experimentados, el desarrollo de este tipo de editores [48]. EuGENia parte de un metamodelo, especificado en formato Ecore o Emfatic, enriquecido con una serie de anotaciones específicas GMF, y a partir de él, generar automáticamente los tres modelos utilizados por GMF para implementar el editor gráfico correspondiente, esto es: 1) modelo que define los elementos gráficos (gmfgraph), 2) el que define la paleta de herramientas (gmftool), y 3) el que establece las correspondencias entre los dos anteriores y cada uno de los elementos del metamodelo de partida (gmfmap).

2.1.4.4 Creación de Editores Textuales

Xtext

Es un framework dirigido a la creación de lenguajes textuales de modelado. Asocia una representación textual (sintaxis concreta) a los conceptos y relaciones del lenguaje, especificados en el metamodelo (sintaxis abstracta), y proporcionan los mecanismos que posibilitan la edición y manipulación de los modelos textuales. Xtext es un proyecto open-source que forma parte de Eclipse Modeling Project [100] y cuyo desarrollo está liderado por la empresa Itemis AG [35]. Es un plug-in de Eclipse basado en EMF, lo que permite que los modelos textuales creados con editores Xtext puedan ser utilizados por otros plug-ins de Eclipse, también basados en EMF, como ATL, JET o MOFScript.

EMFText

Es un plug-in de Eclipse que permite definir una sintaxis textual para un metamodelo Ecore. Dentro de sus principales características se destacan que el código generado no contiene dependencias de EMFText, todo el código es totalmente personalizable, permite la generación automática de sintaxis por defecto (HUNT y Java-like), permite realizar el análisis completo de la sintaxis para advertir sobre posibles problemas [25].

TCS

TCS es un componente de Eclipse/GMT que permite la especificación de una sintaxis concreta textual de un DSLs adjuntando información sintáctica de los metamodelos [93]. Con TCS es posible realizar un análisis de sentencias DSL (Texto-a-Modelo) y (Modelo-a-Texto). TCS ofrece un editor de Eclipse que cuenta con resaltado de la sintaxis, un esquema, e hipervínculos por cada sintaxis DSL que es representada en el DSL.

2.1.4.5 Transformación de Modelos

ATL

ATL (Atlas Transformation Language) es un lenguaje de transformación de modelos desarrollado por el grupo de investigación AtlanMod (INRIA & LINA) como respuesta a la propuesta OMG MOF/QVT RFT (Request For Proposal) y forma parte de la plataforma AMMA (Atlas Model Management Architecture). ATL es un lenguaje híbrido que implementa los paradigmas declarativo e imperativo [40]. Sus autores recomiendan el uso del estilo declarativo, puesto que permite expresar de manera sencilla las relaciones existentes entre los elementos del modelo origen y destino, mientras que las construcciones imperativas facilitan la codificación de transformaciones complejas. Una transformación ATL está compuesta por reglas mediante las que se define cómo se crean e inicializan los elementos del modelo (o modelos) destino a partir de elementos del modelo/s origen.

QVT

El OMG como creador de la propuesta MDA. En un intento por unificar las diferentes propuestas para el desarrollo de transformaciones, a finales de 2008 produjo el estándar para el mismo, Query/View/Transformations (QVT). Esta especificación define una familia de lenguajes para la definición de transformaciones: dos lenguajes para el usuario final (QVT Operational Mappings [imperativo] y QVT Relations [declarativo]), y un lenguaje de bajo nivel que puede verse como byte-code de QVT (QVT Core) [75].

RubyTL

RubyTL (Universidad de Murcia) es un lenguaje de transformación basado en reglas, que ha sido construido como un lenguaje embebido dentro de Ruby. Es un lenguaje de transformaciones híbrido, cuya parte declarativa está basada en bindings al estilo de ATL, mientras que la parte imperativa viene dada por las construcciones proporcionadas por el lenguaje Ruby [79]. Actualmente RubyTL es un lenguaje de transformación estable, que dispone de un entorno de programación para Eclipse, llamado AGE, que incluye otra serie de lenguajes embebidos para realizar tareas típicas del Desarrollo de

Software Dirigido por Modelos (DSDM) como son la generación de código, la validación o la creación de otros lenguajes.

2.1.4.6 Lenguajes de Generación de Código

Acceleo

Acceleo es una implementación del estándar MOF-to-Text del OMG realizada por la empresa Obeo, forma parte del proyecto M2T de Eclipse [2]. Ofrece varias ventajas: facilidad para realizar el proceso de generación de código, alta capacidad de personalización, interoperabilidad, gestión de la trazabilidad, etc. Acceleo utiliza el mecanismo de *plantilla* para la generación de código (transformación M2T). La plantilla se construye para manipular modelos que conforman a cierto metamodelo. La idea básica es recorrer el modelo generando texto fijo y dejando "huecos" que se rellenarán en función del modelo de entrada cuando se ejecute la transformación. Normalmente la ejecución de la transformación la realiza un motor de plantillas que interpreta la plantilla, generando el texto y leyendo el modelo de entrada para rellenar los huecos.

JET

JET (Java Emitter Templates) es una herramienta para generar uno o más ficheros de salida a partir de un modelo de entrada haciendo uso de plantillas [29]. Es decir se parte de una plantilla (por ejemplo, el esqueleto genérico con la estructura de una clase C++) y a partir de la información contenida en el modelo de entrada de la transformación (por ejemplo un modelo de clases UML), se podrá generar el código de las clases C++ representadas en el modelo UML, con los nombres de los paquetes, métodos y atributos correspondientes. De esta manera se pueden particularizar las plantillas para cada transformación concreta. Por ello, las plantillas se componen no solo de texto normal (que es común a todas las transformaciones de esta plantilla) sino también de etiquetas JET.

MOFScript

MOFTScript fue desarrollado en el contexto del Proyecto Europeo del FP6 MODEL WARE y se presentó al RFP (Request For Proposal) para la definición del estándar de transformaciones de Modelo-a-Texto del OMG [60]. MOFScript proporciona un conjunto de plug-ins para el entorno de programación Eclipse que engloban un metamodelo, una sintaxis textual concreta, un editor con coloración de sintaxis y un motor de ejecución de transformaciones.

Xpand

Xpand puede generar código tomando como base modelos DSL definidos con Xtext. Es un lenguaje de plantillas tipado estáticamente con invocación de plantillas polimórficas, programación orientada a aspectos, extensiones funcionales, validación de modelos, etc.

Mof2Text

MOF Modelo de Transformación Modelo a Texto (Mof2Text o MOFM2T) es una especificación de un lenguaje de transformación de modelos de la OMG. MOFM2T es parte de la arquitectura basada en modelos del OMG (MDA), reutiliza conceptos de MOF, utiliza arquitectura de metamodelado del OMG. Mientras MOFM2T se utiliza para expresar transformaciones M2T, QVT de OMG se utiliza para expresar transformaciones M2M.

2.1.5 Técnicas de diseño de DSL

Desde la perspectiva del usuario de las herramientas MDE, el DSL constituye el elemento fundamental y la base para que el usuario pueda desarrollar sus aplicaciones, a través de la implementación de una notación textual o gráfica o una combinación de las dos, que le resulte familiar en su dominio de aplicación. Voelter en [96] considera que el desarrollo de software con DSL requiere un proceso compatible, es decir, que mucho de lo que es requerido para construir el DSL es similar a lo que es requerido para trabajar en el desarrollo de la aplicación objetivo, un proceso viable deber ser establecido entre quien construye los artefactos reutilizables y quien los utiliza. Los requerimientos tienen que fluir en una dirección (construcción de artefactos MDE) y un final estable, probable y un producto documentado tiene que ser liberado en la otra dirección (desarrollo de la aplicación con artefactos MDE), en este sentido el utilizar DSL puede ser un cambio fundamental que envuelve todo.

Acerca de los requerimientos para el DSL, se plantean varias preguntas: ¿Qué debería expresar el DSL?, ¿Cuáles son las abstracciones relevantes y sus notaciones?, estas son preguntas no triviales, de hecho son las preguntas claves del desarrollo con DSL. Esto requiere un tanto de experiencia en el dominio, reflexión e iteración. La clave del problema no es entender un problema sino más bien un conjunto de problemas. Entender el grado y naturaleza de este conjunto de problemas puede emplear mucho trabajo. Voelter en [96] plantea tres escenarios básicos para obtener las primitivas del DSL:

- El primero donde el lenguaje del DSL es a menudo obtenido de un framework existente, librería, arquitectura o patrón de arquitecturas. El conocimiento existe y construir el DSL está principalmente relacionado al traslado del conocimiento al nuevo lenguaje. Esto es complementado con la revisión de diferentes fuentes que consideran el mismo domino de aplicación (proceso inductivo).
- El segundo escenario es el proporcionado por un experto del dominio (proceso deductivo). Por ejemplo en dominios muy comunes, como el manejo de seguros, en la ciencia o logística, los expertos de dominio por su experiencia en dicho ámbito son absolutamente capaces de precisar el conocimiento del dominio.
- En el tercer caso, el conocimiento del dominio no está disponible, se tiene que hacer un análisis del dominio a través de los requerimientos proporcionados por los stakeholders y analizando aplicaciones similares existentes. Las personas pueden ser expertas, pero muchas veces no son

capaces de contextualizar el dominio de manera estructurada, en cuyo caso los diseñadores de lenguajes deben proporcionar una estructura consistente, necesaria para definir el lenguaje.

Los DSLs incluyen lenguajes dedicados para ser aplicados en muchos ámbitos como por ejemplo para la ingeniería web, especificación de requisitos [42], modelado de negocios, consulta de datos a través de SQL. Estos lenguajes específicos no sólo son aplicables en el ámbito de la informática, son también útiles en diversas áreas y disciplinas, como la biología, la física, la gestión o la educación, donde los expertos de dominio no son necesariamente expertos en computación. Para desarrollarlos, existen varias técnicas e incluso herramientas que cubren todo o parte de este proceso. Por ejemplo en [16] se resume dos técnicas para construir el metamodelo, una considerada el enfoque tradicional y la otra propuesta por el grupo de investigación; también se presenta una técnica denominada Modelado Dirigido por Ejemplos.

Primera técnica

Esta técnica hace referencia al enfoque "tradicional" para construir un metamodelo, la Figura 2.4 tomada de [16], considera que primero se debe realizar una reunión de recolección de requisitos con los expertos del dominio (1), requisitos que son expresados utilizando lenguaje natural, a continuación se construye un metamodelo, representado como un diagrama de clases con restricciones adicionales (2), posteriormente el metamodelo es revisado por los usuarios finales (3). Si se observan defectos, se proporciona la retroalimentación correspondiente y el metamodelo se vuelve a trabajar. Este proceso se itera hasta que el metamodelo reúna todos los conceptos del dominio, a partir del cual las otras herramientas MDE se puedan desarrollar (editores, mecanismos de transformación de modeles, generadores de código). Se debe considerar que a veces los defectos pueden ser encontrados sólo una vez que la herramienta se encuentre disponible, es cuando los usuarios finales detectan elementos que faltan, por lo que se requiere reiniciar el proceso.

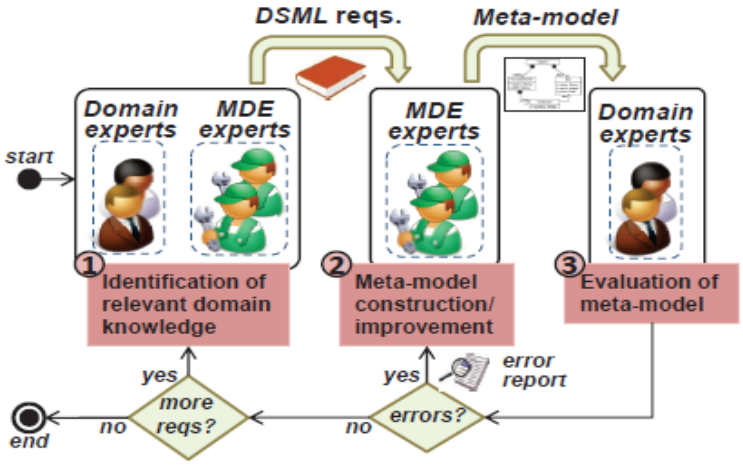


Figura 2.4: Esquema tradicional de construcción de un metamodelo

Segunda técnica

La técnica propuesta por el grupo de investigación considera un proceso colaborativo utilizando ejemplos para la captura de requisitos para construir el metamodelo, con este mecanismo se logra alejar al usuario final de tener que lidiar con conceptos abstractos y demasiado técnicos, los cuales son dejados a los desarrolladores. Los ejemplos (esbozos) son desarrollados por el usuario final utilizando su propia nomenclatura y representan el funcionamiento del sistema desde su propio punto vista, este proceso permite involucrar aún más al usuario debido a que se trata de proceso evolutivo y que además es apoyado por una herramienta para automatizar el desarrollo del metamodelo.

Esta técnica propone cinco pasos, como se señala en la Figura 2.5 especificada en [16]: (1) proceso de bootstrapping – provisión inicial de ejemplos (esbozos), (2) inducción del meta-modelo, (3) evaluación y discusión de cambios, (4) votación de cambios incorporados, y (5) obtención de la versión final del lenguaje a ser desarrollado.

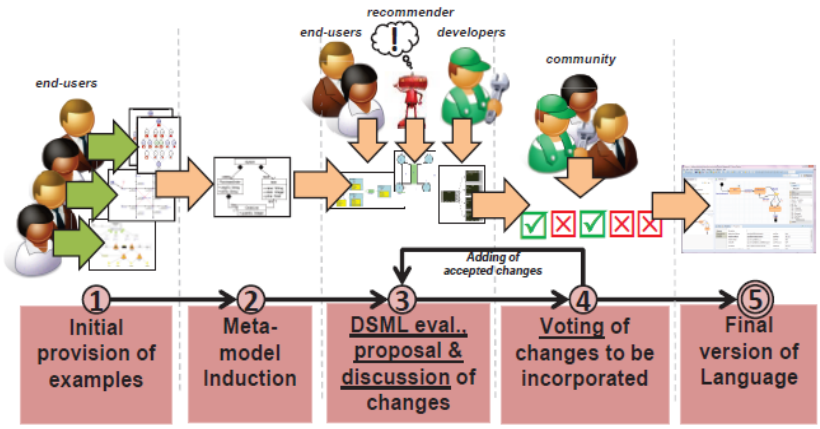


Figura 2.5: Proceso colaborativo manejado por ejemplos

Tercera técnica

Bąk en [7] propone una técnica que se encuentra en etapa de desarrollo, denominada EDM (Example-Driven Modeling) cuyo enfoque se centra en utilizar sistemáticamente ejemplos para elicitar, modelar, verificar y validar el conocimiento de negocios complejos, bajo la consideración que el modelado del dominio es una de las actividades clave en la ingeniería de requisitos, situación que se podría utilizar en el diseño de metamodelos. Esta técnica pretende realizar la transferencia de conocimiento del funcionamiento del sistema entre el Sujeto Experto en la Materia (SME, Subject Matter Expert) y el Analista del Negocio (BA, Business Analyst) a través de una serie de ejemplos expresados de forma verbal. Esta alternativa surge debido a la búsqueda de nuevos mecanismos para obtener requisitos,

puesto que ésta tarea resulta complicada porque muchos de los conocimientos son difíciles de obtener y comunicar [36]. Se considera que a pesar que el uso sistemático de modelos ofrece muchos beneficios en la ingeniería de software [30], éstos están orientados a expertos altamente capacitados que trabajan con abstracciones y no a un público más amplio [24]. Sin embargo, la mayoría de los interesados que se benefician de los modelos son los usuarios no expertos. Razón por la cual se puede entender y trabajar con modelos a través de ejemplos.

La investigación propuesta plantea el escenario en que una persona que requiere la construcción de un sistema, trata de transmitirle el conocimiento del dominio del problema al desarrollador a través de una serie de ejemplos del funcionamiento del entorno, bajo la consideración que los seres humanos aprenden mejor a partir de ejemplos de la información que requiere y la construcción de modelos mentales [29], donde se determina que los seres humanos aprenden abstracciones y a resolver problemas analógicos más eficazmente si el nuevo material se presenta con múltiples ejemplos, también se demostró que múltiples ejemplos mejoran la calidad de las abstracciones construidas. Con respecto a la validación, se considera que los desarrolladores han estado utilizando durante décadas ejemplos para probar software. Los enfoques modernos diseñan estrategias de casos de pruebas escritas, considerando ejemplos de su utilización antes de escribir el código. En la propuesta planteada la validación de los requisitos se viabiliza a través de la eficacia de la comunicación entre las partes interesadas, y no necesariamente a través de pruebas exhaustivas. Esta propuesta al igual que [81] hace uso de ejemplos para adquirir conocimiento del entorno de análisis, la diferencia es que lo plantean con diferente nivel de detalle.

2.1.6 Escenarios de creación de artefactos MDE

Se considera que se podría utilizar uno de los dos escenarios señalados a continuación para cubrir todo el ciclo de creación de los artefactos MDE:

Primer escenario

Utilizar un proceso iterativo incremental, donde en cada iteración se desarrolle uno de los artefactos MDE, es decir en una sola iteración crear el metamodelo completamente refinado y probado, en otra iteración el DSL, y así sucesivamente los otros artefactos. El inconveniente de este escenario que si bien utiliza ciclos para crear cada uno de los artefactos hasta obtener todos los artefactos requeridos, se tendría por lo menos que esperar hasta llegar a la iteración de generación de código para saber si la solución MDE es la adecuada. Escenario que a pesar de utilizar iteraciones, no pasa de ser un modelo de desarrollo en cascada. Este escenario resulta inadecuado más aún cuando el equipo de desarrollo es novato en el desarrollo de MDE donde se tiende a obtener metamodelos y lenguajes equivocadas porque no están acostumbrados a un "meta pensamiento"[96].

Segundo escenario

En el segundo escenario también se considera un proceso iterativo incremental, con la diferencia que para evitar los inconvenientes del escenario anterior, en cada iteración (ciclo) se deberían tomar una porción de requerimientos que más alta prioridad y conocimiento se tenga, para con éstos construir una versión reducida del metamodelo, para luego crear su correspondiente editor textual/gráfico, inmediatamente probar las características del nuevo lenguaje mediante la construcción de un modelo básico real, para finalmente crear una versión inicial del generador para verificar que en realidad se pueden generar artefactos pertinentes. Las iteraciones continuarían hasta obtener los artefactos que se requieran.

2.1.7 Recomendaciones para la construcción de artefactos MDE

Las recomendaciones se las ha obtenido de Voelter [96]. Los artefactos MDE son usualmente desarrollados por un pequeño grupo de personas y usados por un gran grupo. Por lo que se debe asegurar que estén bien definidas las planificaciones de las diferentes versiones. Se espera que el desarrollo sea corto con incrementos predefinidos. Los cambios a los requerimientos y problemas deben ser reportados, para lo cual se debe realizar el seguimiento correspondiente. Los errores deben ser encontrados relativamente rápido. Se debe contar con suficiente documentación. El personal de soporte debe estar disponible para ayudar en los problemas que se presenten. Se debe considerar como inevitable la curva de aprendizaje.

Se recomienda como buena práctica el rotar a las personas del equipo cada cierto tiempo, hacer a los desarrolladores de la aplicación parte del equipo de desarrollo del lenguaje para que puedan apreciar los cambios de un nivel inferior, hacer que las personas del equipo de desarrollo del lenguaje participes en el desarrollo de la aplicación, para asegurarse que ellos concibieron los productos como esperarían las personas para quienes desarrollan los productos. Se considera también que crear artefactos MDE correctos y adecuados no es suficiente para hacer el enfoque exitoso, se debe comunicar a los usuarios finales como utilizarlos en el contexto del mundo real. Específicamente se debe contar con documentos de: estructura y sintaxis del lenguaje, como utilizar los editores y los generadores, cómo y dónde escribir código manualmente, así como las especificaciones de la plataforma/framework.

2.2 Fundamentos del Análisis de Dominio

El objetivo final del Análisis de Dominio (DA, Domain Analysis) es la definición de un modelo de dominio que represente las propiedades de los sistemas y las relaciones entre las mismas en un dominio determinado [23]. Esto resulta útil cuando en MDE al utilizar un DSL se pretende representar las características comunes de un conjunto de sistemas de un determinado dominio.

2.2.1 Consideraciones básicas

El análisis de dominio como parte de la Ingeniería de Domino, es el proceso de identificación, recopilación, organización y representación de información relevante de un dominio, basado en el estudio de los sistemas existentes y su historia de desarrollo, conocimientos obtenidos de los expertos de dominio, la teoría subyacente y tecnología emergente dentro de un dominio [43]. Al examinar un tipo de sistemas de software relacionados y la teoría común subyacente de estos sistemas, el análisis de dominio puede proporcionar un modelo de referencia para la descripción de este tipo de sistemas.

El descubrimiento y la explotación sistemática de elementos comunes en todos los sistemas de software relacionados es un requisito técnico fundamental para lograr con éxito la reutilización del software [73]. De acuerdo con Neighbors en [66] la clave para la reutilización de software no se encuentra en la forma de programar el código, sino en la correcta especificación del dominio de aplicación. El análisis del dominio permite determinar qué factores del sistema a desarrollar son comunes a otros sistemas de información del mismo dominio. Entre los objetivos del análisis del dominio se encuentran:

- Reunir y relacionar toda la información referida a un elemento del dominio. El proceso facilita la futura evaluación de la información a fin de permitir su reutilización.
- Establecer modelos y patrones comunes entre un conjunto de sistemas. El estudio comparativo de sistemas permite deducir conocimiento intrínseco al dominio.
- Representar el conocimiento mediante lenguajes y herramientas específicas que faciliten su posterior reutilización. Un mismo lenguaje puede no ser igual de efectivo para representar el conocimiento de dos dominios distintos; por este motivo, es preciso adaptar los artefactos utilizados al dominio analizado.

Por su parte, el crecimiento exponencial que ha experimentado la industria del software y la necesidad de reducir tiempos y costos en la construcción de productos software, han hecho de la reutilización de componentes una práctica común hoy en día. En este contexto surgió la idea de aplicar técnicas utilizadas por otras industrias, como la automotriz o la electrónica [72]. Los criterios señalados en [60, 66,67] han sido utilizados desde hace varios años para desarrollar nuevas formas de abordar el desarrollo de software, entre ellos Línea de Productos de Software (LPS) y MDE. Una Línea de Producto tiene éxito si una compañía puede explotar los productos comunes para conseguir una producción más económica [13]. Una Línea de Productos Software es un conjunto de sistemas que comparten un conjunto administrable de características que satisfacen las necesidades de un segmento específico del mercado o misión, sistemas que se desarrollan a partir de un conjunto de activos clave (core assets - elementos software reutilizables) de una manera planificada [20]. Hoy en día una amplia variedad de compañías que emplean este criterio de desarrollo han logrado incrementar la calidad de sus productos, bajar sus costos de producción, mantenimiento y tiempo [11].

Actualmente existen numerosos métodos de Análisis de Dominio. Cada método se centra en el aumento de la comprensión del dominio mediante la captura de la información en uno o varios modelos. Sin embargo, se establecen una serie de actividades comunes recogidas en distintos métodos [23]:

- **Definición del dominio:** por dominio se entiende un área de conocimiento con una terminología y conceptos propios que deben ser considerados al especificar un sistema. La definición del dominio establece el alcance (límite y vocabulario) del dominio analizado.

- **Modelado de dominio:** se identifican y establecen relaciones entre los datos, funcionalidades y conceptos del dominio. Evalúa y selecciona los elementos susceptibles de reutilización. Un modelo de dominio determina que combinaciones de elementos del dominio deben ser tomados en cuenta y cuáles no.

2.2.2 Análisis de Dominio Orientado a Características (FODA)

Uno de los métodos de análisis de dominio es el Análisis de Dominio Orientado a Características (FODA, Feature-Oriented Domain Analysis). Orientado a características porque se basa en el énfasis puesto en el método de identificación de las características y sus relaciones, que un usuario espera comúnmente en una familia de sistemas de un dominio dado. El método FODA establece un proceso de análisis dividido en dos fases: 1) Análisis de Contexto. La actividad establece los límites del dominio considerado; y 2) Modelado del Dominio [23]. Donde se desarrolla principalmente un Modelo de Características del dominio de interés, sin embargo, para completar la especificación del dominio también se pueden utilizar otros modelos.

2.2.3 Modelo de Características

Una de las aportaciones más interesante del análisis de dominio constituye el modelado de características. En general, la literatura sobre ingeniería de dominio proporciona varias definiciones de feature (característica), en [86] se define como, “una feature es una característica distinguible de un concepto (por ejemplo, un sistema, componente, etc.) que es relevante para alguno de los interesados en el concepto”. Esta definición genérica está más próxima a los criterios de reutilización y generación de familias de sistemas [23]. Las características de los sistemas de un dominio se representan mediante un Modelo de Características.

Descripción del Modelo de Características

El Modelo de Características representa las capacidades generales de un dominio de aplicaciones. Entendiéndose por dominio de aplicación al conjunto actual y futuro de aplicaciones que comparten un conjunto de capacidades comunes y datos. Estas capacidades pueden incluir características tales como: servicios prestados, rendimiento, plataforma hardware requerida, costo, entre otros [43]. El enfoque de análisis se centra en la perspectiva del usuario final de la funcionalidad de las aplicaciones, es decir, los "servicios" prestados por las aplicaciones y los entornos operativos en los que se ejecutan las aplicaciones. Dado que el interés principal está en el carácter común de una familia de aplicaciones, el modelo de características debe capturar las características comunes y las diferencias de las aplicaciones en el dominio.

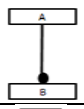
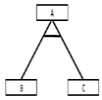
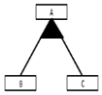
El modelo de características sirve como un medio de comunicación entre usuarios y desarrolladores. Para los usuarios, el modelo de características muestra cuáles son las características estándar, qué características puede elegir, y cuándo puede elegirlas. Para los desarrolladores, el modelo de características indica qué se debe parametrizar (adecuar para que se ajuste a las necesidades - establecer parámetros) en los otros modelos y como debe realizarse la parametrización [43].

Diseño del Modelo de Características

De acuerdo a FODA un modelo de características se encuentra compuesto por dos elementos: las características y las relaciones. Las características se organizan en una estructura jerárquica en forma de árbol las cuales se unen a través de las relaciones. Las relaciones entre las características pueden ser de dos tipos: jerárquicas y no jerárquicas. FODA propone cuatro tipos de relaciones jerárquicas y dos no jerárquicas:

La Figura 2.3 muestra los tipos de relaciones entre las características y los símbolos utilizados para su representación:

Tabla 2.3: Tipos de relaciones entre características

TIPO DE RELACIÓN	SÍMBOLO	DESCRIPCIÓN
JERÁRQUICAS		
Obligatoria (Mandatory)		Relación que indica que cuando la característica padre forma parte de un dominio particular, la característica hija también debe pertenecer como parte del dominio.
Opcional (Optional)		Este tipo de relación muestra que cuando la característica padre forma parte de un dominio particular, la característica hija puede o no ser incluida en el dominio.
Alternativa (Alternative) XOR (1)		Es la relación entre la característica padre y un conjunto de características hijas, la cual indica que cuando la característica padre forma parte de un dominio en particular, sólo una de las características del grupo de hijas debe pertenecer como parte del dominio.
OR (1 o más)		Indica que cuando la característica padre forma parte de un dominio particular, una o más de sus características hijas debe ser parte del dominio.
NO JERÁRQUICAS (representan restricciones)		
Excluye (excludes)		Este tipo de relación indica que una característica X excluye a Y, significa que si la característica X es incluida en el dominio, la característica Y no debe ser incluida y viceversa.
Requiere (requires)		La relación muestra que una característica X requiere a Y, significa que si la característica X es incluida en el dominio, la característica Y debe también ser incluida y no viceversa.

Ejemplos de tales características son las funciones de desvío de llamadas y transferencia de llamada de un sistema de conmutación telefónica, características de transmisión automática y manual de un automóvil, representado en la Figura 2.6 tomada de [43], o el de compras por internet (E-Shop), Figura 2.7 tomada de [30]:

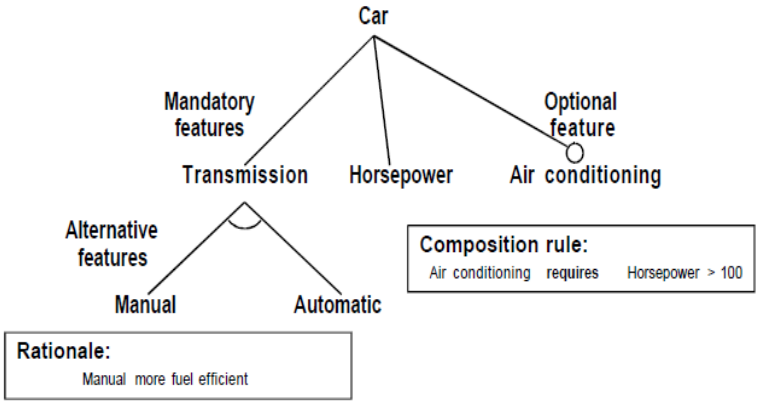


Figura 2.6: Modelo de Características de un automóvil

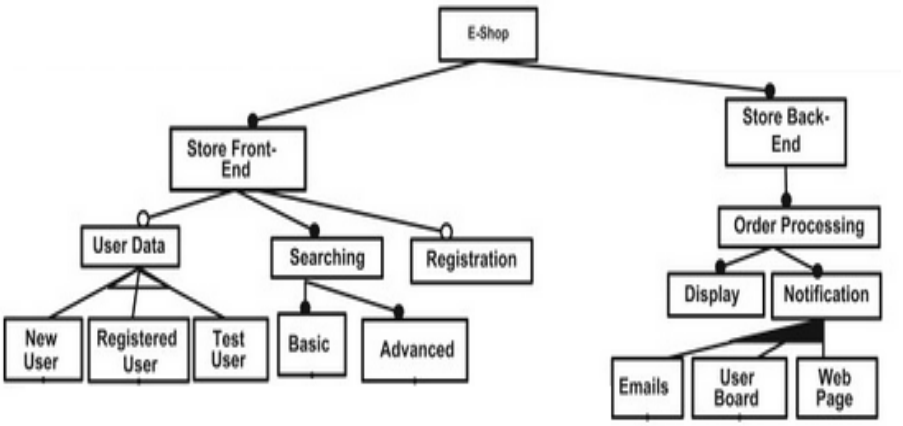


Figura 2.7: Modelo de Características de compras a través de Internet (E-Shop)

2.3 Metodologías de Desarrollo de Software

A continuación se procede a realizar una breve descripción de la evolución de las metodologías de desarrollo de software desde el punto de vista dialéctico, con la finalidad de analizar los fundamentos y principales características de las metodologías tradicionales y las metodologías ágiles, de tal manera que permitan formular las directrices de la propuesta.

2.3.1 El patrón dialéctico

La evolución de la gestión de los proyectos de software sigue un patrón dialéctico, basado en los conceptos formulados por Platón [70]. La gestión de proyectos de software al igual que otras áreas del conocimiento, se mueven según un triple patrón, que comienza con una etapa que se llama tesis, seguida de una etapa que la niega, llamada antítesis y por último se llega a la etapa final, llamada

síntesis. La síntesis es a la vez el final de un proceso dialéctico y el comienzo de otro, en el que la síntesis del antiguo funciona como una tesis de nuevo.

Nonaka y Takeuchi, en su libro Hitotsubashi on Knowledge Manegement [92], afirman estar convencidos de que este patrón dialéctico de tesis, antítesis y síntesis dirige la evolución del conocimiento. Cuestionar lo conocido es el motor de la evolución del conocimiento. De este modelo la espiral de evolución del conocimiento para la gestión de proyectos de software tiene su síntesis en las metodologías predictivas (tradicionales o procedurales), a las cuales se les contrapone una antítesis con modelos conceptuales absolutamente diferenciados (agilidad), y finalmente el conocimiento evoluciona a una síntesis que se ha ido generando en estos últimos años, tomando lo más valioso de ambos extremos. Posteriormente esa síntesis se convertirá en una tesis, que permitirá la evolución del modelo de conocimiento.

Con este principio, se realizará una breve descripción de los tres enfoques, donde:

- La tesis: gestión tradicional
- La antítesis: agilidad
- La síntesis: métodos ágiles de desarrollo

2.3.2 La tesis: Gestión Tradicional o Predictiva

Varios han sido los problemas que originaron la “crisis del software”. En este contexto se analizan los principales: incertidumbre en cronogramas y presupuestos, problemas en la especificación de requisitos, deficiencia de los procesos de desarrollo, y falta de gestión. La tesis, o alternativa tradicional, fue la primera corriente que buscó dar solución a los inconvenientes mencionados.

2.3.2.1 Incertidumbre en cronogramas y presupuestos

A medida que el software va ganando mayor incidencia sobre el hardware se hace evidente la necesidad de contar con elementos de gestión y estimación de proyectos de desarrollo de software, ya que son muy comunes los problemas relativos a errores en la definición de fechas de finalización, duración de los proyectos y costes. En el ámbito de la estimación aparecen varias técnicas, como COCOMO (Constructive COst MOdel) que constituye un modelo matemático de base empírica utilizado para estimación de costos de software y que está orientado a determinar la magnitud del producto final midiendo el "tamaño" del proyecto principalmente en líneas de código; también se cuenta con técnicas como Puntos de Función y Wideband Delphi [99].

2.3.2.2 Problemas en la especificación de requisitos

La falta de claridad en los requisitos así como la complejidad para su entendimiento y resolución, y la imposibilidad de lograr un eficiente manejo de los pedidos de cambio, dio origen al desarrollo de la “Ingeniería de Requisitos”, que es el proceso de descubrir, documentar y gestionar los requisitos para un

sistema informático. El objetivo de la ingeniería de requisitos es producir un conjunto de requisitos del sistema que en la medida de lo posible, sea completa, consistente, relevante y refleja lo que el cliente quiere realmente [87]. Establecer los requisitos es una de las actividades más importantes en el desarrollo de software, ya que si no se puede especificar con exactitud qué es lo que se necesita, es inútil implementarlo. Por ello se puede desarrollar el mejor software del mundo, pero si no es lo que se necesita, se habrá fallado.

2.3.2.3 Deficiencia de los procesos de desarrollo

Sobre el principio de calidad de Juran (Quality Control Handbook) [41], empleado con buenos resultados en los procesos de producción industrial, considera que la calidad del resultado depende básicamente de la calidad de los procesos empleados en su producción. Se desarrollaron también para la industria del software modelos de procesos como ISO 90003, CMMI (Capability Maturity Model for Integration), ISO 15505, ISO 12207, para que las empresas puedan alcanzar los cuatro beneficios clave de la producción basada en procesos: 1) repetitividad de resultados, 2) escalabilidad, 3) mejora continua, y 4) know-how propio.

2.3.2.4 Falta de gestión (predictiva)

Los proyectos han existido siempre. Cualquier trabajo para desarrollar algo único es un proyecto, pero la gestión de proyectos es relativamente reciente que comenzó a forjarse en los años sesenta. Bernard Schriever, arquitecto del desarrollo de misiles balísticos, es considerado el padre de la gestión de proyectos, por la introducción del concepto de “concurrencia”, para integrar todos los elementos del plan del proyecto en un solo programa y presupuesto. El objetivo de la concurrencia era ejecutar las diferentes actividades de forma simultánea, y no secuencialmente, y al aplicar a otro proyecto se redujeron considerablemente los tiempos de ejecución. La industria automotriz siguió los pasos del ámbito militar, aplicando técnicas de gestión de proyectos para la coordinación del trabajo entre áreas y equipos diferentes. Comenzaron a surgir técnicas específicas, histogramas, cronogramas, con conceptos de ciclo de vida del proyecto o descomposición de tareas (WBS Work Breakdown Structure) [70]. En 1960, Meter Norden del laboratorio de investigación de IBM en un seminario de Ingeniería de Presupuestos y Control presentado ante el American Management Association, señaló:

- Es posible relacionar los nuevos proyectos con otros pasados y terminados para estimar sus costes.
- Se producen regularidades en todos los proyectos.
- Es absolutamente necesario descomponer los proyectos en partes de menor dimensión para realizar planificaciones.

En los 80 se definieron los objetivos que la gestión de proyectos debía cumplir para poder considerar que el trabajo concluyó con éxito:

- Se ejecuta en un tiempo planificado.
- Sin desbordar el presupuesto estimado.
- Satisfaciendo las necesidades del cliente:
    - Realiza las funciones que necesita.
    - Las realiza correctamente y sin errores.

La gestión de proyectos predictiva o clásica es una disciplina formal, basada en la planificación, ejecución y seguimiento a través de procesos sistemáticos y repetibles. En este contexto la planificación se realiza sobre un análisis detallado del trabajo que se quiere realizar y su descomposición en tareas. Parte por tanto de los requisitos detallados de lo que se quiere hacer, sobre esa información se desarrolla un plan adecuado a los recursos y tiempos disponibles; y durante la construcción se sigue de cerca la ejecución para detectar posibles desviaciones y tomar medidas para mantener el plan, o determinar qué cambios va a experimentar. Se trata de una gestión "predictiva", que vaticina a través del plan inicial cuáles van a ser la secuencia de operaciones de todo el proyecto, su coste y tiempos. Su principal objetivo es conseguir que el producto final se obtenga según lo "previsto"; y basa el éxito del proyecto en las agendas, costes y calidad. En conclusión la gestión tradicional o predictiva:

- Establece como criterios de éxito: obtener el producto definido, en el tiempo previsto y con el coste estimado.
- Asume que el proyecto se desarrolla en un entorno estable y predecible.
- El objetivo de su esfuerzo es mantener el cronograma, el presupuesto y los recursos.
- Divide el desarrollo en fases a las que considera "ciclo de vida", con una secuencia de tipo: necesidad, requisito, diseño, desarrollo, pruebas y cierre.

2.3.3 La Antítesis: Agilidad

Muchas empresas trabajan en escenarios que se parecen ya muy poco a los que impulsaron la gestión de proyectos predictiva y necesitan estrategias diferentes para gestionar el lanzamiento de sus productos: estrategias orientadas a la entrega temprana de resultados tangibles, y con la suficiente agilidad y flexibilidad para trabajar en entornos inestables y rápidos. Quizás ya no hay "productos finales", sino productos en evolución, mejora o incremento continuo, desde la primera versión beta.

El modelo predictivo no es el único posible en este escenario, quizá el interés de una empresa sea poner en el mercado antes que nadie un producto valioso para los clientes, y estar continuamente desarrollando su valor y funcionalidad. En algunos proyectos de software el empeño en aplicar prácticas de estimación, planificación, ingeniería de requisitos es vano. Tal vez la causa de los problemas no sea la mala aplicación de las prácticas, sino la aplicación de prácticas inapropiadas. Se están aplicando criterios de gestión predictiva, cuando se trata de proyectos que no necesitan tantas garantías de previsibilidad en la ejecución, como valor y flexibilidad para trabajar en un entorno cambiante.

2.3.3.1 Manifiesto Ágil

En marzo de 2001, 17 críticos de los modelos tradicionales de desarrollo de software, convocados por Kent Beck que había publicado un par de años atrás el libro "Extreme Programming Explained" [9] en el que se exponía una nueva metodología denominada Extreme Programming, se reunieron en Salt Lake City para discutir sobre el desarrollo de software. En la reunión se acuño el térmico "Métodos Ágiles" para definir a los que estaban surgiendo como alternativa a las metodologías formales. En dicha reunión se resumieron cuatro postulados lo que ha quedado denominado como "Manifiesto Ágil", que son la base sobre la que se asientan estos métodos.

Manifiesto Ágil

Estamos poniendo al descubierto mejores métodos para desarrollar software, haciéndolo y ayudando a otros a que lo hagan. Con este trabajo hemos llegado a valorar:

- A los individuos y su interacción, por encima de los procesos y las herramientas.
- El software que funciona, por encima de la documentación exhaustiva.
- La colaboración con el cliente, por encima de la negociación contractual.
- La respuesta al cambio, por encima del seguimiento de un plan.

Aunque hay valor en los elementos de la derecha, valoramos más los de la izquierda.

2.3.3.2 Principios ágiles

La agilidad, sin importar la metodología o marco del que se trata, concuerdan en el cumplimiento de doce principios básicos. Son características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto a metas a seguir y organización del mismo.

Principios Ágiles

1. Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.
2. Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo.
3. Entregar con frecuencia software que funcione, en un período de un par de semanas hasta un par de meses, con preferencia en períodos breves.
4. Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana a través del proyecto.
5. Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea.
6. La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro del equipo de desarrollo es mediante la conversación cara a cara.
7. El software que funciona es la principal medida del progreso.
8. Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.
9. La atención continua a la excelencia técnica enaltece la agilidad.
10. La simplicidad como arte de maximizar la cantidad de trabajo no hecho, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de equipos que se auto-organizan.
12. En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.

2.3.4 La Síntesis: Métodos ágiles de desarrollo

Los métodos ágiles de desarrollo, no son ni tesis ni antítesis, en este momento hacen de síntesis, aprendiendo de los aciertos y de los errores de los dos enfoques de gestión analizados, y que es gracias al conocimiento aportado por los dos enfoques el poder avanzar en el espiral del conocimiento que permita mejorar el trabajo de la ingeniería de software.

2.3.4.1 Términos a tomar en cuenta

Existen varios términos fundamentales de esta forma de construir software:

- **Agilidad:** capacidad para producir partes completas del producto en períodos breves de tiempo.
- **Flexibilidad:** capacidad para adaptar el curso del desarrollo a las características del proyecto, y a la evolución de los requisitos.
- **Fases del desarrollo solapado:** el concepto de "fase" que implica el trabajo secuencial, se cambia ahora por el de "actividad". Requisitos, análisis, diseño, desarrollo no son fases ejecutadas en un orden determinado. Son actividades que se pueden realizar en cualquier momento de forma simultánea; "a demanda" cuando las necesita el equipo. En el ciclo de vida secuencial de software se habla de "modificación de requisitos", este término lleva implícito el concepto que se está cambiando algo que quedó cerrado en la fase de requisitos. En el desarrollo ágil, los requisitos evolucionan, se desarrollan y enriquecen durante todo el ciclo de vida, igual que el diseño y el código.

2.3.4.2 El ciclo de desarrollo ágil

El desarrollo ágil parte de la visión del concepto general del producto, y sobre ella el equipo produce de forma continua incrementos en la dirección apuntada por la visión; y en el orden de prioridad que necesita el negocio del cliente. Los ciclos breves de desarrollo, se denominan iteraciones y se realizan hasta que se decide no evolucionar más el producto. Se encuentra formado por cinco fases: Concepto, Especulación, Exploración, Revisión y Cierre

Concepto: en esta fase se crea la visión del producto y se determina el equipo que lo llevará a cabo. Partir sin una visión genera esfuerzo inútil. La visión es un factor crítico para el éxito del proyecto. Se necesita tener el concepto de lo que se quiere, y conocer el alcance del proyecto. Es además una información que deben compartir todos los miembros del equipo.

Especulación: una vez que se sabe qué es lo que hay que construir, el equipo especula y formula hipótesis basadas en la información de la visión, que de por sí es muy general e insuficiente para determinar las implicaciones de un desarrollo (requisitos, diseño, costes, etc.). En esta fase se determinan las limitaciones impuestas por el entorno de negocio: costes y agendas principalmente y se cierra la primera aproximación de lo que se puede producir. La gestión ágil investiga y construye a partir

de la visión del producto. Durante el desarrollo confronta las partes terminadas: su valor, posibilidades y la situación del entorno en cada momento. La fase de especulación se repite en cada iteración, y teniendo como referencia la visión y el alcance del proyecto consiste en:

- Desarrollo y revisión de los requisitos generales.
- Mantenimiento de una lista con las funcionalidades esperadas.
- Mantenimiento de un plan de entrega: fechas en las que se necesitan las versiones, hitos, iteraciones del desarrollo. Este plan refleja ya el esfuerzo que consumirá el proyecto a lo largo del tiempo.
- En función de las características del modelo de gestión y del proyecto puede incluir también una estrategia o planes para la gestión de riesgos. Si las exigencias formales de la organización lo requieren, también se produce información administrativa y financiera.

Exploración: se desarrolla un incremento del producto, que incluye las funcionalidades determinadas en la fase anterior.

Revisión: equipo y usuarios revisan lo construido hasta el momento. Trabajan y operan con el producto real contrastando su alineación con el objetivo.

Cierre: al llegar a la fecha de entrega de una versión de productos (fijada en la fase de concepto y revisado en las diferentes fases de especulación), se obtiene el producto esperado. Posiblemente éste seguirá en el mercado, y por emplear gestión ágil, es presumible que se trate de un producto que necesita versiones y mejoras frecuentes para no quedar obsoleto. El cierre no implica el final del proyecto. Lo que se denomina "mantenimiento" supondrá la continuidad del proyecto en ciclos incrementales hacia la siguiente versión para ir acercándose a la visión del producto.

2.3.4.3 Principales modelos de gestión ágil

Si hubiera que determinar cuál es el origen de la gestión de proyectos con este enfoque, a falta de mejor información, habría que situarlo en las prácticas adoptadas en los 80 por empresas que hacen referencia Takeuchi y Nonaka [92]. La industria del software ha sido la primera en seguir su adopción, y muchos de sus profesionales han documentado y propagado las formas particulares en las que han implementado los principios de la agilidad en sus equipos de trabajo. Algunos de los modelos que se encuentran inscritos en la organización Agile Allience [3] diferenciados por las actividades que realizan y su alcance, son: AUP (Agile Unified Process), Crystal, FDD (Feature Driven Development), Scrum (utiliza ciclos iterativos llamados sprints como aspecto más representativo de desarrollo), XP (eXtreme Programming) con las historias de usuario.

2.3.4.4 Modelos Clásicos de Desarrollo vs. Modelos Ágiles

La Tabla 2.4 tomada de [70] recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales ("no ágiles"). Estas diferencias afectan no sólo al proceso en sí, sino también al contexto del equipo, así como a su organización.

Tabla 2.4: Diferencias entre las metodologías tradicionales y ágiles

Metodologías Tradicionales	**Metodologías Ágiles**
Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo	Basadas en heurísticas provenientes de prácticas de producción de código
Cierta resistencia al cambio	Especialmente preparados para los cambios durante el proyecto
Impuestas externamente	Impuestas internamente (por el equipo)
Proceso mucho más controlado, con numerosas políticas/normas	Proceso menos controlado, con pocos normas
Existe un contrato prefijado	No existe contrato tradicional o al menos es bastante flexible
El cliente interactúa con el equipo de desarrollo mediante reuniones	El cliente es parte del equipo de desarrollo
Grupos grandes y posiblemente distribuidos	Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio
Más artefactos	Pocos artefactos
Más roles	Pocos roles
La arquitectura del software es esencial y se expresa mediante modelos	Menos énfasis en la arquitectura del software

Los equipos ágiles empiezan a trabajar sin conocer con detalle cómo será el producto final. Parten de la visión general, y sobre ella, producen regularmente incrementos de funcionalidad que incremental el valor al producto.

2.4 Desarrollo por el Usuario Final

2.4.1 Generalidades

El Desarrollo por el Usuario Final (EUD, End-User Development) es una área de investigación en el campo de las Ciencias de la Computación, específicamente de HCI (Human-Computer Interaction), que se define según Lieberman en [52] como “el conjunto de métodos, técnicas y herramientas que permiten en algún momento a los usuarios de sistemas de software, actuar como desarrolladores de software no profesionales para crear, modificar o extender un artefacto de software”. Los usuarios de computadoras han aumentado rápidamente en número y diversidad [83]. Estos incluyen gerentes, contadores, ingenieros, amas de casa, maestros, científicos, trabajadores de la salud, personal de seguros, vendedores, asistentes administrativos. Muchas de estas personas trabajan en tareas que varían rápidamente de forma anual, mensual, o incluso a diario. Por lo tanto, sus necesidades de software son diversas, complejas y en constante cambio. EUD permite a los usuarios finales diseñar o personalizar la interfaz de usuario y funcionalidad de software. Esto es valioso porque los usuarios finales conocen su propio contexto y necesidades mejor que nadie, y que a menudo tienen conocimiento en tiempo real de los cambios en sus respectivos dominios [33].

La herramienta EUD más popular es la hoja de cálculo [15]. Debido a su naturaleza sin restricciones, las hojas de cálculo permiten que usuarios de ordenadores relativamente poco sofisticados escriban programas que representan modelos de datos complejos, evitando la necesidad de aprender programación con lenguajes de bajo nivel [1]. Los primeros intentos en el desarrollo de programas por parte de los usuarios finales se centraron en la adición de simples scripts en algún lenguaje de programación para extender y adaptar una aplicación existente, como una suite de oficina. Los

artefactos definidos por los usuarios finales pueden ser objetos que describen un comportamiento automatizado o secuencia de control, tales como peticiones de base de datos o definición de reglas gramaticales; y que pueden ser descritos con paradigmas de programación tales como Programación por Demostración , Programación por Ejemplos , Programación Visual , o Generación de Macros [21]. También pueden ser parámetros que se eligen entre las alternativas de comportamientos predefinidos de una aplicación. Otros artefactos de desarrollo de usuario final también pueden referirse a la creación de contenido generado por el usuario, tales como anotaciones, que puede ser o no computacionalmente interpretables.

Hay dos razones básicas por las que EUD se ha vuelto popular. Una de ellas es porque las organizaciones se enfrentan a retrasos en los proyectos y el uso de EUD puede reducir efectivamente el tiempo de finalización de un proyecto. La segunda razón es que las herramientas de software cada vez son más potentes y fáciles de usar [55], por lo que pueden apoyar en las labores donde los usuarios tengan conocimiento y experiencia en el dominio. Ejemplos del desarrollo de usuario final incluyen la creación y modificación de programas de creación de modelos 3D, aplicaciones de animación , archivos de configuración (por ejemplo, filtros de correo electrónico), modificaciones del juego para introducir los preferencias de los usuarios, creación de guiones utilizados en Call Centers, modelos científicos utilizados en la simulación por ordenador, scripts y macros que agregan o extienden tareas automatizadas en una suite de oficina, programación visual en forma de lenguajes visuales como LabVIEW, desarrollo de páginas web como con Google Sites, desarrollo de Wikis, Mashups Web, prototipos y programas de un dominio específicos escritos por personal técnico para que los usuarios puedan utilizarlos en sus requerimientos específicos, como puede ser el caso de desarrollar aplicaciones de propósito específico con una gama de artefactos MDE.

Según Sutcliffe en [90], EUD asigna actividades de desarrollo a un usuario final, evidentemente el usuario final tendrá que realizar un poco de esfuerzo para aprender una herramienta EUD, la motivación de los usuarios debe estar ligada a la confianza que va a potenciar su trabajo, en el ahorro de tiempo de trabajo y/o aumento de la productividad. Los beneficios que proveen la utilización de este tipo de herramientas se las transmite a través de estrategias de marketing, demostraciones, y recomendaciones entre personas. Una vez que la tecnología es utilizada, la experiencia de los beneficios reales se convierte en la clave de motivación. Los costos de utilización de la herramienta se definen como la suma de:

- **Coste técnico:** el precio de la tecnología y el esfuerzo para instalarlo
- **Costo de aprendizaje:** el tiempo necesario para entender la tecnología
- **El costo del desarrollo:** el esfuerzo para desarrollar aplicaciones utilizando la tecnología
- **Costo de prueba y depuración:** el tiempo necesario para verificar el sistema

Los dos primeros costos se los realiza una sola vez, durante la adquisición de la herramienta. Mientras que se incurre en el tercer y cuarto costo cada vez que se desarrolla una aplicación. A pesar de los beneficios señalados por la utilización de estas herramientas, este enfoque de desarrollo tiene varias críticas. Principalmente parten del hecho que los usuarios finales no saben cómo probar y asegurar la

calidad de sus aplicaciones. Se considera que el usuario pueda dejar de lado aspectos de seguridad y que son desarrolladas sin emplear prácticas de desarrollo generalmente aceptadas. Como la especificación antes de la codificación, las pruebas sistemáticas, etc. Este punto de vista supone que todos los usuarios finales son igualmente ingenuos cuando se trata de la comprensión del software, aunque Pliskin en [71] sostienen que este no es el caso general, consideran que existen usuarios finales sofisticados que si son capaces de desarrollar aplicaciones software con este enfoque.

2.4.2 Ingeniería de Software de Usuario Final

Para disminuir los aspectos negativos de EUD, se han realizado varios estudios de ingeniería de software para el desarrollo de software por parte de los usuarios finales, a esta área se la denomina Ingeniería de Software de Usuario Final (EUSE, End-User Software Engineering). Como lo establece Ko en [46], la programación se ha convertido en una habilidad técnica de millones de personas que no son necesariamente del área informática, según las estadísticas de la Oficina de Trabajo de EE.UU. y Estadística, en 2012 en los Estados Unidos habría menos de 3 millones de programadores profesionales, pero más de 55 millones de personas utilizando hojas de cálculo y bases de datos en el trabajo, muchos para escribir fórmulas y consultas para apoyar su trabajo [83].

La mayoría de los programas de hoy en día no están escritos por los desarrolladores profesionales de software, sino por personas con experiencia en otros ámbitos de trabajo para alcanzar objetivos que necesitan apoyo computacional [46]. Por ejemplo, un profesor podría escribir una hoja de calificación para realizar el seguimiento de calificaciones de los estudiantes, un fotógrafo podría escribir una secuencia de comandos de Photoshop para aplicar los mismos filtros a un centenar de fotos, o un cuidador podría escribir un script para ayudar a que una persona con discapacidades cognitivas sea más independiente [17]. En estas situaciones, el programa desarrollado por un usuario final constituye un medio para un fin y se podría utilizar una variedad de herramientas para lograr esta meta. La Tabla 2.5 tomada de [46], proporciona una idea de la diversidad de clases de personas que pueden crear aplicaciones.

Tabla 2.5: Lista parcial de Clase de Personas que escriben programas

Clase de personas	**Actividades de programación, herramientas y lenguajes utilizados**
Administradores de sistemas	Escribir scripts para los sistemas, utilizando editores de texto y lenguajes de scripting
Diseñadores de interacción	Prototipos de interfaces de usuario con herramientas como Visual Basic y Flash
Artistas	Crean arte interactivo con lenguajes de Processing (http://processing.org)
Profesores	Enseñan ciencia y matemáticas con hojas de cálculo
Contadores	Tabular y resumir datos financieros con hojas de cálculo
Actuarios	Evaluar riesgos financieros utilizando herramientas de simulación como MATLAB
Arquitectos	Modelado y diseño de estructuras utilizando Autocad , FormZ y otros modeladores 3D
Niños	Crean animaciones y juegos con Alice (http://www.alice.org)
Niñas de escuela intermedia	Utilice Alice para contar historias
Webmasters	Gestionar bases de datos y sitios web utilizando Access, FrontPage, HTML, Javascript
Trabajadores de salud	Escribir especificaciones para generar formularios de informes médicos
Científicos/ingenieros	Utilizan MATLAB y Prograph para realizar pruebas y simulaciones
Usuarios de e-mail	Escribir reglas de correo electrónico para gestionar, ordenar y filtrar
Jugadores de videojuegos	Configuraciones del entorno de juego, de una jugador, multi-jugador en línea
Músicos	Crear música digital con sintetizadores y lenguajes de flujo de datos musicales
VCR y usuarios de TiVo	Grabar programas de televisión con antelación, especificando parámetros y calendarios
Propietarios de viviendas	Escribir calendarios de control para la calefacción y sistemas de iluminación con X10

Usuarios de Apple OS X	Automatización del flujo de trabajo con AppleScript y Automator
Usuarios de calculadora	Procesos de datos matemáticos y generación de gráficos con lenguajes de scripting
Administradores	Crear bases de datos y generar reportes con Crystal Reports

Como se señaló anteriormente la intención de los usuarios finales de crear programas es que les permita apoyar las labores donde tiene un amplio conocimiento y experiencia en el dominio a través de entornos de programación y herramientas fáciles de utilizar como los DSLs donde no tienen que lidiar con aspectos de bajo nivel de abstracción como los GPLs y actividades de ingeniería de software, mientras que la intención del desarrollador de software profesional es crear aplicaciones de cualquier índole que pueden ser utilizadas por miles de usuarios, por lo debe considerar a detalle cuestiones de calidad del software como la confiabilidad, reutilización, usabilidad, seguridad, facilidad de mantenimiento y actividades que refuerzan estas cualidades, como pruebas, verificación, y depuración.

A pesar que este usuario final que programa alguna aplicación puede no tener la mismos objetivos que los desarrolladores profesionales, tiene que hacer frente a muchos de los mismos retos de ingeniería de software, incluyendo la comprensión de sus requisitos, así como la toma de decisiones sobre el diseño, la integración, reutilización, probar y depurar, aunque sea de una manera intuitiva y rudimentaria, debido a que estas actividades le resultan secundarias al objetivo de que el programa le está ayudando a lograr. Debido a esta diferencia en las prioridades [12], las personas que están participando en la programación del usuario final rara vez tienen el tiempo o el interés en actividades de una ingeniería de software sistemática y disciplinada. Situación que se puede apreciar en la Tabla 2.6 tomada [46].

Tabla 2.6: Diferencias cualitativas en aspectos de ingeniería de software entre un profesional y un usuario final

Actividad de Ingeniería de Software	**Profesional de Ingeniería de Software**	**Ingeniería de Software de Usuario Final**
Requerimientos	explícito	implícito
Especificación	explícito	implícito
Reutilización	planificado	sin planificación
Pruebas y Verificación	precaución	confiado
Depuración	sistemática	oportunista

La investigación en Ingeniería de Software de Usuario Final es un entorno interdisciplinario, donde están involucradas áreas de la ciencia de la computación, como la ingeniería de software, la interacción persona ordenador, áreas como la educación, la psicología y otras. Por lo tanto, hay varias dimensiones a lo largo de la cual se podrían discutir este tema, incluyendo herramientas y nuevos enfoques de lenguajes de programación. Debido a la amplia variedad de aplicaciones que permiten al usuario final adquirir habilidades para desarrollar o personalizar sus entornos de trabajo a través de programación, éste lo hace sin tomar en consideración las actividades formales de la ingeniería de software y está lejos de que lo haga, por lo que se considera que cada grupo necesita apoyo por lo menos básico de ingeniería de software a la medida de su ámbito de práctica [46], por lo que EUSE se limita a realizar recomendaciones generales de las actividades que se podrían realizar. Así por ejemplo:

Requerimientos y Diseño

Los requisitos describen lo que un programa debe hacer, y el diseño se refiere a la determinación de cómo un programa debe hacerlo. Por ejemplo, un requisito podría ser que un programa debe ser capaz de ordenar una lista de direcciones de correo, y su diseño podría detallar el algoritmo de clasificación que se utiliza. Conseguir los requisitos correctos es un aspecto crítico desde la perspectiva de EUSE, por su énfasis en la calidad. Se espera que los desarrolladores profesionales investiguen, documenten y refinen los requisitos antes de empezar a diseñar o codificar una aplicación. Por el contrario, los usuarios finales suelen vivir en su dominio todos los días y lo saben muy bien, por lo que a menudo ya tienen una idea de cuáles son los requisitos, y no necesitan ningún trabajo extra para llegar a ellos, ni siquiera documentarlos. Sin embargo, pueden haber ocasiones donde el usuario final no conoce los requisitos con antelación y no aspiran a un "diseño" de por sí, sino que pueden esperar que estos asuntos se aclararon de acuerdo a la evolución de la ejecución del programa [21].

Los desarrolladores profesionales a veces no conocen bien los requisitos con antelación, pero se toman medidas para hacer frente a esa situación, tales como el empleo de un método para llegar a los requisitos, como el desarrollo de prototipos, en lugar de omitir por completo el concepto. En este caso, los usuarios finales que desarrollan pueden acceder directamente a la codificación sin tomarse el tiempo para documentar sus requisitos o buscar inconsistencias [77]. Debido al estrecho acoplamiento de EUD a un dominio, los cambios externos en el dominio pueden causar la evolución de las necesidades, por ejemplo, cambios en las reglas de contabilidad pueden requerir un analista financiero para calcular datos diferentes, lo que podría a su vez causar modificaciones a una hoja de cálculo existente. Debido a su naturaleza altamente iterativa, el refinamiento de requisitos en EUD ha sido comparado con una forma de programación altamente ágil. Por lo tanto, las necesidades del usuario final que desarrolla aplicaciones tienden a ser emergentes y estrechamente entrelazadas con el diseño, ante este hecho, los enfoques de diseño que han sido dirigidos a este tipo de desarrolladores tienen como objetivo apoyar el desarrollo evolutivo y/o exploratorio [52]. Una de las técnicas que se podrían utilizar es el “proceso de diseño exploratorio” el cual es asistido por un crítico experto del dominio, que puede revisar el diseño del usuario y dar sugerencias de mejora (Figura 2.8) tomada de [26].

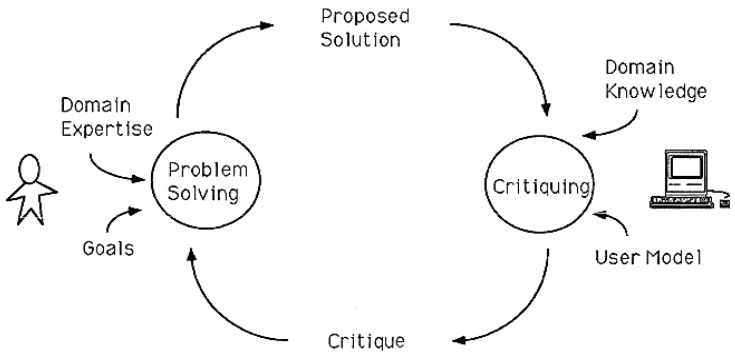


Figura 2.8: Visión general del proceso de diseño exploratorio

Verificación y Validación

La verificación y/o validación (V & V) incluyen actividades que tratan de asegurarse de que un programa hace lo que se supone que debe hacer. La prueba es el método más común para el V & V (incluso entre los desarrolladores profesionales). Uno de los trabajos de apoyo a V & V en EUD es ayudar a los usuarios a evaluar si sus programas contenían errores, alentándolos a hacer pruebas de los resultados de sus aplicaciones. Uno de los enfoque de prueba de usuario final mejor desarrollado es "lo que ves es lo que pruebas" (WYSIWYT) aplicado para hojas de cálculo [27], lo que sugiere que las pruebas a ser aplicadas dependerá del tipo de aplicación.

Depuración

Cuando se detecta un error de programación, éste debe ser eliminado por depuración. Algunas de las técnicas de depuración utilizados por los desarrolladores profesionales se han adaptado para su uso en herramientas EUP. Por ejemplo el hecho de poder mostrar el valor de una variable o de algún elemento mientras se ejecuta un programa, a través de la ejecución de instrucciones paso a paso (opciones de trace en algunos compiladores) en busca de operaciones incorrectas [51].

Reutilización

Después que el código está escrito, la reutilización puede acelerar la creación de programas posteriores. Sin embargo, el apoyo en la reutilización de los programas de usuario final es un reto porque los usuarios finales rara vez tienen la oportunidad o la formación necesaria para diseñar programas altamente reutilizables. Otro reto es que los usuarios finales desarrolladores pueden cometer errores al crear programas u otros archivos para adaptar a las aplicaciones, y la reutilización de estos errores se puede propagar a través de la organización [54]. Por lo tanto, a pesar de que los sistemas como los repositorios o servidores de archivos pueden resultar fáciles de emplear para unos, para otros puede resultar extremadamente difícil, por otro lado se requiere mucho tiempo para que otros desarrolladores puedan evaluar la capacidad de reutilización de estos programas. Para ayudar a reducir la dificultad de los programas de reutilización, se están desarrollando varios esfuerzos para crear modelos de repositorios de reutilización para los usuarios finales, pero relacionados con los intereses particulares del usuario [82]. Fuera de los repositorios, no se ha comenzado a estudiar la manera de ayudar a los usuarios extraer piezas reutilizables [69].

Capítulo 3

Impacto de MDE en la Empresa

Para determinar el impacto que tiene MDE en la empresa, se realizó una revisión detallado de la literatura en ámbitos como: experiencias con MDE, aspectos de calidad en MDE, artefactos de soporte al proceso de desarrollo con MDE y procesos de desarrollo de MDE en base a DSL.

3.1 Experiencias con MDE

Kulkarni y Reddy en [49] señalan que la curva de aprendizaje para adoptar el enfoque de desarrollo MDE es empinada y que se requiere una alta inversión inicial en la creación del entorno de desarrollo, por lo que considera que estos dos factores son los principales impedimentos para que el desarrollo de software con este enfoque sea empleado en pequeñas y medianas empresas.

Clark y Muller en [19] describen la experiencia de la creación de dos empresas dedicadas a revolucionar el desarrollo de software elevando el nivel de abstracción a través del modelado. Estas empresas no lograron sobrevivir debido al enfoque eminentemente técnico de sus creadores lo que les hizo descuidar aspectos comerciales y financieros, además sostienen porque quizás el mercado no está aún preparado para el nuevo enfoque de desarrollo. Los autores mencionan un aspecto importante por las cuales empresas no pueden salir a flote aunque sus propuestas sean tecnológicamente adecuadas y bien estructuradas, señalando “de nada sirve tener los estándares técnicos más altos posibles y saber cómo hacer algo de manera correcta, si nadie va a escuchar”.

Sánchez et al. en [80] presenta la experiencia de Transferencia de Tecnología (ToT) en dos empresas pequeñas. Se considera que aunque MDE está ganando cada vez más aceptación en la comunidad de ingeniería de software su adopción por la industria es muy limitada. Existe un creciente número de empresas que han aplicado MDE con éxito, pero su uso por la industria sigue siendo una excepción y no la norma. Se señala que la adopción de innovación (utilización de una nueva tecnología) en el proceso de desarrollo de software en la industria no es inmediata, sino que abarca un período considerable de

tiempo. Se determina que para alcanzar las condiciones necesarias, se requiere: una base estable y madura, herramientas útiles y robustas, profesionales calificados y fundamentalmente que las empresas tomen conciencia de los beneficios de la nueva tecnología. Por lo cual se sugiere varias acciones para que MDE sea adoptado, como más investigación y desarrollo para superar los desafíos técnicos, incluir MDE en los planes de estudio de las Universidad, desarrollar proyectos de ToT destinados a incrementar el conocimiento en la industria, y poder proporcionar a los desarrolladores un nuevo enfoque de desarrollo de software.

Zohaib et al. en [101] describe la experiencias en la aplicación de un entorno de desarrollo empleando MDE utilizando UML/MARTE (Modeling and Analysis of Real-Time and Embedded Systems) en proyectos industriales de RTES (Real-Time and Embedded Systems) en diferentes dominios. Sostienen que la utilización de UML/MARTE en los diferentes proyectos requiere el apoyo de una metodología completa que identifique el subconjunto UML/MARTE que se debe utilizar para tratar problemas en contextos específicos. Consideran que una metodología completa basada en UML/MARTE debe derivarse de un determinado propósito para hacer frente a un problema particular en un dominio específico. Para ello, se debe partir de un análisis exhaustivo en base a un Análisis Dominio, que permita identificar la información que se va a manejar.

El estudio de investigación empírica realizado por Hutchinson et al. en [32] en un período de doce meses, determina la percepción de MDE en la industria. Establecen que hay muchos ejemplos de historias de éxito, pero también hay muchos casos de fracaso. Se considera que MDE tiene varios beneficios potenciales, principalmente la ganancia en productividad, portabilidad, facilidad de mantenimiento e interoperabilidad. Pero también presenta varios inconvenientes como es el aspecto cultural de varias empresas reacias al cambio, consideran que es un enfoque que tiene aún altos riesgos de adopción debido a varios factores, entre los principales el desconocimiento, utilización de herramientas costosas, y la falta de un proceso formal de desarrollo.

Por su lado Mohagheghi et al. en [64] señala que MDE se encuentra todavía en una fase temprana de adopción en la industria, menciona un caso de éxito y uno de fracaso. Considera que este paradigma debe demostrar su superioridad sobre otros paradigmas de desarrollo en base al apoyo de un rico ecosistema de herramientas estables, compatibles y estandarizadas. Se considera varios retos que debe vencer MDE, entre ellos, hacer frente a modelos muy grandes y complejos, manejo de trazabilidad, ayuda de herramientas y la utilización de mecanismos de gestión de modelos.

El estudio de la literatura realizado por Mohagheghi y Dehlen [62] hace una recopilación de evidencias de los beneficios y limitaciones de MDE en algunas empresas. Consideran que el enfoque MDE ha promovido la premisa que el desarrollo de software con este paradigma incrementa la productividad debido a la automatización de varias de las tareas de trabajo, aumenta la calidad debido a la disminución de errores principalmente en la codificación y permite manejar la complejidad del desarrollo de software elevando el nivel de abstracción al manejar modelos y lenguajes de dominio específico. El estudio concluye estableciendo que no hay la suficiente evidencia que permita generalizar los postulados considerados por MDE debido a varios factores que afectan su desarrollo.

3.2 Aspectos de calidad en MDE

Debido a que los elementos primarios del desarrollo de software con MDE lo constituyen los modelos y sus transformaciones, el análisis realizado por Mohagheghi et al. en [63] en base a la revisión de la literatura existente, toma en consideración los criterios que involucra la calidad de los modelos en MDE y cómo ésta puede ser mejorada. Se considera que si los defectos en los modelos son detectados en etapas tempranas y éstos son corregidos lo más pronto posible, permitirá mejorar la calidad de los artefactos que se van creando, lo que se traduce en última instancia en la reducción de los costos de mantenimiento. Se identifican seis objetivos de calidad del modelo: exactitud, exhaustividad, coherencia, comprensibilidad, confinamiento y mutabilidad.

3.3 Artefactos de soporte al proceso de desarrollo con MDE

En el trabajo realizado por Cánovas et al. en [16] se propone una infraestructura de colaboración donde todos los actores (técnicos y no técnicos) trabajen juntos para el desarrollo del DSL y asegurar que el sistema satisfaga las expectativas requeridas. Esta infraestructura permite en el desarrollo de un DSL presentar propuestas de cambio, recomendar posibles soluciones, y realizar comentarios surgidos durante su desarrollo y evolución.

En el estudio inspirado en los fundamentos de la programación interactiva por Sánchez et al. en [81] se propone un proceso iterativo e interactivo en la creación de metamodelos y entornos de modelado, en base a la colaboración de expertos de dominio (sin necesariamente contar con competencias en metamodelado) con los ingenieros de software. Este proceso plantea que tanto el experto del dominio como los ingenieros de software vayan desarrollando fragmentos del modelo desde su punto de vista, estos fragmentos deben ser revisados, refinados y completados hasta llegar a obtener un metamodelo que se ajuste a los requerimientos planteados inicialmente y que posteriormente se pueda compilar en una plataforma específica.

En el trabajo realizado por Mernik et al. en [58] se realiza un análisis de las características, ventajas y desventajas de los DSLs, los cuales son considerados como Lenguajes Orientados a la Aplicación, señala cuándo y cómo utilizar un DSL, se considera que un DSL no puede ser una solución a todos los problemas de ingeniería de software. Se señala ciertas categorías de las aplicaciones en las cuales resulta conveniente utilizar el enfoque basado en DSL:*1) Tareas de automatización*, cuando el desarrollo de la aplicación tiene el mismo patrón de tareas de programación GPL, *2) Línea de Productos de Software* cuando la aplicaciones comparten una arquitectura común y se desarrollan a partir de un conjunto común de elementos básicos, *3) Representación de Estructuras de Datos*, muchas veces las estructuras de datos resultan complicadas de escribir y mantener, por lo cual a menudo se expresan más fácilmente utilizando un DSL, *4) Estructura de los Datos Transversal*, a menudo las estructuras de datos se expresan mejor y de forma más fiable con un DSL adecuado, *5) Sistemas front-end* basados en DSLs debido a que a menudo se puede usar para el manejo de la configuración y adaptación de un sistema, *6)*

Interacción, la interacción basada en texto o menús de las aplicaciones software pueden ser complicadas y repetitivas, por ejemplo Excel en modo interactivo se complementa con el lenguaje de macros.

3.4 Procesos de desarrollo de MDE en base a DSL

Strembeck y Zdun en [88], considerando la experiencia en numerosos proyectos de desarrollo de DSLs, la realización de estudios, experimentos de campo y seguimiento de proyectos, proponen un enfoque para el desarrollo sistemático de DSLs. La propuesta establece cuatro actividades principales: 1) definición del core del lenguaje de modelado del DSL, 2) definición del comportamiento de los elementos del lenguaje DSL, 3) definición de la sintaxis concreta del DSL, y 4) integración de artefactos DSL con la plataforma/infraestructura. Estás actividades conforman un proceso iterativo e incremental.

En el estudio realizado por Kolovos en [47] se presenta un análisis de los requisitos fundamentales que debe cumplir el desarrollo de un DSL, el cual debe centrarse en la identificación de las necesidades de los stakeholders y en el establecimiento de los límites de éste (donde terminan un DSL y donde comienzan los GPLs). La adecuada comprensión de los requisitos del DSL en contextos específicos, es fundamental para mejorar la calidad del DSL y éste tendrá un impacto sobre los atributos de calidad del proceso de desarrollo global de los sistemas y los productos resultantes. De igual manera se considera que un DSL tiene su propio ciclo de vida.

3.5 Análisis de la literatura revisada

MDE es un enfoque que se basa en muchas de las técnicas aplicadas con éxito en la ingeniería de software, se caracteriza por muchos aspectos, entre ellos porque aprovecha la utilización de modelos en todas las fases de desarrollo, crea lenguajes de dominio específico como marco que cubrir un dominio en particular, aprovecha las transformaciones para automatizar tareas repetitivas y mejorar la calidad del software, eleva el nivel de abstracción al ocultar detalles específicos de la plataforma. Sin embargo, varios son los obstáculos que se tienen que superar para que éste nuevo enfoque de desarrollo de software pueda ser utilizado por la industria (la tecnología orientación a objetos tardó dos décadas). Estos obstáculos se pueden entender desde varios ámbitos, como el tecnológico, económico, educativo, cultural y la falta de apoyo de las grandes empresas de desarrollo de herramientas de soporte.

Ámbito tecnológico

Dentro de los obstáculos tecnológicos se pueden destacar el cubrir y mejorar varios aspectos, como la sincronización de modelos, ingeniería directa e inversa, mecanismos para administración de modelos grandes y complejos, trazabilidad, gestión de modelos, soporte de herramientas de desarrollo, metodologías formales de desarrollo.

Ámbito económico

Varios estudios acerca de la aplicabilidad de MDE en la industria consideran que la empresa podría utilizar el nuevo paradigma de desarrollo siempre y cuando les permita mejorar la manera como actualmente lo realizan, lo que involucra disminuir el tiempo y costes del desarrollo, y aumentar la calidad del producto generado. El balance costo/beneficio al emplear MDE sería positivo a largo plazo, tras el desarrollo de un cierto número de proyectos, debido a que en etapas iniciales de su aplicación se requiere entre otros factores, capacitar a las personas e invertir en nuevas herramientas de automatización.

Ámbito educativo

La falta de profesionales capacitados adecuadamente en el proceso de desarrollo de software dirigido por modelos. Al ser una nueva tendencia de desarrollo los centros de educación podrían incluir en sus planes de estudio la revisión de MDE y propiciar la participación en proyectos de ToT destinados a incrementar el conocimiento en la industria, sobre todo en la nueva forma de pensar de los desarrolladores.

Ámbito cultural

Por lo general, se tiene como regla considerar que las personas tienden a hacer las cosas de la misma manera una y otra vez. Además representantes de la industria manifiestan que no sienten la necesidad de cambiar algo que por ahora funciona bien. Consideran que la nueva tecnología debe venir acompañada por el desarrollo de herramientas adecuadas, tutoriales y/o literatura especializada. La pregunta principal que una organización se hace es ¿realmente necesito MDE?, la segunda pregunta se relaciona a si tiene la capacidad para adaptar sus procesos al punto de vista de MDE. Las empresas pueden ser reacias a cambiar su estructura o parte de ella. El factor más importante en toda organización son las personas, se considera que hay personas que simplemente no quieren cambiar, ven lo negativo en todo, sin embargo el escepticismo a veces es realista, debido que hay personas que no pueden extrapolar un nuevo enfoque de desarrollo, simplemente no lo entienden o no lo quieren entender. Existen varias empresas que han empleado MDE en proyectos reales, mientras que otros piensan que MDE no es todavía lo suficientemente maduro como para ser considerada en proyectos a escala industrial.

Apoyo decidido de grandes empresas de desarrollo de herramientas de soporte

Varias empresas grandes de desarrollo de software han apoyado la iniciativa MDE pero no con el suficiente empeño o entereza. Por ejemplo IBM le ha apostado a dar impulso a MDA debido al apoyo del OMG, para el efecto ha creado una variedad de herramientas CASE para ser aplicados con este enfoque como Rational Software Architect (RSA), el inconveniente su elevado costo. Microsoft por su lado desarrolló DSL Tools que se puede integrar a su entorno de desarrollo Visual Studio. También existen iniciativas aunque no muchas de medianas empresas y de la comunidad investigadora.

Capítulo 4

Directrices de la Metodología

En este capítulo se determinan las directrices que debe considerar la metodología a proponer en base a la literatura revisada.

4.1 Definición de Metodología de Desarrollo de Software

Existen muchas definiciones de lo que es una metodología de desarrollo de software, todo depende de la fuente y el autor. A continuación se señalan dos de ellas:

- Metodología de desarrollo de software en ingeniería de software es un marco de trabajo usado para estructurar, planificar y controlar un proyecto de desarrollo, que permite llevarlo a cabo con altas posibilidades de éxito [84].

- Las metodologías imponen un proceso disciplinado sobre el desarrollo de software con el fin de hacerlo más predecible y eficiente. Lo hacen desarrollando un proceso detallado con un fuerte énfasis en planificar, inspirado por otras disciplinas de la ingeniería [34].

Una metodología de desarrollo de software contempla un conjunto de pasos y procedimientos que se deben seguir para desarrollar software, considera: las etapas en las que se debe dividir un proyecto, las tareas se llevan a cabo en cada etapa, las heurísticas para llevar a cabo dichas tareas, las salidas que se producen y cuándo se deben producir, las restricciones que se deben aplicar, las herramientas que se pueden utilizar, y las actividades de gestión y control del proyecto.

4.2 Criterios que debe considerar la metodología a proponer

Tomando en consideración las aportaciones de las investigaciones realizadas, se procede a formular un listado de los principales aspectos que debería tomar en cuenta la propuesta de la metodología a estructurar.

1. Basada en MDE y en DSL para permitir un mayor ámbito de manejo de la complejidad al no estar limitado únicamente a la utilización de UML. Si bien es cierto que UML es el estándar de modelado, el lenguaje no es del todo destinado a ser utilizado para resolver problemas particulares de un dominio específico [101].
2. Se debe delimitar el alcance de las aplicaciones en las que MDE puede proporcionar mayores beneficios en comparación con los enfoques tradicionales de desarrollo, debido a que el DSL está diseñado específicamente para las necesidades de un determinado problema o dominio específico.
3. La metodología debe estar basada en las mejores prácticas de las metodologías de desarrollo tradicional y ágiles.
4. Debe ser lo más simple posible de tal manera que permita a los desarrolladores entenderla y aplicarla rápidamente.
5. Se debe propender el involucramiento de forma directa de los usuarios y expertos del domino en el desarrollo de las herramientas MDE, con la finalidad de conformar un equipo colaborativo de trabajo principalmente en el desarrollo del DSL.
6. Se debe utilizar un proceso iterativo incremental que permita manejar la volatilidad de los requisitos y que proporcione al cliente beneficios del proyecto de forma rápida e incremental.
7. La metodología debe contar con una etapa preliminar que se encargue de conocer y determinar la visión y alcance del proyecto con la finalidad de poder conformar el equipo de desarrollo, determinar los aspectos de logística requeridos para el desarrollo del proyecto y sobre todo que permita identificar y representa las características del dominio de la aplicación.
8. Se debe considerar una segunda etapa encargada de crear el conjunto de artefactos MDE que permitan crear aplicaciones de un dominio específico. Artefactos como DSLs (metamodelos-editores textuales/gráficos), mecanismos de transformación de modelos, mecanismos de análisis de modelos, generadores de código, scripts de configuración y despliegue, casos de prueba, manuales, documentación.
9. Se debe incluir mecanismos de reutilización de artefactos como una opción para incrementar la productividad y calidad del proceso de desarrollo.
10. Considerar para el ámbito comercial, la creación de herramientas tipo CASE (Computer Aided Software Engineering) con tecnología MDE para diferentes áreas profesionales, debido a que una de los beneficios de MDE es que personal no informático pueda crear sus propias aplicaciones.
11. Proponer un mecanismo de desarrollo de aplicaciones ligeras con los artefactos MDE creados, tomando en consideración los principios del Desarrollo de Usuario Final.

4.3 Escenarios de desarrollo de aplicaciones con MDE y DSL

4.3.1 Escenarios para la creación de la solución MDE

La etapa de creación de la solución MDE contempla dos escenarios:

Aplicación a Medida: Si se pretende crear una solución MDE que permitan desarrollar una única aplicación que se ajusta 100% a las necesidades específicas de una organización (a medida), el enfoque MDE no es el recomendado debido a que se invertirá más esfuerzo y recursos para crear primero la solución MDE y luego la aplicación requerida.

Familia de Aplicaciones de Dominio Específico: Uno de los beneficios de MDE es la utilización de lenguajes de alto nivel (DSLs) para desarrollar aplicaciones de un dominio específico (particular) de forma automática mediante generadores de código. Aplicaciones de un dominio específico significa que comparten las mismas o la mayoría de sus características funcionales y arquitectónicas. Aprovechando esta situación se podría construir una solución MDE de un dominio específico que permitan desarrollar aplicaciones infinidad de veces.

4.3.2 Escenarios para el desarrollo de aplicaciones con soluciones MDE

La etapa de desarrollo de aplicaciones con soluciones MDE contempla dos escenarios:

Desarrollo de Aplicaciones Ligeras: En el caso de personas que requieren construir aplicaciones software pero que no tienen conocimientos de GPL como C, C++, Java, PHP, etc. Consiste en un conjunto de artefactos MDE, como DSLs y generadores de código, que están a disposición de personas de diferentes áreas profesionales, para que ellos mismo desarrollen la aplicación que requieran utilizando un DSL que les permite elevar el nivel de abstracción del problema, es decir empleando una notación que se ajuste a su área de conocimientos para diseñar un modelo mediante el cual se pueda generar código en un GPL y plataforma específica. Por ejemplo, Umbra Designer una herramienta para el desarrollo gráfico de servicios de telefonía "server-side", la herramienta facilita la construcción de servicios por personas no expertas [14]; también se puede mencionar a WebRatio que establece un entorno de desarrollo MDE que permite producir aplicaciones web Java y ejecutables en un entorno Web/SOA [98].

Desarrollo de Aplicaciones Complejas: Es el caso de empresas de tecnología que mediante la utilización de infraestructura tecnológica basada en MDE cubran parte o todas las fases del Ciclo de Vida del Desarrollo de Software (SDLC, Systems Development Life Cycle) de aplicaciones complejas de un determinado dominio específico de forma automática; se podrían desarrollar aplicaciones complejas, como el manejo de transacciones bancarias, desarrollo de aplicaciones de telefonía móvil, desarrollo de servicios de telecomunicaciones. Este conjunto de artefactos MDE al estar orientadas a aplicaciones de propósito específico sirven para generar aplicaciones de ese dominio, pero casi nunca en un 100%

debido a que se debe considerar que prácticamente no hay dos aplicaciones idénticas por más que exista un solapamiento significativo de los requisitos funcionales para una misma intención de aplicación [49], por lo que se tendrá que refinar los artefactos MDE manualmente para que cubra los nuevos requisitos. Por ejemplo, MasterCraft es un conjunto de artefactos MDE que cubre todo el SDLC mediante el uso de entornos de modelado y generadores de código [56].

4.4 Selección de los escenarios de desarrollo

La metodología propuesta deberá contemplar todos los criterios señalados en la sección 4.2. Por ello de acuerdo a los criterios 2 y 10, la metodología deberá considerar las actividades para el desarrollo de Aplicaciones Ligeras y de una Familia de Aplicaciones de Dominio Específico.

Capítulo 5

Propuesta de la Metodología

Para fines de referencia rápida la Metodología de Desarrollo de Software Dirigida por Modelos y Lenguajes de Dominio Específico propuesta, se la denominará MDM1 (Metodología Dirigida por Modelos Versión 1). La propuesta contempla fundamentalmente los temas tratados en capítulo de marco teórico y el impacto de MDE en la empresa. Se debe destacar además que se utilizan principalmente criterios de desarrollo ágil basados en Scrum debido a la utilización de sprints como mecanismo de iteración e incremento de los artefactos a ser creados.

5.1 Arquitectura de MDM1

ETAPA 1: PREPARACIÓN DEL PROYECTO

1. Determinación de la Visión del Producto
2. Conformación del Equipo de Desarrollo
3. Determinación de Aspectos de Logística
4. Análisis de Dominio

ETAPA 2: SPRINT

1. Planificar
   - Reunión de Planificación del Sprint
   - Elaboración de la Planificación del Sprint
2. Hacer
   - Implementación de tareas
3. Verificar
   - Revisión de ejecución de tareas
   - Reunión Diaria

   * Mini-sprint entre Hacer y Verificar
4. Revisar
   - Presentación de incremento
   - Reunión de Revisión de Incremento
5. Retrospectiva
   - Reunión de Retrospectiva

Procesos transversales de Gestión y Soporte

- Gestión del Proyecto
- Aseguramiento de la Calidad

Interacción con el Repositorio de Activos Reutilizables

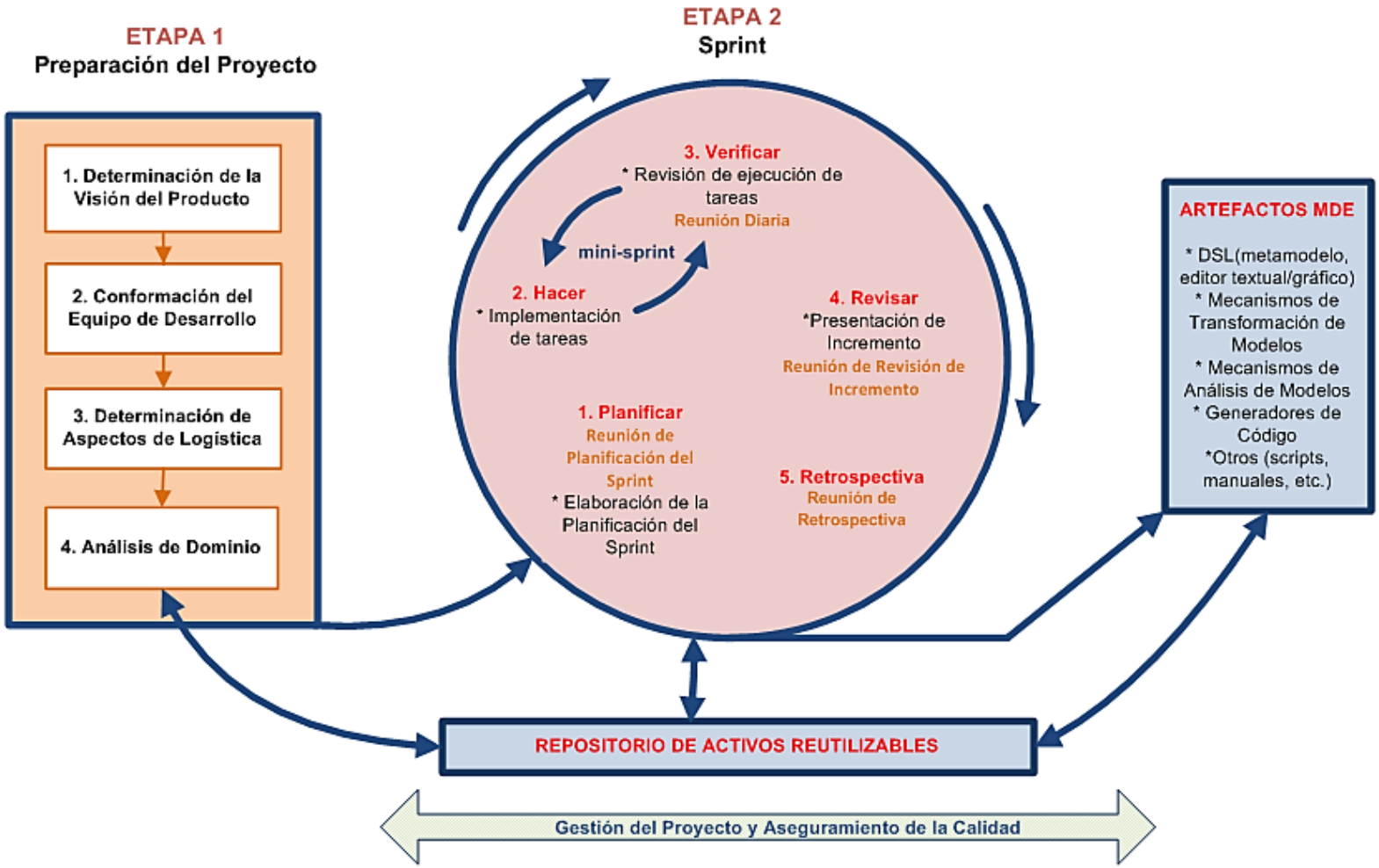


Figura 5.1: Representación gráfica de la Arquitectura de MDM1

5.2 Descripción de las actividades de MDM1

Esta sección describe cada una de las actividades técnicas y administrativas planteadas por la metodología, en algunos casos se ha creído pertinente realizar una explicación previa antes de describir la actividad con la finalidad de contextualizar el tema, de igual manera se incluyen en algunos casos propuestas de trabajos realizados con el objetivo de dar a conocer sus aportes.

5.2.1 ETAPA 1: PREPARACIÓN DEL PROYECTO

Contempla cuatro actividades: Determinación de la Visión del Producto, Conformación del Equipo de Desarrollo, Determinación de Aspectos de Logística, Análisis de Dominio.

5.2.1.1 Determinación de la Visión del Producto

Esta actividad se realiza en una primera reunión entre el interesado en contar con un conjunto de artefactos MDE y el representante de la empresa de desarrollo. Su objetivo es permitir que el representante de la empresa pueda conocer de forma general la visión y alcance del proyecto, producto de lo cual se determinará el conjunto de artefactos y servicios que se obtendrán.

Si se manifiesta interés por parte del interesado, se comienza un proceso de negociación, en el cual un representante de la empresa desarrolladora dará forma al proyecto y lo cotizará (en tiempo y dinero). Si el interesado acepta la Propuesta de Negocio se firmará un contrato, el interesado de esta manera se convierte en cliente e inicia el proyecto.

5.2.1.2 Conformación del Equipo de Desarrollo

Se encarga del staffing, es decir de la selección y entrenamiento de personas para que conformen el equipo de desarrollo. Al respecto se recomienda tomar en cuenta cinco principios de selección del personal [10]:

- Máximo talento. Usar poco y buen personal.
- Trabajo adecuado. Asignar tareas según la habilidad y motivación de la gente disponible.
- Progreso profesional. Ayudar a la gente a actualizarse por sí misma.
- Equilibrio de Equipo. Seleccionar personas que se complemente y armonice con los demás (capacidad de trabajo en grupo).
- Eliminar la inadaptación. Eliminar y reemplazar a los miembros problemáticos del equipo lo antes posible.

A más de los aspectos señalados anteriormente, el personal que conformaría el equipo de desarrollo debe contar con ciertas capacidades y conocimientos del entorno MDE, entre las principales:

- Conocimientos de programación orientada a objetos.
- Modelado de sistemas de software.
- Utilización de herramientas de modelamiento de software.
- Utilización de herramientas de generación de código automático.
- Conocimiento de plataformas tecnológicas clásicas y contemporáneas.
- Experiencia en reutilización de software.

5.2.1.3 Determinación de Aspectos de Logística

La finalidad de esta actividad es dar soporte al proyecto con las adecuadas herramientas, procesos y métodos. Brinda una especificación de las herramientas que se van a necesitar en cada momento, así como define cada instancia concreta del proceso que se va a seguir. Dentro de las responsabilidades de esta actividad se señalan las siguientes:

- Seleccionar y adquirir hardware, software base y de desarrollo (espacio tecnológico MDE).
- Establecer y configurar las herramientas para que se ajusten a lo requerido.
- Configurar y mejorar los procesos.
- Capacitar al personal en caso de requerirlo.
- Encargarse del soporte técnico.

5.2.1.4 Análisis de Dominio

En vista de que MDM1 está orientada a la creación de artefactos MDE (algunos autores lo denominan herramientas) para el desarrollo de Aplicaciones Ligeras y de una Familia de Aplicaciones de Dominio Específico, el análisis de domino permitirá determinar y representar las características de un determinado dominio de aplicación, para establecer un modelo o patrón común que permita representar su conocimiento mediante lenguajes y herramientas específicas que faciliten su posterior reutilización [83]. Por lo cual esta actividad permitirá obtener los elementos para estructurar el metamodelo, para que sobre éste, en la siguiente etapa se construyan los artefactos MDE.

Como sucede con cualquier modelo, la creación de un metamodelo es una actividad de modelado conceptual, y en el contexto de MDE se suele aplicar un modelado conceptual orientado a objetos en el que los metamodelos se crean a partir de conceptos de clase, atributo, agregación, referencias y generalización, por lo que se puede representar un metamodelo mediante un diagrama de clases UML, o en forma de un árbol de agregación de elementos. De este modo, el diseño de un metamodelo se puede abordar de la misma forma que se realiza el modelado de

cualquier dominio de aplicación, solo que ahora se modela un lenguaje y se trata de representar los conceptos y relaciones entre ellos que subyacen en él. Si se utiliza los conceptos orientados a objetos, el modelado de un DSL se puede realizar teniendo en consideración que [29]:

- Los conceptos del lenguaje se modelan como clases (las metaclases del metamodelo).
- Las propiedades de los conceptos, como atributos de una metaclase.
- Las relaciones entre conceptos como referencias (asociaciones dirigidas) entre metaclases y en particular si se trata de relaciones de agregación (también conocidas como "parte de") o de composición.
- La especialización de un concepto como una relación de generalización entre metaclases (también se puede utilizar el término "herencia de clases" aunque no es lo más apropiado).
- La definición del lenguaje se encapsula en un paquete.

MDM1 propone la realización de un análisis de dominio previo a la construcción del metamodelo. Para ello, utiliza una versión adaptada del método FODA debido a que éste fue desarrollado a inicios de la década de los 90 y hace uso de diagramas y modelos que de cierta manera se los podría considerar en la actualidad obsoletos (diagramas de flujo de datos, modelos entidad-relación, etc.), razón por la cual se proponen tres productos a desarrollar en las actividades de Análisis de Contexto y el Modelado del Dominio. El mecanismo a ser utilizado para lograrlo es a través de una serie de reuniones entre los usuarios (expertos del dominio) y los desarrolladores (expertos en MDE). Para lo cual se podría utilizar una adaptación de uno de los tres escenarios descritos en la sección 2.1.5 (Técnicas de diseño de DSL).

Análisis de Contexto

En esta actividad interactúan los expertos del dominio y el analista de requisitos para delimitar el alcance y especificar los requisitos del sistema del dominio propuesto. Para ello el analista recoge información de fuentes primarias, principalmente de la opinión de expertos del dominio. También puede recurrir a otras fuentes como estándares, análisis de otros sistemas de dominios similares, manuales, entre otros. En el caso de la metodología propuesta en esta fase se debe elaborar la Lista de Requisitos del Dominio expresado en lenguaje natural, esta lista debe especificar los requisitos funcionales, no funcionales, de índole legal y/o reglamentaria. La Lista de Requisitos del Dominio constituye el principal documento donde se describen las funcionalidades del sistema hasta un determinado momento del proyecto. Debido a que se considera que los requisitos pueden cambiar durante el desarrollo del proyecto. Situación que no se debe temer y que no debe tener mayor impacto en el desarrollo del proyecto debido a los criterios de agilidad que adopta la metodología.

García en [29], señala que de alguna forma el proceso de desarrollo es un proceso en el que se pasa de modelar el espacio del problema a obtener modelos para el espacio de la solución, y que en el caso de MDE se lo consigue a través de la automatización de la transformación de modelos.

Por lo señalado, y siendo la manera de desarrollar software con cualquier enfoque de desarrollo, donde en primera instancia hay que abstraer el problema, expresarlo por lo general utilizando lenguaje natural para luego hacer uso de modelos que permitan representar el problema y plantear una solución, a este proceso se lo podría referenciar como una transformación Texto-a-Modelo (T2M).

Modelado del Dominio

Esta actividad realiza una discriminación de los datos, funcionalidades y conceptos propios del dominio. Evalúa y selecciona los elementos susceptibles de reutilización. Por lo cual se debe elaborar por parte del equipo de desarrollo el Modelo de Características que captura las características fundamentales de los sistemas del dominio analizado, y un Diagrama de Clases que represente los objetos (clases) del dominio y sus relaciones**.**

Ejemplo de los artefactos obtenidos en el análisis de dominio de un proyecto

Para poder ilustrar los artefactos (productos) y los formatos utilizados en el análisis de domino se hace uso de un proyecto de desarrollo de una cola de simulación para optimizar el proceso de abordaje de pasajeros a un avión, tomado de [16], el cual ha sido adaptado a las necesidades particulares de la presente propuesta.

Como se mencionó en la fase de Análisis del Contexto se obtiene la Lista de Requisitos del Dominio, para ello se recomienda la utilización de la plantilla que se muestra en la Tabla 5.1:

Tabla 5.1: Lista de Requisitos del Dominio

PROYECTO: Desarrollo de una cola de simulación para optimizar el proceso de abordaje de pasajeros a un avión. **OBJETIVO:** Optimizar el proceso de atención a los pasajeros en el check-in y abordaje. **USUARIOS:** Supervisores de operaciones del terminal. Personal de gestión del aeropuerto. Gestores de terminales de la compañía aérea.	**Cliente:** Aeropuerto de Barajas - Madrid **Experto de Dominio:** Carlos Contreras **Administrador del Proyecto:** Marco Andrade

LISTA DE REQUISITOS DEL DOMINIO

Id.	Descripción de la funcionalidad	Prioridad	Riesgo	Observaciones
Requisitos Funcionales				
1	Señalar si el operador de check-in está atendiendo o no (pueden habe varias colas de atención)	alta	alto	
2	Representar dinámica y gráficamente la llegada de los pasajeros a la cola para realizar el check-in	alta	alto	
3	Representar dinámica y gráficamente la atención del operador de check-in	alta	alto	
4	Representar dinámica y gráficamente la llegada del pasajero al kiosco de atención de check-in	alta	alto	
5	Registrar equipaje del pasajero	alta	alto	
6	Incluir procesos de seguridad de vuelos	alta	medio	
7	Representar dinámica y gráficamente la llegada de los pasajeros a la cola de abordaje del avión	alta	alto	
8	Especificar si el orden de abordaje es por prioridad o por número de asiento asignado	media	bajo	
9	Especificar si el desplazamiento de los pasajeros al avión es por un corredor o mediante el desplazamiento en bus	media	bajo	
10	Se deberá señalar la capacidad de pasajeros del avión	alta	bajo	
11	Se deberá señalar el número de pasajeros que han abordado	alta	bajo	La capacidad será proporcianada de acuerdo al tipo de avión
12	Se indicará el tiempo total de atención a los pasajeros en la cola de check-in	baja	bajo	
13	Se indicará el tiempo promedio de atención a los pasajeros en la cola de chek-in	baja	bajo	
14	Se indicará el tiempo total de abordaje de todos los pasajeros en el avión	baja	bajo	
15	Se indicará el tiempo promedio de atención a los pasajeros en el abordaje del avión	baja	bajo	
Requisitos No Funcionales				
1	El sistema debe residir en el servidor del terminal aéreo, donde los operarios podrán acceder a el a través de portátiles o dispositivos móviles.	alta	medio	
2	El sistema no deberá proporcionar a los operarios información confidencial de los pasajeros.	alta	alto	

* La prioridad y riesgo de los requisitos está ligada a la prioridad y riesgo en la implementación de la funcionalidad

Con la Lista de Requisitos del Dominio se procede a realizar un Diagrama de Características (Tabla 5.2) con la finalidad de identificar y representar las características del dominio. El mecanismo considerado para el efecto es el de identificar los términos y conceptos más relevantes de la Lista de Requisitos del Dominio para ser representados como características en el Modelo de Características. Es podría interpretarse como una transformación de Texto-a-Modelo, que permite "visualizar" las características del sistema de simulación de colas, de forma sencilla, agrupada y con un significado derivado de la notación empleada para realizar el modelo.

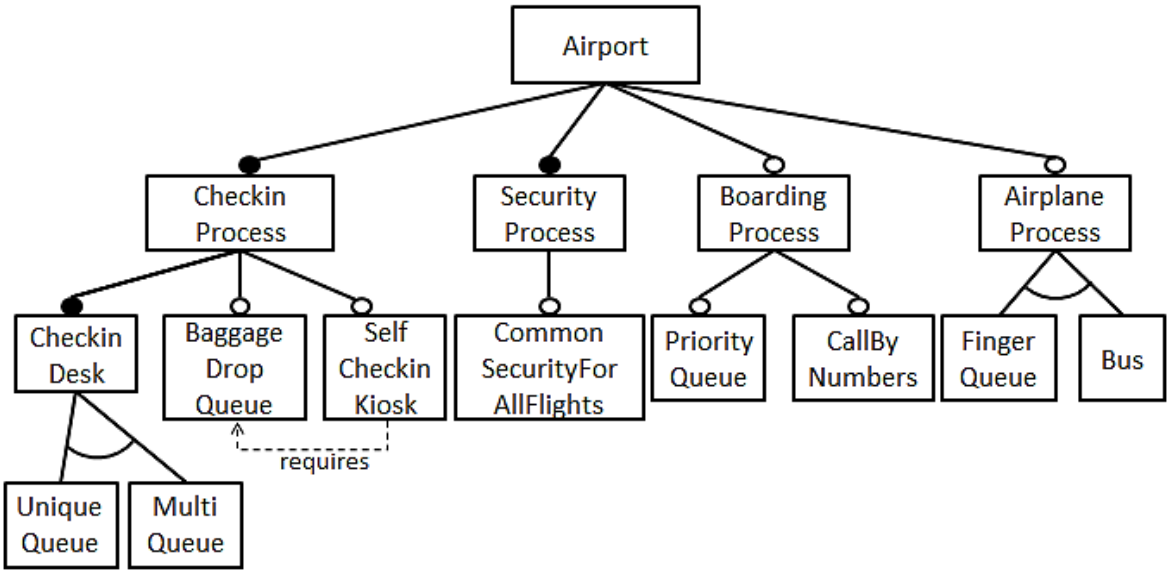


Figura 5.2: Modelo de Características

La siguiente actividad toma como insumo de entrada el Diagrama de Características y la Lista de Requisitos del Dominio con los cuales se procede a realizar el Diagrama de Clases. La Figura 5.3, muestra una primera versión del Diagrama de Clases de sistema analizado. El mecanismo empleado para el efecto es el de tomar las características del Modelo de Características, las mismas que son consideradas "clases candidatas" del Diagrama de Clases. Se debe considerar que en el desarrollo tradicional de software las clases representan los conceptos relevantes del dominio del sistema. Las relaciones entre las características representan relaciones de agregación, referencias y generalización, todo dependerá del diseño del Diagrama de Clases. Los atributos de las clases se deben tomar de la Lista de Requisitos del Dominio, debido a que el Diagrama de Características no permite representar las propiedades de las características identificadas. Mientras que las restricciones especiales habrá que representarlas mediante OCL. El Diagrama de Clases creado es considerado la primera versión del metamodelo del dominio analizado. El metamodelo podrá ser refinado si así se lo requiere en la siguiente etapa de la metodología propuesta.

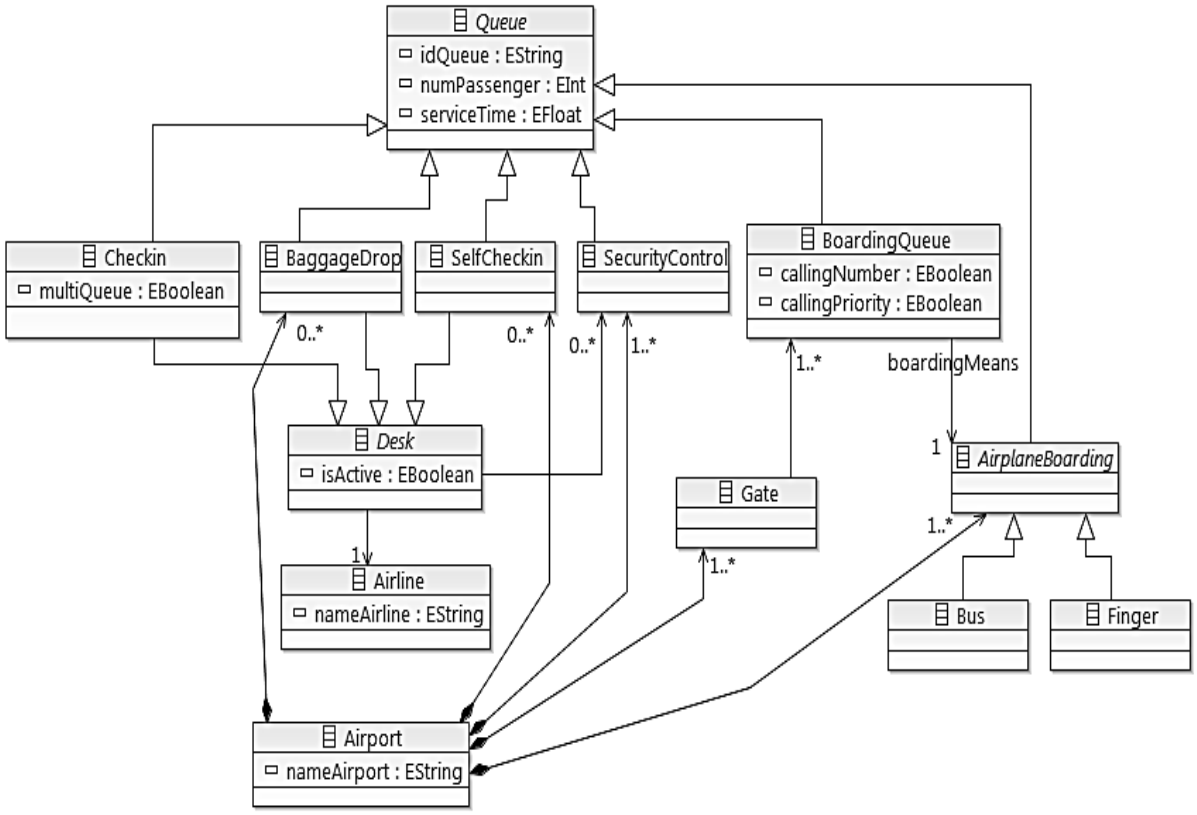


Figura 5.3: Diagrama de Clases

Alternativa para realizar el Análisis de Dominio

Como se mencionó anteriormente existen varias métodos para realizar el análisis de dominio, en el trabajo realizado por Tairas et al. en [91] se propone una técnica en base a la utilización de ontologías en el análisis de dominio para diseñar lenguajes de dominio específico. El uso de ontologías puede ayudar a la determinación de la información pertinente en un determinado dominio (por ejemplo, la conceptualización del dominio y la determinación de los elementos comunes y variables del dominio) que debe ser modelada en un lenguaje de un dominio específico. A pesar de la propuesta de este método, en general se carece de directrices claras para el uso de técnicas de análisis de dominio en el proceso de desarrollo de DSLs [58].

Chandrasekaran et al. en [18] propone dos elementos relacionadas con ontologías: el primero es un vocabulario de representación de algún dominio especializado, esto representa el vocabulario de objetos, conceptos y otras entidades sobre el dominio; el segundo es el cuerpo de conocimiento del dominio utilizando este vocabulario representativo. Este conocimiento puede obtenerse a partir de las relaciones de las entidades que han sido representados por el vocabulario. Las ontologías buscan representar los elementos de un dominio a través de un

vocabulario y las relaciones entre estos elementos con el fin de proporcionar algún tipo de conocimiento del dominio. La relación entre una ontología y el diseño del DSL se relaciona en base a un conjunto de componentes:

- La definición del dominio que determina su alcance.
- La terminología del dominio (establecimiento del vocabulario - ontología).
- La descripción de los conceptos del dominio.
- Un modelo de características que describa las similitudes y variaciones de los conceptos del dominio y sus interdependencias.

A continuación se señalan los elementos generados por el método propuesto para un dominio de Control del Tráfico Aéreo (ATC) en un aeropuerto [91]. El propósito es desarrollar un DSL que pueda estandarizar el lenguaje de la comunicación entre el piloto y la torre de control, si bien el inglés es el lenguaje estándar en este ámbito, los pilotos de países de no-habla inglesa pueden experimentar problemas de comunicación.

Tabla 5.2: Ontología propuesta para el análisis de dominio

Class	Description	Slots		
		Name	Description	Values
`Aircraft`	Arriving or departing aircraft	`Airline ID`	Name of the airline	Two letters
		`Flight Number`	Flight Identification	Integer
`GroundControl`	Controller in charge of airport ground traffic			
`Tower`	Controller in charge of take-offs and landings			
`Runway`	Available take-off and landing locations	`Runway Number`	Runway Identification	1 – 36 (i.e., runway heading 10° – 360°)
		`Runway Orientation`	To distinguish parallel runways	Class `Left` or `Right`
`Taxiway`	Paths connecting runways to ramps	`Taxiway Name`	Taxiway Identification	One or two letters (digits)
`Ramp`	Aircraft parking area	`Ramp Name`	Ramp Identification	String
`Gate`	Passenger embarkation and disembarkation	`Gate Letter`	Gate Identification	One letter
		`Gate Number`	Gate Identification	Integer

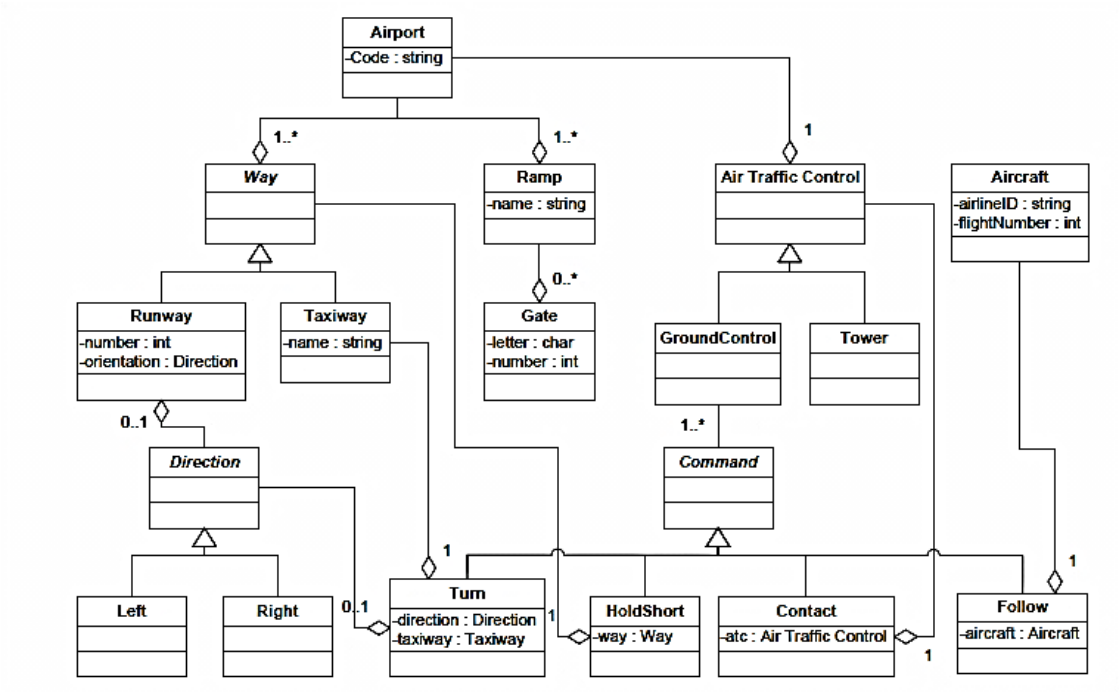


Figura 5.4: Diagrama Conceptual de Clases obtenido de la ontología definida

El método propuesto además permite desarrollar la sintaxis concreta (textual) a través de la definición de reglas OO, lo hace a través del formato de definir una Gramática Libre de Contexto (GLC). Las actividades representadas en la Tabla 5.3 y Figura 5.5 de acuerdo a MDM1 serían parte de la siguiente etapa (Sprint).

Tabla 5.3: Transformación del Diagrama de Clases Conceptual a una GLC

```
AIRPORT       ::= WAYS RAMPS ATC
WAYS          ::= WAYS WAY | WAY
WAY           ::= RUNWAY | TAXIWAY
RUNWAY        ::= number DIRECTION
TAXIWAY       ::= name
RAMPS         ::= RAMPS RAMP | RAMP
RAMP          ::= name GATES
GATES         ::= GATES GATE | ε
GATE          ::= letter number
ATC           ::= GROUNDCONTROL | TOWER
GROUNDCONTROL ::= COMMANDS
COMMANDS      ::= COMMANDS COMMAND | COMMAND
COMMAND       ::= CONTACT | FOLLOW | HOLDSHORT | TURN
CONTACT       ::= ATC
FOLLOW        ::= AIRCRAFT
HOLDSHORT     ::= WAY
TURN          ::= DIRECTION TAXIWAY
DIRECTION     ::= LEFT | RIGHT | ε
AIRCRAFT      ::= airlineID flightNumber
```

```
// description of BHM airport
runway 6 runway 24 runway 18 runway 36
taxiway A taxiway A1 taxiway A2 taxiway A3 taxiway A4 taxiway B  taxiway B1
taxiway F taxiway G  taxiway G1 taxiway H  taxiway H2 taxiway H4
ramp Cargo
ramp Terminal gate B1 gate B2 gate B3 gate C1 gate C2 gate C3 gate C4 gate C5

// commands from Ground Control
turn right on A
turn left on M
hold short runway 18
contact tower
```

Figura 5.5: Ejemplo de un programa diseñado con la GLC

5.2.2 ETAPA 2: SPRINT

Este proceso considera la creación de los artefactos MDE a través de iteraciones denominadas "sprints", cada uno de los sprints desarrollará una versión funcional de todos los artefactos MDE que se consideren, e ir evolucionando en cada sprint hasta alcanzar la solución MDE esperada. De acuerdo a recomendaciones de criterios ágiles de desarrollo, cada sprint debe tener una duración de dos a cuatro semanas. Otra de las características importantes de este proceso es la realización de reuniones a lo largo del proyecto, consideradas como actividades de gestión del proyecto y control de calidad de los artefactos creados. Para ello se establece una reunión al inicio de cada sprint para determinar el trabajo que se va a realizar, reuniones diarias que realiza el equipo para su autogestión, una reunión para evaluar el resultado de cada sprint, y una reunión de retrospectiva (feedback) de las actividades realizadas en el sprint.

Cada sprint contempla cinco actividades: Planificar, Hacer, Verificar, Revisar y Retrospectiva, las cuales son derivadas de la estrategia de Mejora Continua de la Calidad establecida por W. Edwards Deming, en el denominado Ciclo de Deming [38] o también conocido Modelo de Mejora Continua de Procesos, o simplemente Círculo PDCA, siglas del acrónimo de Plan, Do, Check, Act (Planificar, Hacer, Verificar, Actuar), MDM1 modifica brevemente las actividades del proceso Actuar y se lo renombra como Revisión, también se considera al finalizar cada sprint la realización de una reunión de retrospectiva.

Los principales artefactos MDE que deben ser creados (de acuerdo al proyecto) en esta etapa son DSLs (metamodelos y editores textuales/gráficos), mecanismos de transformación de modelos, mecanismos de análisis de modelos, generadores de código; también se deben considerar, elementos como scripts de configuración y despliegue, script de base de datos, casos de prueba, diseños, documentación, manuales, y los que se requieran.

5.2.2.1 Planificar

Esta es la primera de las actividades del sprint en la cual se realiza la Reunión de Planificación del Sprint tendiente a obtener la planificación de actividades y tareas del sprint.

REUNIÓN DE PLANIFICACIÓN DEL SPRINT

Esta actividad se inicia tomando en consideración los productos generados en el análisis de dominio que permitió determinar y representar las características del dominio de aplicación a través de la Lista de Requisitos del Dominio, Modelo de Características y una versión inicial del Diagrama de Clases que representa la base del metamodelo [53], sobre el cual se deben crear los demás artefactos MDE.

Cada uno de los sprints tomará como entrada la porción de la Lista de Requisitos del Dominio que más alta prioridad se tenga en ese momento, las características y la porción del diagrama de clases que los representen, de ser el caso se los volverá a analizar en la Reunión de Planificación del Sprint. Con estos elementos se procurará diseñar y construir una versión definitiva del metamodelo, con el cual se podrá crear su representación textual o gráfica, con la versión creada del DSL inmediatamente se puede probar las características del nuevo lenguaje (semántica) mediante la construcción de un modelo real básico, para finalmente crear una versión del generador para comprobar que en realidad se pueden generar artefactos pertinentes. Cabe señalar que este proceso debe comprometer la participación activa y permanente de los usuarios y del experto del domino.

Con el procedimiento y actividades señaladas, en la Reunión de Planificación del Sprint se debe elaborar la Planificación del Sprint para cada sprint, considerado éste como instrumento de planificación, seguimiento y control del progreso del proyecto. La Reunión de Planificación del Sprint constituye una jornada de trabajo previa al inicio de cada sprint, en la que se determina cuál va a ser el trabajo y los objetivos que se deben conseguir en cada sprint. Se trata de una reunión conducida por el responsable del funcionamiento de la metodología, en esta caso denominado Administrador del Proyecto y a la que deben asistir el Cliente (Experto del Dominio) y el Equipo de Desarrollo, y a la que también pueden asistir otros implicados en el proyecto.

En esta reunión debe encontrar presente el experto de dominio debido a que en el caso de que exista confusión en los requisitos él sería el encargado de realizar las correspondientes aclaraciones, más aún pueden producirse cambios en los requisitos de la Lista General del Dominio desde la última versión revisada. Situación que no debería afectar mayormente el progreso del proyecto debido a las planificaciones de cada sprint (flexibilidad del desarrollo ágil).

Pre-condiciones

- La organización tiene determinados los recursos disponibles para llevar a cabo el sprint.
- El experto del dominio conoce perfectamente el dominio de la aplicación que se debe desarrollar, la misma que representa las necesidades del negocio.

- El equipo tiene conocimientos suficientes de las tecnologías empleadas y del negocio del producto, como para realizar estimaciones basadas en "juicio de expertos", y para comprender los conceptos del negocio que expone el propietario del producto.

Entradas

- Lista de Requisitos del Dominio actualizada.
- Los artefactos desarrollados hasta la fecha a través de los sucesivos incrementos (excepto si se trata del primer sprint, donde se utilizan los elementos del Análisis de Dominio).
- Consideraciones del negocio del cliente y del escenario tecnológico empleado. Pueden cambiar el escenario comercial y tecnológico de la solución MDE mientras se la está creando.

Resultados

- Objetivo del sprint.
- Especificación detallada de las actividades y tareas en la Planificación del Sprint.
- Duración estimada en completar las tareas del sprint.
- Hora de las Reuniones Diarias.
- Fecha y Hora de Reunión de Incremento.
- Fecha y Hora de Reunión de Retrospectiva.
- Fecha y Hora de Reunión de Planificación del próximo Sprint.

Formato de la reunión

- Esta reunión marca el inicio de cada sprint.
- Duración máxima de la reunión: un día.
- Deben asistir: el experto del dominio, el equipo de desarrollo y el administrador del proyecto.
- Pueden asistir todos aquellos interesados (asesoran y observan) que aporten información útil, ya que es una reunión abierta.

Consta de dos partes separadas por una pausa de café o comida, según la duración.

Primera parte de la reunión (duración de 1 a 4 horas)

Experto del Dominio

- Presenta las funcionalidades de la Lista de Requisitos del Dominio en el caso del primer sprint. Si han habido modificaciones durante los siguientes sprints, las expone.
- En el caso de dudas, estas deben ser despegadas con un nivel de detalle suficiente para transmitir al equipo toda la información necesaria para desarrollar el incremento.

Equipo de Desarrollo

- Realizar preguntas y solicita las aclaraciones necesarias.
- Propone sugerencias, modificaciones y soluciones alternativas.

El papel del experto del dominio es atender a dudas y comprobar que el equipo comprende y comparta su objetivo. El administrador del proyecto actúa de moderador de la reunión. Los aportes del equipo pueden suponer modificaciones a la Lista de Requisitos del Dominio. Esta reunión es un punto importante de la metodología para favorecer la fertilización cruzada de ideas y añadir valor a la visión del producto. Tras reordenar y replantear las funcionalidades de la Lista de Requisitos de Dominio si así se lo requiere, el equipo define el "objetivo del sprint" o frase que sintetiza cuál es el valor que se le va a entregar al cliente. Exceptuando sprints dedicados exclusivamente a refactorización o a una colecciones de tareas de correcciones y ajustes (que deberían ser lo menos posible), la elaboración de este lema de forma conjunta en la reunión es una garantía de que todo el equipo comprende y comparte la finalidad del trabajo; y durante el sprint sirve de criterio de referencia en las decisiones que auto gestiona el equipo.

Segunda parte de la reunión (duración de 1 a 4 horas)

Asiste el administrador del proyecto y el equipo de desarrollo. El equipo de trabajo toma como base la funcionalidad de los artefactos MDE implementada hasta el momento, la Lista de Requisitos del Dominio actualizada, las analiza y las traslada a actividades y tareas relacionadas al desarrollo del siguiente sprint, las mismas que deben ser registradas en la Plantilla de Planificación del Sprint. Esta segunda parte debe considerarse como una "reunión del equipo", en la que deben estar todos los miembros y ser ellos quienes descomponen, estiman y asignan el trabajo.

Administrador del Proyecto

El Administrador del Proyecto es responsable y garante de:

1. Crear una atmósfera que apoye y motive a la gente para alcanzar los resultados finales deseados.
2. Realizar esta reunión antes de cada sprint.
3. Ayudar a mantener el diálogo entre el experto del domino y el equipo de desarrollo.
4. Ayudar al equipo de desarrollo a comprender la visión y necesidades del negocio del cliente.
5. Asegurar que el equipo de desarrollo ha realizado una descomposición y estimación del trabajo realista, y ha considerado las posibles tareas necesarias de análisis, investigación o apoyo.

ELABORACIÓN DE LA PLANTIFICACIÓN DEL SPRINT

La Planificación del Sprint la realiza el equipo de desarrollo con supervisión del administrador del proyecto durante la segunda parte de la Reunión de Planificación del Sprint, asignando cada tarea a una o más personas dependiendo de su complejidad y criticidad dentro del proyecto. Esta información deberá ser registrada en la Plantilla de Planificación del Sprint en la que también se debe registrar el tiempo estimado en terminar cada tarea. Esto resulta útil porque descompone el proyecto en unidades de tamaño adecuadas para determinar el avance diario e identificar el riesgo y problemas de retrasos sin necesidad de procesos complejos de gestión. Es también una herramienta de soporte para la comunicación directa del equipo.

Condiciones

- Realizada de forma conjunta por todos los miembros del equipo, es decir por el administrador del proyecto y el equipo de desarrollo.
- Cubre todas las tareas identificadas por el equipo para conseguir el objetivo del sprint.
- Sólo el equipo de desarrollo lo puede modificar durante el sprint con autorización del administrador del proyecto.
- El tamaño de cada tarea está en un rango de 2 a 24 horas de trabajo.
- Es visible para todo el equipo de desarrollo. Para ello se puede utilizar una pizarra en la pared en el mismo espacio físico donde trabaja el equipo, una hoja de cálculo, o mediante el empleo de una herramienta propia de gestión de proyectos.

Plantillas de Planificación de Sprint

Partiendo de la consideración de que las actividades y tareas que se tienen que realizar para la creación de artefactos MDE en base a DSL presentan prácticamente un patrón similar (esto dependerá de las necesidades del proyecto), en el sentido de crear un metamodelo en base a los requerimientos del usuarios, definir una sintaxis textual o gráfica (no estrictamente necesario para definir generadores de código o transformaciones de modelos), implementar mecanismos de transformación y análisis de modelos, y desarrollar los generadores de código; la presente propuesta metodológica plantea la utilización de Plantillas de Planificación del Sprint predefinidas para cada iteración (sprint). Como ejemplo ilustrativo se diseña y muestra la Plantilla de Planificación de Sprint para el primer sprint, Tabla 5.4, sin que esto signifique que las actividades y tareas sean fijas, simplemente se pretende guiar e ilustrar en su contenido. De igual manera las Plantillas de Planificación del Sprint de las otras iteraciones (sprints) utilizarían el mismo formato y criterio.

Tabla 5.4: Plantilla de Planificación de Sprint para el primer sprint

PROYECTO:	**Cliente:** Aeropuerto de Barajas - Madrid
Desarrollo de una cola de simulación para optimizar el proceso de abordaje de pasajeros a un avión.	**Experto de Dominio:** Carlos Contreras **Administrador del Proyecto:** Marco Andrade **Inicio:** Lun 2, Sep. de 2013 **Fin:** Vie 15, Nov. de 2013

PLANIFICACIÓN DEL SPRINT	**Fecha:**													
	Tareas pendientes													
	Horas pendientes													

Sprint No: 1	**Inicio:**	**Fin:**				**Objetivo del Sprint**											
Categoría	**Actividad**	**Tareas**	**Responsable**	**Estimando (horas)**	**Estado**	**Proveer de funcionalidad inicial básica a los artefactos MDE**											
Metamodelo	Definición de la sintaxis abstracta del DSL	1. Revisión de documentos del Análisis de Dominio															
		2. Refinamiento de documentos de Análisis de Domio (Lista de Requisitos del Dominio, Modelo de Características y Diagrama de Clases)															
		3. Selección de los requisitos de mayor nivel de prioridad															
		4. Construcción de la primera versión del metamodelo (tendiente a obtener la versión definitiva)															
		5. Evaluación y mejoras del metamodelo															
Editor	Definición de la sintaxis concreta del DSL	1. Definición de los elementos textuales/gráficos del DSL															
		2. Construcción del editor															
		3. Evaluación y mejoras del editor															
	Prueba de semántica del DSL	4. Utilización del DSL para diseñar un modelo real básico															
Transformación	Creación de mecanismos de transformación	1. Definición de reglas de transformación															
		2. Desarrollo del programa de transformación															
		3. Ejecución del programa en un motor de transformación															
		4. Evalución y pruebas de la transformación															
Generador	Desarrollo del generador de código	1. Determinación del GPL para el cual generar código															
		2. Selección del mecanismo de generación de código															
		3. Desarrollo del generador de código															
		4. Pruebas de generación de código															
	TOTAL HORAS PLANIFICADAS:																

Reuniones Diarias:	Reunión de Revisión de Incremento:	Reunión de Retrospectiva:	Reunión de Planificación de próximo Sprint:

5.2.2.2 Hacer

Se refiera a la implementación de las actividades y tareas especificadas en la Planificación del Sprint, cada elemento implementado deberá ser probado para verificar que su funcionamiento esté acorde con lo detallado en el análisis y diseño. Para el efecto se debe utilizar la infraestructura hardware y software que permita construir los diferentes artefactos MDE. Puede ser que en la implementación de algunos artefactos se entre en un ciclo iterativo incremental (mini-sprint) entre esta actividad y la de verificación para determinar el cumplimiento de las tareas. Por lo cual se deberá corregir los defectos encontrados lo más pronto posible.

5.2.2.3 Verificar

Esta actividad contempla la revisión de la ejecución de las tareas planificadas en el sprint, a través de Reuniones Diarias de todo el equipo de desarrollo.

REVISIÓN DE EJECUCIÓN DE TAREAS

Debido a la utilización de criterios de desarrollo ágil, donde se prioriza la calidad y rapidez de la implementación de las tareas, se debe realizar una verificación de que las actividades se van cumpliendo de acuerdo a lo planificado. Esto se logra mediante una revisión y control diario de las tareas desarrolladas para evaluar si se ha producido lo esperado a través de la realización de reuniones diarias, en las cuales utilizando la Plantilla de la Planificación del Sprint se va registrando el avance del desarrollo de los artefactos MDE. En este punto la Plantilla de Revisión del Sprint se convierte en una especie de Cuadro de Mando del Proyecto.

REUNIÓN DIARIA

Esta reunión se la desarrolla al inicio de cada jornada laboral con una duración de no más de 20 minutos, donde cada miembro expone al resto del equipo tres cuestiones:

1. Tarea en la que trabajó ayer.
2. Tarea o tareas en las que trabajará hoy.
3. Si ha tenido algún inconveniente, o va a necesitar alguna cosa especial o prevé algún impedimento para realizar el trabajo.

El administrador del proyecto actualiza la Plantilla de Planificación del Sprint con las tareas completas, y los tiempos de trabajo que les quedan a las otras tareas.

Pre-condiciones

- Los miembros del equipo han desarrollado las tareas definidas.
- Se han identificado inconvenientes en el caso de haberlos.

Entradas

- Plantilla Planificación del Sprint actualizada con la información de la reunión anterior.
- Información de las tareas realizadas por cada integrante del equipo.

Resultados

- Plantilla de Planificación del Sprint actualizada.
- Identificación de necesidades e impedimentos.

FORMATO DE LA REUNIÓN

Se recomienda realizarla de pie y empleando una proyección en una pizarra o pared de la Plantilla de Planificación del Sprint, para que todo el equipo pueda ver y aportar. En la reunión está presente todo el equipo, y pueden asistir también otras personas relacionadas con el proyecto o la organización, pero estas no pueden intervenir. Cada persona proporciona información de las tareas asignadas para la actualización de la Plantilla de Planificación del Sprint, se marca como Tarea Completa la tarea finalizada (color verde), como Tarea Retrasada (color rojo) la que lleva más tiempo de lo estimado, y como Tarea Pendiente (ningún color) la tarea que no ha sido completada pero se encuentran dentro del tiempo estimado.

Al final de la reunión, el equipo refresca la Plantilla de Planificación del Sprint, con las estimaciones actualizadas. El administrador del proyecto comienza la gestión de necesidades e impedimentos identificados. La Tabla 5.5 muestra la utilización de la Plantilla de Planificación del Sprint para realizar el seguimiento y control del proyecto.

Tabla 5.5: Seguimiento y control del proyecto en base a la Plantilla de Planificación del Sprint

PROYECTO:	**Cliente:** Aeropuerto de Barajas - Madrid **Experto de Dominio:** Carlos Contreras **Administrador del Proyecto:** Marco Andrade **Inicio:** Lun 2, Sep. de 2013 **Fin:** Vie 15, Nov. de 2013
Desarrollo de una cola de simulación para optimizar el proceso de abordaje de pasajeros a un avión.	

PLANIFICACIÓN DEL SPRINT	**Fecha:**	Lun, 9 Sep.	Mar, 10 Sep.	Mie, 12 Sep.	Jue, 12 Sep.	Vie, 13 Sep.	Lun, 16 Sep.	Mar, 17 Sep	Mie, 18 Sep.	Jue, 19 Sep	Vie, 20 Sep	Lun, 23 Sep.	Mar, 24 Sep.	Mie, 25 Sep.
	Tareas pendientes	17	14	14	14	13	12							
	Horas pendientes	124	116	108	100	94								

Sprint No: 1	**Inicio: Lunes 9, Sep.**	**Fin: Martes, 1 Oct.**				**Objetivo del Sprint**												
Categoría	**Actividad**	**Tareas**	**Responsable**	**Estimando (horas)**	**Estado**	**Proveer de funcionalidad inicial básica a los artefactos MDE**												
Metamodelo	Definición de la sintaxis abstracta del DSL	1. Revisión de documentos del Análisis de Dominio	José y María	2	Completa	2												
		2. Refinamiento de documentos de Análisis de Domio (Lista de Requisitos del Dominio, Modelo de Características y Diagrama de Clases)	José y María	4	Completa	4												
		3. Selección de los requisitos de mayor nivel de prioridad	José y María	2	Completa	2												
		4. Construcción de la primera versión del metamodelo (tendiente a obtener la versión definitiva)	José y María	20	Completa		8	8	4									
		5. Evaluación y mejoras del metamodelo	José y María	6	Retrasada				4	4								
Editor	Definición de la sintaxis concreta del DSL	1. Definición de los elementos textuales/gráficos del DSL	María	4	Pendiente					4								
		2. Construcción del editor	María	20	Pendiente													
		3. Evaluación y mejoras del editor	María	4	Pendiente													
	Prueba de semántica del DSL	4. Utilización del DSL para diseñar un modelo real básico	José	4	Pendiente													
Transformación	Creación de mecanismos de transformación	1. Definición de reglas de transformación	Xabier	4	Pendiente													
		2. Desarrollo del programa de transformación	Xavier	18														
		3. Ejecución del programa en un motor de transformación	Xabier	4	Pendiente													
		4. Evalución y pruebas de la transformación	Xabier	4	Pendiente													
Generador	Desarrollo del generador de código	1. Determinación del GPL para el cual generar código	Xabier y María	2	Pendiente													
		2. Selección del mecanismo de generación de código	Xabier y María	6	Pendiente													
		3. Desarrollo del generador de código	Xabier y María	16	Pendiente													
		4. Pruebas de generación de código	Xabier y María	4	Pendiente													
	TOTAL HORAS PLANIFICADAS:			124														

Reuniones Diarias: 9H00	Reunión de Revisión de Incremento: Martes 1, Oct. - 9Hh00	Reunión de Retrospectiva: Martes 1, Oct. - 10Hh00	Reunión de Planificación de próximo Sprint: Miércoles, 2 Oct. - 15H00

5.2.2.4 Revisar

Permite realizar una revisión del producto generado en un sprint.

PRESENTACIÓN DE INCREMENTO DE LA HERRAMIENTA

Para el efecto se debe realizar una reunión denominada Reunión de Revisión de Incremento, la cual se la realiza al final del sprint. Su duración es de máximo una hora donde el equipo de desarrollo presenta al propietario del producto el incremento construido en el sprint.

REUNIÓN DE REVISIÓN DE INCREMENTO

Esta reunión permite al experto de domino obtener información objetiva del sistema. También permite marcar a intervalos regulares el ritmo de construcción de las herramientas MDE y la trayectoria que va tomando la visión del producto (hitos de control). Al ver y probar el incremento, el experto de domino, y el equipo en general obtienen feedback clave para evaluar el producto generado. Otros ingenieros y programadores de la empresa también pueden asistir para conocer cómo trabaja la tecnología empleada. El administrador del proyecto obtiene retro-información sobre buenas prácticas y problemas durante el sprint, necesarias para las prácticas de ingeniería de procesos y mejora continua de la implementación de la metodología.

Pre-condiciones

- Se ha concluido el sprint.
- Asiste todo el equipo de desarrollo, el experto de domino, el administrador del proyecto y todas las personas implicadas en el proyecto que lo deseen.

Entradas

- Incremento terminado.

Resultados

- Feedback para el experto de domino: hito de seguimiento de la construcción de las herramientas MDE.
- Feedback para el administrador del proyecto: buenas prácticas y problemas durante el sprint.
- Convocatoria a la Reunión de Planificación del siguiente sprint.

FORMATO DE LA REUNIÓN

Es una reunión informal. El objetivo es ver el incremento y trabajar en el entorno del cliente. Están prohibidas las presentaciones gráficas y "powerpoints". El equipo no debe invertir más de una hora en desarrollar la reunión, y lo que se muestra es el resultado final: terminado, probado y operando en el entorno del cliente (incremento). Según las características del proyecto puede incluir también documentación de usuario, o técnica. Se trata de una reunión informativa. No tiene una misión orientada a tomar decisiones, ni a criticar el incremento.

Un protocolo recomendado es el siguiente:

- El equipo expone el objetivo del sprint, se muestra la versión de los artefactos MDE desarrollados. Se abre un turno de preguntas y sugerencias sobre lo visto. Esta parte genera información muy valiosa para que el experto de domino y el equipo en general, puedan mejorar el valor de la visión del producto.
- El administrador del proyecto, de acuerdo con las agendas del experto de domino y el equipo establecen la fecha para la reunión de preparación del siguiente sprint.

5.2.2.5 Retrospectiva

Dada la premisa de flexibilidad de los modelos de desarrollo ágil, se puede considerar también la realización de reuniones retrospectivas al final de cada sprint, o de cada versión de producto o cada cierto periodo de tiempo. Una reunión retrospectiva no es lo mismo que la Reunión de Revisión del Incremento. Se puede realizar luego de la Reunión de Revisión de Incremento y puede tener una duración de una a dos horas. Al igual que los modelos de procesos incorporan prácticas de "ingeniería de procesos" para conseguir una mejora continua de su capacidad, en agilidad también van surgiendo prácticas para lo que sería el equivalente de mejora continua de la agilidad de la organización [70].

El objetivo de la Revisión del Incremento es analizar "qué" se está construyendo, mientras que una Reunión Retrospectiva se centra en "cómo" se está construyendo, "cómo" se está trabajando, con el objetivo de analizar problemas y aspectos mejorables, se discute lo bueno y lo malo que se ha producido en el sprint, se liman asperezas entre los miembros del equipo de trabajo, se retroalimenta a todas las personas.

5.2.3 Procesos Transversales

Consta de dos procesos: la Gestión del Proyecto y el Aseguramiento de la Calidad.

5.2.3.1 Gestión del Proyecto

La gestión de proyectos contempla la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades de un proyecto para satisfacer los requisitos del proyecto.

Planificación del Proyecto

Se debe elaborar un plan del proyecto, el cual debe ser mantenido como la base para la gestión del proyecto. El plan de proyecto es un documento formal y aprobado utilizado para gestionar y controlar la ejecución del proyecto. Se basa en los requerimientos del proyecto y las estimaciones establecidas. La planificación del proyecto debe garantizar que todos los planes que afectan al proyecto son coherentes con el plan de proyecto. La metodología propuesta realiza la planificación de las actividades y tareas de la creación de una versión completa del conjunto de artefactos MDE en la Plantilla Planificación del Sprint realizada en la Reunión de Planificación del Sprint.

Seguimiento y Control del Proyecto

El propósito de esta actividad es proporcionar un entendimiento del progreso del proyecto, para que puedan tomarse las acciones correctivas apropiadas cuando los avances del proyecto se desvían significativamente del plan. El plan documentado del proyecto es la base para supervisar las actividades y tareas, comunicar el estado, y tomar acciones correctivas. El progreso es determinado principalmente comparando el producto de trabajo, atributos de las tareas, esfuerzo, costo y calendario real, contra lo planificado. Una visión apropiada permite tomar acciones correctivas oportunas cuando el desempeño se desvía significativamente del plan.

La metodología propuesta realiza el seguimiento y control de las actividades en la creación de los artefactos MDE en las cuatro reuniones planteadas en la metodología:

- Reunión de Planificación del Sprint
- Reunión Diaria
- Reunión de Revisión de Incremento
- Reunión de Retrospectiva

Y específicamente a través del registro de actividades en la Plantilla de Planificación del Sprint (Cuadro de Mando).

5.2.3.2 Aseguramiento de la Calidad

De acuerdo a CMMI, el aseguramiento de la calidad es una Área de Proceso de la disciplina de Soporte, en esta caso se denomina Aseguramiento de la Calidad del Producto y del Proceso debido a que se considera que la calidad de un producto está determinada en gran medida por la calidad del proceso utilizado para desarrollarlo y mantenerlo.

En el caso de la creación de los artefactos MDE, el propósito del aseguramiento de la calidad del proceso es proporcionar al equipo de desarrollo, al administrador del proyecto y a la gerencia una visión objetiva de la ejecución de los procesos. Su objetivo fundamental es garantizar que los procesos definidos están siendo cumplidos, así como poder detectar deficiencias en la forma de trabajar establecida, para tomar acciones correctivas. En la metodología propuesta el administrador del proyecto tiene la responsabilidad de cumplir y hacer cumplir los procesos establecidos, esto a pesar que se considera que en el desarrollo ágil se cuenta con equipos de desarrollo responsables, disciplinados y con capacidad de auto-organizan .

Como instrumento para ello, se emplea la Plantilla de Planificación del Sprint, donde se van registrando día a día los avances generados, de tal manera que podrá tomar acciones en el caso de presentarse retrasos e inconvenientes que afecten el normal desenvolvimiento del proyecto. Se debe tomar en cuenta que en la Plantilla de Planificación del Sprint al establecerse un objetivo, estimación de cumplimiento de tareas, responsables y fecha de entrega, es decir al basar las acciones en un resultado esperado, la exactitud y cumplimiento de las especificaciones a lograr se convierten también en un elemento a mejorar. Por otro también se incorpora en cada sprint las fases adaptadas de la estrategia de Mejora Continua de la Calidad establecida por Deming (PDCA), para mantener todas las actividades a desarrollar bajo un esquema de control y mejora continua.

Desde el punto de vista del aseguramiento de la calidad del producto, es decir para garantizar la calidad de los artefactos MDE, se tienen varias métodos y técnicas que podrían emplearse. Uno de ellos es el establecido por Ma en [53], donde plantea un método cuantitativo para evaluar la estabilidad y la calidad del diseño de metamodelos en base a UML. El método lleva a cabo evaluaciones objetivas a metamodelos UML usando adaptaciones de las métricas aplicadas a orientación a objetos (OO). Considera que para medir la calidad del diseño de metamodelos expresados en notación UML se puede hacer uso de métricas OO, debido a que la sintaxis abstracta (metamodelo) se presenta como un modelo que se describe en un subconjunto UML, que consiste en un diagrama de clases UML. En tal caso, se puede mapear (asignar) los componentes de la sintaxis abstracta del metamodelo expresado en notación UML en la del diagrama de clases de diseño OO, de acuerdo a la siguiente consideración, ver Tabla 5.6:

Tabla 5.6: Mapeo de elementos del metamodelo a elementos del diagrama de clases

Metamodelo (expresado en notación UML)	**Diagrama de Clases**
meta-clase	clase
meta-atributo	atributo
meta-asociación	asociación

La calidad del metamodelo UML, involucra que el metamodelo cumpla con seis Factores o Atributos de Calidad, tales como: Reusabilidad, Flexibilidad, Comprensibilidad, Funcionalidad, Extensibilidad y Eficacia. Como se muestra en la Tabla 5.7:

Tabla 5.7: Definición de atributos de calidad

Atributos de Calidad	Definición
Reusabilidad	Cohesión en el acoplamiento inferior y superior en el diseño del metamodelo con la intensión de reutilización en diferentes dominios.
Flexibilidad	Características que permitan la incorporación de cambios en un diseño de metamodelo.
Comprensibilidad	Propiedades del diseño del metamodelo que le permita ser fácilmente aprendido y comprendido.
Funcionalidad	Las responsabilidades asignadas a las metaclases en el diseño del metamodelo.
Extensibilidad	El diseño del metamodelo permite la incorporación de nuevos requisitos.
Eficacia	Capacidad del diseño del metamodelo para lograr la funcionalidad deseada.

Sin embargo, se debe pasar de una definición subjetiva teórica de lo que es la calidad del metamodelo a algo cuantificable o medible, según Ma [53] esto se logra a través de las métricas utilizadas en la programación OO añadiendo seis nuevas métricas para el diseño de metamodelos con UML, especificadas en negrilla en la Tabla 5.8 :

Tabla 5.8: Descripción de las métricas de la arquitectura

Métrica	Descripción
DSC	Tamaño del diseño en meta-clases
NOH	Número de jerarquías. Es un recuento del número de clases que no heredan que tienen hijos en un meta-modelo
MNL	Número máximo de nivel de herencia
NSI	Número de herencia simple en meta-clases
NMI	Número de herencia múltiple en meta-clases
ADI	Promedio de la profundidad de la estructura de herencia de meta-clases
AWI	Promedio de la anchura de la estructura de herencia de meta-clases
ANA	Promedio del número de ancestros
ANDC	Promedio del número de distintas meta-clases que están asociadas con meta-clases
ANAT	Promedio del número de meta-atributos
ANAG	Promedio del número de meta-agregaciones
ANS	**Promedio del número de estereotipos** (no aplica a DSL)
AWF	**Promedio del número de invariantes OCL**
AAP	**Promedio del número de operaciones adicionales (operaciones OCL)**
NAC	**Número de meta-clases abstractas**
NCC	**Número de meta-clases específicas**
NEK	**Número de meta-clases que no tienen padre y ningún hijo en el meta-modelo. La mayoría de ellos destacan por ser tipo enumeración.**

Los Atributos de Calidad pueden ser calculados por once Propiedades de Diseño como se señala en la Tabla 5.9. La importancia relativa de las Propiedades de Diseño individuales que influyen en un Atributo de Calidad se pondera proporcionalmente de acuerdo a un criterio establecido por un esquema de ponderación, uno de ellos se presenta en la Tabla 5.10 adoptada por [8], donde cada una de las Propiedades de Diseño se puede asignar (mapear) directamente en una o más Métricas que evalúan la estabilidad de los metamodelos UML.

Tabla 5.9: Fórmulas de cálculo para los Atributos de Calidad de las Propiedades de Diseño

Atributos de Calidad	Definición
Reusabilidad	-0.25*Acoplamiento+0.25*Cohesión+0.5*Mensajería+0.5*Tamaño de Diseño
Flexibilidad	0,25*Encapsulamiento-0,25*Acoplamiento+0.5*Composición+0.5*Polimorfismo
Comprensibilidad	-0.33*Abstracción+0.33*Encapsulamiento-0.33*Acoplamiento+0.33*Cohesión-0.33* Polimorfismo-0.33*Complejidad-0.33*Tamaño de Diseño
Funcionalidad	0.12*Cohesión+0.22*Polimorfismo+0.22*Mensajería+0.22*Tamaño de Diseño + 0.22 *Jerarquías
Extensibilidad	0.5*Abstraction-0.5*Acoplamiento+0.5*Herencia+ 0.5*Polimorfismo
Eficacia	0.2*Abstracción+0.2*Encapsulamiento+0.2*Composición+0.2*Herencia 0.2*Polimorfismo

Tabla 5.10: Mapeo de las Métricas de Diseño a Propiedades de Diseño y valores de las propiedades de metamodelos UML

Propiedad de Diseño	Métrica de Diseño	UML 1.1	UML 1.3	UML 1.4	UML 1.5	UML 2.0
Tamaño de Diseño	DSC	120	133	192	194	260
Jerarquías	NOH	2	2	3	3	1
Abstracción	ANA	2.50	2.68	2.95	2.97	4.26
Encapsulamiento	-	1	1	1	1	1
Acoplamiento	ANDC	0.86	0.90	1.40	1.65	1.03
Cohesión	-	1		1	1	1
Composición	ANAG	0.32	0.29	0.51	0.70	0.65
Herencia	ADI	2.46	2.45	2.92	2.93	3.87
Polimorfismo	NAC	10	13	25	26	46
Mensajería	ANAT	0.64	0.62	0.55	0.66	0.39
Complejidad	ANS+AWF+AAP	1.86	1.97	2.25	2.21	1.90

5.2.4 Interacción con el Repositorio de Activos Reutilizables

La finalidad de esta actividad es mantener la integridad de los artefactos que se crean en el proceso para conformar un repositorio de reutilización. Se debe tener en cuenta que construir componentes reutilizables incrementan el esfuerzo en las tareas de desarrollo, porque se requiere un diseño más genérico de software, elaboración de documentación y pruebas exhaustivas, sin embargo se pueden utilizar en la generación de diferentes aplicaciones con funcionalidad parecida para diferentes organizaciones [89]. La era de la información y las crecientes presiones para obtener software de calidad y en el menor tiempo posible acentúan la necesidad de aprovechar óptimamente el esfuerzo de desarrollo. Por esta razón se ha propuesto la reutilización como una respuesta para incrementar significativamente la productividad a la vez que se mantiene o se incrementa la calidad del software [78].

Cada una de las actividades propuesta en la metodología generan productos o artefactos que deberán registrarse en un repositorio digital, pudiendo utilizar para el efecto herramientas automáticas de gestión de versionamiento y si es posible herramientas orientadas al control de versionamiento para artefactos MDE. Este es el caso del Sistema de Control de Versiones (VCS) para entornos MDE creado en el proyecto AMOR (Adaptable Model Versioning) [5], el cual

permite a los desarrolladores trabajar en paralelo en el control de versionamiento de algunos artefactos MDE que se van creando en un proyecto.

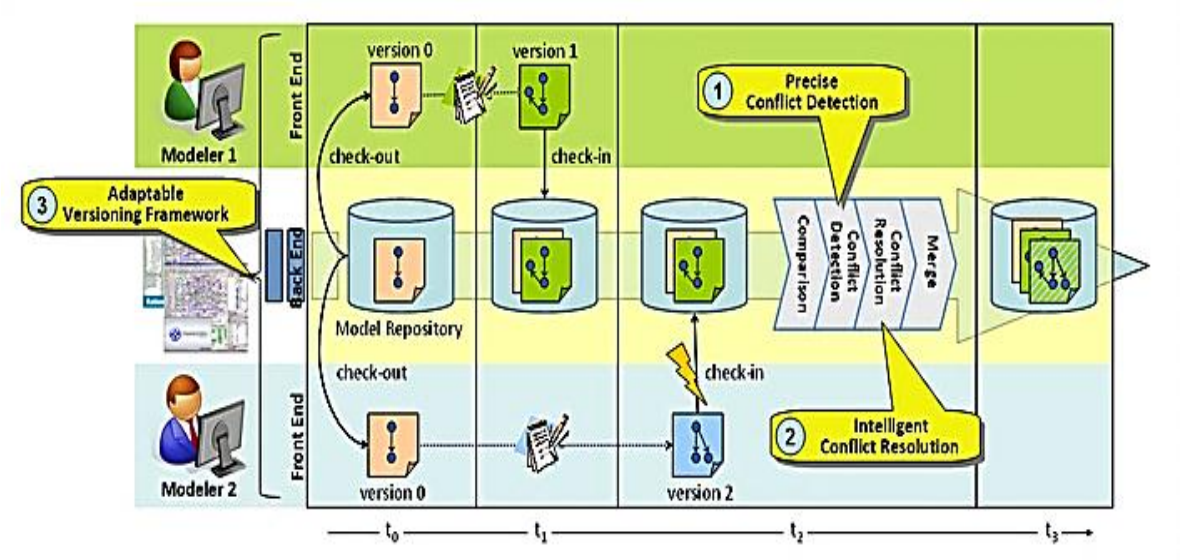


Figura 5.6: Esquema del control de versionamiento del Proyecto AMOR [5]

Las innovaciones de AMOR se manifiestan en tres objetivos principales de investigación. En primer lugar, AMOR apunta a: 1) la detección precisa de conflictos, es decir fallos no detectados previamente; 2) la resolución de conflictos inteligente, proporcionando técnicas para la representación de las modificaciones en conflicto; 3) plasmar los objetivos de AMOR en un Framework de Versionamiento Adaptable, proporcionando al usuario la flexibilidad de balancear entre el esfuerzo de adaptación razonable y el apoyo de versionamiento adecuado mientras garantiza la aplicabilidad genérica para diversos lenguajes DSL y herramientas asociadas.

Por otro lado, también se recomienda que el administrador del proyecto registre en el repositorio digital correspondiente los instrumentos de gestión del proyecto, para mantener un histórico de los proyectos desarrollados que le permitan realizar mejores estimaciones de tiempo y recursos en futuros proyectos.

Capítulo 6

Proceso de desarrollo de Aplicaciones Ligeras con artefactos MDE

En este capítulo se propone el proceso de desarrollo de aplicaciones ligeras con los artefactos MDE creados con MDM1 por parte del usuario final, tomando en consideración el enfoque proporcionado por el Desarrollo por el Usuario Final en el capítulo 2 y las generalidades del desarrollo de aplicaciones con DSLs.

6.1 Generalidades del desarrollo de aplicaciones con DSL

De acuerdo a Voelter en [96] un DSL limita libertades a los usuarios finales en algunos aspectos, ellos solo pueden expresar cosas que se encuentran delimitadas por el DSL. Específicamente no pueden manejar aspectos del nivel que no estén considerados en el DSL, debido a que su alcance es manejado por el motor de ejecución. Sin embargo, los usuarios pueden realizar las tareas que estén concebidas en su dominio de aplicación. Los usuarios finales al igual que con el manejo de otras aplicaciones pueden cometer errores, otros incluso pueden hacer mal uso de él, por lo que un DSL demasiado expresivo puede presentar mayor riesgo de operar.

Como parte de los procesos de desarrollo de aplicaciones con DSLs se debe asegurar la realización de revisiones periódicas del modelo, esto es crítico, especialmente en la fase de adopción de una nueva herramienta, donde las personas están todavía aprendiendo el lenguaje y su entorno operativo. Sin embargo, las revisiones resultan más fáciles y eficientes de realizar en el nivel del DSL que en el nivel de código, puesto que los programas DSL son más concisos y apoyan una mejor separación de las preocupaciones de su especificación equivalente en código GPL.

Si durante el desarrollo de la aplicación con el DSL se observan errores recurrentes en actividades en que el usuario realiza con regularidad, pueda significar que los diseñadores del lenguaje tengan que rediseñar este aspecto, a lo mejor lo que los usuarios esperan es realmente correcto. Hacer bien las cosas utilizando DSL requiere mucho tiempo de trabajo en un proyecto [96], sin embargo si se organizan adecuadamente las actividades que se tienen que realizar se podría configurar adecuadamente el lenguaje para ser utilizado en varios proyectos.

6.2 Criterios que debe considerar el proceso de desarrollo

Considerando las características del tipo de aplicaciones que se podrían desarrollar con los artefactos MDE por parte del usuario final (no necesariamente informático), se procede a formular un listado de los principales aspectos que debería tomar en cuenta el proceso de desarrollo de aplicaciones.

- Ser lo más fácil de aprender y aplicar por parte del usuario final.
- Debe ser práctico y lo más simple posible de emplear, de tal manera que permita al usuario final obtener en relativamente corto plazo la solución que requiere.
- Se debe considerar un proceso de naturaleza iterativa-incremental que permita el desarrollo evolutivo y exploratorio de la aplicación.
- Debe contemplar la realización de actividades básicas de ingeniería de software que permitan al usuario desarrollar una solución acorde a sus requerimientos. Para lo cual la herramienta MDE podría implementar actividades que de cierta manera "obliguen" al usuario a realizar actividades de ingeniería de software. Como por ejemplo antes de comenzar a diseñar el modelo con el DSL, el usuario podría generar una Lista de Verificación (Check List) de los requisitos. También se podrían implementar algún mecanismo de pruebas automáticas del diseño generado, como lo hacen algunas herramientas CASE.
- Se debe procurar la participación de usuarios finales reales durante la creación de los artefactos MDE, de tal manera que vayan probando aspectos de funcionalidad y facilidad de uso (usabilidad).

6.3 Proceso de desarrollo propuesto

En base a los criterios que debe considerar el proceso de desarrollo, se plantea la utilización de un modelo de desarrollo en base a prototipos evolutivos de software. La creación de un prototipo de software es un proceso de abajo hacia arriba, donde algunas funciones básicas son implementadas rápidamente, probadas por el usuario, y mejoradas; a continuación se aplican requisitos adicionales para el usuario, y el ciclo continúa hasta que se termine de desarrollar el producto [72]. La propuesta así planteada demanda que la *funcionalidad del proceso de desarrollo por el usuario final* sea concebida e implementada como un artefacto más de la solución MDE. A través

de la implementación de actividades que “obliguen” al usuario final a realizar tareas básicas de ingeniería de software.

El proceso de desarrollo por el usuario final contempla cinco fases:

- Establecimiento de los objetivos del prototipo
- Descripción de la funcionalidad del prototipo (requerimientos)
- Desarrollo del prototipo
- Evaluación del prototipo
- Despliegue

* Mini-sprint entre el Desarrollo y Evaluación del prototipo

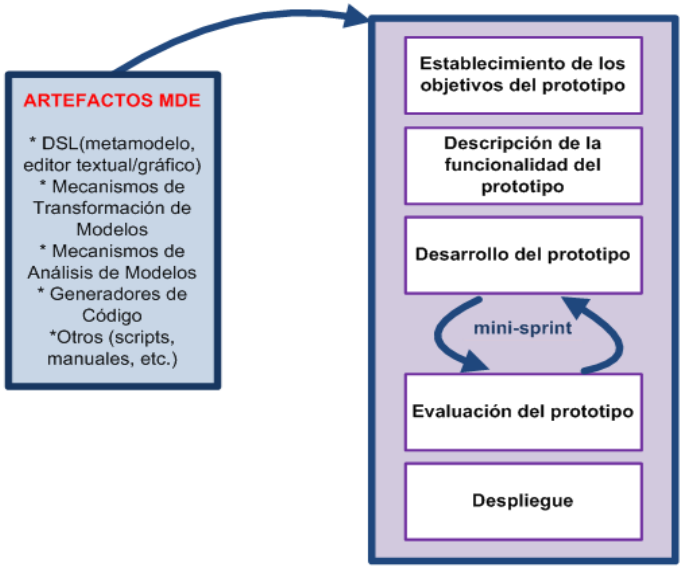


Figura 6.1: Proceso de desarrollo de aplicaciones por el usuario final con enfoque MDE

6.3.1 Establecimiento de los Objetivos del Prototipo

El proceso de desarrollo se originaría con la definición de los objetivos de la aplicación a construir, los objetivos del desarrollo del prototipo deben ser claros y lo más específicos posible de tal manera que denoten la aplicación a construir, el marco tecnológico y temporal a emplear. Los objetivos deben comenzar con una palabra en infinitivo.

6.3.2 Descripción de la Funcionalidad del Prototipo

Posteriormente se debe realizar la identificación y descripción de los requerimientos (funcionalidad). Los requisitos de la aplicación a construir con el conjunto de artefactos MDE deben ser un subconjunto de los requisitos utilizados para construir el DSL [96], puesto que constituyen los requisitos de la funcionalidad de la aplicación objeto a desarrollar. Considerando

también que los artefactos MDE, lo que proveen, es un mecanismo alternativo de desarrollo de software que permite elevar el nivel de abstracción. Donde el desarrollador no tiene que lidiar con entornos y herramientas de desarrollo tradicional (UML, RUP, GPLs como C, C++, java, C#, etc.).

6.3.3 Desarrollo del Prototipo

Se debe considerar que debido a la construcción de la aplicación mediante un prototipo evolutivo, éste va completándose a través de varias iteraciones, por lo cual en la primera iteración se deben tomar los requisitos más representativos para construir una primera versión. Luego ir incrementado e implementando en las siguientes iteraciones otros requisitos al prototipo, hasta obtener una versión completa.

6.3.4 Evaluación del Prototipo

Se establece un proceso iterativo (mini-sprint) entre el desarrollo del prototipo y su evaluación puesto que a medida que se van incrementando nuevos requisitos al prototipo, este debe ser verificado, validado y refinado por el desarrollador y de ser posible por otro usuario (experto del dominio) que asuma un rol crítico. Además se puede hacer uso de mecanismos de pruebas automáticas del diseño del prototipo como se señaló en anteriormente.

6.3.5 Despliegue

En la fase de despliegue se implanta la aplicación desarrollada en el entorno de producción considerado.

Capítulo 7

Conclusiones y Trabajos Futuros

El desarrollo de software ha pasado por muchas etapas en un periodo de tiempo muy corto y en cada una ha ido ganando complejidad, añadiendo capas de abstracción y mejorando todo lo existente anteriormente de manera acelerada. Los desarrolladores han tenido que ir aprendiendo muchas veces a las malas que las herramientas para resolverlos han ido mutando y creciendo aceleradamente hasta formar un conjunto complejo de plataformas, estándares y personas, que deben funcionar juntos.

Este es el caso de MDE, una nueva disciplina dentro de la ingeniería de software que se ocupa de la utilización sistemática de modelos de software para mejorar la facilidad, productividad y calidad del desarrollo de software. La visión de la construcción del software que promulga esta nueva disciplina ha mostrado su potencial para dominar la complejidad arbitraria del software al proporcionar una mayor abstracción y elevando el nivel de automatización. El paradigma MDE considera como elementos fundamental de la disciplina que: 1) un modelo representa total o parcialmente un aspecto de un sistema de software, 2) estos modelos son representados con DSLs también denominados “lenguajes de modelado”, 3) un metamodelo es empleado para representar formalmente un DSL, 4) la automatización es normalmente conseguida a través de la traducción de los modelos a código mediante transformaciones de modelos.

A pesar de los beneficios que plantea MDE como soporte para el apoyo del desarrollo de software, de los esfuerzos de una gran comunidad de investigadores y algunas empresas de desarrollo de software que impulsan este enfoque, todavía queda mucho camino que recorrer para que los posibles usuarios de esta tecnología tengan la suficiente confianza como para comenzarla a utilizar masivamente. Por tal motivo y queriendo aportar con un grano de arena a

esta nueva iniciativa de desarrollo de software, se estructuró una primera versión de una metodología de desarrollo de software dirigida por modelos basada en DSLs. Las investigaciones en el área son muchas, cada una trata un tema en particular, por lo que de alguna manera este trabajo logró integrar la mayor cantidad de elementos posibles. Sin embargo lejos de que la presente propuesta metodológica sea un producto terminado, listo para utilizarlo, puede ser utilizado como una guía de soporte y ayuda para las personas interesadas en adoptar este enfoque de desarrollo de software.

La investigación realizada ha permitido identificar que existen varias áreas que a futuro pueden ser desarrolladas y mejoradas en el ámbito MDE con DSL, entre ellas: modelos de ejecución y simulación, testing y validación, técnicas de aseguramiento de calidad, administración de proyectos, desarrollo de sistemas de versionamiento de artefactos MDE especialmente con capacidad de gestión, validación, refinamiento y refactorización de modelos, desarrollo de aplicaciones con artefactos MDE y DSL por el usuario final empleando criterios de ingeniería de software y principios de usabilidad, la inclusión de elementos específicos a la presente propuesta metodológica para poderla probar en entornos reales para determinar su conveniencia.

Referencias

[1] Abraham, R., Burnett, M & Erwig, M. (2009). Spreadsheet programming. Wiley Encyclopedia of Computer Science and Engineering, pp.1–10.

[2] Acceleo, http://www.eclipse.org/acceleo

[3] Agile Allience, www.agileallience.org

[4] Alan Brown, http://www.ibm.com/developerworks/ rational/library/3100.html

[5] AMOR Adaptable Model Versioning, http://www.modelversioning.org/

[6] Atkinson, C., & Kuhne, T. (2003). Model-driven development: a metamodeling foundation. Software, IEEE, 20(5), 36-41.

[7] Bąk, K., Zayan, D., Czarnecki, K., Antkiewicz, M., Diskin, Z., Wąsowski, A., & Rayside, D. (2013, May). Example-driven modeling: model= abstractions+ examples. In Proceedings of the 2013 International Conference on Software Engineering (pp. 1273-1276). IEEE Press.

[8] Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. Software Engineering, IEEE Transactions on,28(1), 4-17.

[9] Beck, K., & Andres, C. (2004). Extreme programming explained: embrace change. Addison-Wesley Professional.

[10] Boehm, B. (1989). Software risk management (pp. 1-19). Springer Berlin Heidelberg.

[11] Bosch, J. (2002). Maturity and evolution in software product lines: Approaches, artefacts and organization. In Software Product Lines (pp. 257-271). Springer Berlin Heidelberg.

[12] Brandt, J., Guo, P., Lewenstein, J., and Klemmer, S. R. 2008, Opportunistic programming: How rapid ideation and prototyping occur in practice, In Proceedings of the Workshop on End-User Software Engineering (WEUSE).

[13] Buchmann, T., Dotor, A., Westfechtel, B. (2013). MOD2-SCM: A model-driven product line for software configuration management systems, Information and Software Technology vol. 55, pp. 630-650.

[14] Buezas, N., Guerra, E., de Lara, J., Martín, J., Monforte, M., Mori, F., & Cuadrado, J. S. (2013). Umbra Designer: Graphical Modelling for Telephony Services. In Modelling Foundations and Applications (pp. 179-191). Springer Berlin Heidelberg.

[15] Burnett, M., Scaffidi, C., "End-User Development". Interaction-Design.org. at "Encyclopedia of Human-Computer Interaction".

[16] Cánovas, J.,Cabot, J.,López-Fernández, J., Sánchez, J., Guerra, E., de Lara, J. (2012). Engaging End-Users in the Collaborative Development of Domain-Specific Modelling Languages.

[17] Carmien, S. P., & Fischer, G. (2008, April). Design, adoption, and assessment of a socio-technical environment supporting independence for persons with cognitive disabilities. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 597-606). ACM.

[18] Chandrasekaran, B., Josephson, J. R., & Benjamins, V. R. (1999). What are ontologies, and why do we need them?. Intelligent Systems and Their Applications, IEEE, 14(1), 20-26.

[19] Clark, T., & Muller, P. A. (2012). Exploiting model driven technology: a tale of two startups. Software and Systems Modeling, 1-13.

[20] Clements, P., & Northrop, L. (2002). Software product lines. Boston: Addison-Wesley.

[21] Costabile, Maria Francesca, Fogli, Daniela, Mussio, Pero and Piccinno, Antonio (2006): End-user development: The software shaping workshop approach. In: Lieberman, Henry, Paterno, Fabio and Wulf, Volker (eds.). "End User Development (Human-Computer Interaction Series)". Springerpp. 183-205

[22] De Lara, J., & Guerra, E. (2010). Deep meta-modelling with METADEPTH. In Objects, Models, Components, Patterns (pp. 1-20). Springer Berlin Heidelberg.

[23] Díez Cebollero, D. (2009). ComBLA: la aplicación del análisis de dominios al desarrollo de sistemas de aprendizaje asistido por ordenador.

[24] Dobing, B., & Parsons, J. (2006). How UML is used. Communications of the ACM, 49(5), 109-113.

[25] EMFText, http://www.emftext.org/index.php/EMFText

[26] Fischer, Gerhard, Lemke, Andreas C., Mastaglio, Thomas W. and Morch, Anders (1990): Using Critics to Empower Users. In: Carrasco, Jane and Whiteside, John (eds.) Proceedings of the ACM CHI 90 Human Factors in Computing Systems Conference 1990, Seattle, Washington,USA. pp. 337-347

[27] Fisher, Marc, Rothermel, Gregg, Brown, Darren, Cao, Mingming, Cook, Curtis R. and Burnett, Margaret M. (2006): Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. In ACM Trans. Softw. Eng. Methodol., 15 (2) pp. 150-194

[28] Fuentes, L., & Vallecillo, A. (2004). Una introducción a los perfiles UML. Revista Novatica–Asociación de Técnicos de Informática-España.., & Vallecillo, A. (2004). Una introducción a los perfiles UML. Revista Novatica–Asociación de Técnicos de Informática-España.

[29] García, M. García, F., Pelechano, V., Vallecillo, A., Vara, J., Vicente-Chicote, C. (2013). Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas. Ra-Ma.

[30] Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer.Cognitive psychology, 15(1), 1-38.

[31] Gröner, G., Bošković, M., Silva Parreiras, F., & Gašević, D. (2012). Modeling and validation of business process families. Information Systems.

[32] Hutchinson, J., Whittle, J., Rouncefield, M., & Kristoffersen, S. (2011, May). Empirical assessment of MDE in industry. In Proceedings of the 33rd International Conference on Software Engineering (pp. 471-480). ACM.

[33] Interacction Design Fundation, http://www.interaction-design.org/encyclopedia/end-user_development.html

[34] ITCorpSoft, http://www.itcorpsoft.com/servicios/7-creacion-y-desarrollo-de-software-web-para-empresas. Html

[35] Itemis, http://www.itemis.com/

[36] Jackson, M. A., “Problems & Requirements,” in RE, 1995.

[37] Johan den Haan; The Enterprise Architect Building an Agile Enterprise; http://www.theenterprisearchitect.eu/archive/2009/05/06/dsl-development-7-recommendations-for-domain-specific-language-design-based-on-domain-driven-design

[38] Johnson, C. N. (2002). The benefits of PDCA. Quality Progress, 35(5), 120.

[39] Jouault, F., & Bézivin, J. (2006). KM3: a DSL for Metamodel Specification. In Formal Methods for Open Object-Based Distributed Systems (pp. 171-185). Springer Berlin Heidelberg.

[40] Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. Science of Computer Programming, 72(1), 31-39.

[41] Juran, J., Gryna, F., Bingham, R. (1974). "Quality control handbook", McGraw-Hill.

[42] Kaindl, H. (2012). Specifying Business Requirements through Interaction Design. Institute of Computer Technology. Vienna University of Technology. Vienna, Austria

[43] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study (No. CMU/SEI-90-TR-21). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[44] Kent, S. (2002, January). Model driven engineering. In Integrated Formal Methods (pp. 286-298). Springer Berlin Heidelberg.

[45] Kleppe, A. (2008). Software language engineering: creating domain-specific languages using metamodels. Pearson Education.

[46] Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., ... & Wiedenbeck, S. (2011). The state of the art in end-user software engineering.ACM Computing Surveys (CSUR), 43(3), 21.

[47] Kolovos, D. S., Paige, R. F., Kelly, T., & Polack, F. A. (2006). Requirements for domain-specific languages. In Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD).

[48] Kolovos, D. S., Rose, L. M., Abid, S. B., Paige, R. F., Polack, F. A., & Botterweck, G. (2010). Taming EMF and GMF using model transformation. In Model Driven Engineering Languages and Systems (pp. 211-225). Springer Berlin Heidelberg.

[49] Kulkarni, V., & Reddy, S. (2005). Model-driven development of enterprise applications. In UML Modeling Languages and Applications (pp. 118-128). Springer Berlin Heidelberg.

[50] Kulkarni, V., Barat, S., & Ramteerthkar, U. (2011). Early experience with agile methodology in a model-driven approach. In Model Driven Engineering Languages and Systems (pp. 578-590). Springer Berlin Heidelberg.

[51] Leshed, Gilly, Haber, Eben M., Matthews, Tara and Lau, Tessa (2008): CoScripter: automating & sharing how-to knowledge in the enterprise. In: Proceedings of ACM CHI 2008 Conference on Human Factors in Computing Systems April 5-10, 2008. pp. 1719-1728

[52] Lieberman, Henry, Paterno, Fabio, Klann, Markus and Wulf, Volker (2006): End-user development: An emerging paradigm. End User Development. In: Lieberman, Henry, Paterno, Fabio and Wulf, Volker (eds.). "End User Development (Human-Computer Interaction Series)". Springerpp. 1-8

[53] Ma, H., Shao, W., Zhang, L., Ma, Z., & Jiang, Y. (2004). Applying OO metrics to assess UML meta-models. In UML 2004 - The Unified Modeling Language. Modelling Languages and Applications (pp. 12-26). Springer Berlin Heidelberg.

[54] Mackay, Wendy E. (1990): Patterns of Sharing Customizable Software. In: Halasz, Frank (ed.) Proceedings of the 1990 ACM conference on Computer-supported cooperative work October 07 - 10, 1990, Los Angeles, California, United States. pp. 209-221

[55] Mantramindia PVT, http://www.mantramindia.com/development.aspx

[56] MasterCraft: Component-based Development Environment. Technical Documents. Tata Research Development and Design Centre, http://www.tata-mastercraft.com.

[57] Mellor, S. J., Scott, K., Uhl, A., & Weise, D. (2002). Model-driven architecture. In Advances in Object-Oriented Information Systems (pp. 290-297). Springer Berlin Heidelberg.

[58] Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4), 316-344.

[59] MetaCase, http://www.metacase.com

[60] MOFScript , http://umt-qvt.sourceforge.net/mofscript/docs/MOFScript-User-Guide.pdf

[61] Mohagheghi, P., & Dehlen, V. (2008). A metamodel for specifying quality models in model-driven engineering. In Proc. The Nordic Workshop on Model Driven Engineering (pp. 51-65).

[62] Mohagheghi, P., Dehlen, V. (2008). Where is the Proof? - A Review of Experiences from Applying MDE in Industry. Springer Berlin Heidelberg. Volume 5095, 2008, pp 432-443.

[63] Mohagheghi, P., Dehlen, V., & Neple, T. (2009). Definitions and approaches to model quality in model-based software development–A review of literature. Information and Software Technology, 51(12), 1646-1669.

[64] Mohagheghi, P.,Fernandez, M.,Martell, J., Fritzsche, M.,Gilani, W. (2009). MDE Adoption in Industry: Challenges and Success Criteria. Volume 5421, 2009, pp 54-59.

[65] Montenegro Marín, C. E., GAONA GARCÍA, P. A., CUEVA LOVELLE, J. U. A. N., & SANJUAN MARTÍNEZ, O. S. C. A. R. (2011). Application of Model-Driven Engineering (Mda) for the Construction of a Tool for Domain-Specific Modeling (Dsm) and the Creation of Modules in Learning Management Systems (Lms) Platform Independent. DYNA, 78(169), 43-52.

[66] Neighbors, J. (1987). Report on the Domain Analysis Working Group Session. In Proceedings of the Workshop on Software Reuse. Rocky Mountain.Institute of Software Engineering. Boulder. CO.

[67] OMG (2003); OMG Model Driven Architecture (MDA). En http://www.omg.org/mda.

[68] OMG Specifications, http://www.omg.org/

[69] Oney, Stephen and Myers, Brad A. (2009): FireCrystal: Understanding interactive behaviors in dynamic web pages. In: IEEE Symposium on Visual Languages and Human-Centric Computing - VL/HCC 2009 20-24 September, 2009, Corvallis, OR, USA. pp. 105-108

[70] Palacio, J., Ruata C., (2012). Gestión de Proyectos con Scrum Manager, <URL: http:// www. Scrum manager.net>

[71] Pliskin, N., Shoval, P., (1987), "End-user prototyping: sophisticated users supporting system development". ACM SIGMIS Database 4 (4): 7–17. doi:10.1145/1017816.1017817. Retrieved 2008-05-29.

[72] Pomberger, G., Bischofberger, W. R., Kolb, D., Pree, W., & Schlemm, H. (1991). Prototyping-Oriented Software Development - Concepts and Tools.Structured Programming, 12(1), 43-60.

[73] Prieto-Diaz, R. (1987). Domain Analysis for Reusability. In Proceedings of COMPSAC 87:The Eleventh Annual International Computer Software & Applications Conference, pages 23-29. IEEE Computer Society, Washington, DC.

[74] Proyectos y procesos software, http://www.monografias.com/trabajos96/proyectos-y-procesos-software/proyectos-y-procesos-software.shtml

[75] QVT, http://www.omg.org/spec/QVT/1.1/

[76] Rodriguez, J. and Gonzales, S. Líneas De Productos Software. 2007

[77] Rosson, Mary Beth, Sinha, Hansa and Edor, Tisha (2010): Design Planning in End-User Web Development: Gender, Feature Exploration and Feelings of Success. In: Hundhausen, Christopher D., Pietriga, Emmanuel, Diaz, Paloma and Rosson, Mary Beth (eds.) IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010 21-25 September 2010, 2010, Leganés-Madrid, Spain. pp. 141-148

[78] Sampat, N.; Luker, P.; Zedan, H. A Model for Software Reuse. Software Technology Research Laboratory, School of Computing Sciences, De Montfort University.

[79] Sánchez , J., García, J., Menarguez, M. (2006, January). Rubytl: A practical, extensible transformation language. In Model Driven Architecture–Foundations and Applications (pp. 158-172). Springer Berlin Heidelberg.

[80] Sánchez, J., Cánovas, J., Garcia, J. (2012). Applying Model-Driven Engineering in Small Software Enterprises.

[81] Sánchez, J., de Lara, J., Guerra, E. (2012). Bottom-Up Meta-Modelling: An Interactive Approach. Model Driven Engineering Languages and Systems, 3-19.

[82] Scaffidi, Christopher, Bogart, Christopher, Burnett, Margaret M., Cypher, Allen, Myers, Brad A. and Shaw, Mary (2010): Using traits of web macro scripts to predict reuse. In J. Vis. Lang. Comput., 21 (5) pp. 277-291

[83] Scaffidi, Christopher, Shaw, Mary and Myers, Brad A. (2005): Estimating the Numbers of End Users and End User Programmers. In: VL-HCC 2005 - IEEE Symposium on Visual Languages and Human-Centric Computing 21-24 September, 2005, Dallas, TX, USA. pp. 207-214

[84] Selecting a Development Approach, Revalidated: March 27, 2008. Retrieved 27 Oct 2008.

[85] Selic, B. V. (2004). On the semantic foundations of standard UML 2.0. In Formal Methods for the Design of Real-Time Systems (pp. 181-199). Springer Berlin Heidelberg.

[86] Simos, M., Creps, R., Klingler, C., & Lavine, L. (1995). Software Technology for Adaptable Reliable Systems (STARS). Organization Domain Modeling (ODM) Guidebook, Version 1.0 (No. STARS-VC-A023/011/00). UNISYS DEFENSE SYSTEMS RESTON VA.

[87] Sommerville, I., & Sawyer, P. (1997). Requirements engineering: a good practice guide. John Wiley & Sons, Inc..

[88] Strembeck, M., & Zdun, U. (2009). An approach for the systematic development of domain-specific languages. Software: Practice and Experience,39(15), 1253-1292.

[89] Sunkle, S., & Kulkarni, V. (2012). Cost estimation for model-driven engineering. In Model Driven Engineering Languages and Systems (pp. 659-675). Springer Berlin Heidelberg.

[90] Sutcliffe, A., (July 2005), "Evaluating the costs and benefits of end-user development" (PDF). ACM SIGSOFT Software Engineering Notes (ACM) 30 (4): 1–4. doi:10.1145/1082983.1083241. Retrieved 2008-05-29.

[91] Tairas, R., Mernik, M., & Gray, J. (2009). Using ontologies in the domain analysis of domain-specific languages (pp. 332-342). Springer Berlin Heidelberg

[92] Takeuchi, H., & Nonaka, I. (2004). Hitotsubashi on Knowledge Manegement, ISBN 0470820748, John Wiley & Sons.

[93] TCS, http://wiki.eclipse.org/TCS

[94] TCS, http://wiki.eclipse.org/TCS/Language_Project

[95] Tratt, L. (2005). Model transformations and tool integration. Software & Systems Modeling, 4(2), 112-122.

[96] Voelter, M. (2013). DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. CreateSpace.

[97] Voelter, M., & Stahl, T. (2006). Model-Driven Software Development.

[98] WebRatio, http://www.mantramindia.com/development.aspx.

[99] Wright, D. (2011). Software Life Cycle Management. It Governance Ltd.

[100] Xtext, http://www.eclipse.org/Xtext/

[101] Zohaib, M., Ali, S., Yue, T.,Briand L. (2012). Experiences of Applying UML/MARTE on Three Industrial Projects. Springer Berlin Heidelberg. Volume 7590, 2012, pp 642-658.