Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Generación automática de código a partir de modelos: Mecanismos para la gestión y verificación de modelos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Óscar Romero Jiménez

Tutores: Patricio Letelier Torres

Curso 2020/2021

Resumen

Cumplir con cada uno de los puntos definidos en el documento de especificación de requisitos puede significar el éxito de un proyecto. Estos requisitos de software están en continua evolución, por tanto, los desarrolladores necesitan herramientas para hacer frente a los cambios del cliente. Las herramientas basadas en la generación automática de código son una solución, pues nos permiten producir el código fuente de una aplicación a partir de un modelo, reduciendo el trabajo de codificación a la hora de realizar cualquier modificación.

En este Trabajo Final de Grado (TFG) se ha desarrollado un Add-in en Visual Studio para el apoyo a la gestión y verificación de los modelos dentro del contexto de un Lenguaje de Dominio Específico (DSL). Los modelos son el principal artefacto dentro de un desarrollo específico de dominio, de ahí que sea muy importante que estos modelos puedan evolucionar fácilmente. Este TFG se enmarca en una práctica de empresa realizada por el autor, donde gracias a esta herramienta los analistas del domino serán capaces de gestionar estos modelos.

Palabras claves: Lenguaje de Dominio Específico (DSL), Generación automática de código, Add-in.

Abstract

Achieving each of the points defined in the requirements specification document could mean the success of a project. These software requirements are constantly evolving; therefore, developers need tools to easily react to customer changes. Tools based on automatic code generation are probably the best option, since they allow us to produce the source code of an application from a model, reducing the coding phase and making maintenance easier.

In this final degree project (TFG) we have developed a Visual Studio Add-in to support the management and verification of models within the context of a Domain-Specific Language (DSL). Models are the main part of a domain specific development; hence it is very important that these models were easily evolutionary. The development of this TFG is part of a company internship carried out by the author where thanks to this tool, domain analysts will be able to manage these models.

Key words: Domain-Specific Language (DSL), Automatic Code Generation, Add-in.

Tabla de contenidos

1. Introducción 9
   1.1. Motivación 9
   1.2. Objetivos 12
   1.3. Estructura de trabajo 12
2. Generación automática de código 14
   2.1. ¿Qué es la Generación Automática de Código? 14
   2.2. Ventajas e inconvenientes de la generación automática de código 17
   2.3. Otras formas de generación automática de código 19
3. Desarrollo de dominio específico 20
   3.1. Introducción 20
   3.2. Motivación 21
      3.2.1. Productividad. 21
      3.2.2. Calidad 22
      3.2.3. Economía 22
   3.3. Arquitectura de un desarrollo de dominio específico. 23
      3.3.1. Lenguajes de dominio específico (DSL) 23
      3.3.2. Modelos 25
      3.3.3. Generador de código 31
4. Definición del problema: Referencias modelbus que dejan los modelos inutilizables 32
5. Estado del arte de las herramientas como apoyo a la evolución de modelos 38
   5.1. BISM Normalizer 38
      5.1.1. Instalación 38
   5.2. Agility CMS Model Updater 41
   5.3. Otras herramientas para la evolución de modelos 42
   5.4. Conclusiones 44
6. Propuesta de solución para la evolución de modelos 45
   6.1. Requisitos 45
   6.2. Cómo crear un add-in en *Visual Studio* 45
   6.3. Estructura de la solución .NET 51
      6.3.1. Carpeta ModelFilePathAbstractCommand 53

6.3.2. Carpeta MoveModels....55
6.3.3. Carpeta RenameModels....57
6.3.4. Carpeta Utils....58
6.4. Desafíos del desarrollo. 60
6.4.1. Validación de modelos....60
6.4.2. Serialización y cargado de modelos en memoria 61
6.5. Cronología del desarrollo.... 64
6.6. Guía de uso.... 65
6.6.1. Mover un modelo 65
6.6.2. Renombrar un modelo 68
7. Conclusiones y trabajo futuro 69
7.1. Conclusiones.... 69
7.2. Conclusiones personales.... 71
7.3. Relación con asignaturas cursadas 71
7.4. Trabajo futuro.... 72
8. Referencias 73
9. Anexos 78

Tabla de figuras

Figura 1. Actividades en el ciclo de vida del software [8]. 10
Figura 2. Esfuerzo dedicado a cada una de las actividades en un desarrollo de software [9]. 10
Figura 3. Pirámides de tiempos de desarrollo [1]. 11
Figura 4. Desarrollo dirigido por código vs desarrollo dirigido por modelos [13]. 15
Figura 5. Terminología del desarrollo dirigido por modelos [14]. 17
Figura 6. Low-code vs Model-Driven Development [22]. 19
Figura 7.Productividad en un desarrollo de dominio específico [24]. 21
Figura 8. Comparativa costes de desarrollo [13]. 23
Figura 9. Modelo, metamodelo y meta-metamodelo [27]. 24
Figura 10. DSL Tools desarrolladas. 25
Figura 11. Modelo de la DSL Tool de Dominio. 27
Figura 12. Modelo de la DSL Tool de Aplicación. 28
Figura 13. Modelo de la DSL Tool de UI. 29
Figura 14. Modelo basado en la DSL Tool de BI. 30
Figura 15. Modelo basado en la DSL Tool de Unidades de Medida. 30
Figura 16. Relación de asociación unidireccional. 33
Figura 17. Relación de asociación bidireccional. 34
Figura 18. Relación de generalización. 34
Figura 19. Ejemplo de referencia simple. 35
Figura 20. Ejemplo de referencia compuesta. 36
Figura 21. Instalación BISM Normalizer. 39
Figura 22. Comparación de modelos en BISM Normalizer. 39
Figura 23. Selección de modelos en BISM Normalizer. 40
Figura 24. Acciones disponibles en BISM Normalizer. 40
Figura 25. Link model en Agility CMS. 41
Figura 26. Actualizar modelo en Agility CMS. 42
Figura 27. Worklow para evolución de modelos [34]. 42
Figura 28. Modelos de datos MM-evolver. 44
Figura 29. Creación proyecto VSIX. 46
Figura 30. Añadir comando al proyecto. 47
Figura 31. Localización de un comando al hacer click en un elemento del Explorador de Soluciones. 48

Figura 32. Comando en Menú Proyecto. 48
Figura 33. Comando en editor de código... 49
Figura 34. Asignar icono a botón. 49
Figura 35. Definición de los comandos. 50
Figura 36. Definición de la prioridad de los botones. 50
Figura 37. Estructura de la solución para evolucionar modelos. 51
Figura 38. Estructura interna de la solución para evolucionar modelos. 52
Figura 39. Formulario para renombrar un modelo. 57
Figura 40. Desarrollo incremental. 64
Figura 41. Cronología del desarrollo. 65
Figura 42. Primer paso para mover un modelo. 65
Figura 43. Segundo paso para mover un modelo. 66
Figura 44. Aviso de ficheros modificados al mover un modelo. 66
Figura 45. Cambios pendientes al mover un modelo. 67
Figura 46. Cambios pendientes en modelos basados en una DSL Tool distinta. 67
Figura 47. Primer paso para renombrar un modelo. 68
Figura 48. Segundo paso para renombrar un modelo. 68
Figura 49. XML de un modelo mostrado desde Team Foundation Server. 70

CAPÍTULO 1

Introducción

1.1. Motivación

Una de las grandes metas de un desarrollador es satisfacer las necesidades de los clientes, y esto solo será posible si se cumplen los requisitos que se han definido para el proyecto. Usualmente estos requisitos cambian, por ello, debemos disponer de herramientas que nos ayuden a reaccionar rápidamente ante estos cambios. Por ejemplo, imaginemos que en una aplicación dirigida al sector sociosanitario se requiere registrar si un usuario ha dado positivo en COVID. Si disponemos de herramientas que nos ayuden a automatizar la implementación de ciertos cambios, como pueden ser añadir un campo a una tabla o añadir una propiedad a un modelo, podremos reducir los tiempos de desarrollo.

Una oportunidad de automatización está en las herramientas internas del equipo, pues a menudo se pierde tiempo realizando tareas repetitivas que podrían automatizarse. Para mejorar los tiempos de desarrollo y ajustarse a los tiempos estimados, es aconsejable minimizar estas tareas repetitivas, pues generalmente estas son vistas como una de las peores tareas a llevar a cabo por el 40% de los desarrolladores. Además, las aplicaciones que optimizan los procesos internos son vistas como una de las mejores formas para generar ingresos y mejorar la productividad del desarrollo de aplicaciones por el 52% de los desarrolladores [2].

En la Figura 1 podemos observar las tareas que generalmente encontramos en el ciclo de vida del desarrollo de software. Estas son: Definición de necesidades o requisitos, Análisis, Diseño, Codificación, Pruebas, Validación y Mantenimiento.

Figura 1. Actividades en el ciclo de vida del software [8].

Generalmente en un desarrollo de software la mayor parte del tiempo se centra en la actividad de codificación. En la Figura 2 podemos ver que el esfuerzo de un desarrollo de software se dedica generalmente a implementar el sistema. Por tanto, como los requisitos están continuamente evolucionando y dado que la fase de implementación suele ser la más costosa, las herramientas de generación automática de código pueden tomar un papel importante a la hora de disminuir los tiempos de desarrollo.

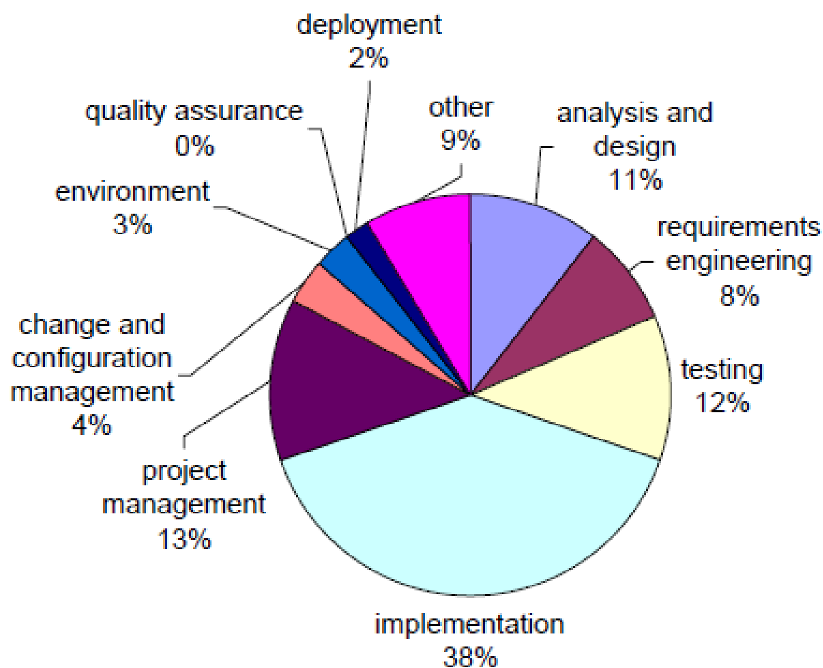


Figura 2. Esfuerzo dedicado a cada una de las actividades en un desarrollo de software [9].

Una alternativa para mejorar la productividad son los enfoques de desarrollo basados en modelos. Estos enfoques son conocidos como *Model-Driven Development (MDD)* [3]. En [4] los autores muestran cómo han obtenido un 33% de mejora de productividad siguiendo un enfoque MDD respecto de un desarrollo manual*.* Además, existen otros enfoques similares como el desarrollo *No-Code* [5] y el *Low-Code* [6], los cuales permiten a programadores y no programadores crear aplicaciones a través de una interfaz gráfica. En [7], Robert asegura que utilizando este enfoque serían necesarias solo 1/6 de las horas de desarrollo comparado con la media de tiempo que se dedica en un desarrollo de software.

Una de las soluciones que se implementó dentro de la empresa donde se enmarca este trabajo para reducir el tiempo de respuesta del equipo de desarrollo, pasó por invertir la pirámide de tiempos de un desarrollo de software clásico. Esto se consiguió siguiendo un desarrollo *MDD,* de tal manera que tanto los analistas como los programadores, analizando los requisitos de los clientes, construyen modelos a partir de los cuales se generará el código de la aplicación.

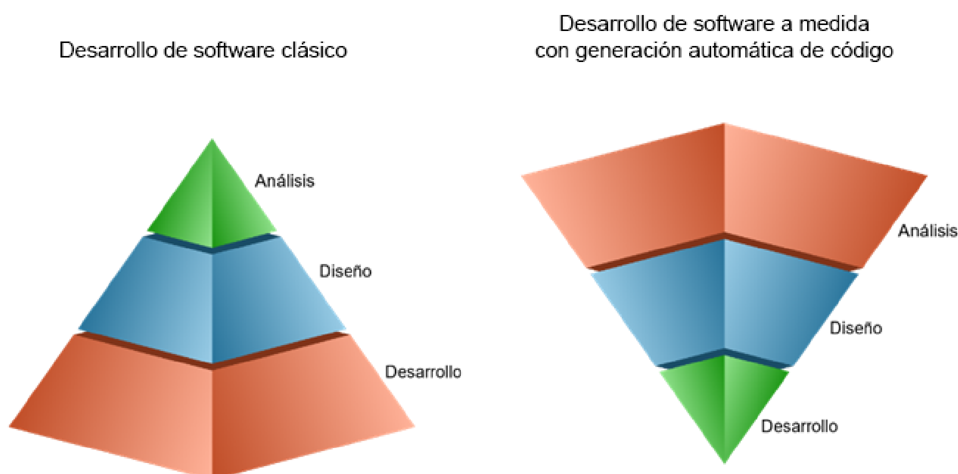


Figura 3. Pirámides de tiempos de desarrollo [1].

Como se muestra en la Figura 3 en un desarrollo tradicional el tiempo dedicado a la codificación es el máximo, por ello el éxito del proyecto tiende a depender del desarrollador. Esto quiere decir que cualquier error cometido en las actividades de análisis y diseño es arrastrado a la fase de desarrollo, haciendo que el software sea más propenso a errores.

En el desarrollo de software a medida con generación automática de código podemos observar que el esfuerzo se centra en el análisis de requisitos con el cliente. Entender qué es lo que el cliente solicita será esencial para el éxito del proyecto, pues las siguientes actividades dependen de que este análisis de requisitos sea completo.

El producto principal de la empresa de desarrollo de software en la que se enmarca este trabajo tiene como producto principal un *ERP* dirigido al sector sociosanitario. Dado que este software empezó a desarrollarse en el año 1997 y es de gran envergadura, en el departamento de I+D+i se está diseñando desde cero una nueva versión que contiene mejoras tanto en funcionalidad como en las tecnologías usadas y que, en el futuro, sustituirá al actual. Para esta nueva versión se decidió utilizar un enfoque *MDD,* apoyándose en la tecnología *DSL-Tools* de *Microsoft* [30].

1.2. Objetivos

Como hemos comentado anteriormente, en este tipo de desarrollos los modelos constituyen una parte fundamental, de ahí la importancia de poder gestionarlos y evolucionarlos fácilmente. Por eso*,* la motivación principal de este trabajo es facilitar al equipo de desarrollo una herramienta de ayuda para la generación, mantenimiento y evolución de los modelos para mejorar los procesos internos y disminuir tiempos en el desarrollo del software.

Así, el objetivo de este Trabajo Final de Grado (TFG) es construir un add-in para *Visual Studio* que, a la hora de mover o renombrar un modelo desde el explorador de la solución, analice todas las referencias a otros modelos y las actualice. Con esto conseguimos que estos modelos sean más mantenibles, pues además de que nos sirve como herramienta de gestión, también nos sirve como herramienta de evolución y verificación.

1.3. Estructura de trabajo

En el capítulo 1 – *Introducción*, se realiza una introducción a este trabajo, poniendo en contexto al lector y explicando cuál es la motivación y objetivos del autor.

En el capítulo 2 – *Generación automática de código*, se explica en qué consiste esta generación de código y qué tipos o variantes existen, todo con el fin de poner en contexto el trabajo realizado, pues el trabajo se realizó para un equipo de desarrollo que trabaja con la generación automática de código.

En el capítulo 3 – *Desarrollo de domino específico,* se describe qué son los dominios específicos de dominio, pues este trabajo está construido dentro de un entorno basado en este tipo de desarrollo.

En el capítulo 4 – Definición del problema: Referencias modelbus que dejan a los modelosinutilizables, se presenta la problemática que causa la generación de este trabajo, detallando cuál era la necesidad.

En el capítulo 5 - *Estado del arte,* se presentan otras herramientas y artículos relacionados con la gestión y evolución de modelos.

En el capítulo 6 – *Propuesta de solución para la evolución de modelos*, se describe la solución que se decidió desarrollar para solucionar el problema que se planteaba. Además, se explica cómo desarrollar una extensión para *Visual Studio* y qué requisitos necesitamos para ello.

Finalmente, en el capítulo 7 – *Conclusiones y trabajo futuro,* se exponen las conclusiones del trabajo, así como qué es lo que ha aprendido el autor de este. Además, se proponen algunas mejoras y trabajo futuro que surge a partir de este trabajo.

CAPÍTULO 2

Generación automática de código

En este capítulo se presenta qué es la generación automática de código. Como forma parte del contexto de este trabajo, es necesario explicar cuáles son la ventajas y desventajas, así como qué es lo que aporta a un desarrollo de software. En nuestro caso esta generación automática de código se lleva a cabo gracias a un desarrollo de dominio específico utilizando *Microsoft DSL Tools*, que veremos en los próximos capítulos.

2.1. ¿Qué es la Generación Automática de Código?

Un generador automático de código es una herramienta que produce, a partir de ciertos artefactos, el código fuente de una aplicación. El uso de estos generadores hace que podamos disminuir los tiempos de desarrollo y reducir los errores en programación, pues evitamos realizar ciertas tareas repetitivas.

La generación automática de código tiene una mala reputación debido a que, históricamente, se implementaba de manera muy pobre por dos razones [10]. La primera es que el código generado por los generadores de código debía ser modificado por los programadores, pues este código autogenerado era repetitivo y, por lo tanto, los programadores tenían que modificarlo para refactorizarlo. Este código generado era muy difícil de mantener, pues cada vez que se generaba código, este se sobrescribía y no había manera de diferenciar qué código era el generado y cuál era el modificado por el programador. Una solución que consiguieron adoptar las herramientas modernas era simplemente separar el código generado del código escrito manualmente por el programador, de manera que el código generado no pudiera modificarse, sino que se utilizara el código manual para complementar este código autogenerado. El segundo problema con las herramientas de generación de código era su integración en entornos de desarrollo. Hoy en día esto ha cambiado y, por ejemplo, *Visual Studio* ya tiene integrada la generación de código XAML (*eXtensible Application Markup Language*) [30].

Dentro del ámbito de la generación automática de código existen diferentes propuestas. Las primeras serían las que se basan en la generación de código a partir de diagramas UML, como en [11], donde la autora propone algunas prácticas aplicables en proyectos de desarrollos de sistemas a partir de diagramas UML

Existen otras aproximaciones, como la relacionada con la generación de interfaces de usuario. En este tipo de generación automática lo que se persigue es generar toda la interfaz de usuario de una aplicación a partir de ciertos artefactos. En [12], los autores combinan la generación automática de interfaces de usuario con el *Big Data* y muestran cómo se puede ahorrar hasta un 20% del tiempo que se dedica a la codificación de las interfaces de usuario. Además, existen muchas otras aproximaciones, pero en este trabajo nos vamos a centrar en la generación automática de código a partir de modelos.

Generalmente, los desarrolladores diferencian entre modelar y codificar. Los modelos se usan comúnmente para diseñar sistemas, entender mejor los dominios o especificar los requisitos, para más tarde realizar la implementación de estos diseños, depurar, realizar pruebas y mantener estos sistemas. Aunque hay varios enfoques en los que modelar y codificar van por separado, también hay otros en los que se conecta el código a los modelos.

Como se muestra en la Figura 4, en un extremo no crearíamos modelos, sino que especificaríamos la funcionalidad directamente en el código. Este enfoque funcionará bien si el sistema que estamos desarrollando es pequeño, por ejemplo, si se trata de una pequeña aplicación que tendrá un ejecutable, el cual puede ser depurado y probado. Si algo necesita modificarse se cambiaría el código, no el ejecutable.

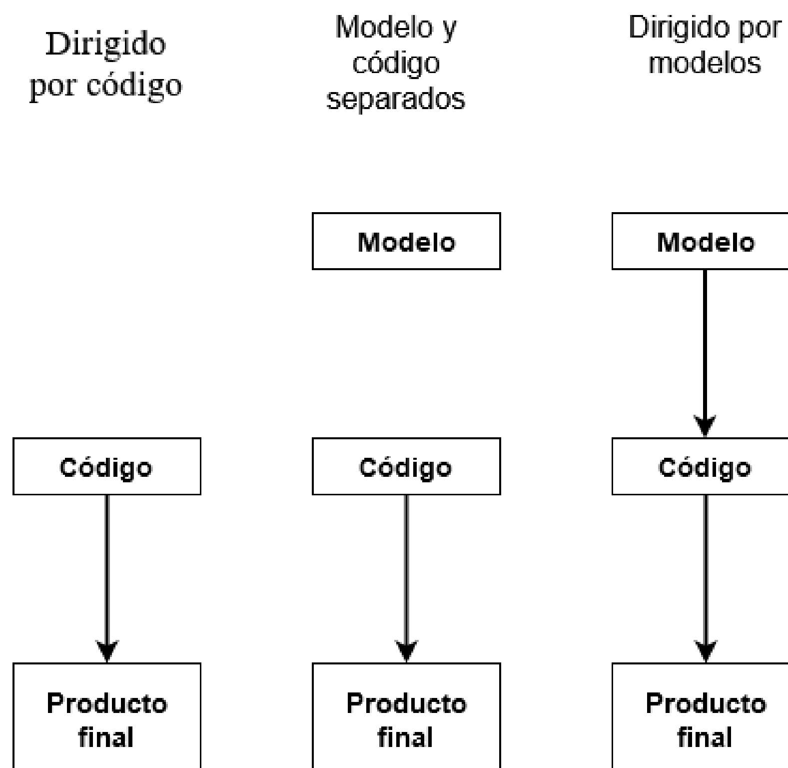


Figura 4. Desarrollo dirigido por código vs desarrollo dirigido por modelos [13].

Sin embargo, la mayoría de los desarrolladores también crean modelos. Esto es así debido a que sin modelos es muy difícil plasmar los requisitos. Además, los modelos permiten aumentar el nivel de abstracción y, gracias a esto, poder desarrollar el sistema en un lenguaje más parecido al lenguaje natural. En cambio, en un desarrollo de software tradicional, no hay una transformación automática de los modelos a código, si no que los desarrolladores leen los modelos y los interpretan para codificar la aplicación produciendo un ejecutable. Estos modelos usualmente no suelen actualizarse durante el mantenimiento, pues únicamente se modifica el código y no se modifican los modelos. Esto se debe simplemente a que el coste de mantener los modelos actualizados en gran medida no se amortiza, pues se trata de un proceso manual, tedioso y propenso a errores.

En el desarrollo dirigido por modelos, utilizamos los modelos como artefacto principal, es decir, tenemos modelos como fuente de datos y no código. Este enfoque debe utilizarse siempre que sea posible pues aumenta el nivel de abstracción, y esto hace que lleguemos a comprender en mayor profundidad el sistema que estamos desarrollando.

Una vez que los modelos se han creado, el código puede generarse para ser compilado o interpretado y ejecutado, y para ello deberemos tener un generador. Este código generado es completo y no será necesario modificarlo, aunque como es obvio puede complementarse con código manual, como hemos comentado anteriormente.

Como podemos observar en la Figura 5, hay varias terminologías dentro del desarrollo dirigido por modelos. El término *Model-Based Engineering* (MBE) es la versión más ligera del *Model-Driven Engineering* (MDE). MBE se refiere a los procesos donde los modelos juegan un papel importante, pero donde no son el principal artefacto. Este término se podría utilizar en aquellos desarrollos donde se utilizan modelos, pero donde este desarrollo no está dirigido por los modelos. Un ejemplo de MBE sería como hemos visto anteriormente, aquellos desarrollos donde el modelo y el código van por separado, es decir, donde los diseñadores de modelos (o analistas) construyen los modelos donde se plasman los requisitos para que después los programadores usen estos modelos para escribir el código manualmente.

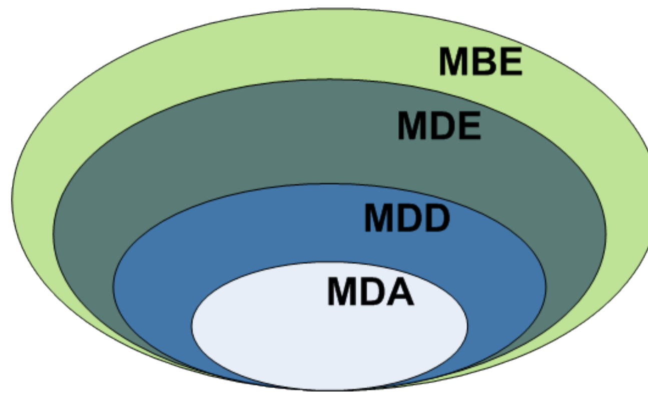


Figura 5. Terminología del desarrollo dirigido por modelos [14].

MDE es una versión más genérica del *Model-Driven Development* (MDD), ya que va mucho más allá del propio desarrollo. MDD es el enfoque de desarrollo que utiliza los modelos como artefacto principal en el que se basa un desarrollo. Esto es, se usan los modelos como la parte principal a partir de la cual se genera código, interfaces de usuario, etc.

Model-Driven Architecture (MDA) es la visión particular que propone la *Object Management Group* (OMG) y que se basa en el uso de los estándares de la OMG. Por tanto, podríamos decir que todo desarrollo MDA es un desarrollo MDD, pero no a la inversa.

Por tanto, podemos decir que el presente trabajo se encuentra dentro del contexto de un desarrollo dirigido por modelos (desarrollo MDD), donde los modelos son el principal artefacto a partir de los cuales se genera el código de la aplicación y que es complementado con código manual, *frameworks* y librerías externas.

2.2. Ventajas e inconvenientes de la generación automática de código

A continuación, se presenta una recopilación de las ventajas e inconvenientes de la generación automática de código obtenidos a partir de algunas referencias citadas y del estudio del desarrollo que sirve como contexto de este trabajo.

Ventajas

- **Reduce los tiempos *time-to-market*[1]** [1].
- **Se basa en los requisitos del usuario:** Gracias a esto, permite que el producto final sea justamente lo que el cliente necesita, ya que se desarrolla específicamente para él [1].
- **Permite mejorar la calidad del *software*:** En ambos artículos se ha experimentado una mejora en la calidad del producto final [15], [16].
- **Aumentar la complejidad del sistema:** Por ejemplo, en un desarrollo de dominio específico donde el código es generado a partir de los modelos, el uso de un lenguaje específico hace que el software construido sea más complejo, ya que es posible utilizar un lenguaje natural más cercado a los conceptos del dominio [17].
- **Independiente del lenguaje de programación** [1].
- **Transparencia en el código:** No es necesario conocer el lenguaje de programación en el que se va a generar el código [1].
- **Mejora de la documentación:** Según [18], este beneficio es uno de los más buscados por parte de los desarrolladores cuando se plantean adoptar la generación automática de código en sus desarrollos.
- **Mayor nivel de abstracción:** Los modelos ocultan los detalles irrelevantes, son precisos y, además, son comprensibles [17].

Inconvenientes

- **Mayor esfuerzo en la etapa de construcción de la arquitectura de los modelos** [1].
- **Separar código generado del no generado.** En [18], George habla sobre la necesidad de separar los dos tipos de código. Si no se separa, el mantenimiento futuro puede llegar a ser un problema.
- **Aumento en la complejidad del desarrollo:** En ocasiones, el uso de modelos hace que el número de artefactos se vea aumentado y, por lo tanto, aumente el número de relaciones y dependencias, lo que puede llevar a un aumento en la complejidad del desarrollo [19].

[1] Tiempo comprendido desde que un producto es concebido hasta que está disponible para el usuario final.

2.3. Otras formas de generación automática de código

En este apartado vamos a ver algunas tendencias que se están utilizando hoy en día para reducir al mínimo las tareas manuales de codificación.

Low-code y No *Code*

Las plataformas *Low-code* [6] son plataformas de desarrollo de software que permiten construir una aplicación reduciendo al máximo las tareas de codificación. *Low-code* puede ser visto como un sinónimo del *Model-Driven Development* (MDD), donde la mayor parte del código es generado, pero necesita completarse con código manual. La mayor ventaja del *Low-code* es que no es necesario tener ningún conocimiento sobre programación ni sobre modelado.

Estas plataformas *Low-code* son un enfoque visual donde es posible automatizar el ciclo de vida del software y donde cualquiera puede desarrollar un producto que encaje con su propia idea de negocio, como es posible conseguir adoptando un enfoque de desarrollo de dominio específico.

Según [20] para el año 2024, el desarrollo de aplicaciones basadas en este enfoque será el responsable de más del 65% de desarrollos de software. De hecho, en la actualidad, muchas de las grandes empresas de software ya están adoptando este enfoque a sus desarrollos [21]. Además, si miramos la Figura 6 vemos cómo las plataformas *Low-code* se han vuelto mucho más populares si lo comparamos con el MDD, al haber aumentado el interés en los últimos años.

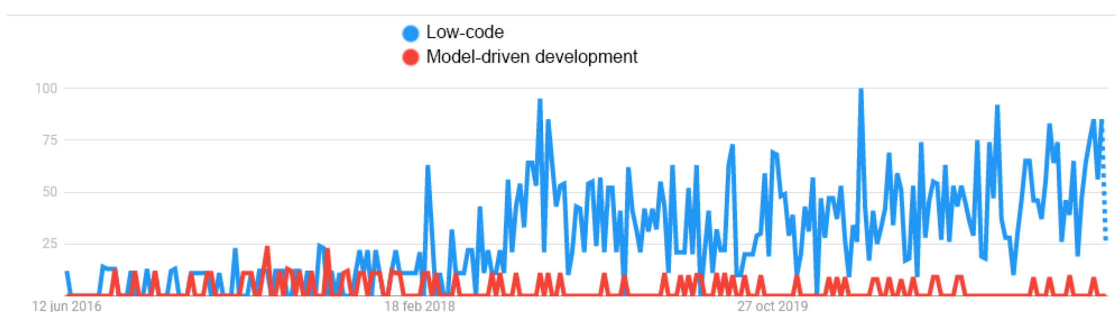


Figura 6. Low-code vs Model-Driven Development [22].

En cuanto a las plataformas *No Code* [5], la idea es muy similar a las de las plataformas *Low-Code*, pero difieren en que en estas paltaformas *No Code* no debería haber ningún trabajo de codificación manual. Este tipo de plataformas será útil para la creación de aplicaciones genéricas, que no requieran de gran complejidad.

CAPÍTULO 3

Desarrollo de dominio específico

En este capítulo explicaremos en qué consiste un desarrollo de dominio específico, puesto que este trabajo se encuentra contextualizado dentro de este tipo de desarrollo. Primero empezaremos con una breve introducción, después describiremos las distintas partes que podemos encontrar en un desarrollo de dominio específico y finalizaremos hablando sobre las características de este tipo de desarrollo, así como qué nos aporta frente a un desarrollo tradicional.

3.1. Introducción

A lo largo de la historia del desarrollo de software, siempre se ha buscado mejorar la productividad a partir de mejorar el nivel de abstracción. Sin embargo, hoy en día los avances en lenguajes de programación tradicionales y lenguajes de modelado no aportan demasiado a la productividad – sobre todo si lo comparamos con la mejora de productividad que supuso pasar de código ensamblador a lenguajes de tercera generación. Por eso nacieron los lenguajes de dominio específico, del inglés *Domain-Specific Language (DSL)* [23], ya que ayuda a aumentar el nivel de abstracción haciendo que el desarrollo se base en conceptos de un determinado dominio.

Un desarrollo de domino específico usualmente puede utilizarse junto el enfoque MDD [3], donde, a partir de modelos, se genera el código final de la aplicación. Esta generación se da gracias a que estos modelos están basados en un lenguaje propio del dominio donde se encuentra el sistema, por lo que los modelos representan conceptos del propio dominio.

Esta generación automática será posible ya que tanto el lenguaje como los generadores de código se basan en conceptos del dominio. Es decir, se define un área de interés – un dominio. Estos dominios pueden ser horizontales, como lo son la persistencia o la interfaz de usuario, o pueden ser verticales, como son telecomunicaciones, banca, seguros, etc. Lo que se busca es que el dominio sea lo más pequeño y cerrado posible, porque así será más sencillo de definir. Por tanto, el desafío al que se enfrentan las empresas cuando optan por un desarrollo de dominio específico es elegir bien el dominio que encaje con lo que quieren desarrollar, pues será la base para construir el sistema que desean [13].

Este trabajo se encuentra contextualizado dentro del desarrollo de dominio específico basado en las *DSL Tools* de *Microsoft* [30], una herramienta que permite diseñar un lenguaje de dominio específico para modelar este dominio y, más tarde, generar el código de la aplicación a través de los generadores de código.

3.2. Motivación

En este apartado vamos a ver qué puede aportar un desarrollo de dominio específico a una empresa, en términos de productividad, calidad y costes.

3.2.1. Productividad.

Según la *Software Productivity Research* (SPR)[2], una consultora dedicada a analizar la productividad, y como podemos observar en la Figura 7, en un desarrollo específico de dominio puede ser hasta un 800% más productivo que los lenguajes de programación tradicionales.

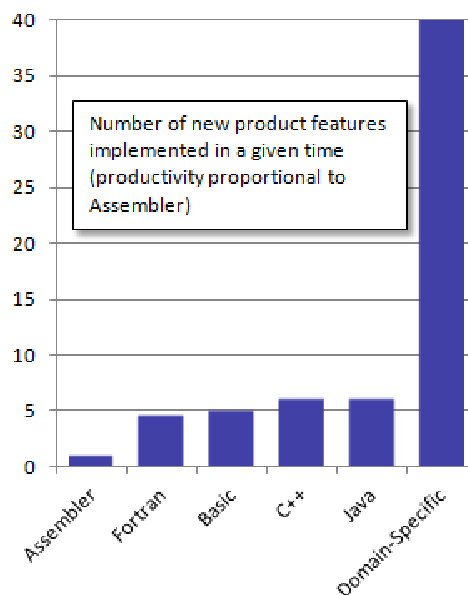


Figura 7. Productividad en un desarrollo de dominio específico [24].

Cuando se comienza a utilizar conceptos de dominio específico, es posible que muchos desarrolladores no estén familiarizados con estos conceptos o incluso no hayan utilizado nunca conceptos relacionados con el modelado. Por eso, las mejoras en productividad tienen mucho más sentido cuando los desarrolladores ya están completamente familiarizados con esta nueva forma de desarrollo.

[2] *Software Productivity Research* (SPR): https://www.ithistory.org/db/companies/software-productivity-research-spr

3.2.2. Calidad.

Un desarrollo de dominio específico nos lleva a mejorar la calidad de las aplicaciones por dos razones: la primera es que los lenguajes de modelado permiten incluir reglas de corrección, reglas que nos sirven para que, a la hora de crear los modelos, no podamos violar ciertas reglas dando lugar a especificaciones incorrectas. La segunda es que los generadores proporcionan un mapeo directo al código, reduciendo el riesgo de que la implementación no se corresponda con el dominio del problema [25].

3.2.3. Economía.

El enfoque basado en un desarrollo de dominio específico (DSL) debe ser utilizado siempre que sea posible, ya que generalmente producen mejores resultados que los enfoques tradicionales. La problemática entonces se encuentra a la hora de decidir si una empresa creará su propio lenguaje DSL o por el contrario modificará uno público; y la decisión a este problema es simplemente una cuestión técnica y económica.

Si una empresa es capaz de encontrar un lenguaje DSL que encaje con su dominio, probablemente sea mejor opción usar o modificar ésta que desarrollar una propia desde cero; sin embargo, aunque se pueda encontrar una solución ya construida que encaje con su dominio, ésta puede estar basada en otros conceptos, tener unas reglas definidas que no encajan con el dominio de esa empresa, que la plataforma de destino no sea la misma a la de esa empresa o generar unos lenguajes de programación con los que no están familiarizados.

La Figura 8 representa estas dos alternativas con relación a los costes acumulativos de desarrollo. El eje X representa la frecuencia con la que un desarrollo de dominio específico es utilizado, mientras que el eje Y representa los costes acumulativos de desarrollo

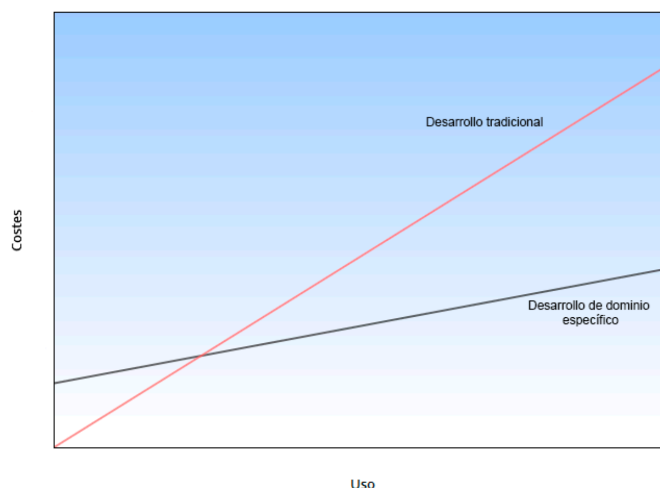


Figura 8. Comparativa costes de desarrollo [13].

Como vemos en la Figura 8, los beneficios de un desarrollo de dominio específico no son gratuitos; alguien debe crear la solución. Podemos observar que inicialmente los costes de este enfoque son mayores que en un desarrollo tradicional, pues será necesario dedicar esfuerzos en recursos humanos, tanto en desarrolladores experimentados como en expertos del dominio. Pero una vez el desarrollo coge forma, los costes son mucho más constantes gracias al aumento de productividad, la disminución del número de errores y al menor coste de mantenimiento.

3.3. Arquitectura de un desarrollo de dominio específico.

En este apartado vamos a introducir los elementos básicos que podemos encontrar en un desarrollo de dominio específico. Generalmente estos suelen ser: lenguaje, modelos y generadores de código. Además, estos elementos se van a describir o ejemplificar basándonos en el desarrollo que se realizó en la empresa donde está contextualizado este trabajo.

3.3.1. Lenguajes de dominio específico (DSL)

Un lenguaje de dominio específico es aquel que nos ofrece una serie de mecanismos para poder manejar la complejidad del dominio, es decir, una serie de conceptos y reglas en un determinado lenguaje que representa las cosas del dominio, en lugar de conceptos de programación. Además de poder definir los objetos del sistema, nos permite que estos objetos tengan una serie de propiedades y puedan relacionarse entre sí.

Los lenguajes están basados en una serie de sintaxis y semántica. La sintaxis especifica la estructura del lenguaje: la construcción del propio lenguaje, sus propiedades y conexiones con otros lenguajes. Pero esta sintaxis no son solo palabras reservadas en el lenguaje del dominio, sino que también contiene las reglas que se deben seguir cuando se construyen los modelos, reglas que son necesarias para evitar errores a la hora de generar el código a partir de los modelos. Estas reglas deben ser comprobadas tan pronto como sea posible, para así asegurar que el código autogenerado está bien formado.

La semántica está asociada al significado de los conceptos del dominio. Cuando añadimos un elemento a un modelo o conectamos elementos, estamos generando significado. La manera en la que se representan los conceptos del dominio puede variar de un programador a otro, por ello debemos tener unos mecanismos para que a la hora de generar significado en los modelos se sigan ciertas directrices y se siga un estándar en la representación de estos conceptos.

Este estándar lo conseguimos gracias a los metamodelos. Según Atkinson [26], los metamodelos son uno de los pilares de la ingeniería dirigida por modelos (MDE) [3]. Un metamodelo es un modelo conceptual del lenguaje del dominio que describe los conceptos del lenguaje, sus propiedades, las conexiones permitidas y las reglas a seguir. Un meta-metamodelo, como podemos ver en la Figura 9, es la descripción de este metamodelo, que generalmente aporta un nivel más alto de abstracción.

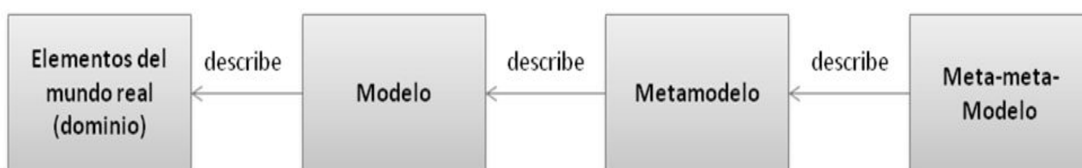


Figura 9. Modelo, metamodelo y meta-metamodelo [27].

El add-in que se ha construido se ha integrado en un desarrollo de dominio específico donde se ha utilizado la herramienta *DSL Tools* de *Microsoft* para definir los lenguajes que se utilizarán para modelar el sistema.

Como vemos en la Figura 10, en el proyecto en el que se enmarca este TFG hay varias *DSL Tool* desarrolladas. Cada una de estas *DSL Tool* constituye un lenguaje que se utiliza para definir cada una de las partes del sistema:

- La *DSL Tool* de UI define la interfaz de usuario.
- La *DSL Tool* de Aplicación define aspectos más técnicos de la aplicación, como pueden ser: tipos de datos, conexiones a microservicios, etc.

- La *DSL Tool* de *BI* define el modelo de negocio, así como las estadísticas para medir el progreso de este.
- La *DSL Tool* de Dominio define los conceptos que se utilizarán en cada uno de los lenguajes y que, por tanto, es la que da significado a los conceptos del sistema.
- La *DSL Tool* de Unidades de Medida define el estándar de medida para cada una de las magnitudes definidas en el sistema.

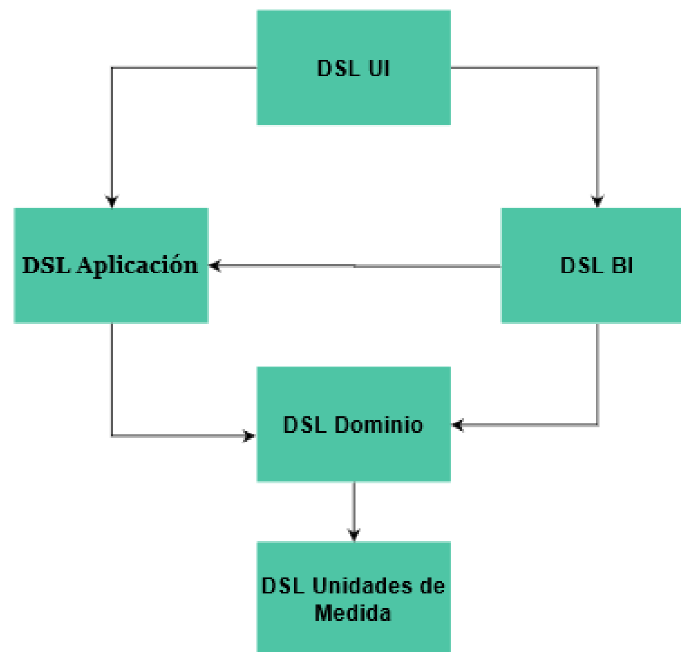


Figura 10. DSL Tools existentes en el contexto de este trabajo.

Además, como vemos en la Figura 10, cada flecha que sale de cada *DSLTool* es una dependencia, pues para la definición y construcción de algunas de ellas se han utilizado otras. Por ejemplo, para definir la *DSL Tool* de Dominio se ha utilizado la *DSL Tool* de Unidades de medida. Esto será importante a la hora de evolucionar los modelos.

3.3.2. Modelos

Los modelos son el artefacto principal en un desarrollo de dominio específico. Estos modelos, además de servir para crear las especificaciones del sistema, sirven como representación de los conceptos del dominio. Gracias a estos modelos podemos tener una visualización gráfica de las reglas y restricciones del dominio y podemos generar el código fuente de la aplicación.

En el contexto de este trabajo hay diferentes tipos de modelos, uno por cada *DSL Tool* desarrollada. Como veremos a continuación, estos modelos son diferentes entre sí. Los modelos están construidos y basados en un XML y este tendrá una estructura diferente dependiendo del tipo de modelo que sea. Además, cada modelo tiene un fichero asociado que tiene extensión *.diagram* que constituye la representación gráfica del mismo. Esto y las dependencias entre *DSL Tool* influirán a la hora de mover y renombrar los modelos. Los diferentes tipos de modelos que podemos encontrar son:

Modelos de dominio

Estos modelos están basados en la *DSL Tool* de dominio. Este tipo de modelos sirve para representar el dominio, definiendo los conceptos del dominio y relacionándolos entre sí. Por ejemplo, en la Figura 11 podemos ver cómo se ha modelado el concepto de *Person*, es decir, cómo se representa una persona en nuestra aplicación. Como podemos ver una persona tendrá una serie de atributos y una serie de relaciones, tanto a elementos del propio modelo como a elementos de otros modelos. Además, podemos observar cómo los conceptos *Proposer, User, Staff, ExternalPerson* y *Doctor* serán un tipo de persona.

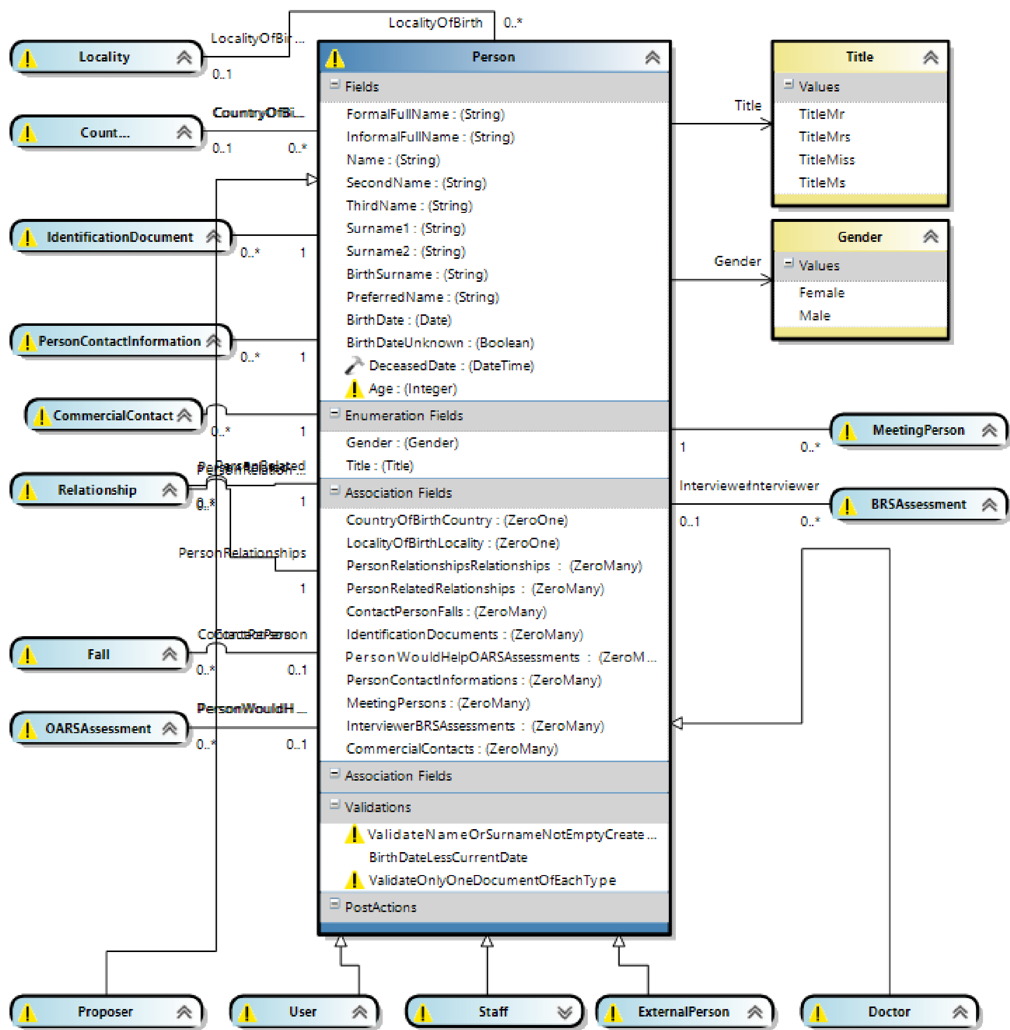


Figura 11. Modelo de la DSL Tool de Dominio.

Modelos de aplicación

Estos modelos están basados en la *DSL Tool* de Aplicación. Estos modelos sirven para definir aspectos de la aplicación como pueden ser conexiones con microservicios. En la Figura 12 podemos observar cómo se representa la exposición del método *GetDomainElementRelatedUIForms* en un microservicio que tendrá la aplicación.

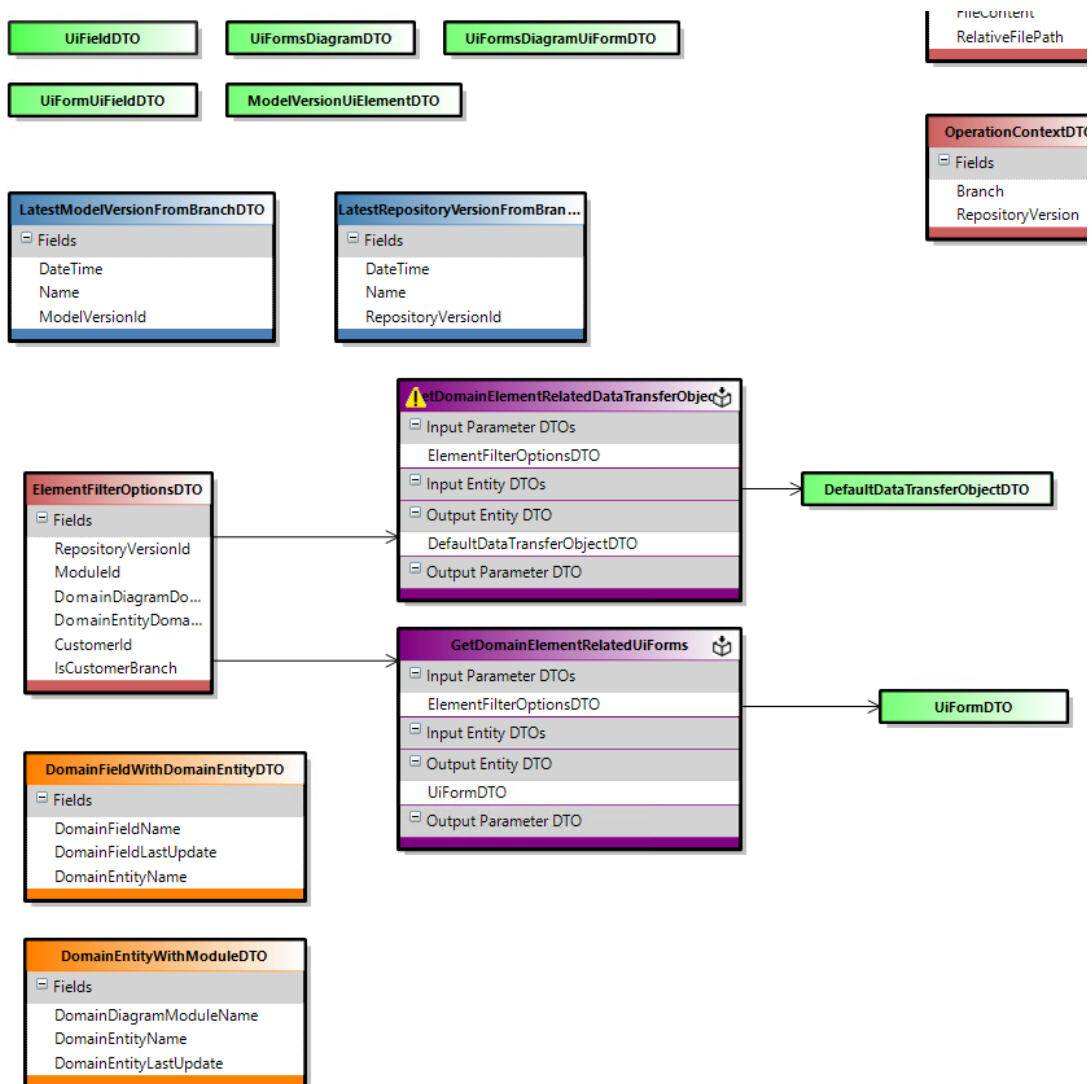


Figura 12. Modelo de la DSL Tool de Aplicación.

Modelos de UI

Estos modelos están basados en la *DSL Tool* de UI. Estos modelos se utilizarán para modelar lo que será la interfaz de usuario, es decir, aquella que el usuario final utilizará para manejar la aplicación. En estos modelos, como vemos en la Figura 13, se definen una serie de formularios que podrán utilizarse de manera genérica en diferentes ventanas de la aplicación, sin necesidad de crear un formulario para cada ventana de nuestra aplicación.

Figura 13. Modelo de la DSL Tool de UI.

Modelos de BI

Estos modelos están basados en la *DSL Tool* de BI. Estos modelos sirven para modelar indicadores. Estos indicadores se utilizan para medir el progreso del negocio o dominio y para comprobar que se están alcanzando los objetivos que se definieron. La Figura 14 ilustra un ejemplo de un modelo de este tipo.

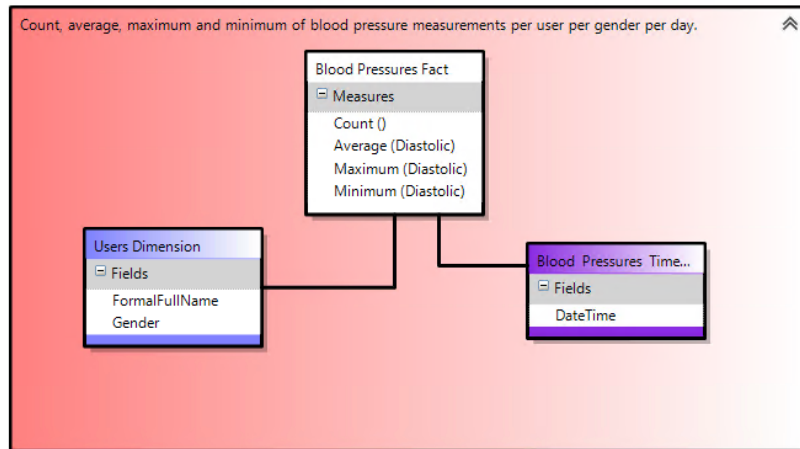


Figura 14. Modelo basado en la DSL Tool de BI.

Modelo de Unidades de Medida

Estos modelos están basados en la *DSL Tool* de Unidades de Medida. Estos sirven para modelar las unidades en las que se miden las magnitudes utilizadas en el dominio. Como vemos en la Figura 15, en el modelo de ejemplo se pueden ver diferentes unidades de medida: *Miligram, Kilogram, Milimeter*, etc.

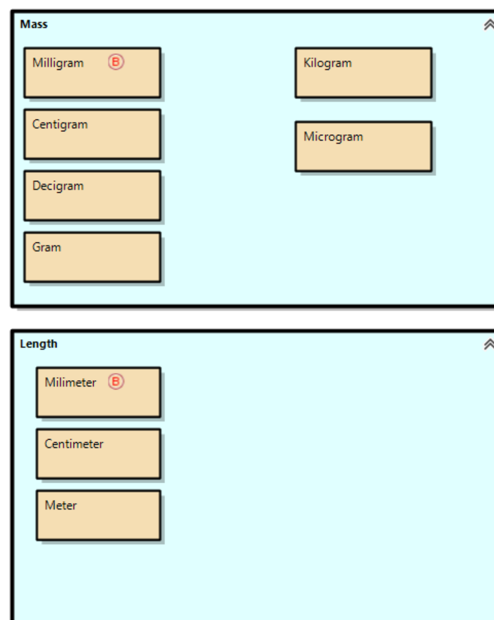


Figura 15. Modelo basado en la DSL Tool de Unidades de Medida.

3.3.3. Generador de código

Un generador de código es un componente que accede a los modelos, extrae información de ellos y la transforma a una sintaxis específica. Este proceso depende y está guiado por el metamodelo, el lenguaje de modelado que hemos visto anteriormente.

El código generado debe ser completo, es decir, después de la generación, el código no necesitaría modificaciones; por eso, tanto el metamodelo como el lenguaje de dominio utilizados deben estar bien definidos. Aunque la mayor parte del código se autogenere a partir de los modelos, puede haber código manual que complemente este código autogenerado.

Existen varias clasificaciones para dividir a los generadores de código. La más usada es la división de estos en generadores declarativos, operacionales o una mezcla de ambos [13]. En los declarativos se describen los mapeos ente los elementos del metamodelo y el lenguaje de programación de destino y en los operacionales se describen los pasos que deben seguirse para producir el código a partir de un modelo origen.

CAPÍTULO 4

Definición del problema: Referencias modelbus que dejan los modelos inutilizables

Como los modelos son el artefacto principal de los desarrollos de dominio específico, están en constante modificación. Por ello, es importante disponer de mecanismos que nos faciliten su mantenimiento y evolución. Los analistas del dominio, que en nuestro caso son aquellos que modelan el sistema, necesitaban una herramienta que les permitiera mover de carpeta y renombrar estos modelos sin que dejaran de ser funcionales, ya que, hasta ahora, al mover o renombrar un modelo, como en las referencias modelbus aparece la ruta relativa al modelo que referencian y este ha sido movido o renombrado, esas referencias modelbus eran incorrectas y el modelo dejaba de ser funcional.

Los modelos están compuestos por entidades. Además, estas entidades necesitan estar relacionadas entre sí para generar el significado del dominio. Por lo tanto, existen relaciones entre entidades dentro del mismo modelo y entre entidades de distinto modelo. Cada una de estas relaciones es una referencia que se encuentra especificada dentro de cada uno de los modelos, de tal manera que cada referencia apunta a la ruta del modelo donde se encuentra la entidad referenciada.

Al igual que en UML, podemos encontrar varios tipos de relaciones. Una asociación es una relación entre dos entidades que describen una conexión entre dos conceptos del dominio. Dentro de estas asociaciones podemos distinguir dos tipos:

- **Unidireccionales:** Son todas las referencias que apuntan de un modelo a otro, pero no al contrario. Un ejemplo de este tipo sería la referencia a un Enumerado que se encuentra en otro modelo. Supongamos que el modelo A contiene una entidad A, y que el modelo B contiene un enumerado EN. Una relación unidireccional se representaría como en la Figura 16.

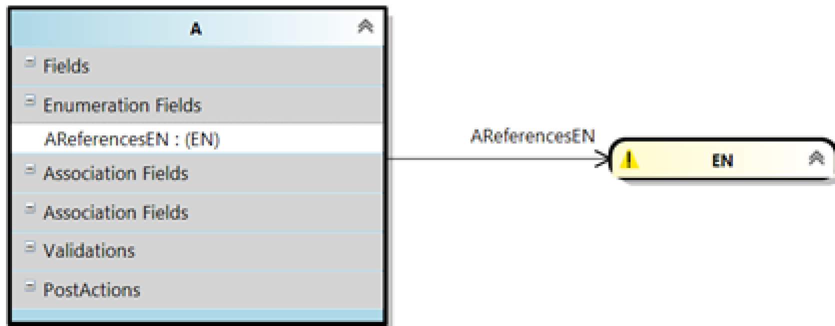


Figura 16. Relación de asociación unidireccional.

Esta relación aparecería en el XML del modelo como una etiqueta *<association>* asociada a la entidad que contiene la relación. Dentro de esta etiqueta, la referencia a la entidad referenciada se encontraría como atributo *reference,* como vemos a continuación.

```
<association
  Id="42011ad1-7ec0-4a0c-b54d-6b533e27f664"
  targetMultiplicity="One"
  sourceMultiplicity="ZeroMany"
  reference="modelbus://ADDInformatica.Agility.DslTools.Domain.D
  omainDSLAdapter/A/Association/..\..\B.ddsl%2f41a02608-e7bf-
  404a-b490-3b957f4d1ccb"
  functionalTODOs=""
  observations=""
  tODOs=""
  order="0"
>
```

Además, las relaciones que hacen referencia a una entidad de otro modelo aparecerán también como etiqueta *<externalEntityReference>*. Por eso, en este caso, la relación entre el modelo A y el modelo B, aparecerá dos veces en el XML del modelo.

```
<externalEntityReferences>
    <externalEntityReference
      Id="94d86c6d-99fa-4c37-9700-83248fc00522"
      name="User"
      reference="modelbus://ADDInformatica.Agility.DslTools.Doma
      in.DomainDSLAdapter/A/..\..\B.ddsl%2f732170cc-18ea-4c0d-
      94fc-00b548972e8a"
    />
</externalEntityReferences>
```

- **Bidireccionales:** Se dan cuando un modelo A apunta a un modelo B, y a su vez el modelo B también apunta al modelo A. Un modelo A con la entidad E1, y un modelo B con una entidad E2, y una relación entre ambas, se representaría como podemos ver en la Figura 17.

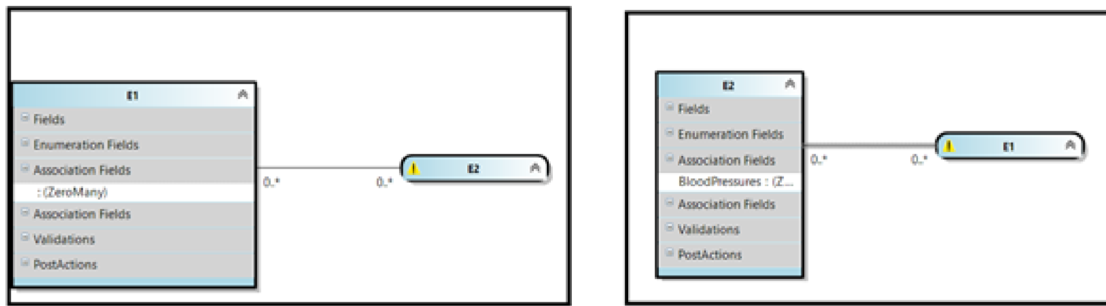


Figura 17. Relación de asociación bidireccional.

Las relaciones bidireccionales se especifican en el XML del modelo de la misma manera que las relaciones unidireccionales. La única diferencia es que en este caso esta relación aparecerá en ambos modelos: en el modelo que está referenciando y en el modelo referenciado.

Otro tipo de relaciones que podemos encontrar son las relaciones de generalización. Estas relaciones se dan entre una entidad del modelo padre y un modelo hijo. En la Figura 18 vemos un ejemplo de este tipo de relaciones.

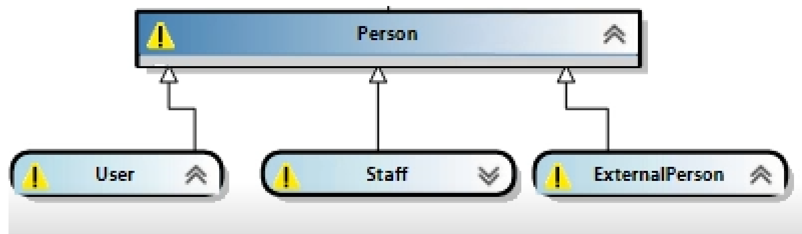


Figura 18. Relación de generalización.

Este tipo de relaciones aparecen como una etiqueta <*generalization*> en los modelos interesados. En el ejemplo de la Figura 18, en cada uno de los modelos hijos (*User, Staff* y *ExternalPerson*) aparecería una etiqueta <generalization> dentro de la etiqueta <*externalEntityReference*>, como podemos ver en el código mostrado a continuación, mientras que en el modelo padre esta etiqueta aparecerá asociada a la entidad que tiene la generalización.

```
<externalEntityReference
  Id="26623c38-ab61-464f-bc1b-4ed0ad880a70"
  name="Person"
  reference="modelbus://ADDInformatica.Agility.DslTools.Domain.DomainD
  SLAdapter/Cocktail.Person/Person/.\Cocktail.Person.ddsl%2fc0441452-
  28b6-4958-a3ae-e21df135cafa"
>
    <subTypes>
        <generalization
          Id="286e32df-1a62-4b64-b7a8-cd3b64e1fd21"
          reference="modelbus://ADDInformatica.Agility.DslTools.Domain.
          DomainDSLAdapter/Cocktail.Person/Generalization/.\Cocktail.Pe
          rson.ddsl%2f9fe8f8e0-3066-4f9a-bfa1-01d585082dea"
    >
```

```
            <entityMoniker
              Id="0a0788f3-e769-4fa7-b7b3-102195391bac"
            />
        </generalization>
    </subTypes>
</externalEntityReference>
```

Para especificar y definir las referencias entre entidades (el valor del atributo *reference*) se utilizó la herramienta Modelbus[3] de *Microsoft.* Modelbus proporciona un método para vincular modelos de lenguajes de dominio específico y modelos UML. En este caso, se utiliza para crear referencias entre las entidades de los modelos y poder vincular los modelos de cada uno de los lenguajes de dominio específico, dando así significado a los conceptos del dominio.

Como podemos observar en la Figura 19, una referencia modelbus está compuesta por tres partes. La primera hace referencia a la entidad del modelo al que se hace referencia, la segunda constituye la ruta relativa al modelo que se está referenciando y la tercera es el GUID que identifica la referencia. La referencia modelbus de la Figura 19 se encuentra en el modelo *Users* y representa una relación entre este modelo y la entidad *Person* del modelo *Cocktail.Person*.

```
         1
"modelbus://ADDInformatica.Agility.DslTools.Domain.DomainDSLAdapter/Cocktail.Person/Person/
.\Cocktail.Person.ddsl!2fc0441452-28b6-4958-a3ae-e21df135cafa"
                     2 3
```

Figura 19. Ejemplo de referencia simple.

Podemos distinguir dos tipos de referencias en base al número de modelos que referencian:

- **Simples:** Aquellas referencias que apuntan a un único modelo. La referencia modelbus de la Figura 19 sería un ejemplo de este tipo.
- **Compuestas:** Aquellas referencias que apuntan a varios modelos. La referencia de la Figura 20 es un ejemplo de una referencia compuesta por tres referencias modelbus.

[3] Modelbus: https://docs.microsoft.com/es-es/visualstudio/modeling/integrating-models-by-using-visual-studio-modelbus?view=vs-2019

```
allowedUnitsOfMeasurementReferences=
```

```
"[modelbus://ADDInformatica.Agility.DslTools.UnitsOfMeasurement.UnitsO
fMeasurementDSLAdapter/UnitsOfMeasurement/Milimeter/..\\..\\..\\..\\..
\\..\\Cocktail\\Core\\Contracts\\UnitsOfMeasurement\\Models\\UnitsOfMe
asurement.uomdsl%2f8b43c8c5-4909-4e0b-a528-5e55602b04a3;
```

```
modelbus://ADDInformatica.Agility.DslTools.UnitsOfMeasurement.UnitsOfM
easurementDSLAdapter/UnitsOfMeasurement/Centimeter/..\\..\\..\\..\\..\
\..\\Cocktail\\Core\\Contracts\\UnitsOfMeasurement\\Models\\UnitsOfMea
surement.uomdsl%2f9477f6f4-1cc2-47eb-b5e9-dcb34f1f6497
```

```
modelbus://ADDInformatica.Agility.DslTools.UnitsOfMeasurement.UnitsOfM
easurementDSLAdapter/UnitsOfMeasurement/Meter/..\\..\\..\\..\\..\\..\\
Cocktail\\Core\\Contracts\\UnitsOfMeasurement\\Models\\UnitsOfMeasurem
ent.uomdsl%2f12bc89bc-39cf-4a51-b7d2-575ab148867b&quot;]"
```

Figura 20. Ejemplo de referencia compuesta.

Esto es importante ya que la manera en la que trataremos ambos tipos de referencias será distinta. Si detectamos una referencia compuesta como la de la Figura 20, la fragmentaremos hasta obtener tres referencias modelbus. Después, estas tres referencias modelbus las trataremos como referencias modelbus simples para finalmente volver a enlazarlas para modificar la referencia.

Una vez explicado todo el contexto de las referencias modelbus, explicaremos el problema que había al mover y renombrar modelos. Los expertos del dominio que modelaban organizan y evolucionan los modelos en base a un criterio propio, pero no podían cambiar de opinión a la hora de elegir la localización dónde iban a destinar cada uno de los modelos ya que una vez construidos, estos dejarían de ser funcionales al intentar moverlos.

La causa principal de la ruptura de los modelos venía dada por la ruta relativa que aparece en las referencias modelbus. Al mover un modelo de localización, cambia la ruta donde este se encuentra; por tanto, las referencias modelbus de este modelo y de todos los que referencien a este serán incorrectas, pues contendrán la ruta relativa antigua. De igual manera ocurre cuando renombramos un modelo, aunque en este caso no cambia la ruta relativa en sí, sí que cambia el nombre del modelo y este es parte de la ruta relativa.

Por lo tanto, la herramienta que hemos construido tenía que resolver toda esta problemática, recorriendo todas las referencias modelbus del propio modelo que queremos evolucionar para así obtener todos los modelos referenciados, cargarlos en memoria y modificar las referencias modelbus que hacen referencia al modelo origen.

Además de todo esto, hay que tener en cuenta las dependencias entre las *DSL Tool*. Como se mostró en la Figura 10 existen una serie de dependencias entre los lenguajes del dominio donde se encuentra contextualizado este trabajo. Por tanto, al evolucionar un modelo tendremos que saber en qué *DSL Tool* está basado para evolucionar primero los modelos que estén basados en esa *DSL Tool* y después, los modelos de las *DSL Tool* dependientes a esta.

CAPÍTULO 5

Estado del arte de las herramientas como apoyo a la evolución de modelos

En este capítulo vamos a presentar otras extensiones de *Visual Studio* construidas con el fin de poder evolucionar los modelos.

5.1. BISM Normalizer

BISM Normalizer[4] es una extensión de *Visual Studio* que trabaja con *Azure Analysis Services* (Azure AS)[5] y con modelos tabulares [30] de *Server Analysis Services* (SSAS). Los modelos tabulares son bases de datos que se ejecutan en memoria y proporcionan un acceso rápido a los datos. Estos modelos se crean en *Visual Studio* con la extensión de *Analysis Services*, la cual proporciona un espacio de diseño con la cual podemos crear tablas, particiones, relaciones, etc.

Esta extensión permite mantener y evolucionar los modelos, pues nos proporciona ciertas tareas que son necesarias cuando trabajamos con modelos. Algunas de las funcionalidades de esta extensión son:

- Fusión de metadatos entre modelos.
- Comparación de modelos.
- Adición y eliminación de objetos en los modelos.
- Reutilización de objetos entre modelos.

5.1.1. Instalación

BISM Normalizer es una extensión VSIX, por lo que contendrá los mismos ficheros base. Esta extensión se encuentra completamente gratuita en el *Marketplace* de *Visual Studio*[6].

Para la instalación de esta extensión se necesita Visual Studio 2015 o posterior. Para instalarla realizaremos los siguientes pasos:

1. Seleccionamos *Extensions* > *Manage Extensions* en la barra de tareas superior de *Visual Studio.*

[4] BISM Normalizer: http://bism-normalizer.com/
[5] Azure Analysis Services (Azure AS): https://azure.microsoft.com/es-es/services/analysis-services/
[6] Marketplace de Visual Studio: https://marketplace.visualstudio.com/

2. Seleccionamos la pestaña *Online* en la parte izquierda del diálogo.
3. Escribimos *ssdt* en el cuadro de búsqueda.
4. Instalamos la extensión *Analysis Services SSDT* si no la tenemos.
5. Instalamos *BISM Normalizer* y reiniciamos *Visual Studio*.

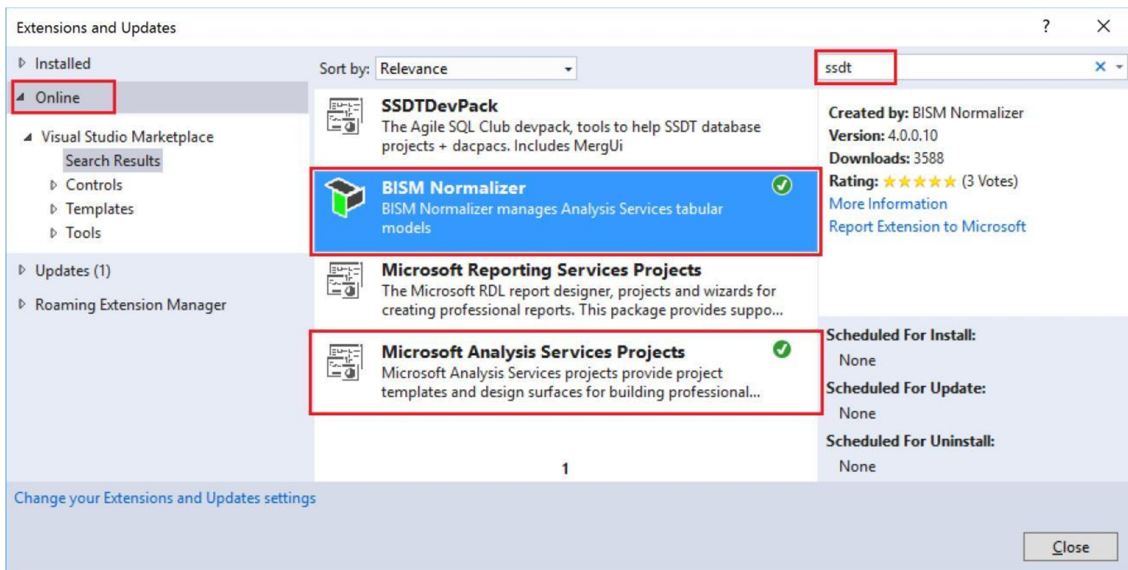


Figura 21. Instalación BISM Normalizer.

5.1.2. Funcionamiento

Para comparar dos modelos, seleccionaremos la opción *Tools* > *New Tabular Model Comparison* desde el menú superior de *Visual Studio.* Después, pulsaremos el botón *Compare* como se muestra en la Figura 22.

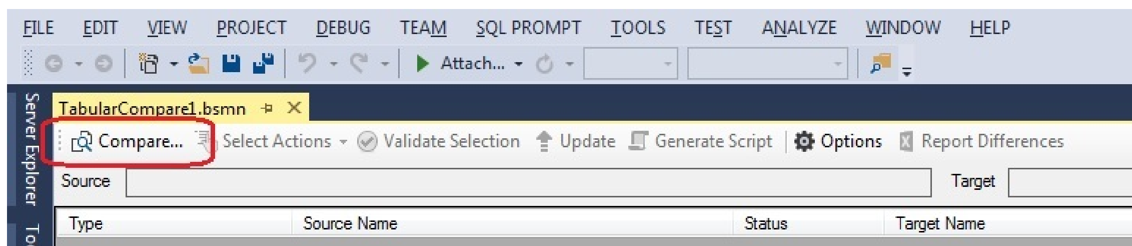


Figura 22. Comparación de modelos en BISM Normalizer.

Al pulsar el botón Compare aparecerá un diálogo donde se seleccionarán los modelos a comparar. Este modelo puede estar en nuestra solución, en una base de datos o en nuestro explorador de archivos. La Figura 23 ilustra la comparación de un modelo ubicado en el disco local y un modelo que se encuentra dentro del proyecto en el explorador de la solución de *Visual Studio.*

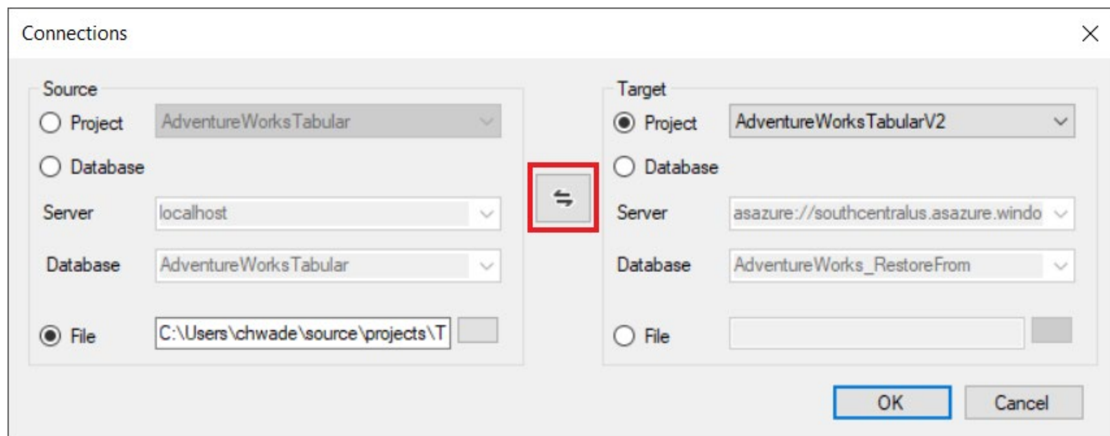


Figura 23. Selección de modelos en BISM Normalizer.

Al pulsar el botón OK nos aparecerán las diferencias que hay entre los dos modelos comparados, tanto en la definición de los objetos como en las relaciones que contiene cada uno. Además, como podemos observar en la Figura 24, existen diferentes acciones que nos permiten evolucionar y mantener estos modelos. Podemos ocultar, borrar y actualizar objetos, así como muchas más acciones que no se han mostrado por falta de relevancia para este TFG.

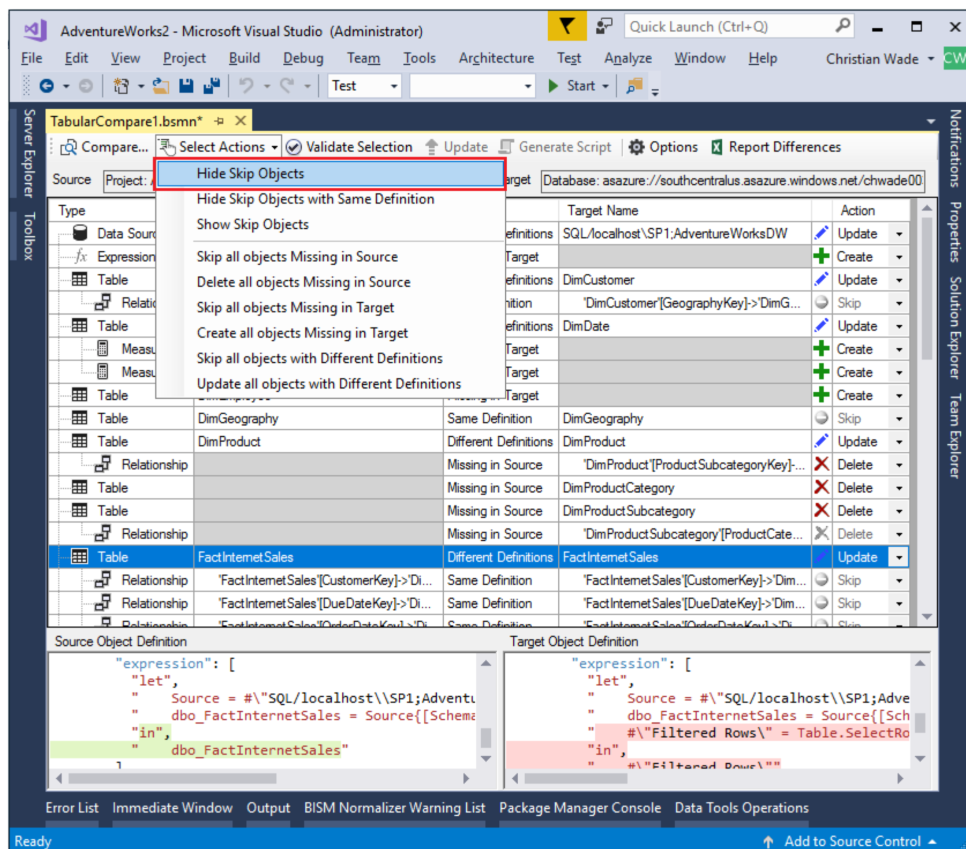


Figura 24. Acciones disponibles en BISM Normalizer.

5.2. Agility CMS Model Updater

Agility CMS Model Updater[7] es una extensión que permite actualizar las clases fuertemente tipadas de un sistema de control de contenidos (CMS). Un CMS es un programa que permite crear un entorno de trabajo para la administración de contenidos como, por ejemplo, en páginas web. Estas clases fuertemente tipadas representan el contenido y los módulos de la arquitectura, por lo que es importante facilitar su evolución.

Según comentan en [28] poder cambiar el nombre de estas clases aumenta la eficiencia de desarrollo de un sistema de contenidos ya que, en lugar de descargar manualmente sus clases desde el administrador de contenido, simplemente habría que seleccionar el archivo de API de C# dentro del proyecto de *Visual Studio* y el nombre de la clase se actualizará automáticamente.

Para renombrar una clase simplemente haremos *click* derecho en una clase dentro del proyecto y pulsaremos el botón *Link Model.* Después, como vemos en la Figura 25 pulsaremos el botón *Update Model.*

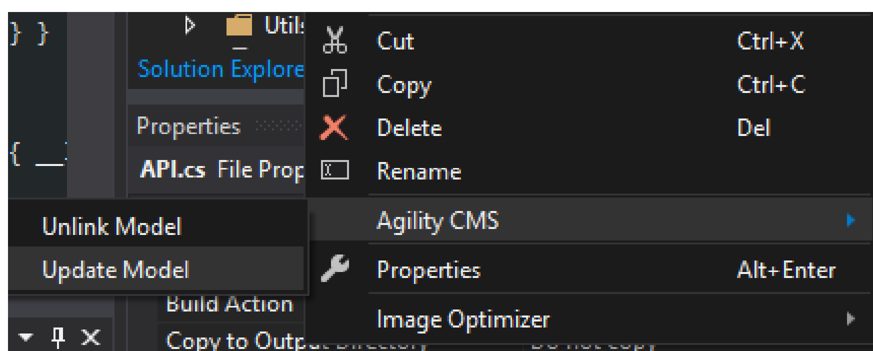


Figura 25. Link model en Agility CMS.

Cuando pulsamos el botón *Update Model* aparecerá un pequeño formulario donde elegiremos el nuevo nombre. Como vemos en la Figura 26 esta extensión tiene un funcionamiento similar al botón de renombrar modelo presentado en este TFG.

[7] Agility CMS Model Updater: https://help.agilitycms.com/hc/en-us/articles/360003991352-Visual-Studio-Extension-Agility-CMS-Model-Updater

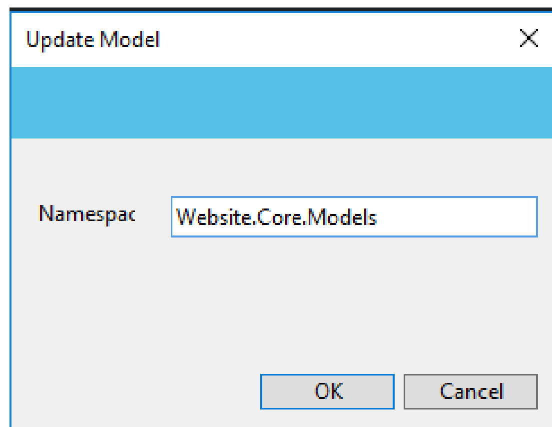


Figura 26. Actualizar modelo en Agility CMS.

5.3. Otras herramientas para la evolución de modelos

Además de las dos extensiones presentadas anteriormente, hemos encontrado otros artículos y herramientas para la evolución de modelos. Estos artículos se basan en desarrollos de herramientas para la evolución de modelos.

En [29] se presenta una herramienta para la gestión y evolución de modelos y metamodelos. Los autores confirman que para ser competitivos son necesarias herramientas que nos ayuden a gestionar los frecuentes cambios que se producen por los avances tecnológicos. Ellos proponen una herramienta basada en un *workflow* que contiene siete actividades, como se muestran en la Figura 27.

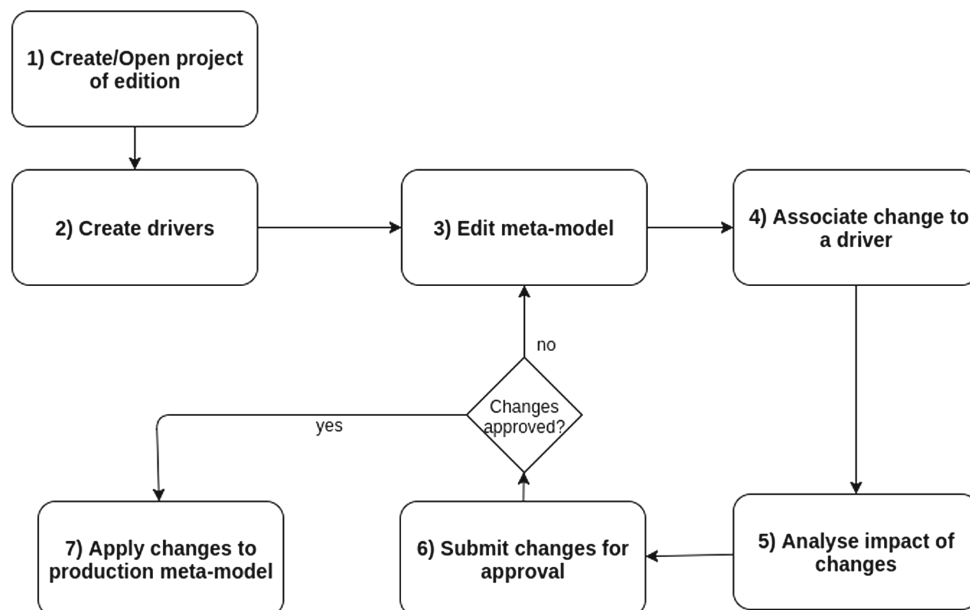


Figura 27. Worklow para evolución de modelos [34].

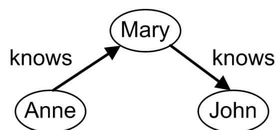
En esta herramienta, el usuario primero deberá elegir el origen de los metamodelos a editar. Después, en el paso 2, se crea el concepto de *Driver,* que hace referencia a uno o más cambios a un metamodelo. Más tarde, se realizan las transformaciones deseadas al metamodelo, interactuando con la representación gráfica del mismo. En el siguiente paso estas transformaciones tendrán que ser vinculadas a un *Driver*, explicando el motivo de para qué son estás transformaciones. Entonces, se procederá a analizar el impacto de los cambios, mostrando el resultado en una serie de modelos tras las modificaciones del metamodelo. Este análisis está basado en indicadores de incoherencias y problemas entre relaciones del metamodelo, con el fin de guiar al usuario en la toma de decisiones. Finalmente, si los cambios son aprobados, estos se aplicarán al metamodelo en producción.

La segunda herramienta es *MM-evolver*[8], la primera herramienta del mercado para manejar la gestión de múltiples modelos de base de datos. Esta herramienta se encuentra a la espera de cualquier modificación en el esquema de una base de datos para propagarlos a todos los modelos a los que afecten esos cambios. Estas modificaciones pueden darse entre modelos de un mismo esquema o entre modelos de diferentes esquemas.

Para ilustrar el reto de la gestión de bases de datos multimodelo, los autores consideran el ejemplo mostrado en la Figura 28. En este ejemplo tenemos datos organizados en distintos modelos de datos. La información de los clientes se almacena en una tabla relacional, mientras que el grafo muestra las relaciones entre los clientes. Además, en documentos JSON se almacenan los pedidos, incluyendo el número y nombre del producto. El último tipo de datos consiste en pares clave/valor relacionando los clientes y los productos.

El reto de este sistema es afrontar la evolución de los datos. Pongamos que se quiere añadir una nueva propiedad a un modelo que no afecta al resto de modelos, por ejemplo, en el JSON para representar los pedidos. Más tarde se decide mover esta propiedad a otro modelo, por ejemplo, al modelo relacional. Esta propiedad puede tener ya relaciones en otros modelos, por lo tanto, MM-evolver modificaría lo necesario para mover este atributo y actualizar todos los modelos de acuerdo con ese cambio.

[8] MM-Evolver: https://openproceedings.org/2019/conf/edbt/EDBT19_paper_310.pdf



a) graph data

```
1 --> "34e5e759"
2 --> "0c6df508"
```

c) key/value pairs

Customer_ID	Name	Credit_limit
1	Mary	5,000
2	John	3,000
3	Anne	2,000

b) relational table

```
{ "Order_no":"0c6df508",
  "Orderlines": [
    { "Product_no":"2724f"
      "Product_Name":"Toy",
      "Price":66 },
    { "Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40 } ] }
```

d) document data

Figura 28. Modelos de datos MM-evolver.

5.4. Conclusiones

Hay que comentar que ninguna de las extensiones que se han mencionado en este capítulo ha servido de precedente para este trabajo, pues el problema que debía solucionar nuestro add-in era muy específico, pues estaba condicionado por las DSL Tools ya desarrolladas en la empresa que va a utilizar nuestra herramienta, por lo que, aunque alguna de estas extensiones es visualmente parecida, ya que añaden un menú con botones a la interfaz, la funcionalidad es totalmente diferente.

No se ha encontrado ninguna extensión en el *Marketplace* que se encuentre enmarcada dentro de un contexto de DSL Tools o dentro de un desarrollo de dominio específico. Por lo tanto, esta sería la única o de las pocas extensiones integradas dentro de un desarrollo de este tipo.

Las extensiones del *Marketplace* de *Visual Studio* que se han presentado se han construido de la misma manera que la extensión presentada en este TFG. Estas extensiones se han construido recientemente, por lo tanto, podemos observar cómo esta manera de extender *Visual Studio* sigue utilizándose a fecha de la realización de este trabajo.

CAPÍTULO 6

Propuesta de solución para la evolución de modelos

En este capítulo se explicará cómo solucionamos el problema anteriormente planteado. Después de evaluar diferentes opciones, se pensó que lo mejor era crear un proyecto de extensibilidad para poder construir un add-in y poder añadir funcionalidades a la interfaz de *Visual Studio.* Estas funcionalidades facilitarán la evolución de los modelos.

Por lo tanto, en este capítulo se va a presentar cómo crear un add-in en *Visual Studio*, qué requisitos previos son necesarios para crearlo, qué proceso seguimos y un breve ejemplo a modo de guía de usuario.

6.1. Requisitos

Para poder extender *Visual Studio* debemos tener instalado el SDK[9]. El SDK es un kit de desarrollo de software que proporciona *Microsoft* para poder extender la funcionalidad que ya nos proporciona el propio IDE de *Visual Studio*. Este SDK contiene bibliotecas, documentación y herramientas necesarias para desarrollar aplicaciones. Por lo tanto, será necesario ya que tendremos que crear un proyecto de extensibilidad para crear nuestra herramienta.

En este caso el add-in se ha implementado bajo *Visual Studio* 2019, la última versión disponible cuando se realizó el desarrollo, pero se pueden crear proyectos de extensibilidad a partir de *Visual Studio* 2015[10].

6.2. Cómo crear un add-in en *Visual Studio*.

Crear un add-in en *Visual Studio* significa extender la propia funcionalidad de este. Para ello, *Microsoft* pone a nuestra disposición unos paquetes de extensibilidad con los que podremos extender la funcionalidad del entorno, por ejemplo, añadiendo botones.

[9] SDK: https://docs.microsoft.com/es-es/visualstudio/extensibility/installing-the-visual-studio-sdk?view=vs-2019
[10] Visual Studio 2015 Extensibility: https://channel9.msdn.com/events/Build/2016/B886

Para añadir un botón a la interfaz de *Visual Studio*, este tiene que estar contenido en una extensión. Para crear una extensión simplemente haremos botón derecho dónde queramos añadir el proyecto dentro de la solución y pulsaremos el botón *Add>New Project.*

Hay dos tipos principales de extensiones: *VSPackages* y extensiones de MEF. Las extensiones *VSPackages* se utilizan para las extensiones que usan o extienden comandos, ventanas de herramientas o proyectos, mientras que las extensiones MEF se usan para extender o personalizar el editor de *Visual Studio*, que es la ventana principal que utilizamos cuando trabajamos con el entorno.

En nuestro caso, lo que queremos es mostrar un botón en la interfaz del entorno de programación. Este botón se hará visible cuando seleccionemos un modelo de nuestra solución. Por eso, elegiremos las extensiones *VSPackages*, pues no necesitamos personalizar el editor de *Visual Studio*. Como vemos en la Figura 29, para crear una extensión *VSPackages*, crearemos un proyecto VSIX.

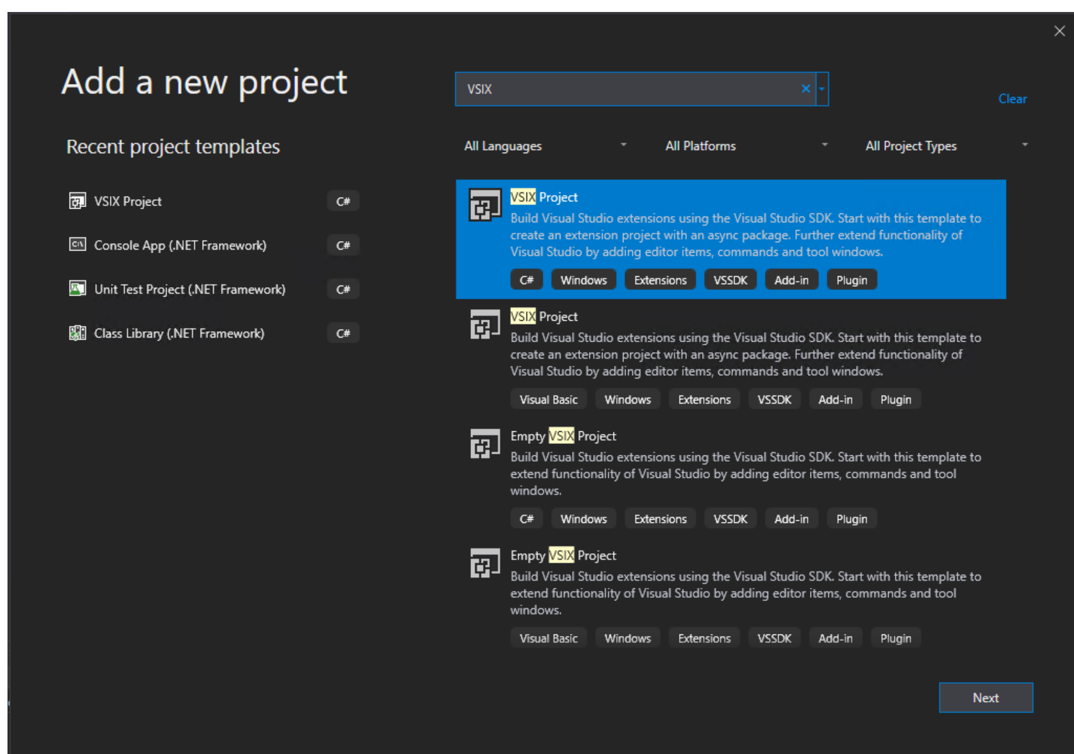


Figura 29. Creación proyecto VSIX.

Tanto las *DSL Tool* desarrolladas como el add-in creado son proyectos de extensibilidad. Esto hace que al ejecutarlos se abra una instancia experimental de *Visual Studio*. Es interesante saber que desde las propiedades del proyecto podremos elegir qué proyecto o solución por defecto queremos que se abra al iniciar la instancia experimental. Esto nos ahorró bastante tiempo en la etapa de pruebas, pues ejecutar y

abrir la nueva instancia es una tarea pesada, por lo que configuramos como solución predeterminada aquella donde se encontraban la mayoría de los modelos.

Una vez tenemos el proyecto VSIX creado, añadiremos un nuevo *Item* al proyecto. Después, como podemos observar en la Figura 30, añadiremos un comando dentro del paquete de extensibilidad *VSPackage*.

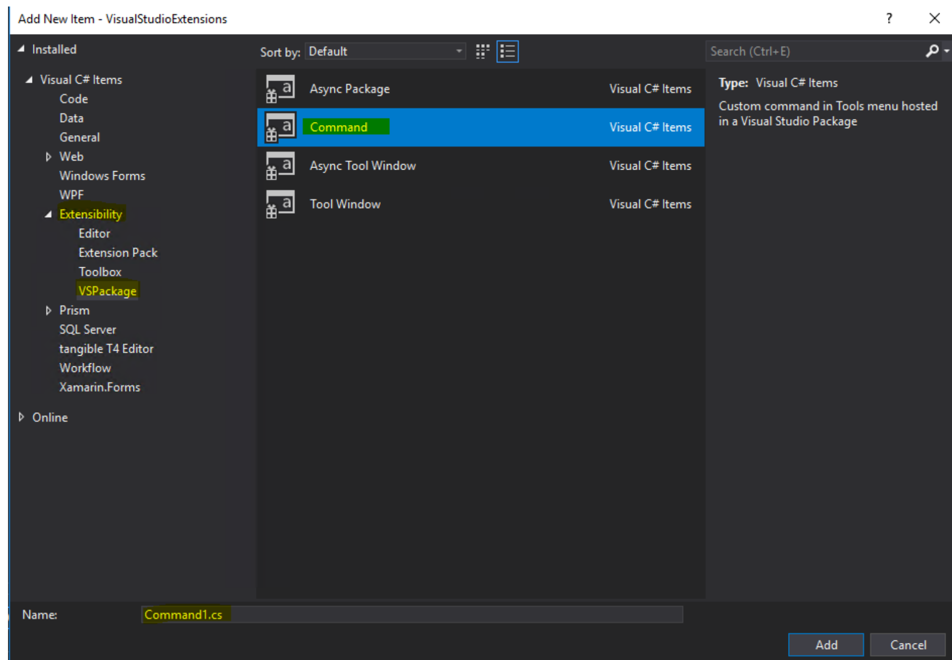


Figura 30. Añadir comando al proyecto.

Si es el primer comando que existe en la extensión, nos generará varios ficheros: *<NombreComando>.cs, <NombrePaquete>.vsct, Resources\<NombreComando>.png.*

- **<NombreComando>.cs**

Contiene la lógica asociada al nuevo comando. En esta lógica se encuentra el proceso de inicialización del botón y el método que se ejecutará cuando el botón se accione. En definitiva, en esta clase se localizará la funcionalidad en sí del botón creado. En nuestro caso, como veremos en el punto *6.3. Estructura de la solución .NET,* será diferente pues hemos separado la lógica en clases parciales.

- **<NombrePaquete>.vsct**

Contiene toda la información necesaria para que *Visual Studio* pueda situar el botón dentro del entorno de desarrollo. Cuando creamos un nuevo comando este fichero se actualiza automáticamente añadiendo las secciones necesarias de ese nuevo botón. Solo será necesario modificarlo si se desea cambiar el grupo en el que va a aparecer el botón, el texto del botón, etc.
Este fichero contiene varias secciones:

○ <*Groups*>: En esta sección se indican los diferentes grupos donde queremos que aparezcan nuestros botones. El valor de la etiqueta <*Parent*> puede ser un grupo por defecto de Visual Studio o uno ya existente definido por nosotros [30].

Además, podemos asignarle una prioridad al grupo. Esta prioridad se utiliza para configurar cómo se ordenan los grupos dentro de un menú. En nuestro caso, y como podemos observar en la Figura 31, nuestros botones estarán en el grupo por defecto de *Visual Studio* cuyo identificador es *IDM_VS_CTXT_ITEMNODE*, que corresponde con el menú que aparece al seleccionar y hacer *click* derecho en un elemento dentro del Explorador de Soluciones.

```
<Groups>
  <Group guid="guidVisualStudioExtensionsPackageCmdSet" id="ModelOperationsGroup" priority="0x0600">
    <Parent guid="guidSHLMainMenu" id="IDM_VS_CTXT_ITEMNODE" />
  </Group>
</Groups>
```

Figura 31. Localización de un comando al hacer click en un elemento del Explorador de Soluciones.

Algunos ejemplos de otros grupos por defecto de Visual Studio serían:

- ***IDM_VS_MENU_PROJECT:*** Con este identificador el botón aparecería al pulsar sobre la opción *Proyecto* de la barra de herramientas superior de *Visual Studio.*

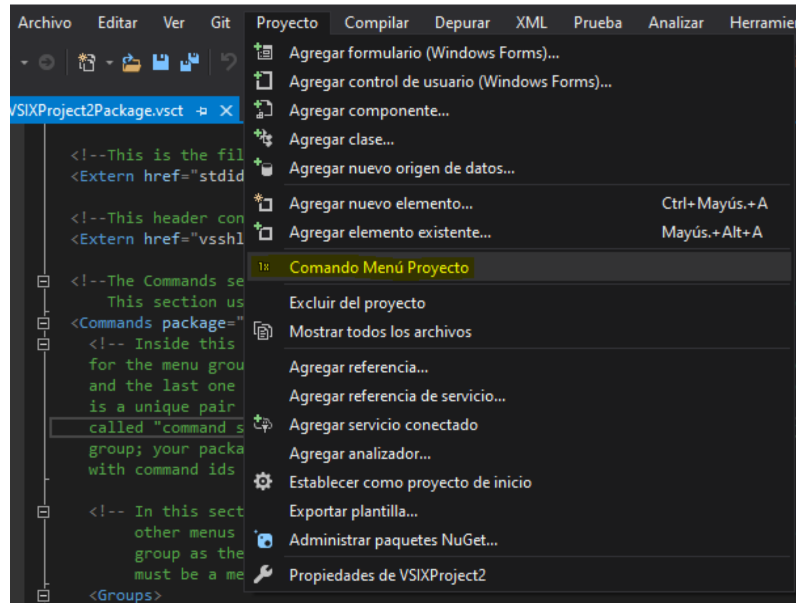


Figura 32. Comando en Menú Proyecto.

- ***IDM_VS_CTXT_CODEWIN:*** Con este identificador el botón aparecería al hacer *click* derecho sobre el editor de código.

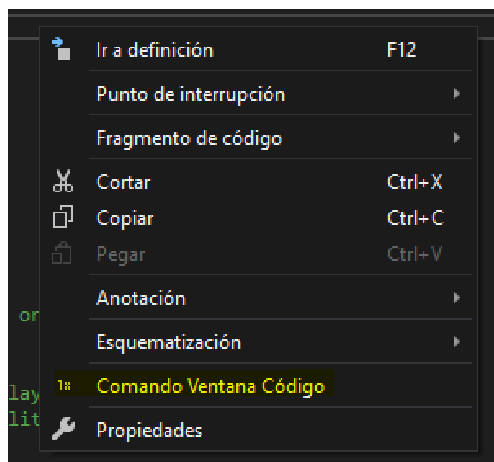


Figura 33. Comando en editor de código.

○ <*Buttons*>: En esta sección se encuentran definidos los diferentes botones que forman parte de la extensión.

- <*Button*>: Hace referencia a un botón. Deberemos tener tantas secciones de estas como comandos hayamos creado. Cada uno de estos botones se identifica con un GUID y un id, además de tener asociado una prioridad y un tipo.

  El GUID debe ser el mismo que el GUID del proyecto que forma parte, el id debe ser único para cada botón y la prioridad permitirá configurar cómo se ordenan los botones dentro de un mismo grupo.

  ▪ <*Icon*>: Permite definir un icono para el botón [30]. En la Figura 34 podemos observar cómo se le ha asignado un icono a un botón de ejemplo. Para ello simplemente habrá que añadir el icono en la carpeta de recursos y hacerle referencia desde el apartado *Icon* mediante el GUID habiendo creado previamente un *Bitmap* en el apartado *Bitmaps.*

```
<Bitmaps>
    <Bitmap guid="guidImages" href="Resources\View.png"/>
</Bitmaps>
```

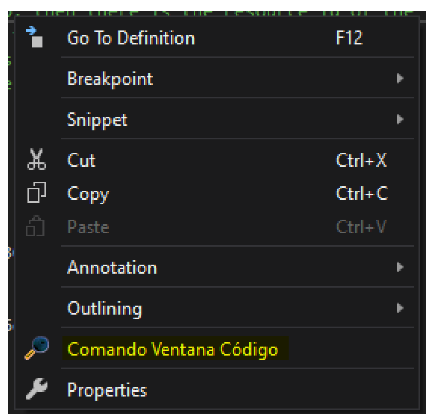


Figura 34. Asignar icono a botón.

- <*Strings*>: Permite añadirle un texto al botón.
- <*CommandFlag*>: permite modificar el comportamiento del botón dentro de la ventana de VS .

En nuestro caso, y como podemos observar en la Figura 35, los botones que se han añadido serán invisibles por defecto, y esta visibilidad será dinámica. Esto es porque nuestros botones solo serán visibles cuando hagamos *click* derecho en elementos que sean un modelo, de tal forma que, si hacemos *click* derecho sobre un elemento que no es un modelo, los botones serán invisibles y no podremos utilizarlos.

```
<Buttons>
  <Button guid="guidVisualStudioExtensionsPackageCmdSet" id="MoveModelsMenuCommandId" priority="0x0100" type="Button">
    <Parent guid="guidVisualStudioExtensionsPackageCmdSet" id="ModelOperationsGroup" />
    <CommandFlag>DefaultInvisible</CommandFlag>
    <CommandFlag>DynamicVisibility</CommandFlag>
    <Strings>
      <ButtonText>Move Models to...</ButtonText>
    </Strings>
  </Button>
  <Button guid="guidVisualStudioExtensionsPackageCmdSet" id="RenameModelsMenuCommandId" priority="0x0200" type="Button">
    <Parent guid="guidVisualStudioExtensionsPackageCmdSet" id="ModelOperationsGroup" />
    <CommandFlag>DefaultInvisible</CommandFlag>
    <CommandFlag>DynamicVisibility</CommandFlag>
    <Strings>
      <ButtonText>Rename Model</ButtonText>
    </Strings>
  </Button>
</Buttons>
```

Figura 35. Definición de los comandos.

- <*Bitmaps*>: Esta sección contiene las diferentes imágenes que se relacionan con los botones.
- <*Symbols*>: Como vemos en la Figura 36, en esta sección se definen los diferentes identificadores: el identificador del proyecto, el de los grupos, el de los botones, etc.

```
<Symbols>
  <!-- This is the package identifier. -->
  <GuidSymbol name="guidVisualStudioExtensionsPackage" value="{f0155523-a7d4-4011-823b-a2ee454ea30f}" />

  <!-- This is the identifier used to group the menu commands together -->
  <GuidSymbol name="guidVisualStudioExtensionsPackageCmdSet" value="{a4d3f527-8a96-4fd4-9a90-0ffb48149107}">
    <IDSymbol name="ModelOperationsGroup" value="0x1020" />
    <IDSymbol name="MoveModelsMenuCommandId" value="0x0100" />
    <IDSymbol name="RenameModelsMenuCommandId" value="0x0200" />
  </GuidSymbol>
</Symbols>
```

Figura 36. Definición de la prioridad de los botones.

Cuando creamos un botón, este fichero se actualiza automáticamente añadiendo las secciones necesarias. Será necesario que las modifiquemos siempre y cuando se quiera cambiar el grupo en el que aparece el botón, el texto, etc.

- **Resources\<NombreComando>.png**

Es la imagen por defecto que utiliza el botón. Puede eliminarse o cambiarse por la que se desee.

6.3. Estructura de la solución .NET

El add-in se encuentra dentro de la solución donde está implementada cada una de las *DSL Tool*, pues al igual que ellas, son proyectos de extensibilidad. Por eso, como podemos ver en la Figura 37, todo nuestro trabajo estará dentro del proyecto *Extensions,* en concreto dentro de la carpeta *VisualStudioExtensions*.

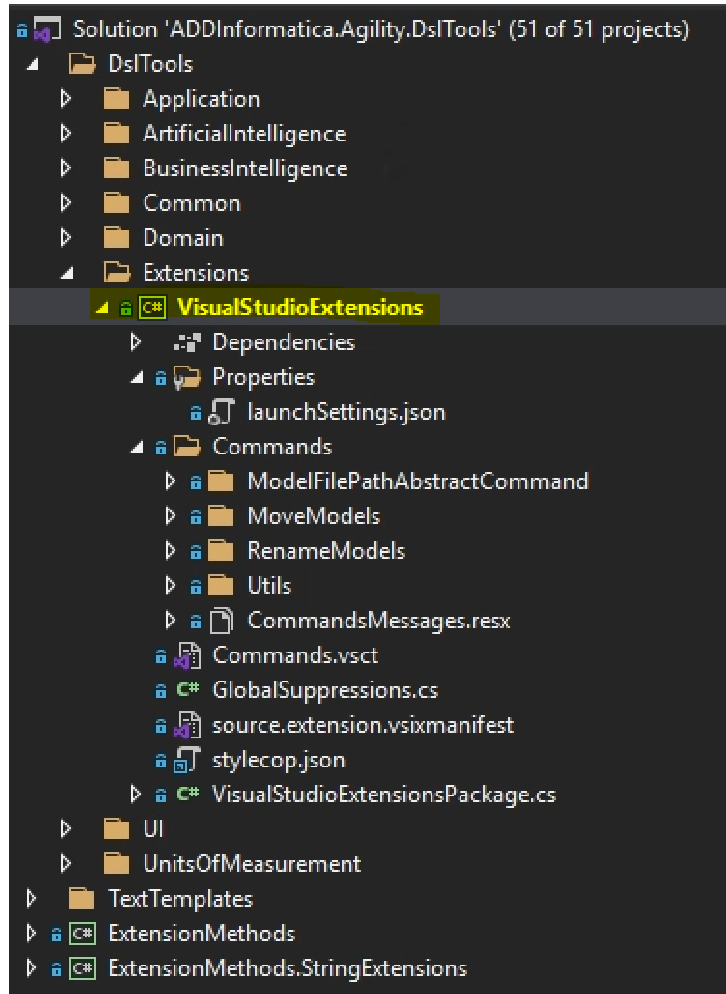


Figura 37. Estructura de la solución para evolucionar modelos.

Dentro de este proyecto, las clases estarán estructuradas de la siguiente manera:

- **Carpeta *ModelFilePathAbstractCommand:*** En esta carpeta se encuentran las clases que contienen la lógica común a ambos botones, es decir, donde se encuentran los manejadores de los eventos de ejecución de los comandos. Dado que los botones son para mover y renombrar modelos, mucha de la funcionalidad de estos botones es común, pues renombrar un modelo es cambiar el nombre de este en la ruta relativa de las referencias modelbus.
- **Carpeta *MoveModels:*** En esta carpeta se encuentran las clases asociadas al botón de mover los modelos. En esta carpeta se encuentra el manejador del

evento que lanza la inicialización del botón, así como la parte de la funcionalidad que es específica de mover modelos.

- **Carpeta *RenameModels:*** Al igual que en la carpeta anterior, se encuentra el manejador del evento que lanza la inicialización del botón de renombrar los modelos. Además de esto, aquí tenemos las clases que contienen funcionalidad específica para el botón de renombrar un modelo, como puede ser el formulario que aparece para elegir el nuevo nombre del modelo, etc.
- **Carpeta *Utils:*** Aquí podremos encontrar toda la funcionalidad relacionada con los ficheros XML. Además, aquí encontramos las clases que controlan la modificación de las referencias modelbus para cada uno de los tipos de modelos.

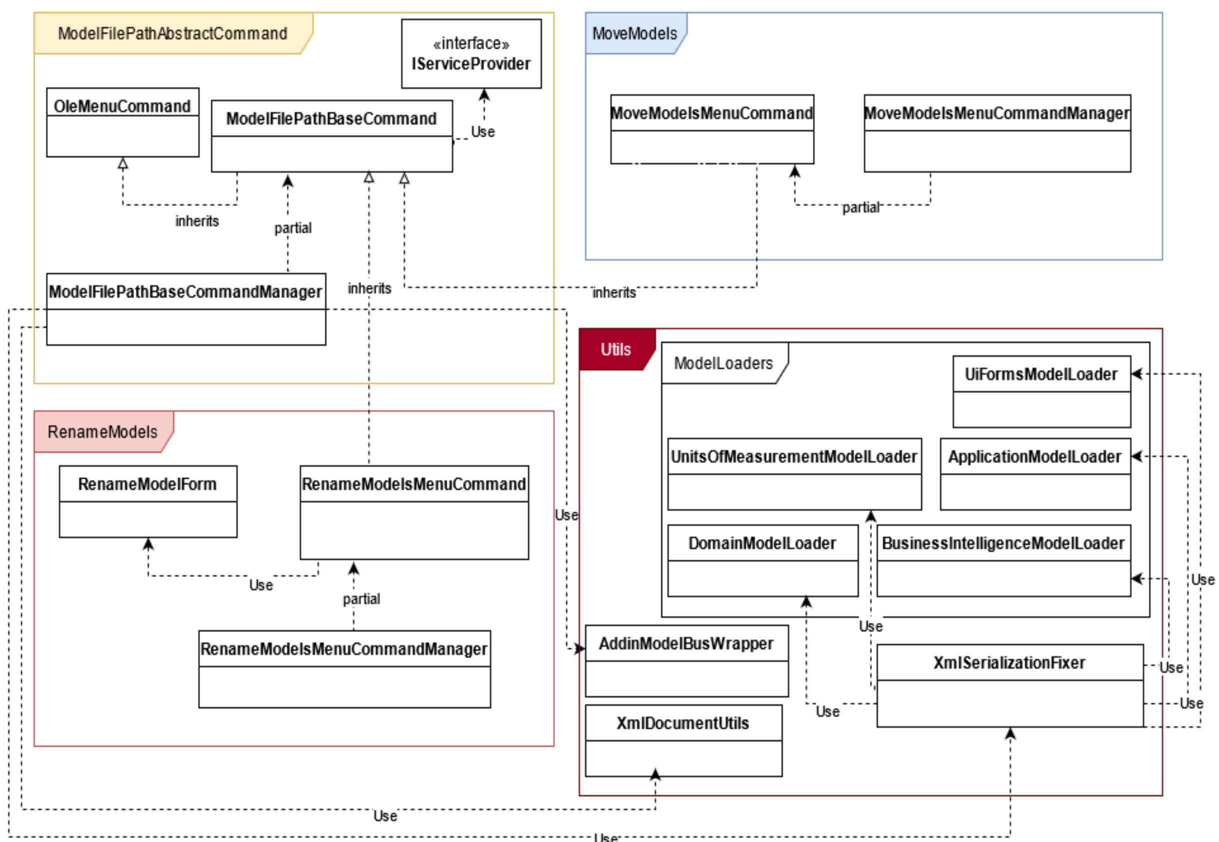


Figura 38. Estructura interna de la solución para evolucionar modelos.

La Figura 38 ilustra el esquema de la estructura del desarrollo realizado. Como podemos observar la parte principal de la extensión se encuentra en la carpeta *ModelFilePathAbstractCommand*, que será la que gestione toda la funcionalidad de la extensión. Cabe mencionar que se apostó por un diseño modular, para que en la medida de lo posible se pudieran añadir más formas de evolucionar los modelos en el futuro.

6.3.1. Carpeta ModelFilePathAbstractCommand

A. La clase *ModelFilePathAbstractCommand*

Los comandos son, por defecto, un *MenuCommand* [30]. En nuestro caso estos comandos están construidos como un *OleMenuCommand* [30], pues los botones se mostrarán de manera dinámica.

Esta clase está definida como abstracta y parcial. Pero, además, es necesario que esta clase herede de *OleMenuCommand*, ya que así comprobaremos si el elemento que estamos intentando mover o renombrar es un modelo y así poder hacer visible el botón correspondiente dinámicamente.

El método que se muestra a continuación es aquel que lanza la comprobación del elemento que estamos intentando evolucionar. Este método se ejecutará antes de inicializar el comando y ejecutará el método *BeforeQueryStatusEventHandler* de la clase *ModelFilePathAbstractCommandManager* que veremos en el siguiente apartado.

```
/// <summary> The handler for the event that is called before
executing the
/// <see cref="ModelFilePathBaseCommand"/>. </summary>
/// <param name="sender"> The source of the event. </param>
/// <param name="eventArgs"> The <see cref="EventArgs"/>.
</param>
private static void CustomCommandBeforeQuery(object sender, EventArgs
eventArgs)
{
      ModelFilePathBaseCommand modelFilePathBaseCommand = sender
      as ModelFilePathBaseCommand;

       modelFilePathBaseCommand?.BeforeQueryStatusEventHandler();
}
```

Para configurar el manejador del evento que ejecutará la funcionalidad del comando, deberemos tener un método como el mostrado a continuación. Este método se lanzará una vez comprobado que el elemento seleccionado es un modelo. Este método es el que inicia la ejecución de la funcionalidad de los comandos, y que desencadenará toda la funcionalidad.

```
/// <summary> The handler for the event that executes the <see
/// cref="ModelFilePathBaseCommand"/>. </summary>
/// <param name="sender"> The source of the event. </param>
/// <param name="eventArgs"> The <see cref="EventArgs"/>.
</param>
private static void CustomCommandExecute(object sender,
EventArgs eventArgs)
{
```

```
    ModelFilePathBaseCommand modelFilePathBaseCommand = sender
    as ModelFilePathBaseCommand;

    modelFilePathBaseCommand?.Execute();
}
```

B. La clase *ModelFilePathAbstractCommandManager*

Definida como clase parcial a la clase anterior, aquí podemos encontrar todo el código necesario para cambiar la ruta de las referencias modelbus de los *XML* de los modelos. Aquí se encuentra todo el grueso de la lógica.

Por tanto, en esta clase se encontrarán todos los métodos que serán llamados desde las clases manejadoras de los comandos a los que pertenece cada botón. Los métodos mostrados a continuación son los llamados desde la clase *ModelFilePathAbstractCommand* y que son los que desencadenan la ejecución y visibilidad de los botones.

```
/// <summary> Event handler that manages the visibility of the menu
command.
private void BeforeQueryStatusEventHandler()
{
      this.Enabled = this.ModelIsSelected();
      this.Visible = this.Enabled;
}

/// <summary> Executes the logic associated to this command.
</summary>
private void Execute()
{
      IEnumerable<string> selectedItemsPaths =
this.GetSelectedFiles();
      if (!selectedItemsPaths.Any())
       {
            return;
       }

      this.CustomCommandExecute(selectedItemsPaths);
}
```

En esta clase también podemos encontrar otros métodos importantes como los que se encargan de recargar los elementos del proyecto una vez hemos evolucionado los modelos, como vemos más abajo, los que se encargan de cargar el XML de los modelos y renombrar las referencias modelbus o como el que determina el tipo de modelo con el que estamos trabajando para saber qué tipos de modelo hemos de modificar; esto es, si estamos trabajando con un modelo de dominio, tenemos que comprobar y modificar los modelos cuyo lenguaje contenga una dependencia al lenguaje de dominio, tal y como explicamos en el capítulo de *Definición del problema*.

```
/// <summary> Reloads the project so that changes are visible in the
Solution Explorer. </summary>
private void ReloadProject()
{
	// Gets the selected items of the current solution opened in the
	Solution Explorer.
	Array selectedItems =
	(Array)this.dte.ToolWindows.SolutionExplorer.SelectedItems;

	// Gets the project to which the selected items belong.
	IEnumerable<string> solutionProjectsName = selectedItems
	.Cast<UIHierarchyItem>()
	.Select(uiItem => uiItem.Object as ProjectItem)
	.Select(projectItem => projectItem.ContainingProject.Name);

	string currentProjectName =
solutionProjectsName.FirstOrDefault();

	// Gets the Solution Explorer Window through the DTE.
	Window solutionExplorerWindow =
	this.dte.ToolWindows.SolutionExplorer.Parent;

	// Activates the solution explorer so that we can reload it.
	solutionExplorerWindow.Activate();
	UIHierarchyItem solutionItem =
	this.dte.ToolWindows.SolutionExplorer.UIHierarchyItems.Item(1);

	UIHierarchyItem currentProjecItem =
	GetProjectItemFromSolution(currentProjectName, solutionItem);

	if (currentProjecItem is null)
	{
		return;
	}

	// Select the project so that we can unload and reload it.

currentProjecItem.Select(vsUISelectionType.vsUISelectionTypeSelect);

	// Unloads the project.
	this.dte.ExecuteCommand("Project.UnloadProject");

	// Reloads the project.
	this.dte.ExecuteCommand("Project.ReloadProject");
}
```

6.3.2. Carpeta MoveModels

A. La clase *MoveModelsMenuCommand*

Clase parcial que hereda de la clase *ModelFilePathAbstractCommand.* Esta clase solo contiene su constructor y el método que inicializa la instancia del comando para mover modelos.

```
/// <summary> Initializes the instance of the <see
cref="MoveModelsMenuCommand"/>. /// /// </summary>
/// <param name="package"> The owner <see cref="AsyncPackage"/>.
</param>
```

```
/// <returns> An empty task that enables this method to be awaited.
</returns>
public static async Task InitializeAsync(AsyncPackage package)
{
      // Switch to the main thread as the call to AddCommand in
      the constructor requires the
      // UI thread.
      Await
      ThreadHelper.JoinableTaskFactory.SwitchToMainThreadAsync(package
      .DisposalToken);
      OleMenuCommandService commandService = await
      package.GetServiceAsync(typeof(IMenuCommandService)) as
      OleMenuCommandService;
      DTE2 dte2 = await package.GetServiceAsync(typeof(DTE)) as
      DTE2;

      Instance = new MoveModelsMenuCommand(package,
      commandService, dte2);
}
```

B. La clase *MoveModelsMenuCommandManager*

Esta clase se encarga de implementar toda la funcionalidad relacionada con el comando para mover modelos a una diferente localización. Contiene el método *CustomCommandExecute*, llamado desde la clase *ModelFilePathAbstractCommandManager* que se lanza al hacer *click* en el botón *Mover modelo.*

Primero, en este método se abre un diálogo del tipo *FolderBrowserDialog* para que el usuario elija la ruta destino a la que queremos mover el modelo. Después se realizan unas comprobaciones para saber si se puede mover el modelo a la localización destino que el usuario ha elegido. Entre estas comprobaciones están:

- Si el XML del modelo con el que estamos trabajando está bien formado.
- Si la ruta destino a la que vamos a mover el modelo existe.
- Si en la ruta a la que vamos a mover el modelo existe ya un modelo con el mismo nombre.

Una vez se ha comprobado que se puede mover el modelo, entonces se realiza una invocación al método *ChangeModelFilePath* de la clase *ModelFilePathAbstractCommandManager* que, como hemos comentado anteriormente, es el que contiene toda la funcionalidad.

Este método recorre todas las referencias modelbus del modelo que estamos moviendo para obtener los modelos a los que referencia y los modelos que le referencian. Al mismo tiempo que obtenemos estos modelos, modificamos las referencias modelbus reemplazando la ruta relativa de esta. Estas referencias modelbus

las obtendremos mediante expresiones regulares, separando el nombre del modelo del resto de la referencia modelbus.

Una vez se han modificado las referencias modelbus del modelo que estamos moviendo, obtendríamos las referencias modelbus del resto de modelos que quedan por tratar. Entonces, recorremos las referencias modelbus de estos modelos obteniendo aquellas que hagan referencia al modelo que estamos moviendo. Finalmente calculamos la nueva ruta relativa al modelo que estamos moviendo para modificar cada una de estas referencias modelbus.

6.3.3. Carpeta RenameModels

Contiene el mismo número de clases que la carpeta anterior para el comando de mover modelos. Las diferencias están en el método CustomCommandExecute de la clase del manejador del comando *RenameModelsMenuCommandManager.* En esta ocasión antes de recorrer las referencias modelbus del modelo que estamos renombrando, abrimos un formulario creado por nosotros en el que mostramos el nombre actual del modelo con el que estamos trabajando y un campo de texto para que el usuario elija el nuevo nombre, como vemos en la Figura 39.

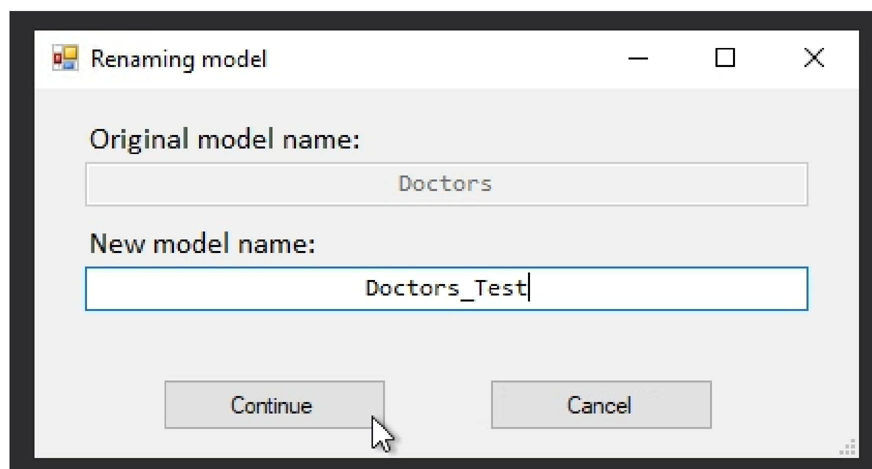


Figura 39. Formulario para renombrar un modelo.

En este caso al modificar las referencias modelbus de los modelos referenciados por el modelo que estamos renombrando, solo tenemos que modificar el nombre de este modelo en esas referencias modelbus, pues como no estamos cambiando la localización, la ruta relativa a este será la misma.

6.3.4. Carpeta Utils

En esta carpeta podemos encontrar una subcarpeta llamada *ModelLoaders,* donde tenemos una clase para cada uno de los tipos de modelo de *DSL Tool* que tenemos. En cada una de estas clases podemos encontrar un método llamado *LoadModel* para cargar el modelo en memoria con el que estamos trabajando y poder modificarlo más tarde.

```
protected override ApplicationModel LoadModel(Store store, string
modelPath)
{
        return
        ApplicationDSLSerializationHelper.Instance.LoadModel(store,
        modelPath, null, null, null);
}
```

Además de esto, podemos encontrar otras clases que contienen la utilidad necesaria para leer y modificar el *XML* de los modelos. En la clase *XmlDocumentUtils* podemos encontrar toda la funcionalidad relativa a operaciones con los modelos. Por ejemplo, el siguiente bloque de código muestra cómo obtenemos el identificador de una referencia modelbus que relaciona dos modelos a través de una expresión regular.

```
/// <summary>
/// Gets the identifier of the External Element referenced by a Model
     External Reference.
/// </summary>
/// <param name="externalReference"> The Reference to the External
     Element. </param>
/// <param name="referenceTagName">
///     The tag of the Model External Reference in the Model.
/// </param>
/// <returns>
/// The identifier of the External Element referenced by the Model
     External Reference.
/// </returns>
private static Guid
GetExternalReferenceReferencedModelElementId(XElement
externalReference, string referenceTagName)
{
        string modelBusReference =
        externalReference.Attribute(referenceTagName).Value;

        // This regular expression gets the identifier of the Model
        Element referenced in the
        // ModelBus Reference, which starts with '%2f'.
        Match match = Regex.Match(modelBusReference, @"%2f+.*$");

        // This check is necessary for some elements such as Forms,
        because it does not need to
        // reference any DTO.
        Guid referencedId = match.Success ?
        Guid.Parse(match.Captures[0].Value.Substring(3)) : Guid.Empty;
```

```
        return referencedId;
}
```

El siguiente código de la misma clase mencionada anteriormente obtiene la ruta relativa de una determinada referencia modelbus, clave a la hora de modificar dicha ruta relativa al mover o renombrar un modelo.

```
/// <summary>
/// Gets the referenced <see cref="ModelElement"/> parent <see
     cref="ModelBase"/>
///     relative path.
/// </summary>
/// <param name="externalReference"> The Reference to the External
Element. </param>
/// <param name="referenceTagName">
///     The tag of the Model External Reference in the Model.
/// </param>
/// <returns>
/// The referenced <see cref="ModelElement"/> parent <see
     cref="ModelBase"/> relative path.
/// </returns>
private static string
GetExternalReferenceReferencedModelElementModelPath(XElement
externalReference, string referenceTagName)
{
        string modelBusReference =
        externalReference.Attribute(referenceTagName).Value;
        // This regular expression gets the relative path of the
        referenced  Model Element parent
        // Model in the ModelBus Reference.
        Match match = Regex.Match(modelBusReference,
        @"[\\].*[.](ddsl|adsl|uifdsl|bidsl|uomdsl)",
        RegexOptions.Compiled);

        // This check is necessary for some elements such as Forms,
        because it   does not need to
        // reference any DTO.
        string referencedModelElementModelPath = match.Success ?
        match.Captures[0]?.Value : string.Empty;

        return referencedModelElementModelPath;
}
```

6.4. Desafíos del desarrollo.

En este apartado se van a mostrar algunos aspectos más complejos que han surgido durante el desarrollo de la extensión.

6.4.1. Validación de modelos

Al principio nuestro método de validación de la extensión era manual. Pero después, hicimos que todas las referencias modelbus actualizadas fueran validadas de forma automática, para así ahorrar tiempo en la búsqueda de los posibles errores que pudieran aparecer.

Toda *ModelBusReference*[11] es serializada para poder ser almacenada en forma de cadena. Estas ModelBusReference contienen un *ReferenceContext,* que es un diccionario en el que se puede almacenar información, como la ruta de acceso al archivo respecto a la cual debe serializarse.

Gracias a la clase *BrokenReferenceDetector*[12] podemos validar automáticamente todas las propiedades de dominio que se encuentren en un almacén de referencias modelbus. Esta comprobación podemos lanzarla cada vez que termina el proceso de actualización de las referencias, es decir, cuando el proceso que lleva a cabo nuestra extensión ha finalizado.

```
private const string INVALID_REF_FORMAT =
      "La propiedad de dominio '{0}' de la referencia"
      + '{1}' tiene el valor '{2}' y es invalido";

[ValidationMethod(ValidationCategories.Save)]
public void ValidateModelBusReferences(ValidationContext context)
{
      BrokenReferenceDetector.DetectBrokenReferences(this.Store,
            delegate(ModelElement element, // parent of property
                  DomainPropertyInfo property, // identifies property
                  ModelBusReference reference) // invalid reference
            {

                  context.LogError(string.Format(INVALID_REF_FORMAT,
                  property.Name,
                  referenceState.Name,
                  new ModelBusReferenceTypeConverter().
                        ConvertToInvariantString(reference)),
                  "ExternalEntityReference",
                  element);
            }
      );
}
```

[11] *ModelBusReference class*: https://docs.microsoft.com/en-us/previous-versions/ee904249(v=vs.140)
[12] *BrokenReferenceDetector class*: https://docs.microsoft.com/en-us/previous-versions/ee904247(v=vs.140)

Este método valida todas las propiedades que se encuentren en el almacén y notifica las referencias que no funcionan en la ventana de errores de nuestra instancia de *Visual Studio*. Así, si al utilizar nuestra extensión los modelos implicados no se abren correctamente, sabremos qué propiedad del XML está mal formada.

6.4.2. Serialización y cargado de modelos en memoria

Cada vez que se modifica un modelo es necesario abrirlo para comprobar que el formato del XML que lo compone sigue el esquema del modelo en lugar del formato estándar de XML. Por tanto, necesitamos un proceso que se lance después de mover o renombrar los modelos y que, en caso de que el XML esté mal formado, lo repare. El siguiente trozo de código se encuentra en el método encargado de guardar los modelos modificados y representa la llamada al método *FixModifiedModelXmlFormat* de la clase *XmlSerializeationFixer* que desencadena el proceso que estamos comentando.

```
// Each time a Model is modified, it is necessary to open it to be
sure that the XML
// format follows the Model schema instead of the standard XML format.
foreach (string modelPath in this.OpenedModelsDictionary.Keys)
{
	if (!this.OpenedModelsDictionary[modelPath].IsModified)
	{
		continue;
	}

	// This check is necessary because the moving Model has already
	been moved so its
	// path must be the destination one.
	if (modelPath.Equals(moveModelFrom,
	StringComparison.OrdinalIgnoreCase))
	{

	xmlSerializationFixer.FixModifiedModelXmlFormat(moveModelTo);
		continue;
	}

	xmlSerializationFixer.FixModifiedModelXmlFormat(modelPath);
}
```

Como podemos observar en la siguiente pieza de código, el método FixModifiedModelXmlFormat contiene una instrucción para cada tipo de modelo. Así, diferenciaremos entre cargar un modelo basado en la DSL Tool de BI, dominio, aplicación, UI o Unidades de Medida, pues cada tipo de modelo contiene un formato XML distinto. Por tanto, tendremos una clase para cada tipo de modelo que se encargue de cargar el modelo que recibe como parámetro.

```
/// <summary> Fixes the XML Document of the Model in the given path.
</summary>
/// <param name="modelPath"> The path of the Model to fix the XML.
</param>
public void FixModifiedModelXmlFormat(string modelPath)
{
	switch (Path.GetExtension(modelPath))
	{
		case ".adsl":

	this.ApplicationModelLoader.LoadModelInPath(modelPath);

			break;
		case ".bidsl":
			this.BusinessIntelligenceModelLoader.LoadModelInPath
			(modelPath);

			break;
		case ".ddsl":
			this.DomainModelLoader.LoadModelInPath(modelPath);
```

El método *LoadModelInPath* crea un objeto de tipo *Store* [13] para crear una representación del modelo en memoria. Entonces, se crea una transacción para cargar el modelo ubicado en la ruta recibida como parámetro.

```
/// <summary> Loads the <typeparamref name="TModelType"/> in the given
path. </summary>
/// <param name="modelPath">
///     The path where the <typeparamref name="TModelType"/> is
located.
/// </param>
/// <returns> The <typeparamref name="TModelType"/>. </returns>
public TModelType LoadModelInPath(string modelPath)
{
	using (Store store = new Store(this.ServiceProvider,
	typeof(TDSLDomainModelType)))
	{
		using (Transaction transaction =
		store.TransactionManager.BeginTransaction())
		{
			return this.LoadModel(store, modelPath);
		}
	}
}
```

El método *LoadModel* es el que contiene la lógica necesaria para cargar el modelo en memoria. En este método, como podemos ver a continuación, se abre el modelo, se aplican las migraciones necesarias asociadas a su versión y se formatea en caso de que no esté en el formato correcto.

[13] Store class: https://docs.microsoft.com/es-es/dotnet/api/microsoft.visualstudio.modeling.store?view=visualstudiosdk-2019

```
public override BusinessIntelligenceModel
LoadModel(SerializationResult serializationResult, Partition
partition, string fileName, ISchemaResolver schemaResolver,
ValidationController validationController, ISerializerLocator
serializerLocator)
{
      using (FileStream fileStream = File.OpenRead(fileName))
      {
            Version modelVersion = GetModelVersion(fileName);

            // XML migrations must be applied before the load of the
            model, since they modify the
            // file directly. They need to run before the rules of the
            DSL Tool.
            ApplyXmlMigrationsToModelFile(fileName, modelVersion);

            BusinessIntelligenceModel model =
            base.LoadModel(serializationResult, partition, fileName,
            schemaResolver, validationController, serializerLocator);
            model.FilePath = Path.GetFullPath(fileName);

            ApplyMigrationsToModelFile(model, modelVersion);

            string existingModelToCompare = File.ReadAllText(fileName,
            Encoding.UTF8);

            string serializedModelToCompare =
            this.GetSerializedModelString(model, Encoding.UTF8,
            writeOptionalPropertiesWithDefaultValue: true, fileName);

            // Only save the model when it has changed.
            if (model.Store.ShouldSaveTransformationChanges() &&
            !CompareModel.AreModelsEquals(existingModelToCompare,
            serializedModelToCompare, this.Version))
            {
                  this.SaveModel(new SerializationResult(), model,
                  fileName, writeOptionalPropertiesWithDefaultValue:
                  true);

                  // If the serialization is running inside a text
                  template transformation process, unnecessary
                  namespaces
                  // might need to be removed. This is due to the
                  // If they are not removed, some classes will not
                  recognize them, and might cause that the model
                  // is not loaded. Because of this, the diagram will
                  be serialized again, but without those namespaces.
                  // See PR#3181 for more information.
                if (model.Store.IsTextTemplatingHost())
                {

                  SerializationHelperUtilities.RemoveInvalidNamespaces
                  FromDiagram(fileName, ValidDiagramNamespaces);

                  SerializationHelperUtilities.FixXmlEndTagSpacing(fil
                  eName);
                }
            }

            return model;
        }
```

6.5. Cronología del desarrollo

En el desarrollo de nuestra herramienta se siguió un desarrollo incremental. Como vemos en la Figura 40, este tipo de desarrollo se caracteriza porque el proyecto es descompuesto en una serie de incrementos, en cada uno de los cuales se entrega una parte de la funcionalidad respecto de la totalidad de los requisitos.

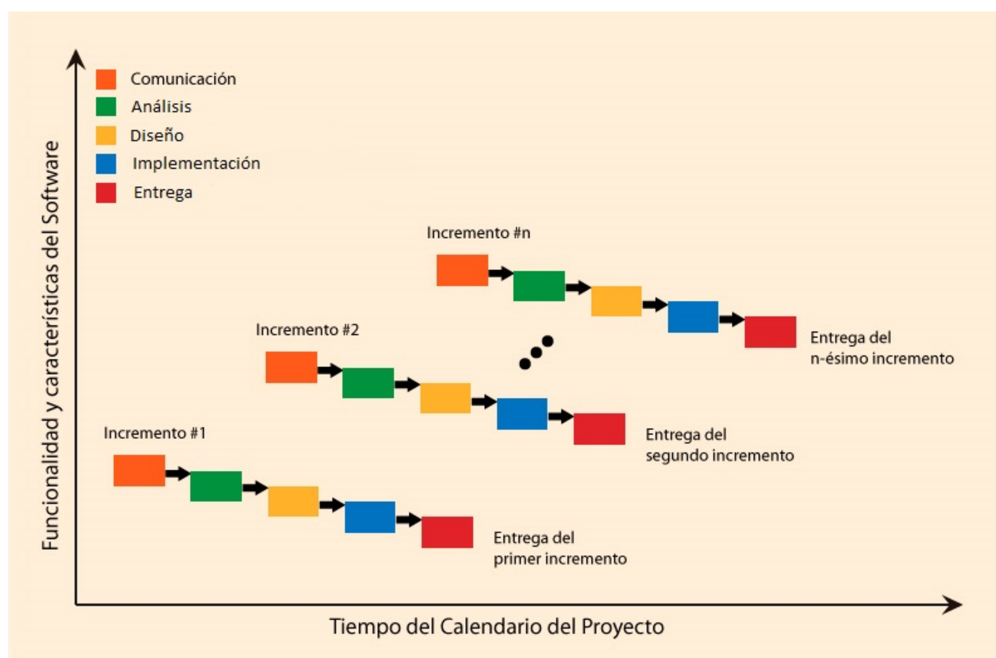


Figura 40. Desarrollo incremental.

Como se puede observar, la primera fase es la fase de comunicación. En esta fase nos reuníamos con los *testers* que, en nuestro caso, eran los analistas del software principal de la empresa donde se enmarca este trabajo y también, usuarios finales de nuestra herramienta.

Por lo tanto, en cada incremento, que coincide con cada vez que se fusionaba nuestra rama con la rama máster en nuestra herramienta de control de versiones, los analistas probaban la herramienta de manera manual y nos reuníamos para preparar los requisitos del siguiente incremento. En las reuniones se abordaban posibles errores o problemas que habían surgido en las pruebas manuales y se les daba prioridad para el siguiente incremento.

La Figura 41 ilustra la cronología del trabajo asociado a este TFG. El trabajo comenzó a plantearse en septiembre del 2020, donde se hicieron reuniones para saber qué es lo que necesitaba el departamento de I+D+i. Una vez supimos cuál era el problema, llegó el momento de conocer el mundo de la generación automática de código y desarrollo de dominio específico a partir de modelos. Cuando se supo el contexto y el problema,

se investigó para conocer cuál era la mejor opción para resolver el problema y se empezó con el desarrollo, que son los tres incrementos que pueden observarse. Finalmente, se realizaron unas pruebas manuales finales para verificar y validar el add-in construido mientras se realizaba la elaboración de esta memoria.

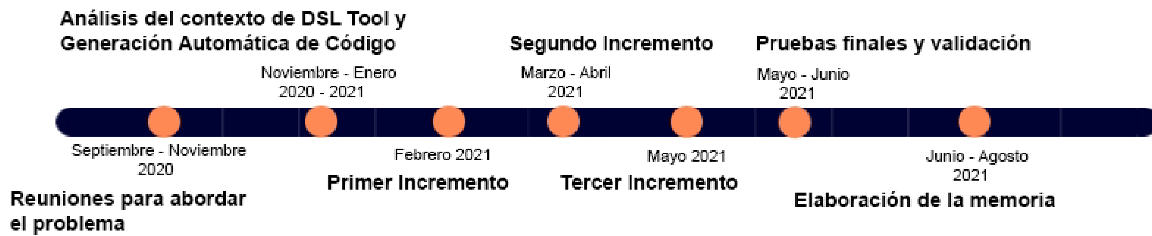


Figura 41. Cronología del desarrollo.

6.6. Guía de uso

6.6.1. Mover un modelo

Para hacer uso de la extensión seleccionaremos un modelo y haremos *click* derecho en él. Después, como podemos observar en la Figura 42, haremos *click* en "*Move Model to...*".

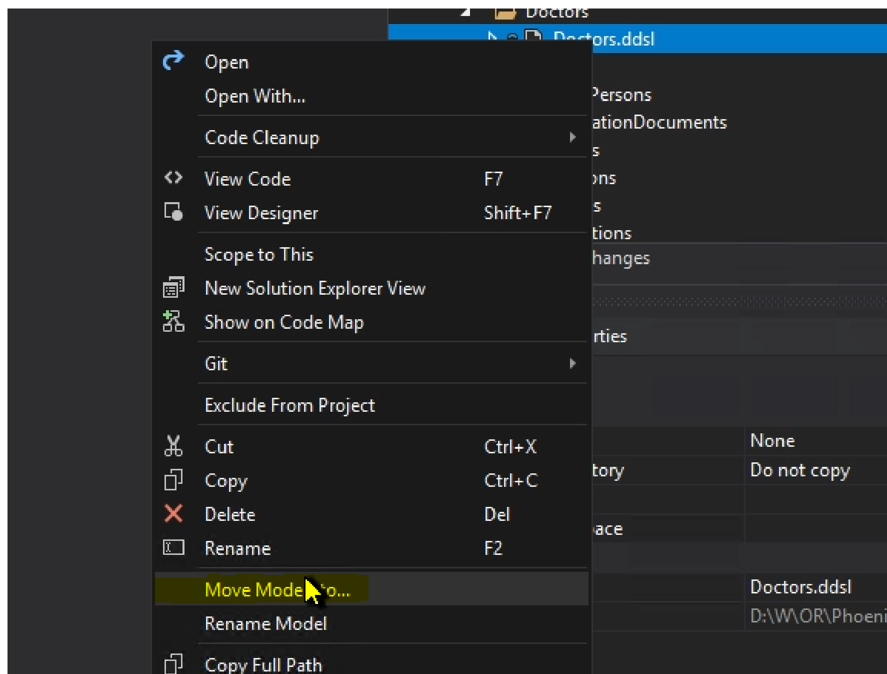


Figura 42. Primer paso para mover un modelo.

Al pulsar el botón para mover un modelo nos aparecerá una ventana. En esta ventana, Figura 43, seleccionaremos la nueva localización donde queremos mover

nuestro modelo y pulsaremos el botón *‘OK’*. Esta ventana también nos da la posibilidad de crear una nueva carpeta.

En este ejemplo vamos a mover el modelo *Doctors.ddsl* basado en la *DSL Tool* de Dominio de la carpeta *Doctors* a la carpeta *Doctors_Test*.

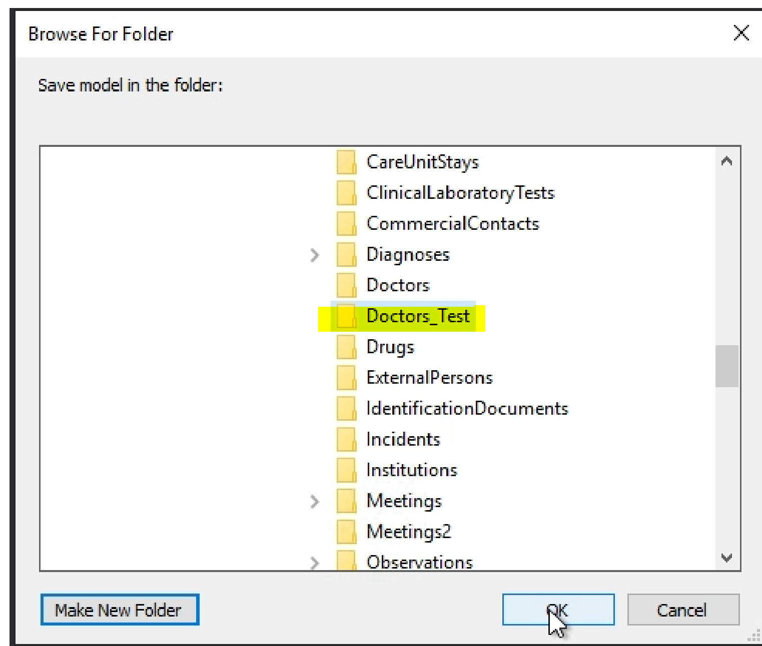


Figura 43. Segundo paso para mover un modelo.

Una vez hacemos pulsamos el botón *Ok* se desencadena todo el proceso necesario para mover el modelo de ruta. En la Figura 44 vemos cómo aparecen unas ventanas en las que se nos indica que ciertos modelos se han modificado. Esto es debido a que las referencias modelbus de esos modelos se han actualizado.

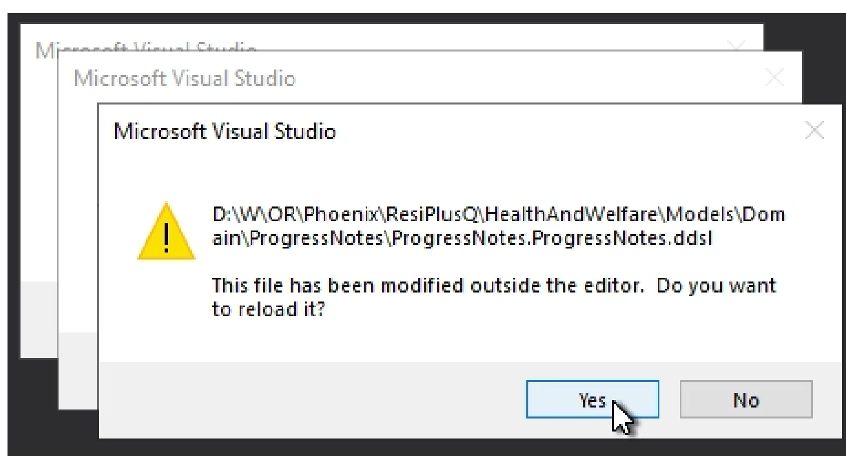


Figura 44. Aviso de ficheros modificados al mover un modelo.

En la Figura 45 y la Figura 46 podemos observar los cambios pendientes una vez ha terminado el proceso. En la Figura 45 podemos ver cómo el modelo aparece ya en la

solución en la nueva localización y aparece borrado en la antigua localización. Esto es así ya que una vez ha realizado todo el proceso de actualización de referencias modelbus, se refresca toda la solución en busca de cambios. Esto lo hacemos ya que si no tendríamos que hacer *Unload* y *Load* de la solución manualmente.

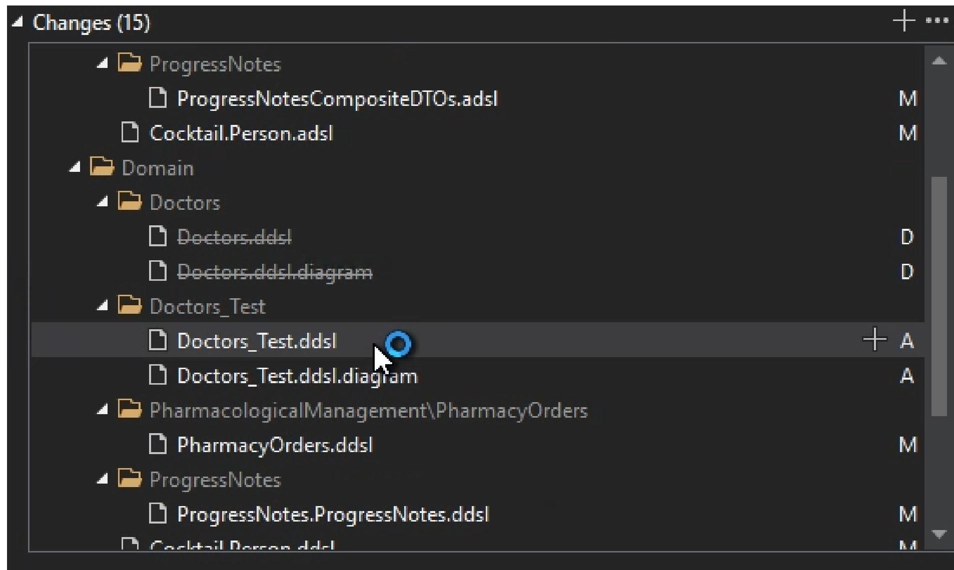


Figura 45. Cambios pendientes al mover un modelo.

Además, en la Figura 46 se ve reflejado cómo también se han actualizado los modelos basados en otras *DSL Tool*. En este caso había referencias al modelo desde modelos basados en la *DSL Tool* de Aplicación y desde modelos basados en la *DSL Tool* de UI.

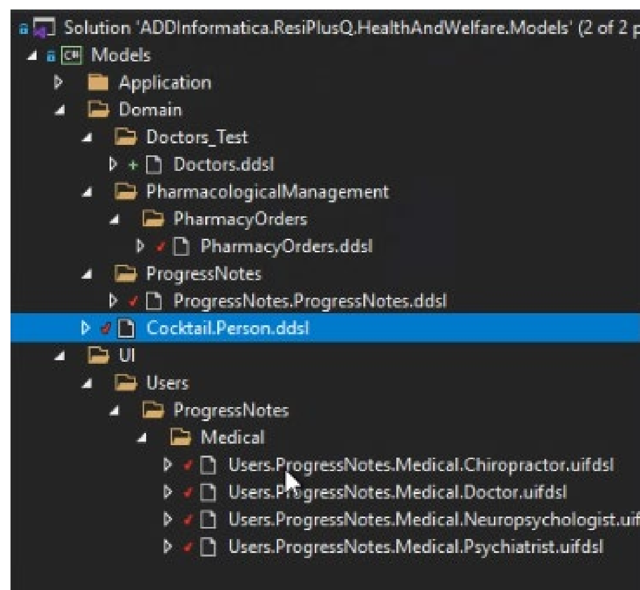


Figura 46. Cambios pendientes en modelos basados en una DSL Tool distinta.

6.6.2. Renombrar un modelo

En este caso seleccionaremos un modelo y una vez hagamos *click* derecho en él, pulsaremos el botón *Rename Model*, como se muestra en la Figura 47.

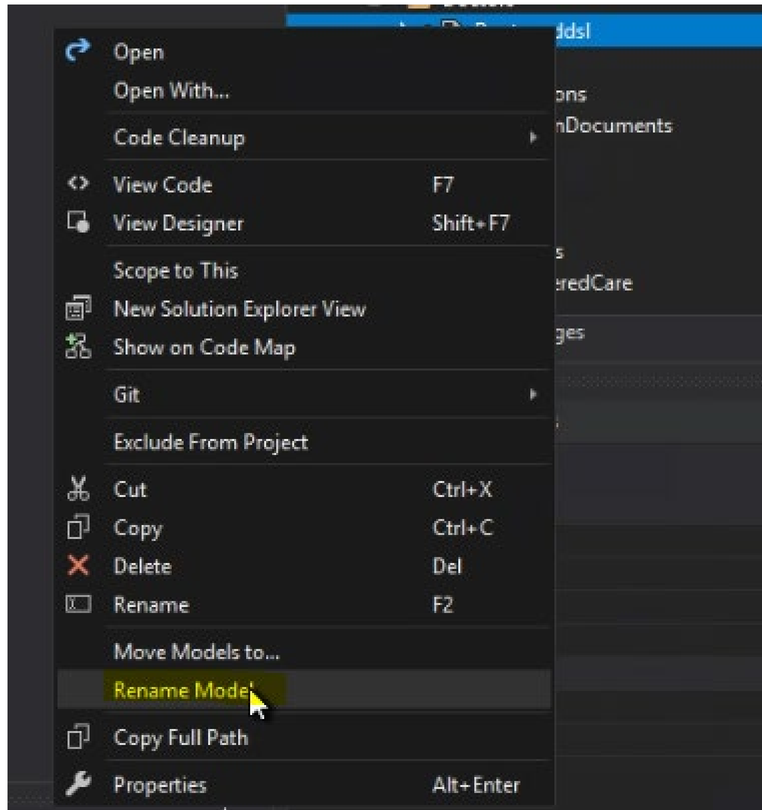


Figura 47. Primer paso para renombrar un modelo.

Entonces se abrirá una ventana donde aparecerá el antiguo nombre del modelo que estamos renombrando y una caja de texto para que indiquemos el nuevo nombre del modelo. En la Figura 48, al presionar el botón *Continue* se desencadenará todo el proceso necesario para cambiar de nombre el modelo.

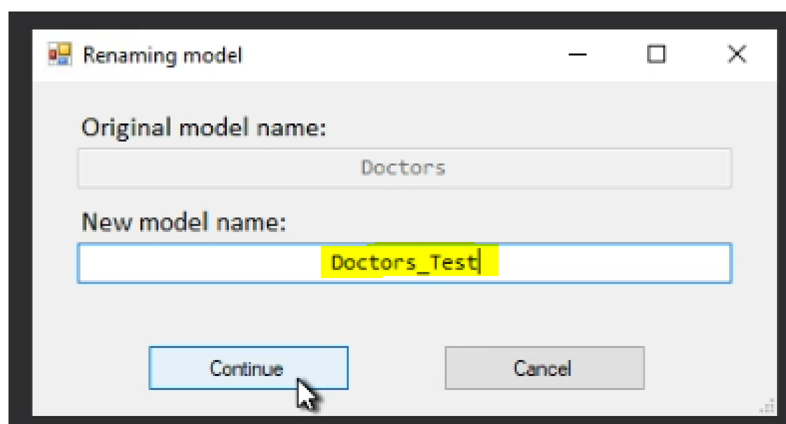


Figura 48. Segundo paso para renombrar un modelo.

CAPÍTULO 7

Conclusiones y trabajo futuro

Para finalizar la memoria, en este capítulo se van a exponer unas últimas ideas sobre el trabajo que se ha realizado; así como algunos de los problemas que nos hemos encontrado, qué nos ha aportado este trabajo y qué hemos aprendido gracias a él. Además, se van a presentar algunas ideas para trabajos futuros.

7.1. Conclusiones

El principal objetivo de este trabajo era poder ayudar a los analistas y a los expertos del dominio a evolucionar los modelos. Se hacía muy complicado modelar, pues no se podían mover ni renombrar los modelos sin que se rompieran y dejaran de funcionar. Pero antes de desarrollar una herramienta que pudiera llevar a cabo esta tarea, primero debíamos conocer el dominio donde ésta se iba a implementar, pues hay que destacar que el autor de este TFG no tenía conocimientos previos sobre lenguajes de dominio específico, ni dominio de la *DSL Tools* de *Microsoft*.

Después de conocer el entorno de trabajo, se pensó cómo se podía dar solución a la problemática principal de la evolución de los modelos. Primero se pensó en acceder al XML de los modelos a través del repositorio en el *Team Foundation Server*, software utilizado para mantener el control de versiones. De esta manera, solo debía abrirse el XML accediendo a través del navegador, modificar las referencias modelbus necesarias y subir esas modificaciones al propio repositorio. La Figura 49 muestra cómo se accedía a un modelo a través del repositorio.

Más tarde nos dimos cuenta de que esta no era la mejor idea, pues se requería una conexión a internet para poder acceder a los modelos, por lo que, en caso de no estar conectados a internet, no se podría trabajar con ellos. Además, esta opción presentaba otro problema y era el hecho de que no podían verificarse los cambios realizados, es decir, no sabríamos si después de las modificaciones realizadas en el XML, estos modelos estarían bien formados y podrían visualizarse correctamente; cosa que solo podría realizarse a través de *Visual Studio*.

Users.ddsl (Latest) - Repos
Not secure | addhqrdrstfs006:8080/tfs/Phoenix/ResiPlusQ/_versionControl?path=%24%2FResiPlusQ%2FHealthAndWelfare%2FModelling%2FMod
Phoenix / ResiPlusQ / Repos / Files / $/ResiPlusQ
$/ResiPlusQ / HealthAndWelfare / Modelling / Models / Domain / **Users.ddsl**
Contents History Compare Annotate Edit Rename Delete Download


```xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <domainModel xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" xmlns:dm1="ht
3   <externalEntityReferences>
4     <domainModelHasExternalEntityReferences Id="b27653b5-5f3a-4862-9b42-b472085c652d">
5       <externalEntityReference Id="a26ee602-e7e6-405e-a303-159836f97d4a" name="Person" description=
6         <subTypes>
7           <generalization Id="631a1d95-d985-4ef8-823e-ea50c9b8e870" reference="modelbus://ADDInform
8             <entityMoniker Id="732170cc-18ea-4c0d-94fc-00b548972e8a" />
9           </generalization>
10        </subTypes>
11      </externalEntityReference>
12    </domainModelHasExternalEntityReferences>
13    <domainModelHasExternalEntityReferences Id="ab75d8f3-d809-4386-9c7c-77010814b57e">
14      <externalEntityReference Id="9a1ed7f7-1e24-4501-8a2c-20bc40f732c1" name="Observation" descrip
15    </domainModelHasExternalEntityReferences>
16  </externalEntityReferences>
17  <entities>
18    <domainModelHasEntities Id="94fe8486-e49c-420e-aa96-3e7a79937327">
19      <entity Id="732170cc-18ea-4c0d-94fc-00b548972e8a" name="User" description="A person attended
20        <targets>
21          <association Id="ab6ec4ab-49b8-4551-8164-c3fad57fb3b1" targetMultiplicity="ZeroMany" targ
22            <externalEntityReferenceMoniker Id="9a1ed7f7-1e24-4501-8a2c-20bc40f732c1" />
23          </association>
24        </targets>
25        <fields>
26          <entityHasFields Id="02a7fd45-2991-4c85-bd41-b203b3408be5">
27            <field Id="06a09e69-67f1-454d-a544-1af111a08068" name="Code" description="The code." di
28          </entityHasFields>
29          <entityHasFields Id="8716bfe6-8359-484d-84e7-2ae8a942cb50">
30            <field Id="e26fd9ac-753b-4a9d-8811-5f2d18d8d8bf" name="PreferredName" description="The
31          </entityHasFields>
32        </fields>
33      </entity>
34    </domainModelHasEntities>
35  </entities>
36 </domainModel>
```


Figura 49. XML de un modelo mostrado desde Team Foundation Server.

Finalmente determinamos que lo mejor era construir una extensión, que se visualizaría como un propio botón en la interfaz de *Visual Studio*. Esto permitiría trabajar con el DTE de *Visual Studio,* una interfaz que sirve para la automatización de los objetos de *Visual Studio* y, por tanto, nos daría un gran abanico de funcionalidades, pues se podrían realizar las modificaciones y que éstas fueran visibles inmediatamente en la solución. Así, sería posible verificar al instante si los cambios realizados en los modelos eran correctos y seguían siendo funcionales tras los cambios.

Por lo tanto, podemos concluir diciendo que los objetivos se alcanzaron completamente, pues lo primordial era dar solución al problema de la evolución de los modelos y ayudar a que los analistas fueran más productivos en las tareas de modelado. Nosotros quedamos satisfechos con el trabajo realizado y la empresa donde se enmarca este trabajo también, ya que actualmente utilizan la extensión a diario en las tareas de modelado. Además, a partir del éxito de esta extensión se están planteando dedicar recursos para que esta funcionalidad se amplie y puedan mover y renombrar entidades sin que los modelos queden inservibles.

7.2. Conclusiones personales

Personalmente, el desarrollo de este trabajo me ha ayudado a analizar los problemas que pueden encontrarse en los procesos internos de una empresa e intentar buscarles una solución. En el desarrollo de un dominio específico son muy importantes los modelos y, en la empresa donde se enmarca este TFG, se estaba lidiando con la evolución y mantenimiento de estos de una forma que no era nada cómoda ni productiva.

Además, he aprendido sobre la terminología de los lenguajes de dominio específico. Antes de la realización de este trabajo no había formado parte de un desarrollo de este tipo, donde el código se genera a partir de los modelos, pues siempre había realizado desarrollos dirigidos por código, donde tenemos una solución que compilamos y generamos los ejecutables o instaladores.

El desarrollo del add-in de *Visual Studio* me ha ayudado a aprender sobre las extensiones de *Visual Studio* con el fin de aprender a interactuar con las diferentes partes del entorno de desarrollo. Y, obviamente, me ha ayudado también en aspectos relacionados con la organización, para poder ser más productivo a la hora de programar y realizar esta memoria.

7.3. Relación con asignaturas cursadas

El presente trabajo tiene relación con muchas de las asignaturas del grado. Hablaremos más en profundidad de solo dos con las que guarda más relación: Proceso de Software y Desarrollo de Software Dirigido por Modelos y Mantenimiento.

La relación con Proceso de Software es clara, pues el objetivo del trabajo era crear una herramienta para mejorar el proceso de desarrollo del dominio. Gracias al add-in desarrollado, los analistas que modelan el sistema, pueden tener una mayor flexibilidad a la hora de mover y renombrar los modelos, ganando en productividad, comodidad y costes.

Con Desarrollo de Software Dirigido por Modelos, la relación también es evidente, y es por el contexto donde se enmarca el trabajo. Se trata de un lenguaje de dominio específico donde, a partir de los modelos desarrollados, se genera el código de la futura aplicación y dónde se ve, como se explicó anteriormente en el punto, la diferencia entre desarrollo dirigido por código y desarrollo dirigido por modelos.

Finalmente, cabe mencionar otras asignaturas del grado con las que también mantiene relación. Introducción a la programación, Programación, Estructuras de datos y Algoritmos e Ingeniería del Software son algunas asignaturas donde se ven conceptos de programación y desarrollo de software que han sido aplicados en el desarrollo de este trabajo.

7.4. Trabajo futuro

Hasta ahora solo se ha automatizado el mover y renombrar modelos, pero no se ha realizado ninguna tarea relacionada con las entidades. Por ello, una propuesta como trabajo futuro sería el de crear dos add-in para poder mover y renombrar entidades.

La idea sería que los nuevos botones fueran visibles cuando se pulsara sobre la entidad en cuestión. Así, como vemos en la figura, al hacer *click* derecho sobre una entidad, nos aparecería la opción de mover o de renombrar entidad.

Además de esto podríamos automatizar la verificación de los cambios. Hasta ahora, una vez hemos movido o renombrado un modelo, manualmente abrimos los modelos modificados para saber si éstos están bien formados o por lo contrario se han quedado en un estado corrupto. Esto es fácilmente automatizable, ya que bastaría con abrir el modelo; si este no lanza ningún error entonces podemos decir que ninguna referencia modelbus se ha roto y, por lo tanto, el modelo está bien formado.

Por último, se podría automatizar el subir el código al TFS. Una vez hemos movido o renombrado un modelo, o hemos movido o renombrado una entidad y hemos verificado que los modelos modificados están bien formados, podemos hacer que el código se suba automáticamente. De la misma forma que la verificación del modelo tras las modificaciones, a través de interacciones con el entorno de *Visual Studio* y el objeto DTE, podemos simular la pulsación del botón *Check-in* que aparece en *Visual Studio*. De esta forma, todo el proceso quedaría automatizado.

Referencias

[1] Desarrollo a medida con generación automática de código. Luis Vilanova. Consultado el día 19/11/2020. URL: https://luisvilanova.es/desarrollo-a-medida-con-generacion-automatica-de-codigo/.

[2] The impact of Low-Code on IT satisfaction. Consultado el día 06/06/2021. URL: https://assets.appian.com/uploads/2019/05/LowCode-and-IT-Satisfaction-Survey-Report.pdf

[3] Model-driven Development (MDD). Consultado el día 22/06/2021. URL: https://searchsoftwarequality.techtarget.com/definition/model-driven-development

[4] Hessa Alfraihi, Kevin Lano, Shekoufeh Kolahdouz Rahimi y Mohammadreza Sharbaf. (2018).

The Impact of Integrating Agile Software Development and Model-Driven Development: A Comparative Case Study: 10th International Conference, SAM 2018, Copenhagen, Denmark, October 15–16, 2018, Proceedings.

[5] What is No-Code Application Development?. Consultado el día 22/06/2021. URL: https://www.bettyblocks.com/no-code-low- code-application-development

[6] What is Low-Code Development?. Consultado el día 22/06/2021. URL: https://www.mendix.com/low-code-guide/

[7] Robert L. Totterdale. (2018). Case study: The utilization of low-code development technology to support research data collection. Consultado el día 23/06/2021. URL: http://www.iacis.org/iis/2018/2_iis_2018_132-139.pdf

[8] Ciclo de vida del software. Consultado el día 22/07/2021. URL: https://ciclodevida.net/del-software

[9] Effort Distribution in Model-Based Development. Werner Heijstek. Consultado el día 22/03/2021. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.4580&rep=rep1&type=pdf

[10] Automated code generation tools can solve problems. Consultado el día 22/03/2021. URL: https://mindmatters.ai/2020/05/automated-code-generation-tools-can-solve-problems/

[11] Mercedes Picón M. 2016. Generación Automática de Código basada en Modelos UML. Consultado el día 22/03/2021.
URL: https://concisa.net.ve/memorias/CoNCISa2016/CoNCISa2016-p134-138.pdf

[12] Chunlin Li y Ben Niu. 2021. User Interface Code Automatic Generation Technology Based on Big Data. Consultado el día 23/03/2021.
URL:
https://www.researchgate.net/publication/346618253_User_Interface_Code_Automatic_Generation_Technology_Based_on_Big_Data

[13] Steven Kelly, Juha-Pekka Tolvanen. Domain-Specific Modeling: Enabling Full Code Generation, páginas 79-85. Wiley-IEEE Press, 2007. ISBN: 9780470249260.

[14] Clarifying concepts: MBE vs MDE vs MDD vs MDA. Consultado el día 10/04/2021.
URL: https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/

[15] Marco Torchiano. (2012). Benefits from modelling and MDD adoption: expectations and achievements. Consultado el día 10/04/2021. URL:
https://dl.acm.org/doi/abs/10.1145/2424563.2424565

[16] Bernhard Schätz, Manfred Broy, Sascha Kirstan, Helmut Krcmar. (2011). What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?. Consultado el día 10/04/2021. URL:
//www.researchgate.net/publication/262203553_What_is_the_Benefit_of_a_Model-Based_Design_of_Embedded_Software_Systems_in_the_Car_Industry

[17] Bran Selic. 2016. Model-Driven Development: Its Essence and Opportunities. Consultado el día 10/04/2021.
URL:
https://www.researchgate.net/publication/221249899_ModelDriven_Development_Its_Essence_and_Opportunities

[18] George Lawton. (2021). Overview of the problems automatic code generation can solve. Consultado el día 11/04/2021
URL: https://searchsoftwarequality.techtarget.com/tip/Overview-of-the-problems-automatic-code-generation-can-solve

[19] B.Hailpern, P.Tarr. Model-Driven development: The good, the bad and the ugly. Consultado el día 10/04/2021. URL:

https://www.researchgate.net/publication/224101620_Modeldriven_development_The_good_the_bad_and_the_ugly

[20] Quadrant for Enterprise Low-Code Application Platforms. Consultado el día 20/06/2021. URL: https://www.gartner.com/en/documents/3991199/magic-quadrant-for-enterprise-low-code-application-platf

[21] The Big Five in Tech bet on modeling and low-code development. Consultado el día 09/07/2021. URL: https://modeling-languages.com/big-five-bet-modeling-low-code/

[22] Google trends. Consultado el día 10/07/2021. URL: https://trends.google.es/trends/explore?date=today%205-y&q=Low-code,Model-driven%20development

[23] Lenguajes de dominio específico. Consultado el día 15/11/2020 URL: https://www.jetbrains.com/es-es/mps/concepts/domain-specific-languages/

[24] Increase in productivity with a Domain-Specific Development. Consultado el día 19/11/2020. URL: https://www.metacase.com/increase.html

[25] João Ricardo, Faria Mendes, Almeida Reis. Improving quality and agility of safety-critical software development using domain-specific languages. Consultado el día 05/12/2020. URL: https://repositorio-aberto.up.pt/bitstream/10216/133760/2/464960.pdf

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

[27] Carlos Enrique Montenegro Marín, Paulo Alonso Gaona-García, Juan Manuel Cueva Lovelle, Oscar Sanjuán. 2011. Aplicación de ingeniería dirigida por modelos (MDA), para la construcción de una herramienta de modelado de dominio específico (DSM) y la creación de módulos en sistemas de gestión de aprendizaje (LMS) independientes de la plataforma. Consultado el día 05/12/2020. URL: https://www.researchgate.net/publication/236366989_Aplicacion_de_ingenieria_dirigida_por_modelos_MDA_para_la_construccion_de_una_herramienta_de_modelado_de_dominio_especifico_DSM_y_la_creacion_de_modulos_en_sistemas_de_gestion_de_aprendizaje_LMS_indep

[28] Agility CMS Model Updater. Consultado el día 22/05/2021. URL: https://marketplace.visualstudio.com/items?itemName=AgilityInc.VS2017-2018-05-17

[29] Tiago Rechau, Nuno Miguel Silva, Miguel Mira da Silva, Pedro Sousa. 2017. A Tool for Managing the Evolution of Enterprise Architecture Meta-model and Models. Consultado el día 06/12/2020. URL: https://www.researchgate.net/publication/319116372_A_Tool_for_Managing_the_Evolution_of_Enterprise_Architecture_Meta-model_and_Models

[30] Artículos de Microsoft consultados:

- DSL Tools de Microsoft. Consultado el día 22/11/2020. URL: https://docs.microsoft.com/es-es/visualstudio/modeling/overview-of-domain-specific-language-tools?view=vs-2019
- Información general sobre XAML. Consultado el día 03/04/2021. URL: https://docs.microsoft.com/es-es/dotnet/desktop/wpf/xaml/?view=netdesktop-5.0
- Información sobre el modelado tabular. Consultado el día 25/05/2021. URL: https://docs.microsoft.com/es-es/analysis-services/tabular-models/tabular-models-ssas?view=asallproducts-allversions
- *Visual Studio* SDK. Consultado el día 02/02/2021. URL: https://docs.microsoft.com/es-es/visualstudio/extensibility/installing-the-visual-studio-sdk?view=vs-2019
- *MenuCommand* Clase. Consultado el día 23/02/2021. URL: https://docs.microsoft.com/eses/dotnet/api/system.componentmodel.design.menucommand?view=net-5.0
- *OleMenuCommand* Clase. Consultado el día 23/02/2021. URL: https://docs.microsoft.com/eses/dotnet/api/microsoft.visualstudio.shell.olemenucommand?view=visualstudiosdk-2019
- Información general sobre las herramientas de los lenguajes específicos de dominio. Consultado el día 01/12/2020. URL: https://docs.microsoft.com/es-es/visualstudio/modeling/overview-of-domain-specific-language-tools?view=vs-2019
- Añadir icono a botón. Consultado el día 10/02/2021 https://docs.microsoft.com/en-us/visualstudio/extensibility/adding-icons-to-menu-commands?view=vs-2019
- Grupos por defecto Visual Studio. Consultado el día 10/02/2021 URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/internals/ide-defined-commands-menus-and-groups?view=vs-2019
- Cambiar comportamiento de un comando. Consultado el día 17/02/2021. URL: https://docs.microsoft.com/es-es/visualstudio/extensibility/command-flag-element?view=vs-2019

- Introducción a la plantilla de proyecto de VSIX. Consultado el día 02/02/2021. URL: https://docs.microsoft.com/es-es/visualstudio/extensibility/getting-started-with-the-vsix-project-template?view=vs-2019
- Extensibilidad de *Visual Studio*. Consultado el día 02/02/2021. URL: https://docs.microsoft.com/es-es/visualstudio/extensibility/?view=vs-2019

Anexos

I. Ejemplo del XML de un modelo

```xml
<?xml version="1.0" encoding="utf-8"?>
<domainModel
xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Cor
e" xmlns:dm1="http://schemas.microsoft.com/dsltools/CommonDSL"
  dslVersion="2021.2.16.999"
  Id="2ebc9cf2-8c50-4d17-bddd-8ad6a6a704e4"
  product="ResiPlusQ"
  namespace="ADDInformatica.ResiPlusQ.HealthAndWelfare"
  application="Health and Welfare"
  secondLanguage="Spanish (Spain)"
xmlns="http://schemas.microsoft.com/dsltools/DomainDSL"
>
  <externalEntityReferences>
    <externalEntityReference
      Id="94d86c6d-99fa-4c37-9700-83248fc00522"
      name="User"
      reference="modelbus://ADDInformatica.Agility.DslTools.Domain.Dom
      ainDSLAdapter/Users/User/..\..\Users.ddsl%2f732170cc-18ea-4c0d-
      94fc-00b548972e8a"
    />
  </externalEntityReferences>
  <entities>
    <entity
      Id="64f2a61b-d3d4-40d9-aaab-a46da5e45e3a"
      name="HealthcareInsurance"
      tODOs=""
      remarks=""
      functionalTODOs=""
      observations=""
      migrationDetail=""
      displayName="Healthcare insurance"
      secondLanguageDisplayName="Cobertura sanitaria"
      description="Contract between the person and the insurance
      company where the person buys a plan and the company agrees to
      cover part of the medical expenses."
      secondLanguageDescription="Contrato entre la persona y la
      compañía de seguros en donde la persona compra un plan y la
      compañía accede a cubrir parte de los gastos médicos."
      pluralDisplayName="Healthcare insurances"
      secondLanguagePluralDisplayName="Coberturas sanitaria"
      collectionDescription="A list of healthcare insurances."
      pluralName="HealthcareInsurances"
      secondLanguageCollectionDescription="Lista de coberturas
      sanitarias."
      defaultData=""
      isCustomizable="true"
      entityType="EntityWithoutTemporalConnotation"
      allowAllPropertiesNullables="false"
      isFindableByImage="false"
      isPlannable="false"
      ignoredByAccessRegistry="false"
      hasDocumentationManagement="false"
      uniqueEntityFieldSet="[]"
```

```
    relatedEntityWithTemporalEvolution=""
  >
    <targets>
      <association
        Id="37c26fad-381b-49e8-b800-702a7553c325"
        targetMultiplicity="ZeroOne"
        sourceMultiplicity="ZeroMany"
        reference=""
        functionalTODOs=""
        observations=""
        tODOs=""
        order="0"
      >
        <entityMoniker
          Id="4355783b-f24d-46fe-844b-7ae9930441bd"
        />
      </association>
      <association
        Id="42011ad1-7ec0-4a0c-b54d-6b533e27f664"
        targetMultiplicity="One"
         sourceMultiplicity="ZeroMany"
         reference="modelbus://ADDInformatica.Agility.DslTools.Domain
         .DomainDSLAdapter/Users/Association/..\..\Users.ddsl%2f41a02
         608-e7bf-404a-b490-3b957f4d1ccb"
        functionalTODOs=""
        observations=""
        tODOs=""
        order="0"
      >
        <externalEntityReferenceMoniker
          Id="94d86c6d-99fa-4c37-9700-83248fc00522"
        />
      </association>
    </targets>
    <fields>
      <field
        Id="6a4e2970-117f-45e3-8d61-3a23cdcf786f"
        tODOs=""
        remarks=""
        functionalTODOs=""
        observations=""
        migrationDetail=""
        name="Comments"
        displayName="Comments"
        secondLanguageDisplayName="Comentarios"
        description="Comments about the healthcare insurance."
          secondLanguageDescription="Comentarios sobre la cobertura
          sanitaria."
        deploymentStatus="Production"
        isRequired="false"
        type="String"
        maxLength="2000"
        minValue="0"
        maxValue="0"
        isCalculated="false"
        isCalculatedOnlyFromEntityInstance="false"
        isDefaultValueCalculated="false"
        order="1"
        isUnique="false"
        decimalCount="0"
        isDefaultOrderField="false"
```

```
        hasIndex="false"
        calculatedPropertyTestCaseCollection=""
        defaultUnitOfMeasurementReference=""
        hasRangeValidation="false"
        ignoredByAccessRegistry="false"
        allowedUnitsOfMeasurementReferences="[]"
        defaultValue=""
      />
      <field
        Id="244d135c-a1b1-4b5f-87ec-970dd88ce5cc"
        tODOs=""
        remarks=""
        functionalTODOs=""
        observations=""
        migrationDetail=""
        name="Number"
        displayName="Number"
        secondLanguageDisplayName="Número"
        description="The identification number of the healthcare
        insurance."
        secondLanguageDescription="El número de identificación del
        seguro sanitario."
        deploymentStatus="Production"
        isRequired="false"
        type="String"
        maxLength="50"
        minValue="0"
        maxValue="0"
        isCalculated="false"
        isCalculatedOnlyFromEntityInstance="false"
        isDefaultValueCalculated="false"
        order="3"
        isUnique="false"
        decimalCount="0"
        isDefaultOrderField="true"
        hasIndex="true"
        calculatedPropertyTestCaseCollection=""
        defaultUnitOfMeasurementReference=""
        hasRangeValidation="false"
        ignoredByAccessRegistry="false"
        allowedUnitsOfMeasurementReferences="[]"
        defaultValue=""
      />
    </fields>
  </entity>
  <entity
    Id="4355783b-f24d-46fe-844b-7ae9930441bd"
    name="InsuranceType"
    tODOs=""
    remarks=""
    functionalTODOs=""
    observations=""
    migrationDetail=""
    displayName="Type of insurance"
    secondLanguageDisplayName="Tipo de cobertura"
    description="The type of insurance contracted by the user. Each
    type of insurance has different characteristics contracted by
    the user that are included in the insurance."
    secondLanguageDescription="El tipo de cobertura contratado por
    el usuario. Cada tipo de cobertura tiene características
```

```
      diferentes contratadas por el usuario que se incluyen en el
      seguro."
      pluralDisplayName="Types of insurance"
      secondLanguagePluralDisplayName="Tipos de cobertura"
      collectionDescription="A list of types of insurance."
      pluralName="InsuranceTypes"
      secondLanguageCollectionDescription="Una lista de tipos de
      coberturas."
      defaultData=""
      isCustomizable="true"
      entityType="MasterEntity"
      allowAllPropertiesNullables="false"
      isFindableByImage="false"
      isPlannable="false"
      ignoredByAccessRegistry="false"
      hasDocumentationManagement="false"
      uniqueEntityFieldSet="[]"
      relatedEntityWithTemporalEvolution=""
    >
      &lt;fields&gt;
        &lt;field
          Id="deb1cffd-d010-477b-b1fd-52df84cdc230"
          tODOs=""
          remarks=""
          functionalTODOs=""
          observations=""
          migrationDetail=""
          name="Name"
          displayName="Type of insurance"
          secondLanguageDisplayName="Tipo de cobertura"
          description="The name of the insurance."
          secondLanguageDescription="El nombre de la cobertura."
          deploymentStatus="Production"
          isRequired="true"
          type="String"
          maxLength="50"
          minValue="0"
          maxValue="0"
          isCalculated="false"
          isCalculatedOnlyFromEntityInstance="false"
          isDefaultValueCalculated="false"
          order="3"
          isUnique="true"
          decimalCount="0"
          isDefaultOrderField="true"
          hasIndex="true"
          calculatedPropertyTestCaseCollection=""
          defaultUnitOfMeasurementReference=""
          hasRangeValidation="false"
          ignoredByAccessRegistry="false"
          allowedUnitsOfMeasurementReferences="[]"
          defaultValue=""
        />
      &lt;/fields&gt;
    &lt;/entity&gt;
  &lt;/entities&gt;
&lt;/domainModel&gt;
```