UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

Facultad de Ciencias Químicas e Ingeniería

Maestría y Doctorado en Ciencias e Ingeniería

Desarrollo de una Herramienta de Acceso y Gestión de Bases de Datos Relacionales para su Interoperabilidad entre Diferentes Plataformas

TESIS

PARA OBTENER EL GRADO DE

MAESTRO EN INGENIERÍA

Presenta:

JOSÉ ROBERTO ARROYO GONZÁLEZ

Bajo la dirección de:

DR. GUILLERMO LICEA SANDOVAL

TIJUANA, BAJA CALIFORNIA, MÉXICO MARZO DEL 2017

A mi madre y a mi padre,
que con su dedicación y sobre todo su ejemplo
me han formado y educado con amor y humildad.

Todos mis triunfos son para ellos.

Universidad Autónoma de Baja California

FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA

COORDINACIÓN DE POSGRADO E INVESTIGACIÓN

FOLIO No. 201

Tijuana, B. C., a 10 de febrero de 2017

C. José Roberto Arroyo González
Pasante de: Maestro en Ingeniería
Presente

El tema de trabajo y/o tesis para su examen profesional, en la Opción TESIS

Es propuesto, por el C. Dr. Guillermo Licea Sandoval

Quien será el responsable de la calidad del trabajo que usted presente, referido al tema: Desarrollo de una Herramienta de Acceso y Gestión de Bases de Datos Relacionales para su Interoperabilidad entre Diferentes Plataformas

el cual deberá usted desarrollar, de acuerdo con el siguiente orden:

I.- INTRODUCCIÓN
II.- MARCO TEORICO
III.- MÉTODOLOGIA
IV.- HERRAMIENTA DE GENERACION Y VERSIONADO DE SERVICIOS
V.- PRUEBAS Y RESULTADOS
VI.- CONCLUSIONES Y TRABAJO FUTURO

UNIVERSIDAD AUTONOMA DE BAJA CALIFORNIA

FACULTAD DE CIENCIAS QUIMICAS E INGENIERIA

Dr. José Luis González Vázquez
Secretario

Dr. Guillermo Licea Sandoval
Director de Tesis

Dr. Luis Enrique Palafox Maestre
Director

Agradecimientos

Agradezco a Dios por darme la fuerza para salir adelante y ponerme siempre en el lugar donde debo de estar.

A la Universidad Autónoma de Baja California (UABC) y la dirección de la Facultad de Ciencias Químicas e Ingeniería por darme la oportunidad de ingresar al Programa de Maestría y Doctorado en Ciencias e Ingeniería (MyDCI).

A mi director de tesis, Dr. Guillermo Licea por su tiempo y orientación en la realización de esta tesis. Por su dedicación como docente durante el tiempo que formé parte del programa MyDCI.

A mis co-directores de tesis, Dr. Manuel Castañón Puga y Dra. Carelia G. Gaxiola Pacheco por su apoyo docente y las oportunidades brindadas durante el tiempo que formé parte de los proyectos de vinculación entre la universidad y la empresa.

A Edaniel Figueroa, por haber confiado en mí como profesionista y brindarme la posibilidad de continuar mis estudios y seguir aprendiendo.

A Yuliana Murillo, por su compañía y apoyo incondicional durante los últimos dos años. Por compartir conmigo momentos de alegría, tristeza y ayudarme a continuar y nunca renunciar.

A todas las personas que ayudaron directa o indirectamente en la realización de este proyecto, gracias.

Roberto Arroyo

Resumen

Este documento presenta una revisión literaria sobre los problemas surgidos de la interoperabilidad de sistemas implementados en plataformas no totalmente compatibles que proveen y subministran información de una base de datos.

Como solución a dicha problemática se documenta la construcción una herramienta capaz de proveer una capa de servicios que manipule estructuras de datos que posibiliten el intercambio de información con plataformas distintas a la que se utilizó para la construcción del sistema.

Se aborda la problemática desde un punto de vista empresarial, y se identifica como parte de una limitante o debilidad operativa para el departamento de desarrollo de software.

Como caso de estudio la herramienta propuesta se integrará al proceso de desarrollo de software llevado a cabo en una empresa de la ciudad de Tijuana, con el fin de observar el impacto que tiene la misma en el desarrollo y mantenimiento de diversos sistemas empresariales.

Se busca que la automatización en la producción de código en arquitecturas basadas en capas provea agilidad a los equipos de desarrollo. De manera que los desarrolladores centran toda su atención en la resolución de problemas lógicos y no en la generación mecanizada de diferentes artefactos de programación, como clases, scripts, etc.

Abstract

This document presents a literary review on problems arising from the interoperability of systems implemented in non-fully compatible platforms that test and subordinate information from a database.

As a solution to this problem, we document the construction of a tool capable of provide a service layer that manipulates data structures that allow the exchange of information with platforms other than the one used for the construction of the system.

The problem is addressed from a business point of view, and is identified as part of a limitation or operational weakness for the software development department.

As a case study, the proposed tool integrates the software development process carried out in a company in the city of Tijuana, in order to observe the impact that it has on the development and maintenance of various business systems.

It seeks to automate the production of code in layer-based architectures to test agility in development teams. This way, the developers focus their attention on solving logical problems and not on the mechanized generation of different programming artifacts, such as classes, scripts, etc.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Las empresas actuales han diversificado el uso que le dan a los sistemas de información. Si bien, su objetivo principal es apoyar los procesos internos proporcionando datos útiles, actualizados y confiables a los miembros de la organización, también han abierto oportunidades estratégicas de negocio al aprovechar la gran capacidad de análisis sobre la información que los mismos sistemas proveen, para tomar decisiones oportunas. [1]

A medida que estos sistemas crecen aumenta su complejidad, por lo que es necesario definir el mecanismo idóneo para que los componentes que modelan la lógica del sistema interactúen de la mejor manera posible con los componentes que almacenan y proveen los datos.

En empresas de desarrollo de software empresarial es sumamente importante definir el proceso mediante el cual será generado dicho mecanismo, a fin de que, se optimice el área de producción de software y se reduzcan los tiempos de re-trabajo manual.

1.1 Antecedentes

Con el surgimiento de las bases de datos relacionales, y debido a su simplicidad para relacionar entidades, se fueron separando los datos de sus procesos asociados en tiempo de ejecución de un programa [1]. Las bases de datos fungían solamente como repositorios donde se depositaban datos de diversos tipos respetando una estructura o esquema y siendo manipulados por un lenguaje propio.

En los años noventa con el fortalecimiento de diversos paradigmas de programación, en específicamente el de la programación orientada a objetos, introducido por Alan Kay en 1970 [2], se rompe esta separación debido a que los fundamentos de este paradigma animan al programador a tratar de igual manera los datos y los procedimientos o funciones, los cuales están contenidos en la representación de una entidad llamada objeto.

Conceptos como el encapsulamiento, la agregación y la herencia proporcionan a los arquitectos de sistemas, herramientas y medios para diseñar modelos que reflejan estructuras y relaciones del mundo real, pero que los obligan a definir las estrategias de mapeo adecuadas si los datos definidos por el modelo tienen que ser persistentes a una base de datos relacionada [3].

A pesar de que la programación orientada a objetos comparte características similares con la manera en que se relacionan las entidades en una base de datos relacional, existen diferencias en la representación y el manejo de la información que imponen ciertas limitaciones que, a su vez, pueden posteriormente generar costos considerables en el desarrollo y mantenimiento de un sistema. [1]

Los posibles problemas ocasionados por este desajuste son bien comprendidos y resueltos por diferentes soluciones actuales, a las que colectivamente nos referimos como herramientas de mapeo objeto-relacional (ORM; del inglés Object-Relational Mapping) [4], las cuales permiten que los datos persistan de forma totalmente automática y transparente a la lógica de la aplicación.

1.2 Planteamiento del Problema

Durante el desarrollo de este proyecto de investigación se analizará una herramienta ORM existente, utilizada en la empresa Punto de Desarrollo de Software S.A. de C.V., la cual crea de manera manual dos capas bien definidas y estructuradas. Una capa de persistencia de datos y una capa de negocios, las cuales se comunican entre sí y mantienen en tiempo de ejecución los datos y objetos que necesita un sistema para funcionar correctamente.

Las limitaciones de esta herramienta que se desean optimizar con el rediseño y creación de una herramienta nueva son que la generación y actualización de las capas

anteriormente mencionadas, de manera que se logre sincronizar automáticamente el esquema de la base de datos, con la estructura de la capa de datos y de negocios. De esta forma objetos utilizados deberán tener automáticamente las propiedades y estructuras necesarias, cuando se ejecute un cambio en el modelo de datos.

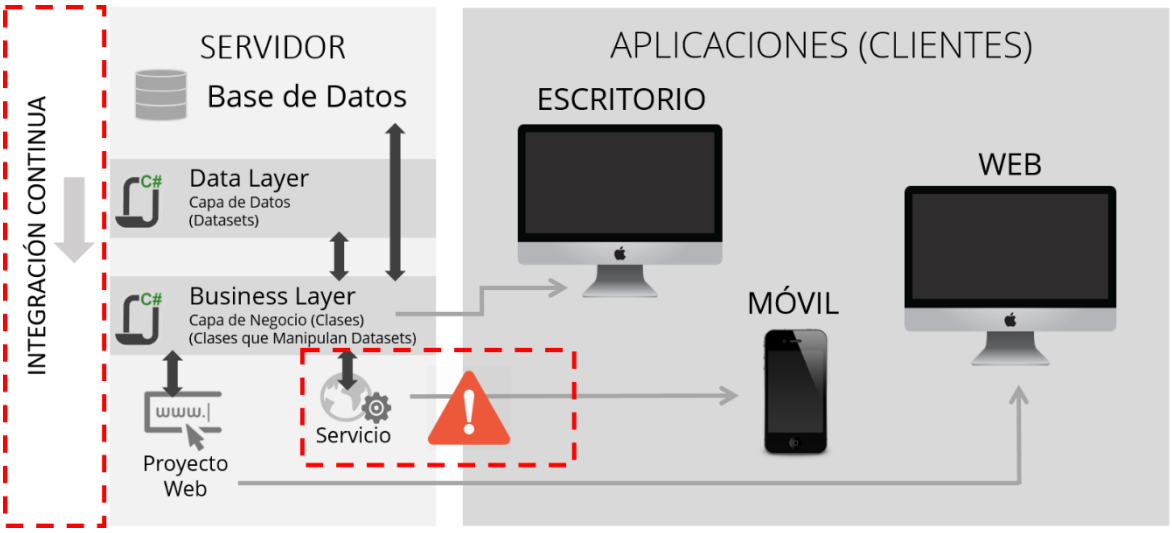

Figura 1.1 Problemática en el Proceso de Desarrollo de la Empresa

En la Figura 1.1 se observa un panorama general del proceso actual de desarrollo de software de la empresa. En el recuadro que se encuentra del lado izquierdo se puede observar los artefactos que forman parte de los proyectos que se encuentran alojados en los servidores locales. Están compuestos de 2 capas (Capa de Datos y Capa de Negocio) que son utilizadas en los proyectos web correspondientes y en los servicios que son consumidos desde aplicaciones en dispositivos con diversos sistemas operativos.

Se detectan dos problemas principales: la integración continua del código y la carencia de compatibilidad de los servicios.

La integración continua del código es una tarea demasiado compleja para el equipo de desarrollo, debido a que las versiones de los cambios realizados que afectan a los servicios se versionan con técnicas muy difíciles de mantener. Es complicado realizar una modificación a los servicios que se encuentran en producción debido a que se afecta directamente a todos los clientes que están consumiendo dicho servicio.

El servicio no mantiene una compatibilidad absoluta con las plataformas que lo consumen, de manera que la integración se realiza mediante consultas utilizando el protocolo http para hacer peticiones POST y GET. Esto provoca que no se mantenga el tipo de dato correspondiente en cada petición, sino que, se tenga que serializar y deserializar a una variable de tipo cadena cada que se requiere proporcionar un parámetro o recibir la respuesta correspondiente del servicio que se está consultado.

La carencia que se debe solucionar, además de las anteriormente mencionadas, es que actualmente los objetos utilizados en la capa de negocios proveen una interface que solo es compatible con lenguajes pertenecientes a la plataforma .NET de Microsoft. Esto provoca que se pierda la compatibilidad aplicaciones desarrolladas en otros lenguajes u otras plataformas, como es el caso de Android, iOS, entre otros.

1.3 Objetivos

1.3.1 Objetivo General

Elaborar una herramienta de acceso y gestión de información que permita la interoperabilidad entre aplicaciones, desarrolladas en diferentes plataformas, que compartan un mismo modelo de datos relacional.

1.3.2 Objetivos Específicos

- Elaborar el diseño arquitectónico y de implementación de una herramienta que automatice la generación de código en aplicaciones basadas en arquitectura en capas.

- Añadir la funcionalidad de sincronización, en caso de que se genere un cambio a nivel esquema de base de datos, para que afecte la capa de datos y de negocios automáticamente.

- Identificar la estructura ideal que debe ser proporcionada por los servicios generados por esta herramienta, para el fácil consumo de aplicaciones desde distintas plataformas.

1.4 Justificación

La razón principal del desarrollo de este proyecto es solucionar el problema de incompatibilidad entre diferentes plataformas que enfrentan muchas empresas en el desarrollo e implementación de diversos sistemas de información, que es muy frecuente y que aún hay un campo amplio de aplicación. Empresarialmente se traduciría en un producto que se presente como una solución eficaz y que pueda ser mejorada con el paso del tiempo.

Dicha solución tendría impacto en la disminución en el tiempo de desarrollo de los implementadores de software. De manera que, si el modelo entidad-relación de la base de datos está debidamente normalizando, estará asegurado que la integración e interacción de entidades en el sistema en conjunto con la lógica de programación, funcionarán correctamente.

Las empresas en la actualidad valoran, además del funcionamiento óptimo del sistema, que la información que es generada o recuperada desde una base de datos, se mantenga integra y no presente inconsistencia de ningún tipo [1].

Identificar una estructura que sea compatible con diferentes plataformas, le da un valor adicional al desarrollo de este proyecto, debido a que cada vez hay más derivados de los lenguajes de primer nivel y el avance tan acelerado en esta cuestión, provoca que los profesionales del área sean tan diversos en el uso de lenguajes para diferentes situaciones que se presentan día a día.

1.5 Alcance

El alcance de este trabajo se limita a la consulta y revisión literaria de los tópicos relacionados con la producción en el desarrollo de software y los problemas de persistencia de datos, al desarrollo de una herramienta que permita la generación de servicios y el control de versiones de los mismos, y la incorporación de dicha herramienta en el entorno de desarrollo que actualmente utiliza la empresa.

La implementación de la herramienta que sea generada como producto del presente trabajo será meramente experimental, con el objetivo de obtener los resultados correspondientes, que se documentarán en el quinto capítulo de este documento.

1.6 Metas

- Desarrollar la herramienta propuesta en el tiempo designado para su elaboración, implementación y pruebas.

- Analizar la posible sustitución de la herramienta que se utiliza actualmente por la herramienta propuesta.

- Analizar la reducción de tiempo de desarrollo y comprobar si la nueva herramienta satisface esta necesidad.

1.7 Organización del Documento

El presente documento consta de cinco capítulos. En el primer capítulo se encuentra la introducción al caso de estudio, los antecedentes y el planteamiento del problema, los objetivos de este trabajo, la delimitación de los alcances del mismo y las metas a cumplir. En el segundo capítulo, se presenta la revisión literaria del estado de las técnicas y metodologías de producción de software y los fundamentos teóricos de la persistencia de datos. En el tercer capítulo, se explica de manera breve la metodología seleccionada para el desarrollo del proyecto. En el cuarto capítulo, se explica la composición y arquitectura de la herramienta desarrollada. En el quinto capítulo, se describen los escenarios de pruebas realizados de acuerdo a las estrategias de versionado de servicios. Por último, en el sexto capítulo se presentan las conclusiones, recomendaciones y oportunidades de trabajo futuro.

Capítulo 2

Marco Teórico

2.1 Metodologías Contemporáneas de Construcción de Software

El crecimiento de la industria del software y la necesidad de reducir los tiempos de desarrollo y de entrega de los proyectos, han hecho que la reutilización de componentes sea una práctica común. En este contexto surgió la idea de aplicar técnicas utilizadas en otras industrias, como la automotriz o la electrónica, en particular las Líneas de Productos.

Una Línea de Productos es, básicamente, un conjunto de productos que comparten funcionalidades que satisfacen ciertas necesidades del mercado y que han sido desarrollados a partir de un conjunto de componentes comunes. El objetivo de aplicar este concepto al desarrollo de software es el de anticiparse al proceso de reutilización llevando esta tarea al extremo, es decir, no solo reutilizar componentes de software sino todo lo referente al proceso de desarrollo de un sistema (análisis, arquitectura, implementación, pruebas, entre otros).

Este trabajo presenta el estado del arte de la construcción de software, centrándose en las Líneas de Productos de Software y las técnicas, procedimientos y paradigmas que derivan de ellas.

2.2 Influencia de Industrias en la Construcción de Software

A través de los últimos años la reutilización en los componentes de software ha ganado importancia en la ingeniería de software. Para los primeros desarrollos los sistemas

de software se consideraban estáticos y cualquier modificación implicaba un tiempo y costo muy elevados [5].

Actualmente los sistemas son dinámicos, es habitual asumir modificaciones en las funcionalidades o nuevos requerimientos lo que implica realizar modificaciones a un sistema existente o incluso la creación de un nuevo sistema a partir de uno ya existente.

Esta idea es la que motivo a empresas acostumbradas al desarrollo de un único producto a pensar en el desarrollo de sistemas mirando a largo plazo y anticipando estos cambios, esto es, planificando la variabilidad futura de los sistemas e investigando que productos pueden ser desarrollados con alguna característica similar entre sí.

Del mismo modo que se requieren buenos procesos de producción para la elaboración de productos tangibles, en el software también se precisan buenos procesos de producción. La idea de reutilizar partes comunes y seleccionar partes variables de un producto es conocida en la ingeniería del software como Líneas de Productos Software (SPL) [5].

2.2.1 Líneas de Productos de Software

Una línea de productos de software (SPL) es un conjunto o familia de productos que comparten un conjunto de features que satisfacen algunas necesidades especiales del mercado y que han sido desarrollado a partir de un conjunto de core-assets con un determinado plan de producción [6].

Esta definición incluye dos conceptos fundamentales para la aplicación de SPL: feature y core-assets.

Una feature es una característica conceptual de un sistema y es usada para describir o distinguir un producto de la línea, algunas features se relacionan con características visibles de los sistemas, otras con la estructura de los sistemas e incluso podrían describir requerimientos no funcionales [7].

Un core-asset es un artefacto de software que es usado en la producción de uno o más productos de una línea de productos de software, puede ser un componente de software, un modelo de procesos, una arquitectura, un documento o cualquier otro resultado de un proceso de construcción de un sistema [7].

El desarrollo de software bajo este paradigma generalmente comprende de dos diferentes procesos relacionados entre sí, conocidos como ingeniería de dominio e ingeniería de aplicación [8]. El nivel de ingeniería de dominio, inicia desde el documento de requisitos que describe una familia de productos relacionados para un segmento específico de mercado. Por lo que, es necesario diseñar una arquitectura y una implementación de referencia para cada familia de productos.

Dicha arquitectura de referencia representará los elementos comunes de todos los productos en la familia y a su vez, debe contener mecanismos que permitan introducir variaciones a los diferentes productos pertenecientes a la familia. En el nivel de ingeniería de aplicación, se comienza desde un documento de requisitos para un producto específico dentro de esa familia.

A continuación, se describen los enfoques actuales aplicables a la construcción y/o desarrollo de software que se basan o utilizan las líneas de productos de software como un mecanismo de producción ágil.

2.2.2 Enfoques Relacionados con Construcción de Software Basados en SLP

2.2.2.1 Desarrollo de Software Dirigido por Modelos

El Desarrollo de Software Dirigido por Modelos (Model-Driven Software Development) es un enfoque de desarrollo de software que trata de mejorar la adaptabilidad de los sistemas ante los cambios, no sólo de los requisitos, sino también los tecnológicos, mediante la definición de modelos a diferentes niveles [9]. Es una aproximación basada en el modelado de sistemas software, la transformación de modelos y la generación final del sistema a partir de ellos.

Sin embargo, al ser sólo una aproximación, no define técnicas a utilizar ni fases del proceso ni ningún tipo de guía metodológica, sólo proporciona una estrategia general a seguir en el desarrollo del software.

En este paradigma los modelos se consideran elementos de primer orden: un modelo se crea con un propósito específico y contiene sólo la información necesaria para llevar a

cabo dicho propósito [10]. Así, distintos modelos contemplan el desarrollo y manipulación de los diversos aspectos del sistema en estudio. La transformación empieza en etapas tempranas y el ingeniero de software completa cada modelo obtenido de un modo más o menos manual. El proceso de transformación de un modelo en otro juega un papel fundamental a lo largo del desarrollo de un sistema bajo este paradigma.

MDSD soporta también distintos niveles de abstracción, relacionados en mayor o menor medida con modelos dependientes de la plataforma de implementación. La gran ventaja de la utilización de esta aproximación es que los modelos no sólo son documentos, sino que realmente son los elementos centrales del desarrollo que finalmente se transforman en el producto final [11].

2.2.2.2 Desarrollo de Software Orientado a Características

El Desarrollo de Software Orientado a Características (Feature-Oriented Software Development) tiene como objetivo encapsular conjuntos coherentes de la funcionalidad de una aplicación informática en módulos bien definidos, independientes y componibles [12].

Dichos módulos reciben el nombre de característica. Una característica se define habitualmente como un incremento, con un propósito común, en la funcionalidad de un sistema.

Una vez descompuesto un sistema en características fácilmente componibles, debería ser posible obtener diferentes versiones del sistema mediante la inclusión/exclusión por composición de determinadas características.

Principalmente, el concepto comprende en implementar características en segmentos de clases, dividiendo el código de éstas en varios fragmentos de clases. Donde cada segmento contiene un incremento de la funcionalidad en comparación con los otros. Los cuales, pueden ser referenciados o combinados entre ellos obteniendo un producto concreto dotado de un conjunto de distintas funcionalidades.

La programación orientada a características aparece estrechamente ligada al concepto de líneas de productos software dado que la primera es un excelente mecanismo para implementar las segundas. Usando programación orientada a características, muchas de las

variabilidades existentes en una línea de productos software pueden implementarse como características.

A la hora de construir un producto específico dentro de una línea de productos software, bastaría con seleccionar las características correspondientes a las variabilidades seleccionadas y componerlas. No obstante, no todas las variabilidades de una línea de producto software se implementan adecuadamente mediante el uso de características. En ciertas ocasiones, como variaciones que suponen incrementos muy pequeños de funcionalidad resultan más adecuadas otras técnicas, como parametrización o la herencia tradicional de la programación orientada a objetos.

La programación orientada a características puede ser vista como una metodología modular de programación que sostiene que la mejor manera de minimizar la redundancia y mejorar la eficiencia, es crear un cierto número de características menores [13]. Este paradigma, se centra en las funcionalidades del sistema, en lugar de los objetos con lo componen (a diferencia de la programación orientada a objeto).

Las funciones deben ser vistas como una herramienta específica que ayudan completar una tarea general.

Apel y Kaestner presentan a FOSD como paradigma integral de desarrollo de software, donde el concepto de feature se utiliza para estructurar el diseño y el código de un sistema de software [14]. Las features son las unidades básicas de la reutilización de este enfoque, y las variantes de un sistema de software varían en las features que proporcionan.

El software se genera en forma eficiente y correcta sobre la base de un conjunto de artefactos de features y de la selección de features que realiza el usuario. Las fases del proceso FOSD pueden resumirse en análisis del dominio, diseño de dominio y especificación, implementación del dominio, configuración y generación del producto.

El software se descompone en términos de las features que provee, para construir software bien estructurado que pueda adaptarse a las necesidades del usuario y al escenario de aplicación. A partir de un conjunto de features comunes se puede generar una variedad de sistemas de software diferentes que comparten features comunes y difieren en otras, de manera similar al enfoque SPL [15].

2.2.2.3 Desarrollo de Software Orientado a Aspectos

El Desarrollo de Software Orientado a Aspectos (Aspect-Oriented Software Development) es un paradigma reciente que define una moderna forma de trabajo que modulariza en entidades independientes las diversas funcionalidades que conforman una aplicación, solucionando dificultades que a la tradicional orientación a objetos le son muy difíciles de resolver [16]. De hecho, son precisamente esos problemas complementarios de código disperso (scattered code o scattering) y código enmarañado (tangled code o tangling) los que, según Jacobson, impiden que las implementaciones resulten más amables y comprensibles [17].

De acuerdo a Pressman [18], el desarrollo de software orientado a aspectos (DSOA), al ser un nuevo paradigma de programación formal, requiere prestar atención a todas y cada una de las etapas del ciclo de desarrollo; a saber: especificación y análisis de requerimientos, diseño, implementación y mantención del software.

2.2.2.4 Desarrollo de Software de Línea de Productos

Ingeniería de Línea de Producto o Desarrollo de Software de Línea de Productos (Software Product Line Engineering) es un enfoque que aplica una forma especializada de reutilización de software mediante el empleo planificado de artefactos básicos de software (core assets) dentro del ámbito de un conjunto de productos relacionados, dichos artefactos de software básicos se utilizan en la producción de un producto en una línea de productos de software [19]. Es decir, se crea una colección de sistemas de software que son similares a partir de un conjunto de activos de software usando un mismo medio de producción.

El proceso de ingeniería se divide en dos equipos de trabajo [20]. El primer equipo se encarga de la Ingeniería de Dominio ó core asset development [6]. Este equipo es responsable de desarrollar los elementos comunes al dominio: estudiar el dominio, definir su alcance (requisitos) dentro del mercado objetivo de la SPL, definir las features, implementar los core assets reutilizables y su mecanismo de variabilidad, y establecer cómo es el plan de producción. Se recogen iterativamente los requisitos comunes para toda la SPL. Estos requisitos se expresan típicamente en forma de features (que ayudan no sólo a que los clientes

puedan distinguir unos productos de otros, sino que dirigen asimismo el desarrollo del código que implemente también esas características).

El segundo equipo se encarga de la Ingeniería de Producto ó product development [6]. Sus cometidos incluyen desarrollar los productos para clientes concretos, a partir de los recursos basados en requisitos concretos de clientes. Para ello, este segundo equipo utiliza los recursos creados por el equipo anterior. Utiliza la infraestructura creada por el equipo de dominio para construir productos concretos para clientes que lo demanden. Durante el desarrollo del producto, las principales tareas son las de configuración y las de composición. Las de configuración instancian los elementos comunes, resolviendo los puntos de variación. Por otro lado, la composición implica la escritura de código alrededor de estos elementos comunes para facilitar su interconexión.

Esta división formal entre dominio y producto es complementada por algunos autores [6] [21] con una parte dedicada a la gestión encargada de asignar, coordinar y supervisar los recursos. Esta gestión se realiza tanto en el nivel técnico (gestión de proyectos concretos) como en el nivel organizativo (estructura organizativa adecuada a los objetivos propuestos). Los artefactos de gestión creados como planificaciones también forman parte de los elementos comunes (core assets).

Para finalizar el primer bloque de la presente revisión literaria se muestra una tabla con información relevante al modo de ejecución de cada uno de los paradigmas relacionados con SPL (Véase Cuadro 2.1).

Cuadro 2.1: Paradigmas Contemporáneos de Construcción, Adaptación y Análisis de Sistemas de Software Relacionados con SPL

Siglas	Nombre	Abstracción	Requisitos	Etapa de Ejecución
MDSD	Model-Driven Software Development	Modelado	Funcionales	Diseño
FOSD	Feature-Oriented Software Development	Modelado	Funcionales y No Funcionales	Análisis, Diseño
AOSD	Aspect-Oriented Software Development	Modelado, Aplicación	Funcionales y No Funcionales	Implementación
SPLE	Software Product Line Engineering	Modelado	Funcionales y No Funcionales	Análisis, Diseño

A pesar de que el concepto de las Líneas de Productos de Software fue definido a principios de la década anterior, el trabajo que se realiza actualmente es basto, debido a su relevancia en la disminución de costos y tiempos.

Una vez abordado el tema de la producción de software, se presenta a continuación la revisión de la literatura relacionada con la persistencia de información en los sistemas de software que utilizan bases de datos relacionales.

Además, se describe el comportamiento de la arquitectura de software basado en capas y la manera en que se realiza el mapeo objeto relacional.

2.3 Persistencia de Objetos

Podemos encontrar diferentes definiciones del término persistencia, según distintos puntos de vista y autores. Veamos dos que, con más claridad y sencillez, concretan el concepto de persistencia de objetos.

La primera definición dice así: “La persistencia de objetos significa que los objetos individuales pueden sobrevivir al proceso de la aplicación; pueden ser guardados a un almacén de datos y ser reconstruidos más tarde.” [22].

En definitiva, la persistencia de objetos es la capacidad que tienen los objetos de sobrevivir al proceso que los creó; permitiendo al programador almacenar, transferir, y recuperar el estado de los objetos.

2.4 Modelo Relacional de Base de Datos

En definitiva, la persistencia de objetos es la capacidad que tienen los objetos de sobrevivir al proceso que los creó; permitiendo al programador almacenar, transferir, y recuperar el estado de los objetos.

Una descripción muy sencilla y directa del modelo relacional, es aquella en donde se define que la responsabilidad de las bases de datos relacionales es modelar la información basándose en relaciones definidas en conjuntos finitos de valores llamados dominios.

Una relación es un conjunto de listas ordenadas de valores llamadas tuplas. Cada tupla es un elemento del producto cartesiano de dominios. Cada ocurrencia de un dominio en la

definición de una relación se denomina atributo, y el mismo dominio puede llegar a repetirse dentro de ella. Los dominios y las relaciones son implementados como tablas con m filas y n columnas. Cada fila corresponde a una tupla y cada columna a un atributo relacional. Por eso es que a lo largo de esta tesis se usará el concepto de tabla, columna y fila en lugar de relación, atributo y tupla.

Cada columna de una tabla tiene un tipo y un nombre para poder ser referenciada sin importar su posición relativa. El tipo de una columna se limita a un conjunto pequeño de tipos predefinidos como integer, varchar o date. Las tablas se definen usualmente con restricciones impuestas a sus filas para evitar la duplicación de datos o dependencias cruzadas. Una restricción sobre una tabla es requerir que cada fila sea identificable unívocamente a partir de un subconjunto de sus columnas. A este subconjunto se lo denomina llave primaria. Cuando esta llave está compuesta por una única columna se la denomina llave simple sino en caso de contar con más de una columna se dice que la llave es compuesta.

El acceso a los datos se realiza a través de un conjunto básico de operaciones: selección, proyección, producto, join, unión, intersección y diferencia. Estas operaciones se expresan a través de un lenguaje denominado SQL (Lenguaje de Consultas Estructurado) usado para guardar, recibir y modificar información en la base de datos.

Para recibir información se debe realizar una selección de las filas, la cual permite especificar ciertas condiciones para los atributos acompañada generalmente con una proyección que significa que sólo un subconjunto de las columnas es seleccionado. Para consultas más complejas se utiliza el operador JOIN que crea una tabla temporal con conjuntos de columnas pertenecientes a dos o más tablas. El resultado de una selección es generalmente un conjunto de filas o RecordSet. Para recorrer este conjunto los lenguajes de programación brindan un cursor y operaciones de navegación encapsuladas en librerías.

De las tres formas de persistencia, sólo las bases de datos relacionales han demostrado ser escalables, robustas y lo suficientemente estándares para las aplicaciones empresariales. No obstante, cuando se quiere persistir los objetos utilizando una de ellas, se puede observar que hay un problema de compatibilidad entre el paradigma de la Orientación a Objetos y el modelo relacional, la también llamada diferencia o impedancia objeto-relacional.

2.5 Impedancia Objeto - Relacional

La Impedancia Objeto-Relacional se define como un conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito bajo el paradigma de la Orientación a Objetos [23].

Un ejemplo claro de esta impedancia se observa en el hecho que en el mundo de la programación orientada a objetos, se tiene un claro sentido de la pertenencia, a cada objeto le pertenecen sus correspondientes atributos; por ejemplo para el objeto Agenda Telefónica podríamos especificar como atributos a una colección de objetos llamados "persona", en la que a cada persona le corresponde su correspondiente atributo "teléfono", al transformar esto hacia el mundo relacional se ocuparía más de una tabla para almacenar la información, este simple hecho, hace notar que las tablas del modelo relacional son inconscientes de cómo están relacionadas con otras tablas a un nivel fundamental, puesto que aun cuando posean constraints para definir sus relaciones, para reconstruir el objeto originalmente persistido se debe construir un query, y dicho query debe especificar explícitamente como se relacionan las tablas entre sí, con esto se demuestra además que el lenguaje SQL a pesar de los constraints se mantiene inconsciente de las relaciones que a nivel de objeto poseen las tablas entre ellas.

Así como lo anteriormente expuesto se pueden enumerar distintos problemas que surgen entre los dos modelos:

1. **Reglas de Acceso:** En el modelo relacional los atributos pueden ser accedidos y/o modificados a través de operadores relacionales predefinidos, mientras que, en el modelo orientado a objetos, se permite que cada clase defina la forma en que serán alterados los atributos, así como la interface que ocupará para ello.

2. **Ataduras del Esquema:** Los objetos del modelo de la programación orientada a objetos, no deben seguir ningún esquema en cuanto a que atributos deben o pueden tener, puesto que son definidos por el programador, mientras que las tablas deben seguir el esquema entidad-relación.

3. **Identificador único:** Las llaves primarias de una fila tienen generalmente una forma de poder representarse como texto visible, mientras que los objetos no requieren un identificador único externamente visible.

4. **Estructura vs Comportamiento:** La orientación a objetos se concentra primordialmente en asegurar que la estructura del programa sea razonable (entendible, extensible, reusable, segura), mientras que los sistemas relacionales ponen el énfasis en tipo de comportamiento que el sistema tendrá una vez en producción (eficiencia, adaptabilidad, rapidez, etc.). Los métodos de la programación orientada a objetos asumen que el principal usuario del código orientado a objetos y sus beneficios es el desarrollador de aplicaciones, mientras que el modelo relacional enfatiza que la forma en que los usuarios finales perciben el comportamiento del sistema es mucho más importante.

De todo esto surge la necesidad de utilizar algún mecanismo para integrar la información contenida en nuestros objetos con los datos almacenados en la base de datos relacional. Esto típicamente se logra a través de una capa de traducción objeto – relacional.

2.6 Arquitectura basada en capas

Para organizar una aplicación empresarial, la industria del software ha convergido en una Arquitectura basada en Capas, dividiendo el sistema en tres capas básicas: la capa de presentación, la capa de lógica de dominio y la capa de persistencia [24].

El principio detrás de esta arquitectura es que cada capa dependa sólo de los elementos contenidos en ella o en las capas situadas por debajo, teniendo responsabilidades bien definidas y evitando cualquier tipo de acoplamiento con las capas superiores.

2.6.1 Capa de Presentación (Vista)

Maneja la interacción entre el usuario y la aplicación. A veces el nombre presentación presta a pensar en sólo salida de la aplicación, pero en realidad maneja la interacción en ambas direcciones. En algunas ocasiones el usuario puede ser otro sistema comunicándose por ejemplo a través de un servicio remoto.

Esta comunicación se hace especialmente notoria en ambientes Web, donde la capa de presentación no sólo tiene que crear documentos entendibles por los usuarios sino manejar los mensajes enviados por el browser como consultas o datos de formularios.

Para esto se suele utilizar un esquema de Controladores y Vistas, donde los controladores son el eje entre las capas inferiores y las vistas, el patrón de diseño Modelo Vista Controlador es un ejemplo popular de esta forma de estructurar la aplicación. Algunos autores dividen los controladores en una capa separada llamada Capa de Aplicación.

2.6.2 Capa de Lógica de Dominio (Capa de Negocios)

Representa conceptos de negocio como reglas o estados. Lo que distingue a esta lógica por sobre el resto de la aplicación es que mantiene los conceptos centrales del proceso, siendo muchas veces la ventaja competitiva por sobre el resto de los productos. Es por ello que, a pesar de que por lo general sólo constituye un conjunto de módulos pequeños comparado el resto de las capas, suele tener mayores requerimientos como pruebas automáticas o revisiones. Esta importancia se realza en las aplicaciones empresariales, la lógica irregular y propensa a cambios requiere un tratamiento especial.

2.6.3 Capa de Persistencia (Capa de Datos)

Esta capa brinda servicios para sincronizar la capa de lógica con un medio de almacenamiento. Para esto se deben identificar los objetos en memoria que deben sobrevivir a la ejecución del programa teniendo así una persistencia de largo plazo.

Dependiendo del tipo de aplicación existen distintos requisitos para esta capa. Por ejemplo, en una aplicación de diseño CAD generalmente cuando se edita un documento se trae un gran conjunto de información a memoria desde el medio de almacenamiento ya que traer subconjuntos de información haría que la aplicación tenga un tiempo de respuesta pobre. En cambio, en una aplicación empresarial generalmente las operaciones se concentran sólo en un grafo limitado de objetos, por lo que traer toda la información disponible es un gasto inaceptable. Para esto debe existir un especial interés en la optimización entre la cantidad de información que se trae a memoria y la realmente utilizada.

2.7 Mapeo Objeto / Relacional (ORM)

El mapeo Objeto/Relacional es “la persistencia automatizada y transparente de las tablas en una Base de Datos relacional, usando metadatos que definen el mapeo entre los objetos y la Base de Datos” [22].

En esencia, ORM transforma datos de una representación en otra.

Por tanto, ORM es una técnica que se utiliza para poder ligar las bases de datos y los conceptos de orientación a objetos creando “bases de datos virtuales”, es decir la aplicación desde dentro utiliza frameworks, los mismos que son intermediarios entre la base de datos relacional y la aplicación totalmente orientada a objetos. Pueden definirse y clasificarse las técnicas a utilizar por parte de la ORM. Existen 4 posibles estrategias que inclusive pueden combinarse según sea el caso: la reconciliación, el uso de patrones, el mapeo y la transformación [4].

Se considera ORM a cualquier capa de persistencia que proporcione autogeneración de SQL a través de metadatos (generalmente XML). No se considera ORM a una capa de persistencia creada por el propio desarrollador.

En el siguiente capítulo se describe la metodología seleccionada para el desarrollo de la herramienta ORM que permita la generación de la capa de datos, negocios y servicios. Se realiza un breve desglose de las etapas que comprende dicha metodología.

Capítulo 3

Metodología

3.1 Descripción de la Metodología

Debido al enfoque práctico de este proyecto la metodología que se eligió para abordar la investigación, en base a su grado de abstracción, es de tipo aplicada. El caso de estudio relacionado demanda que se elabore una solución práctica, tangible y comprobable a un problema suscitado en el mundo real.

3.1.1 Etapa 1: Estudio del Estado del Arte (Revisión Bibliográfica)

Durante esta etapa se realizó la búsqueda y revisión de literatura que sustenta la propuesta de solución. Se lograron obtener contenidos útiles que se presentan en la actualidad y tienen presencia en el contexto del desarrollo de este proyecto.

3.1.2 Etapa 2: Identificación y Definición del Problema

Se llevó a cabo un análisis estratégico del proceso que se realizaba en la empresa para producir software, con el fin de identificar carencias y/o dificultades. Se realizó el prototipo de una propuesta de solución que permitió validar estas necesidades con cada uno de los involucrados en el proceso.

3.1.3 Etapa 3: Desarrollo

Para la elaboración de dicha solución, se adoptó el Método de Desarrollo de Sistema Dinámico (DSDM; del inglés Dynamic Systems Development Method) [25], el cual define el marco de desarrollo en procesos de producción de software, y que encaja perfectamente

debido a que los usuarios, en este caso empleados de la empresa donde se realizó la propuesta de implementación de la herramienta de software, y el equipo de desarrollo, quien desarrolla el presente proyecto de investigación, trabajaron en conjunto en todas las etapas del proceso.

DSDM es el primer antecedente de las metodologías de programación ágil. Es un proceso de tipo iterativo e incremental que se subdivide en 5 fases: Estudio de la viabilidad, Estudio del negocio, Modelado Funcional, Diseño y Construcción, y por último la fase de Implementación. Las ultimas 3 fases son meramente iterativas, y en todas y cada una de ellas, existe la retroalimentación [25].

Al margen de este método, se describen los pasos que se siguieron para lograr los objetivos de la investigación:

1. **Estudio de la Viabilidad:** Se realizaron los pasos definidos en la etapa 1 y 2 de la metodología de investigación de este proyecto. Se determinó si el desarrollo de la herramienta propuesta era factible para ser implementada.

2. **Estudio del Negocio:** Se estudió el proceso que se realizaba en la empresa, previo a la elaboración del proyecto de investigación. Se obtuvieron aquellos factores que propiciaban errores o que no habían sido optimizados para su funcionamiento óptimo.

3. **Modelado Funcional (Iterativa):** Se analizó la composición interna del módulo o componente que forma parte de la herramienta inicial, y se determinó el ambiente donde es óptima su ejecución.

4. **Diseño y Construcción (Iterativa):** Se construyó un bloque del proyecto inicial, realizando evaluaciones constantes con los usuarios finales.

5. **Implementación (Iterativa):** El segmento de la herramienta propuesta se implementó en un entorno controlado y se validó en un ciclo de pruebas, con datos muy similares a los que se usan en proyectos reales. Se realizaron retroalimentaciones y se definieron avances para las iteraciones subsecuentes.

3.1.4 Etapa 4: Pruebas de Funcionamiento y Confiabilidad de los Datos

Se realizaron evaluaciones para determinar si el producto final propuesto, satisfacía la necesidad descrita en el planteamiento del problema y si además cumplía con lo estipulado en los objetivos del proyecto.

3.1.5 Etapa 5: Realización de Experimentos

Se realizaron pruebas de la herramienta utilizando diferentes proyectos, con estructuras diferentes. Los escenarios posibles de pruebas estuvieron basados en estrategias de control de versiones, las cuales están definidas por la cantidad y tipo de cambios que sufre una entidad después de que un servicio es liberado.

Las estrategias de control de versiones que se tomaron en cuenta fueron: estrategia de control de versiones ágil, estrategia de control de versiones semi-estricta, estrategia de control de versiones estricta.

3.1.6 Etapa 6: Ajustes

Una vez terminada la versión inicial, se realizaron los ajustes que estuvieron al alcance del desarrollo del proyecto que se describe en este documento. Esta es una etapa de estabilización de la herramienta, que permite reconsiderar aquellos factores que no fueron tomados en cuenta en el inicio del desarrollo pero que se presentaron de manera importante al realizar las pruebas anteriormente mencionadas.

A continuación, se describe el funcionamiento y composición de la herramienta de generación de capa de datos, negocios y servicios, así como su integración con el entorno de desarrollo Visual Studio.

Capítulo 4

Herramienta de Generación y Versionado de Servicios

El desarrollo de la aplicación se realizó en dos fases en base a las necesidades de funcionalidad y portabilidad de la misma. Primeramente, se definió la estructura de la aplicación de escritorio que define la generación de las capas de datos, negocios y servicios. Posteriormente, esta misma aplicación fue incrustada en un componente de integración al entorno de desarrollo de Visual Studio.

4.1 Arquitectura de la solución

La aplicación está compuesta por un conjunto de formas (vistas) que definen y solicitan al usuario final información relevante para determinar cuál es el origen de la información que se desea generar, es decir, en donde se encuentra la base de datos relacional que se desea tomar como base para la generación de código y cuál es la configuración de esa misma generación de código en base a parámetros que son del completo dominio del usuario final.

La configuración de las cadenas de conexión es una de las partes fundamentales de la aplicación, debido a que sin ellas no es posible realizar cada uno de los procesos subsecuentes de la aplicación. A diferencia de los parámetros de las capas de datos, negocio y servicio, que no tienen una dependencia para la generación de código, excepto la capa de servicios que si necesita que el usuario le indique en donde se encuentra la capa de negocio que tomará como base para definir la implementación de los servicios.

En la figura 4.1 que se encuentra a continuación podemos observar el diagrama general de la solución propuesta en el cual el orden de las capas se expresa de abajo hacia arriba e indica el flujo de la información desde la base de datos hasta los clientes finales.

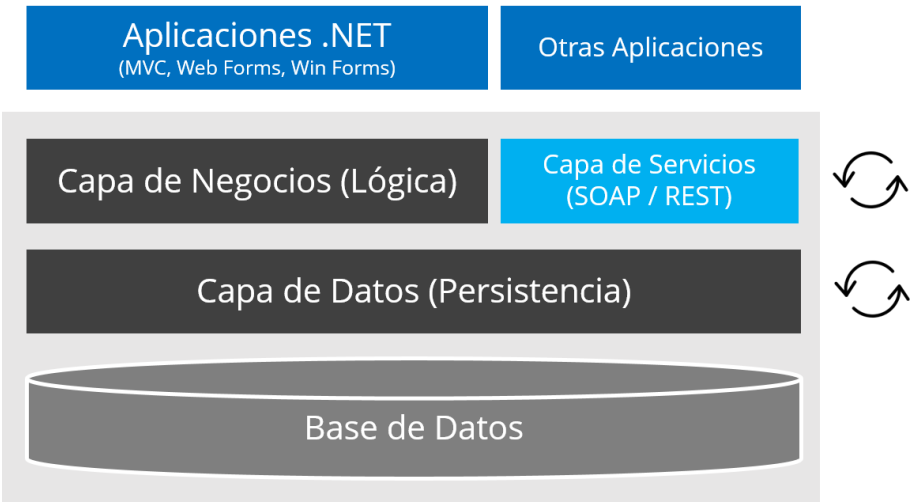

Figura 4.1 Diagrama de Arquitectura de Solución Propuesta

La parte sombreada en color gris corresponde a la aplicación de generación de código y los bloques de color azul fuerte que se encuentra en la parte de arriba representan a las aplicaciones que directa o indirectamente pueden utilizar las capas generadas.

El diagrama está separado verticalmente en dos columnas las cuales nos indican que aplicaciones del entorno .NET de Microsoft son totalmente compatibles con la capa de datos y negocios. Del lado derecho se observa en el recuadro azul la propuesta de generación de la capa de servicios para que otras aplicaciones que no pertenecen al entorno anteriormente mencionado se puedan integrar de la manera más sencilla posible.

El uso de protocolos estandarizados permite que plataformas basadas en diferentes lenguajes de programación puedan consumir sin ningún problema los servicios y garantizar uno de los objetivos de este trabajo, que es precisamente lograr la interoperabilidad entre las mismas.

4.1.1 Conexión a Base de Datos

Para iniciar el proceso de generación de la capa de datos y de negocios, como ya se explicó anteriormente, partimos de la identificación de la base de datos de la cual queremos obtener información. Si el usuario no tiene cadenas de conexión registradas que permitan el acceso a las bases de datos, la aplicación le solicita que lo haga al menos la primera vez.

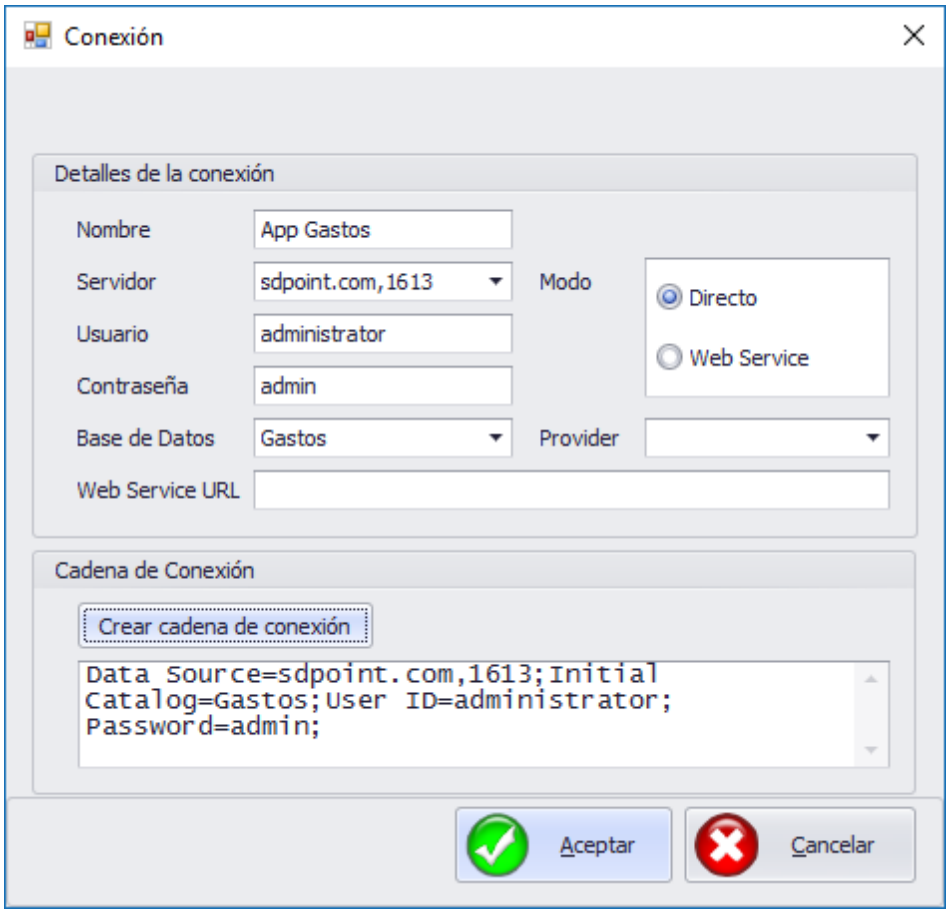

Figura 4.2 Conexión a Base de Datos

Una vez que ya se proporcionaron los datos de acceso la aplicación guarda esas credenciales en un archivo cifrado con extensión .dat en la carpeta Mis Documentos del usuario que esté usando la aplicación para que pueda seguir utilizando dichas credenciales en el futuro sin necesidad de volver a capturar toda la información nuevamente. Este archivo solamente puede ser consultado desde la aplicación debido a que está firmado con certificados de acceso únicos.

4.1.2 Generador de Capa de Datos y Capa de Modelo de Negocio

La capa de datos y la capa de negocios representan la parte medular de la lógica y manejo de información de la aplicación. El usuario puede o no generar estas capas dependiendo de si existen o no versiones previas de estas capas.

Para que el usuario visualice las tablas y las vistas de las que desea generar las capas debe proporcionar las credenciales de acceso a la base de datos como se explicaba anteriormente.

La figura 4.3 ilustra el funcionamiento de la aplicación y la ubicación de los procesos de generación de capa de datos y de generación de capa de negocios. Si bien, los procesos de conexión con base de datos y generación de clases son independientes entre sí, para que la aplicación funcione adecuadamente se necesita una combinación de ambos.

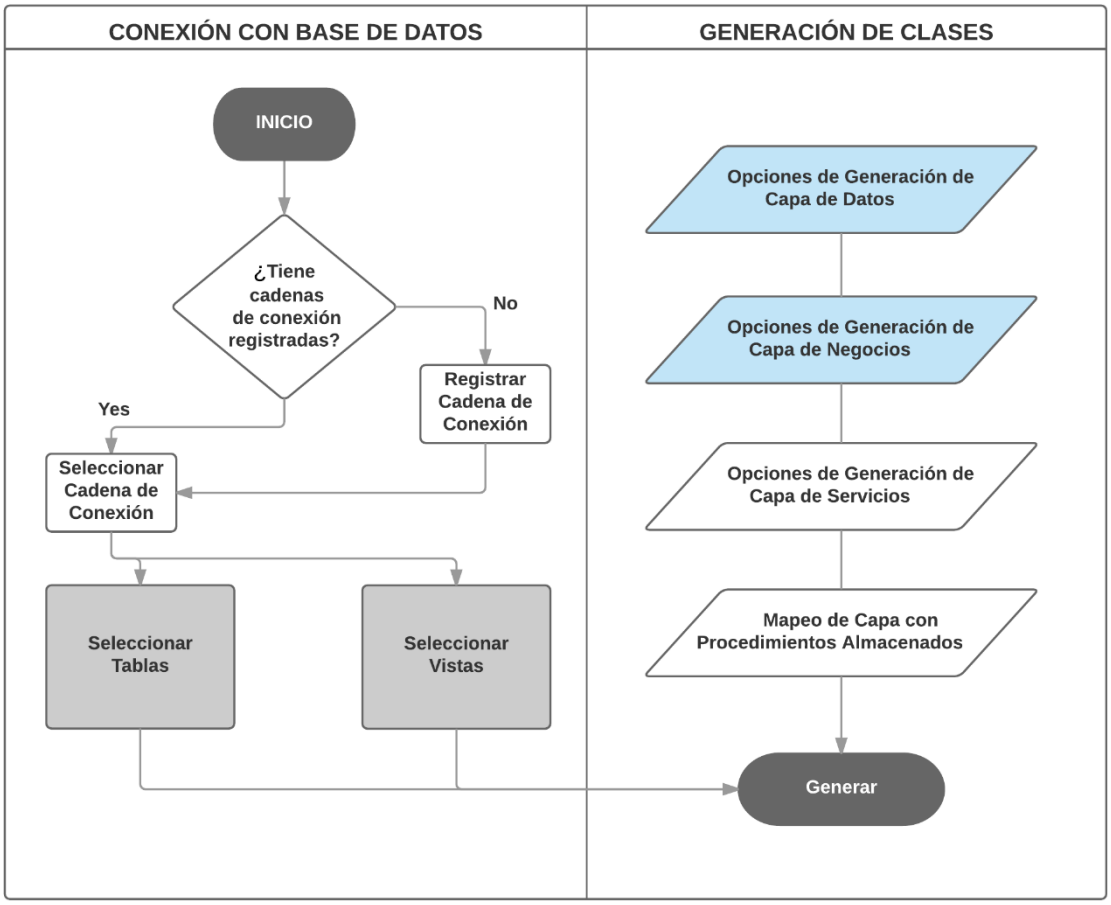

Figura 4.3 Proceso General de Generación de Clases

Después de que el usuario proporciona sus datos de acceso para registrar una cadena de conexión, el usuario la mostrará para que sea seleccionada en el apartado de conexiones disponibles. Al seleccionarla la aplicación muestra al usuario un listado de las tablas y vistas que encontró en dicha base de datos.

El usuario debe realizar una selección de las tablas o vistas que desee considerar para la generación de clases de la conexión a base de datos que tiene activa. De igual manera, pudiera seleccionar todas las tablas y vistas en una sola acción.

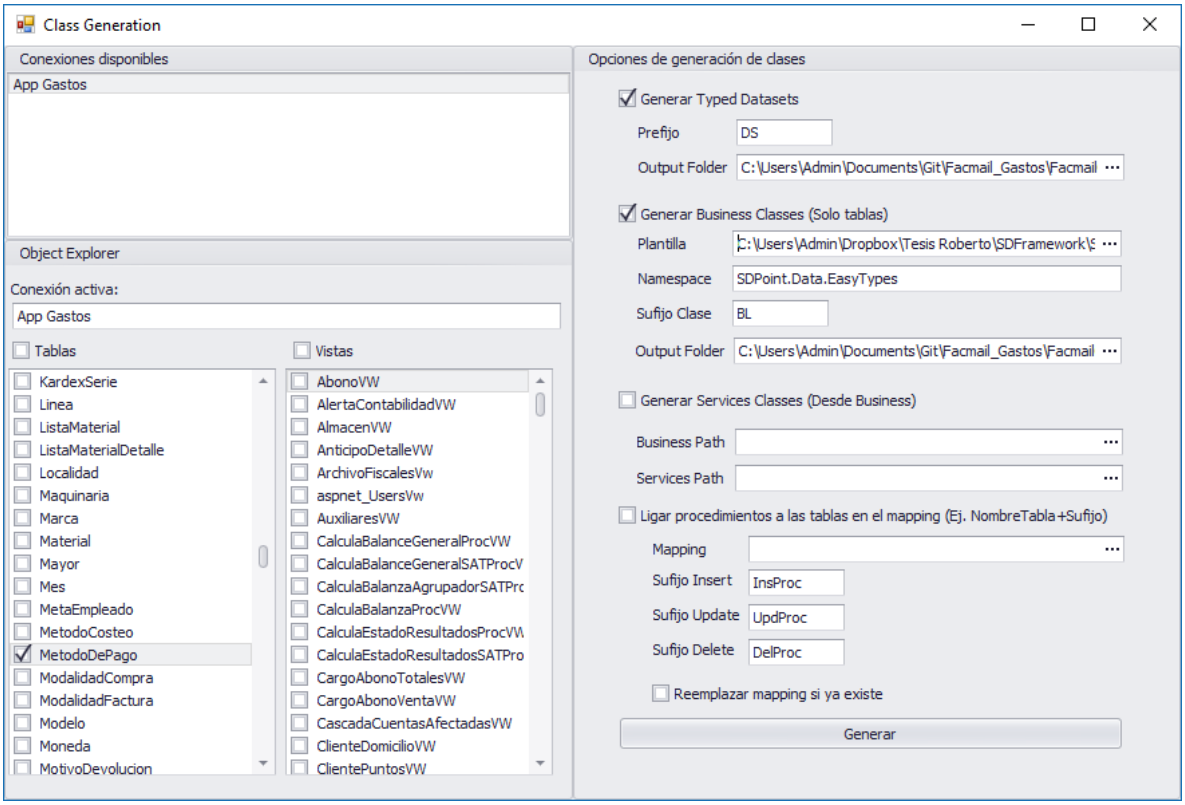

Figura 4.4 Interfaz del Generador de Clases

Como se muestra en la figura anterior, el usuario puede seleccionar que acciones debe ejecutar la aplicación en base a las tablas y vistas previamente seleccionadas. Para generar la capa de datos y de negocios el usuario debe palomear la opción “Generar Typed Datasets” y “Generar Business Classes (Solo Tablas)” respectivamente.

El programa solicitara la ubicación de destino de las clases y algunas otras configuraciones adicionales como prefijo y la dirección de la plantilla para la generación de las clases de negocio, la cual se presenta y explica a continuación:

Cuadro 4.1 Plantilla para Generación de Clases de Capa de Negocio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using CubeERP.Business.Common;
using SDPoint.Data.Connector;
using CubeERP.Data.Datasets;

namespace <namespace>
{
    [DataObject]
    public class <classname> : CubeERP.Business.Common.BaseBusiness
    {
        [DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
        public <datasetname> GetAll<tablename>()
        {
            var result = new <datasetname>();
            using (ConnectorProxy proxy = CreateProxy())
            {
                proxy.Fill(result);
            }
            return result;
        }

        [DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
        public <datasetname> Get<tablename>(<methodkeyfield>)
        {
            var result = new <datasetname>();
            using (ConnectorProxy proxy = CreateProxy())
            {
                proxy.Fill(result, <keyfilter>);
            }
            return result;
        }

        [DataObjectMethodAttribute(DataObjectMethodType.Update, true)]
        public void Update(<datasetname> ds)
```

```
{
    var result = new <datasetname>();
    using (ConnectorProxy proxy = CreateProxy())
    {
        proxy.Update(ds.GetChanges());
    }
    //return result;
}

[DataObjectMethodAttribute(DataObjectMethodType.Update, true)]
public void Update(<methodfields>)
{
    try
    {
        using (<datasetname> ds = Get<tablename>(<keyfield>))
        {
            if (ds.<tablename>.Count > 0)
            {
                <valuesasignation>
                Update(ds);
            }
        }
    }
    catch (Exception ex)
    {
        throw ex;
    }
}

[DataObjectMethodAttribute(DataObjectMethodType.Insert, true)]
public void Insert(<methodfields>)
{
    try
    {
        using (<datasetname> ds = new <datasetname>())
        {
            var dr = ds.<tablename>.New<tablename>Row();
            <valuesasignation>
            ds.<tablename>.Add<tablename>Row(dr);
            Update(ds);
```

```
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }

        [DataObjectMethodAttribute(DataObjectMethodType.Delete, true)]
        public void Delete(<methodkeyfield>)
        {
            try
            {
                using (<datasetname> ds = Get<tablename>(<keyfield>))
                {
                    if (ds.<tablename>.Count > 0)
                    {
                        ds.<tablename>[0].Delete();
                        Update(ds);
                    }
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }
    }
}
```

La plantilla que se presenta en el cuadro anterior representa la forma base que debe adoptar cada una de las clases que se crea por cada tabla seleccionada en el generador. Las operaciones iniciales que se pueden hacer son muy sencillas: obtener todos los registros de una tabla, obtener un registro de una tabla, actualizar un registro, insertar un nuevo registro y, por último, eliminar un registro existente.

El texto que se encuentra entre etiquetas es remplazado por los nombres de la tabla y el nombre de la entidad de datos correspondiente.

4.1.3 Generador de Capa de Servicios

El generador de la capa de servicios forma parte integral de la solución que se ha descrito durante todo este capítulo y representa la solución o aportación a la problemática, también descrita anteriormente.

Para que pueda generar el servicio correctamente se debe palomear la opción "Generar Services Classes" en la interfaz de la aplicación que se muestra en la figura 4.4. El funcionamiento de la generación de clases esta descrita en la siguiente figura:

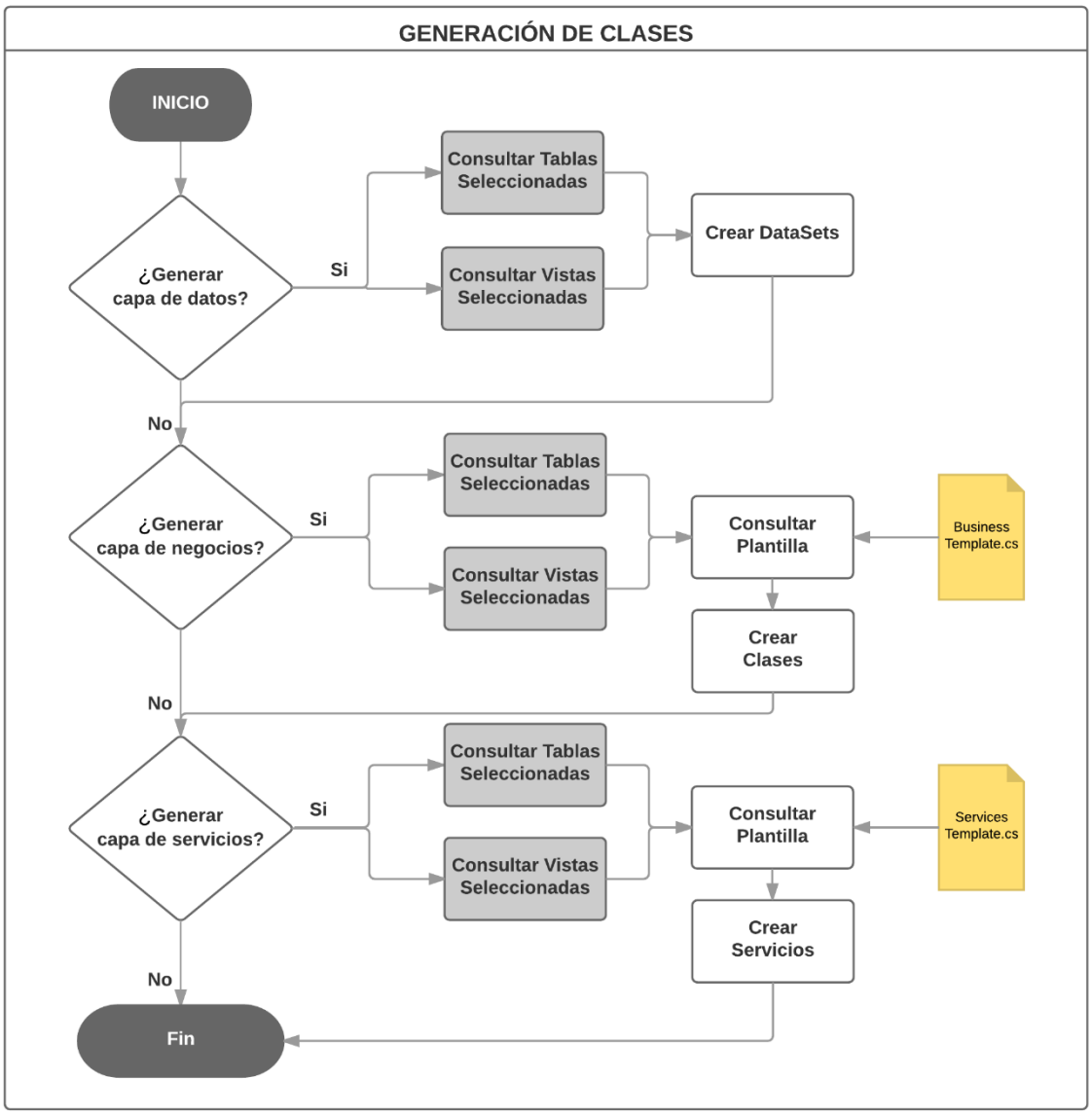

Figura 4.5 Diagrama de Proceso de Generación de Clases en Base a Plantillas

El proceso de generación depende que el usuario seleccione explícitamente que desea que se generen todas las capas. Es posible generar solamente la capa de datos, la de negocios y la de servicios o combinaciones de ellas. Inclusive si el usuario no selecciona la generación de ninguna capa el proceso será finalizado exitosamente, aunque sin resultados.

Al igual que la capa de negocios, la generación de la capa de servicios depende de una plantilla preestablecida que define la estructura, propiedades, métodos del servicio en una sola interfaz que después se debe conectar con la capa de negocios. Dicha plantilla es presentada a continuación:

Cuadro 4.2 Plantilla de Generación de Servicios

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;
using FacmailGastos.Service;
using FacmailGastos.Data.Gastos;
using System.ServiceModel.Web;

namespace <namespace>
{
    public class <classname>
    {
        #region Constructores

        public <classname>()
        {

        }

        public <classname>(<datasetname>.<tablename>Row dr)
        {
            <valuesasignationrows>
        }

        #endregion

        #region Propiedades
```

```
        <valuesasignationproperties>

        #endregion
    }

    [ServiceContract(Namespace = "<namespace>")]
    [XmlSerializerFormat]
    [ServiceKnownType(typeof(<classname>))]
    [ServiceKnownType(typeof(RespuestaGenerica<<classname>>))]
    [ServiceKnownType(typeof(RespuestaGenerica<List<<classname>>>))]
    interface I<classname>
    {
        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<<tablename>> Get<tablename>(Int32 CategoriaID);

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<List<<tablename>>> GetList<tablename>();

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<Int32> Insert<tablename>(<tablename> <tablename>);

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<Int32> Update<tablename>(<tablename> <tablename>);

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<Int32> Delete<tablename>(<methodkeyfield>);
    }
}
```

Todos los archivos generados por la herramienta son posteriormente incorporados a los proyectos de la solución en Visual Studio para que pueda ser utilizados por las aplicaciones web y de escritorio de la plataforma .NET y por las aplicaciones móviles de otras plataformas por medio de los servicios, una vez que sean publicados.

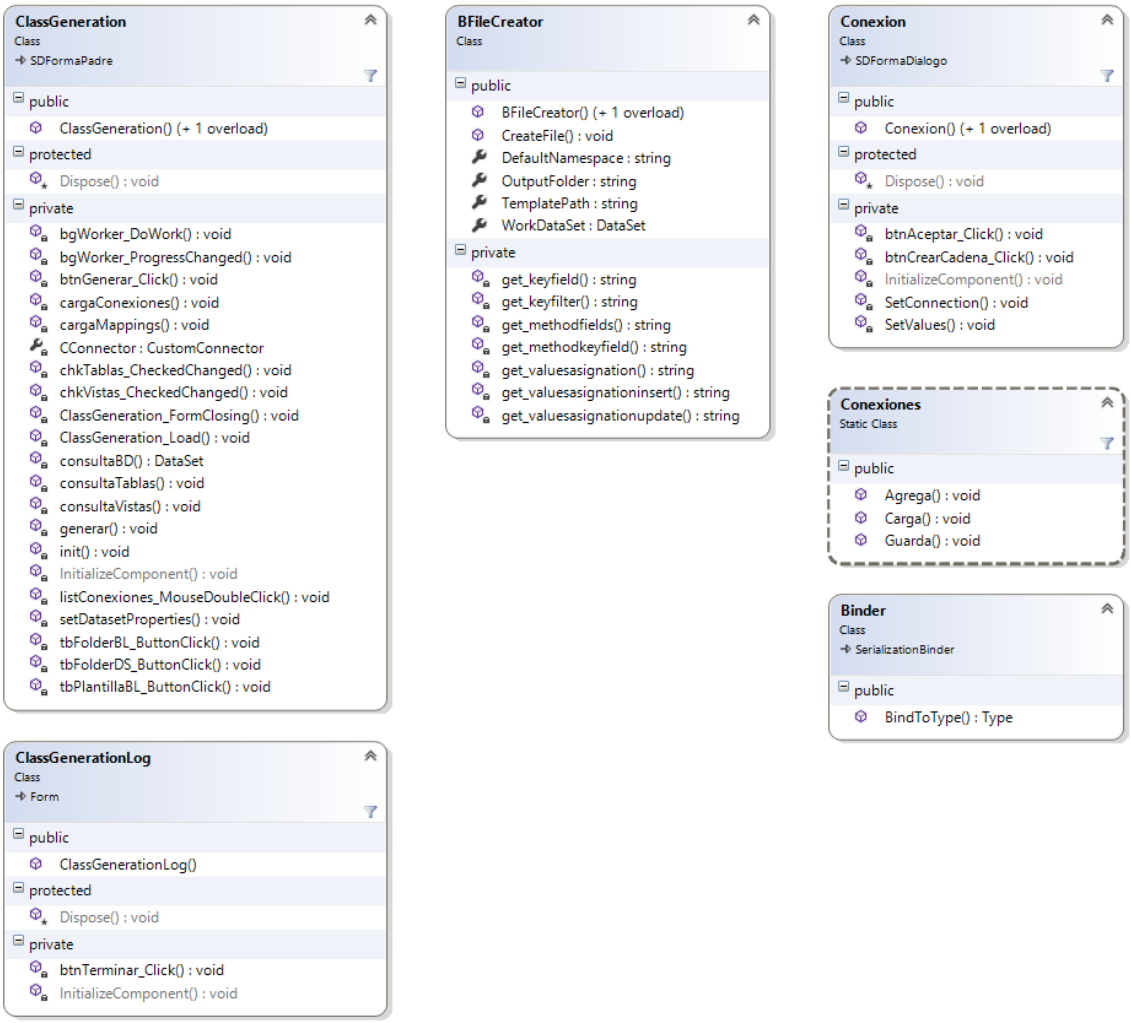

Figura 4.6 Modelo de Clases de la Aplicación

El modelo de clases de la aplicación está compuesto por las entidades que se presentan en la Figura 4.6. Del lado izquierdo se encuentran las clases que representan la generación de clases, en la parte central se encuentra la clase que crea los archivos en base a las plantillas correspondientes, y del lado derecho se encuentran las que se relacionan con las conexiones a la base de datos y las cadenas de conexión.

4.1.4 Versionado de Servicios Generados

Para comunicarse, los clientes y los servicios deben acordar contratos y políticas de consumo. El WSDL (Web Service Description Language) se utiliza precisamente para describir un contrato de interoperación para un servicio, definiendo cuales son las operaciones disponibles en cada endpoint que expone el servicio.

Por cada operación se proporcionan definiciones de esquema XSD para definir mensajes de entrada, salida y error asociados. Los clientes dependen de las definiciones de WSDL y del esquema XSD para generar proxys para comunicarse con el servicio.

Los contratos WSDL pueden incluir opcionalmente secciones de políticas que describen mecanismos de seguridad, codificación, transacciones entre otros requisitos de protocolos para comunicarse con el servicio.

El objetivo es que una vez que se publica un contrado WSDL y un conjunto de políticas para un servicio orientado al público, los clientes puedan generar proxys y código relacionado para comunicarse con dicho los endpoints del servicio, en un futuro previsible.

Idealmente, los contratos y la política no deben cambiarse para preservar la compatibilidad hacia atrás con los clientes que consumen dichos endpoints. Sin duda, esto no quiere decir que los servicios y operaciones relacionadas no puedan surgir cambios.

Windows Communication Foundation (WCF), es un marco de trabajo para la creación de aplicaciones orientadas a servicios. Con WCF, es posible enviar datos como mensajes asincrónicos de un extremo de servicio a otro. WCF expone las políticas de sus servicios con definiciones WSDL y esquemas XSD.

A continuación, se estudian dos enfoques para el versionado de servicios WCF, basados en los contratos de servicios y los contratos de datos.

4.1.4.1 WCF y su Tolerancia a Versiones

Los contratos de WCF son tolerantes a versiones de manera predeterminada, por lo cual la mayoría de las veces, el programador es el que se encarga de definir cuál es el enfoque de versionado que va a utilizar. Las tablas en el Cuadro 4.1 y Cuadro 4.2 encapsulan el efecto de los cambios en cada tipo de contrato que ya utilizan los clientes existentes de un servicio.

Cuadro 4.3 Impacto de los cambios en el contrato de servicio en los clientes existentes

Cambio en el Contrato de Servicio	**Impacto en los Clientes Existentes**
Adición de nuevos parámetros a una declaración de operación o método	Cliente no afectado. Nuevos parámetros inicializados a valores por defecto en el servicio.
Eliminación de parámetros de una declaración de operación o método.	Cliente no afectado. Los parámetros innecesarios enviados por los clientes son ignorados, datos perdidos en el servicio.
Modificación de tipos de parámetros	Se producirá una excepción si el tipo entrante del cliente no puede convertirse en el tipo de datos de parámetro.
Modificación de tipos de variables de retorno	Se producirá una excepción si el valor devuelto del servicio no puede convertirse al tipo de dato esperado en la versión de la declaración de operación del cliente.
Adición de nuevas operaciones	Cliente no afectado. No invocará operaciones de las que no tenga conocimiento.
Eliminación de operaciones	Se producirá una excepción. Se considera que los mensajes enviados por el cliente al servicio utilizan un encabezado de acción desconocido.

Cuadro 4.4 Impacto de los cambios en el contrato de datos en los clientes existentes

Cambio en el Contrato de Datos	**Impacto en los Clientes Existentes**
Agregar nuevos miembros no obligatorios.	Cliente no afectado. Los valores faltantes se inicializan a los valores predeterminados.
Agregar nuevos miembros necesarios.	Se produce una excepción para los valores faltantes.
Eliminar miembros no obligatorios	Datos perdidos en el servicio. No se puede devolver el conjunto de datos completo al cliente, por ejemplo. No genera excepciones.
Eliminar miembros necesarios	Se produce una excepción cuando el cliente recibe respuestas del servicio con valores faltantes.
Modificar los tipos de datos de los miembros existentes	Si los tipos son compatibles, no hay excepción, pero puede recibir resultados inesperados.

Debido a esta tolerancia de versión, la compatibilidad con versiones anteriores se puede lograr con cambios razonables en el contrato WSDL. Para los contratos de servicio, los cambios razonables incluyen la adición de nuevos parámetros que anteriormente no se requerían, la eliminación de parámetros que ya no son necesarios por el servicio o la adición de nuevas operaciones que no existían anteriormente. Aunque soportado en algunos casos, la modificación de tipos de datos para los parámetros y tipos de retorno no es ampliamente recomendado, ya que implica una modificación de la semántica de la operación.

Es precisamente en base a los WSDL que se propone en este trabajo realizar un control de versiones alterno de estos archivos con extensión .xml. De manera que se pueda detectar que información es la que ha cambiado en un método tanto en sus parámetros de entrada o salida, si una operación se añadió por primera vez o si se renombró.

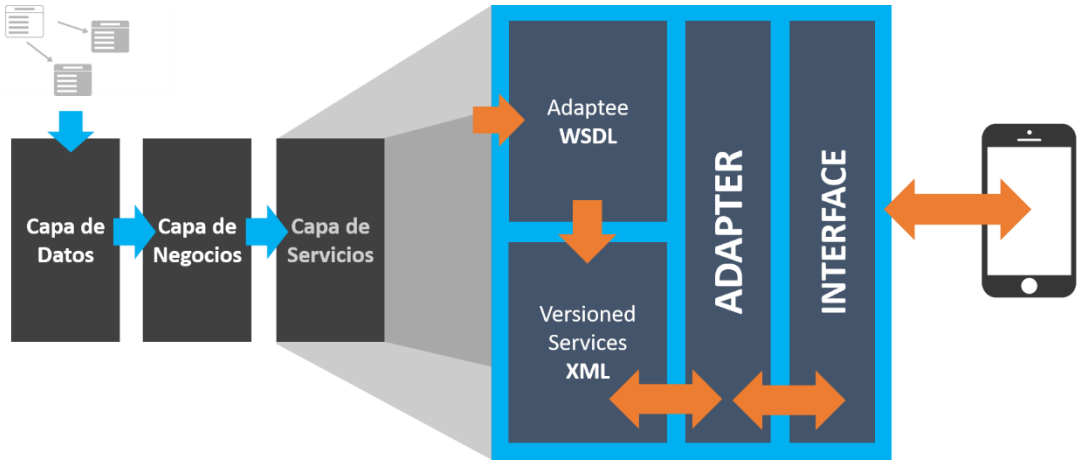

Figura 4.7 Versionado en Capa de Servicios

La figura 4.7 ilustra el proceso mediante el cual, al utilizar el generador de la capa de servicios se implementa un mecanismo que asiste o sugiere al programador diferentes soluciones de versionado dependiendo de la naturaleza de los cambios que pueda surgir dicho servicio a futuro. Los escenarios contemplados, son los anteriormente especificados en el Cuadro 4.3 y Cuadro 4.4. Las pruebas de cada uno de estos escenarios están documentadas en el capítulo cinco del presente documento.

4.2 Paquete (Complemento) para Integración con Entorno de Desarrollo Visual Studio 2015

Con el fin de lograr la total integración de la aplicación de generación de capas de datos, negocios y servicios, se decidió encapsularla en un paquete VSIX fácil de incorporar en el entorno de desarrollo de Visual Studio.

Un paquete VSIX es un archivo que contiene una o más extensiones de Visual Studio, junto con los metadatos que Visual Studio utiliza para clasificar e instalar las extensiones. Los metadatos están contenidos en el manifiesto VSIX y en el archivo Content_Types.xml. Un paquete VSIX también puede contener uno o más archivos de referencia de lenguaje para proporcionar texto de personalizado dependiendo de la ubicación donde se esté utilizando y puede contener paquetes adicionales para instalar las posibles dependencias.

El formato del paquete VSIX sigue el estándar de Open Packaging Conventions (OPC). El paquete contiene binarios y archivos de soporte, junto con un archivo Content_Types.xml y un archivo de manifiesto con extensión .vsix. Además, un paquete puede contener la salida de varios proyectos, o incluso múltiples paquetes que tienen sus propios manifiestos.

Aunque los paquetes de Visual Studio están pensados para compartir de manera masiva diferentes componentes, también en una organización que se dedica al desarrollo de software sirven en demasía para reutilizar componentes entre varios proyectos.

Encapsular el código para que sea reutilizable no solo aplica para equipos de reciente crecimiento, sino que conforme un equipo se establezca y posea una base de código heredada tiene la posibilidad de encapsular y escalar diferentes soluciones para problemas específicos y de esa manera lograr una alta velocidad de desarrollo.

Los paquetes de visual estudio además de tener integridad absoluta con el entorno de desarrollo son fácilmente actualizables. Es decir, si se opera en una empresa donde dos o más equipos de desarrollo utilicen un código base, y ese código base surja cambios en su lógica de programación con el paso del tiempo, es posible que todos los equipos obtengan las últimas actualizaciones.

En Visual Studio el dialogo “Extensiones y Actualizaciones” que se encuentra en el menú de herramientas (Véase Figura 4.8), permite instalar, habilitar, deshabilitar, actualizar y desinstalar paquetes y extensiones. Es posible también, habilitar las actualizaciones automáticas para que Visual estudio compruebe si existe una versión nueva del paquete al iniciar el programa.

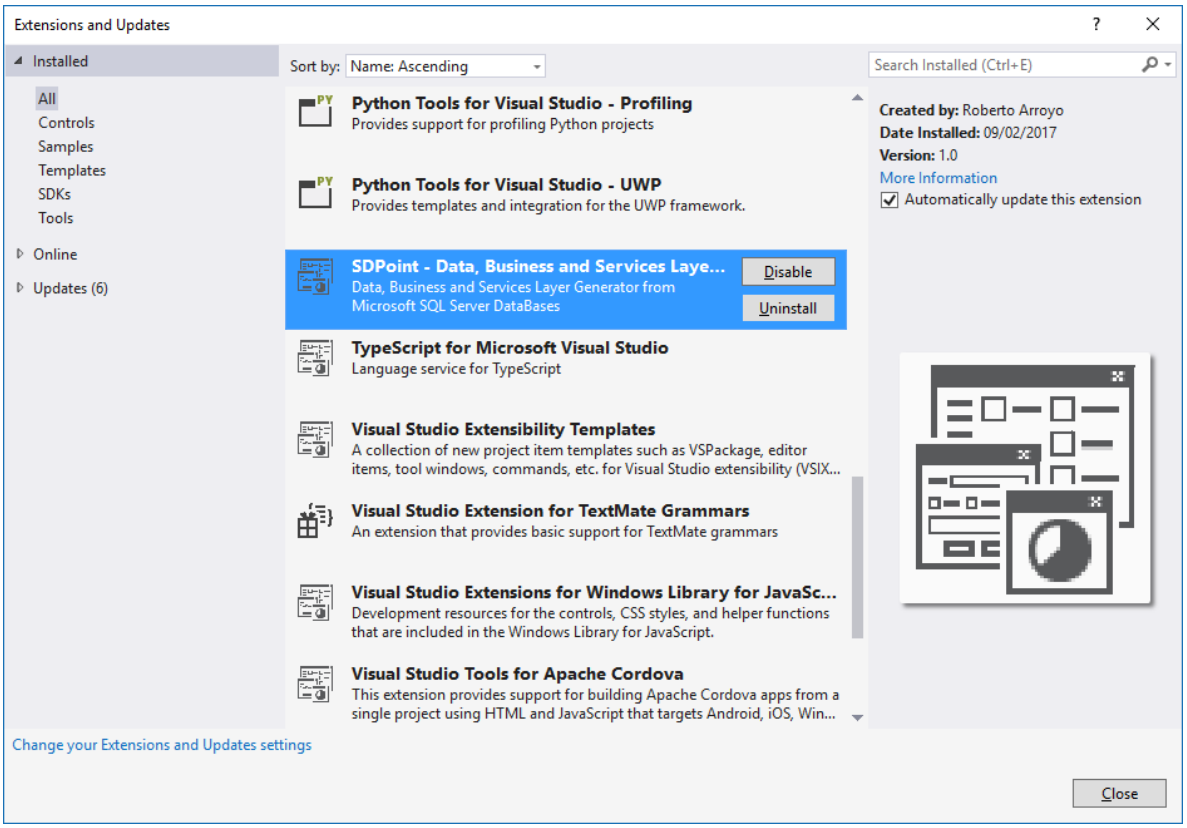

Figura 4.8 Instalación de Complemento en Visual Studio

Del lado izquierdo puede observarse la información más relevante del paquete, la cual fue previamente configurada: creador, fecha de instalación, versión y la imagen de si icono. Para que la aplicación pueda verse en este listado puede instalarse en base al archivo con extensión .vsix, que contiene toda la información necesario para su incorporación, como se mencionaba anteriormente. Otra opción de instalación es la búsqueda en una plataforma que ofrece Microsoft para alojar las extensiones y paquetes. De manera que el usuario puede seleccionar la pestaña que dice “Online” del lado izquierdo y realizar la búsqueda por el nombre o descripción del paquete, y posteriormente instalarlo.

Una vez que el paquete de extensión del generador de las capas de datos, negocio y servicios es instalado, aparecerá el menú de opciones en la barra de acciones superior. En la figura 4.9 se puede observar que el penúltimo botón de acción tiene el nombre de la empresa SDPoint y al presionarlo se expande el submenú que contiene las acciones del administrador de Cadenas de Conexión y del Generador de Clases.

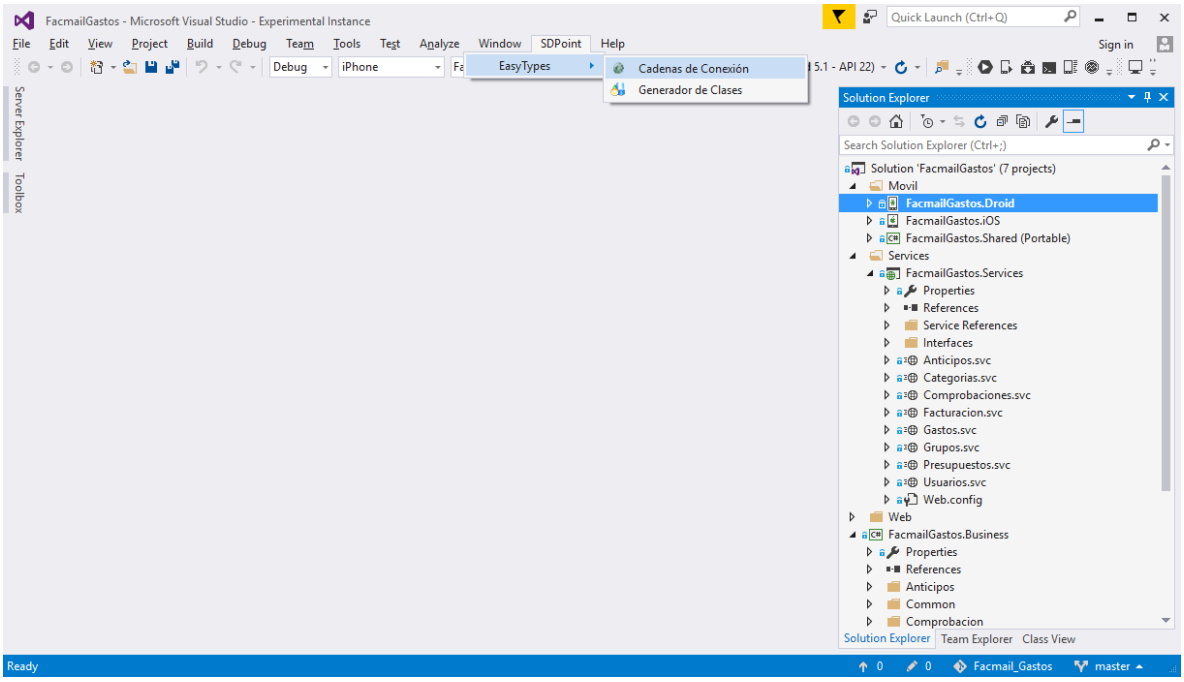

Figura 4.9 Menú de Opciones del Complemento

El módulo de generación de clases podrá realizar acciones sobre los proyectos de la solución que se encuentre abierta en ese momento, con la intención de optimizar el tiempo que es dedicado para estas acciones por parte de los desarrolladores. Tener una herramienta integrada en el entorno de desarrollo beneficia la productividad de los miembros del equipo y fomenta el uso de una base de código estandarizada por el mismo equipo, para ser utilizada en diferentes proyectos.

En el siguiente capítulo se presenta la documentación de las pruebas realizadas con la aplicación integrada en Visual Studio.

Capítulo 5

Pruebas y Resultados

A continuación, se documentan los escenarios de pruebas realizados. En la primera sección se describen los artefactos que son producto de la generación de servicios. Posteriormente se analizan tres estrategias de versionado de servicio aplicables a los servicios generados y se acompañan de las pruebas correspondientes.

En la última sección del presente capítulo se presentan mediciones y análisis del uso de la aplicación que sustentan y evidencian la reducción de tiempo en la generación de servicios, que se traduce en el principal beneficio del presente trabajo.

5.1 Generación de Servicios

Con el objetivo de demostrar la generación de código, se toma como referencia una entidad de una base de datos de prueba llamada Categoría. La tabla tiene una composición sencilla, constituida por una llave primaria y un campo para almacenar la descripción. Este tipo de catálogos sirve para moldear la funcionalidad del sistema y no son accesibles a los usuarios finales de los sistemas que la empresa produce.

Basándose en la plantilla que se presentó anteriormente (Véase Cuadro 4.2 Plantilla de Generación de Servicios) se genera la interfaz del servicio, donde uno de los constructores de la misma acepta un objeto compatible en su totalidad con los que se producen en la capa de negocio. Contiene, además, cada uno de los métodos que se generan en la capa de negocio por lo cual solo se necesita implementar la interfaz, para que adopte toda la lógica contenida en la misma.

Cuadro 5.1 Interfaz de Servicio Generada por la Aplicación

```
namespace FacmailGastos.Services
{
    public class Categoria
    {
        #region Constructores

        public Categoria()
        {

        }

        public Categoria(DSCategoria.CategoriaRow dr)
        {
            CategoriaID = dr.CategoriaID;
            Descripcion = dr.Descripcion;
            Color = dr.Color;
            ImagenIconoUrl = dr.ImagenIconoUrl;
        }

        #endregion

        #region Propiedades
        [DataMember]
        public Int32 CategoriaID;
        [DataMember]
        public String Descripcion;
        [DataMember]
        public String Color;
        [DataMember]
        public String ImagenIconoUrl;

        #endregion
    }

    [ServiceContract(Namespace = "FacmailGastosService")]
```

```
    [XmlSerializerFormat]
    [ServiceKnownType(typeof(Categoria))]
    [ServiceKnownType(typeof(RespuestaGenerica<Categoria>))]
    [ServiceKnownType(typeof(RespuestaGenerica<List<Categoria>>))]
    interface ICategoria
    {
        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<Categoria> GetCategoria(Int32 CategoriaID);

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<List<Categoria>> GetListCategoria();

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<Int32> InsertCategoria(Categoria Categoria);

        [OperationContract]
        [WebInvoke]
        RespuestaGenerica<Int32> UpdateCategoria(Categoria Categoria);

        [OperationContract]
        [DataContractFormat]
        [WebInvoke]
        RespuestaGenerica<Int32> DeleteCategoria(Int32 CategoriaID);
    }
}
```

La aplicación genera una plantilla genérica donde se realiza la implementación de cada uno de los métodos de la capa de negocio para ser expuesta por el servicio. De esta manera, la interfaz que se describe previamente está lista para ser consumida por las aplicaciones clientes, que solamente deberán asignar la referencia correspondiente y utilizar un proxy que en base al contrato WSDL permita la interacción con el servicio. A continuación, se presenta el ejemplo de la implementación del servicio para la entidad de Categoría:

Cuadro 5.2 Implementación de la Interfaz del Servicio con Capa de Negocios

```
namespace FacmailGastos.Services
{
    [ServiceBehavior(Namespace = "FacmailGastosService")]
    public class Categorias : ICategoria
    {
        public RespuestaGenerica<Categoria> GetCategoria(Int32 CategoriaID)
        {
            try
            {
                CategoriaBL objCategoria = new CategoriaBL();
                DSCategoria dsCategoria = objCategoria.GetCategoria(CategoriaID);
                if (dsCategoria.Categoria.Rows.Count > 0)
                    return              RespuestaGenerica<Categoria>.CrearRespuesta(new
Categoria(dsCategoria.Categoria.Rows[0] as DSCategoria.CategoriaRow), true, "OK", null,
1);
                else
                    return RespuestaGenerica<Categoria>.CrearRespuesta(null, false, "No
se encontraron resultados", null, -1);
            }
            catch (Exception ex)
            {
                return     RespuestaGenerica<Categoria>.CrearRespuesta(null,     false,
ex.Message, new Excepcion(), -2);
            }
        }

        public RespuestaGenerica<Int32> InsertCategoria(Categoria g)
        {
            try
```

```
            {
                CategoriaBL objCategoria = new CategoriaBL();
                objCategoria.Insert(g.CategoriaID,       g.Descripcion,       g.Color,
g.ImagenIconoUrl);

                return   RespuestaGenerica<Int32>.CrearRespuesta(g.CategoriaID,   true,
"OK", null, 1);
            }
            catch (Exception ex)
            {
                return  RespuestaGenerica<Int32>.CrearRespuesta(-1,  false,  ex.Message,
new Excepcion(), -2);
            }
        }

        public RespuestaGenerica<Int32> UpdateCategoria(Categoria g)
        {
            try
            {
                CategoriaBL objCategoria = new CategoriaBL();
                objCategoria.Update(g.CategoriaID,       g.Descripcion,       g.Color,
g.ImagenIconoUrl);
                return   RespuestaGenerica<Int32>.CrearRespuesta(g.CategoriaID,   true,
"OK", null, 1);
            }
            catch (Exception ex)
            {
                return  RespuestaGenerica<Int32>.CrearRespuesta(-1,  false,  ex.Message,
new Excepcion(), -2);
            }
        }

        public RespuestaGenerica<Int32> DeleteCategoria(Int32 CategoriaID)
        {
            try
            {
                CategoriaBL objCategoria = new CategoriaBL();
                objCategoria.Delete(CategoriaID);
                return RespuestaGenerica<Int32>.CrearRespuesta(CategoriaID, true, "OK",
null, 1);
```

```
            }
            catch (Exception ex)
            {
                return  RespuestaGenerica<Int32>.CrearRespuesta(-1,  false,  ex.Message,
new Excepcion(), -2);
            }
        }
    }
}
```

5.2 Pruebas de Escenarios de Compatibilidad de Versiones de Servicios

En esta sección del capítulo se explicarán brevemente los diferentes sets de pruebas que se llevaron a cabo en base a diferentes estrategias de versionado de servicios WCF:

5.2.1 Estrategias de Control de Versiones

La tolerancia de versión es aceptable en caso de nuevas operaciones, o la adición y eliminación de parámetros de operación. Pero al mismo tiempo, debe ser evitable cuando se introduce un cambio de ruptura en el servicio, como la eliminación de operaciones de un contrato de servicio o la adición o eliminación de miembros requeridos de un contrato de datos.

Para el ciclo de pruebas que se realizó en este proyecto se abordaron 3 estrategias: control de versiones ágil, control de versiones semi-estricto, control de versiones estricto:

5.2.1.1 Control de Versiones Ágil

Esta estrategia de control de versiones se basa plenamente en la compatibilidad hacia atrás durante el mayor tiempo posible y evita el contrato formal y el control de versiones de extremo hasta que se rompa la compatibilidad. En el enfoque ágil del control de versiones, realizamos cambios en los contratos de servicios y contratos de datos existentes sin su versión, ni en el suministro de nuevos puntos finales. La figura siguiente muestra el enfoque de versiones ágiles:

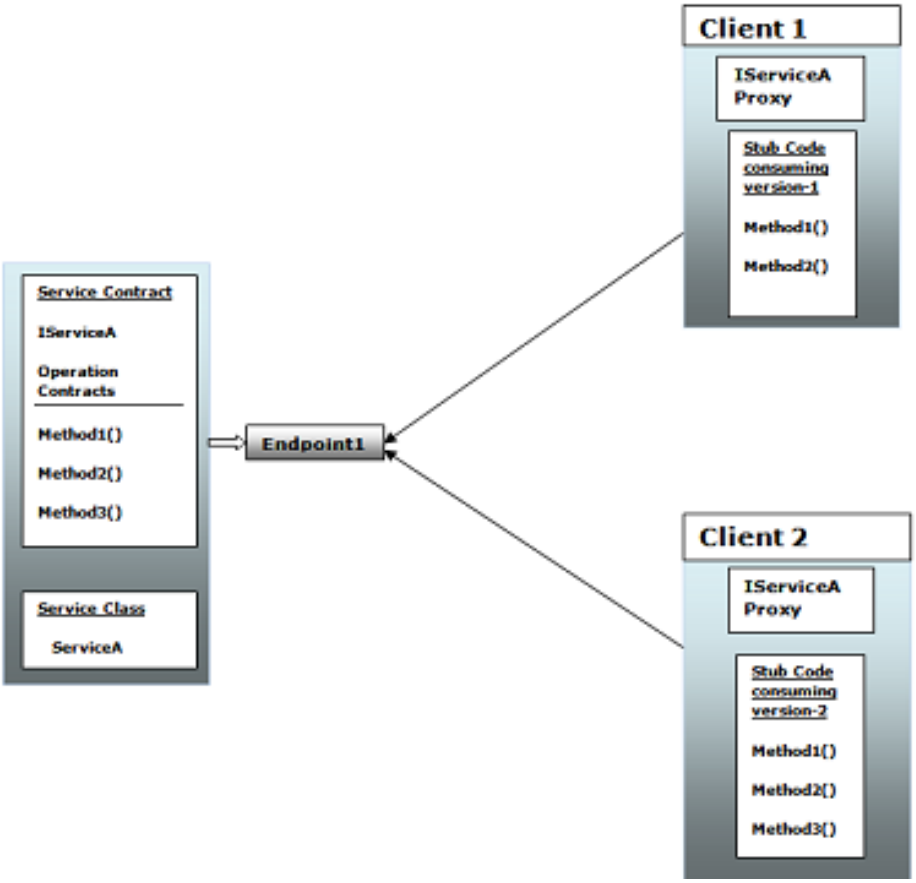

Figura 5.1 Estrategia de Control de Versiones Ágil

El enfoque de versiones ágil es útil en ambientes ágiles que requieren actualizaciones frecuentes del código de producción. La idea del primer set de pruebas es comprobar que el servicio generado por la aplicación es compatible con esta estrategia.

5.2.1.2 Pruebas Utilizando Estrategia de Control de Versiones Ágil

Para ello se toma como base una tabla existente en el modelo de datos con una estructura bastante sencilla. Corresponde a un catálogo de sistema el cual únicamente tiene los campos de ID de identificación de tipo entero y la descripción del método de pago.

El contrato de datos generado para Método de Pago es el siguiente:

Cuadro 5.3 Contrato de Datos en Control de Versiones Ágil

```
[DataContract(Name = "MetodoDePago",
   Namespace="http://schemas.sdpoint.com/Gastos/2017/01")]
```

```
public class MetodoDePago
{
   [DataMember]
   public int MetodoDePagoID;
   [DataMember]
   public string Descripcion;
}
```

El contrato de servicio generado para Método de Pago es el siguiente:

Cuadro 5.4 Contrato de Servicio en Control de Versiones Ágil

```
[ServiceContract(Name = "MetodoPagoServ",
   Namespace = "http://servicios.sdpoint.com/Gastos/2017/01")]
public interface IMetodoDePagoService
{
   [OperationContract]
   MetodoDePago Update(MetodoDePago info);
}
```

Hasta este momento se observa únicamente la parta del código que se expone a los clientes para ser consumido. En la implementación del servicio es donde se hace la conexión entre la capa de negocios y la capa de servicios. En el cuadro que se encuentra a continuación podemos observar que la clase MetodoDePagoService que también fue generado por el servicio hereda de la clase interfaz IMetodoDePagoService y le da funcionalidad a la operación inicial.

Para conectar con la capa de negocios se crea una instancia del objeto que corresponde a la lógica del negocio y se accede a su metodo Update, el cual se encarga de llevar un dataset donde está la información contenida para realizar la actualización.

Todo este proceso se realiza de manera automática por la aplicación y es un proceso transparente para el usuario. Basta con realizar una compilación en el proyecto correspondiente para que el servicio pueda ser consumido, y posteriormente publicado.

Cuadro 5.5 Implementación de Servicio en Control de Versiones Ágil

```
namespace Gastos.Services
{
    public class MetodoDePagoService: IMetodoDePagoService
    {
        public MetodoDePago Update(MetodoDePago metododepago)
        {
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.Update(metododepago);
        }
    }
}
```

La aplicación cliente consume los servicios que ya fueron expuestos haciendo referencia a la versión actual del servicio que se encuentra publicado. De manera que, con la creación de un proxy puede acceder a todas las políticas que el WSDL expresa, como se explicaba anteriormente.

La implementación del servicio a nivel cliente es la siguiente:

Cuadro 5.6 Consumo del Servicio en Cliente 1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using WCFTestClient.MyService;
using System.Runtime.Serialization;

namespace PruebaCliente1
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            using (MetodoPagoServClient proxy = new MetodoPagoServClient())
            {

                var sc = proxy.Update(new MetodoDePago { MetodoDePagoID = 1,
Descripcion = "Efectivo" });
                Console.WriteLine("MetodoDePagoID: " + sc.MetodoDePagoID);
                Console.WriteLine("Descripción: " + sc.Descripcion);

            }
            Console.ReadLine();
        }
    }
}
```

Al ejecutar la aplicación en consola del Cliente1 muestra los siguientes resultados:

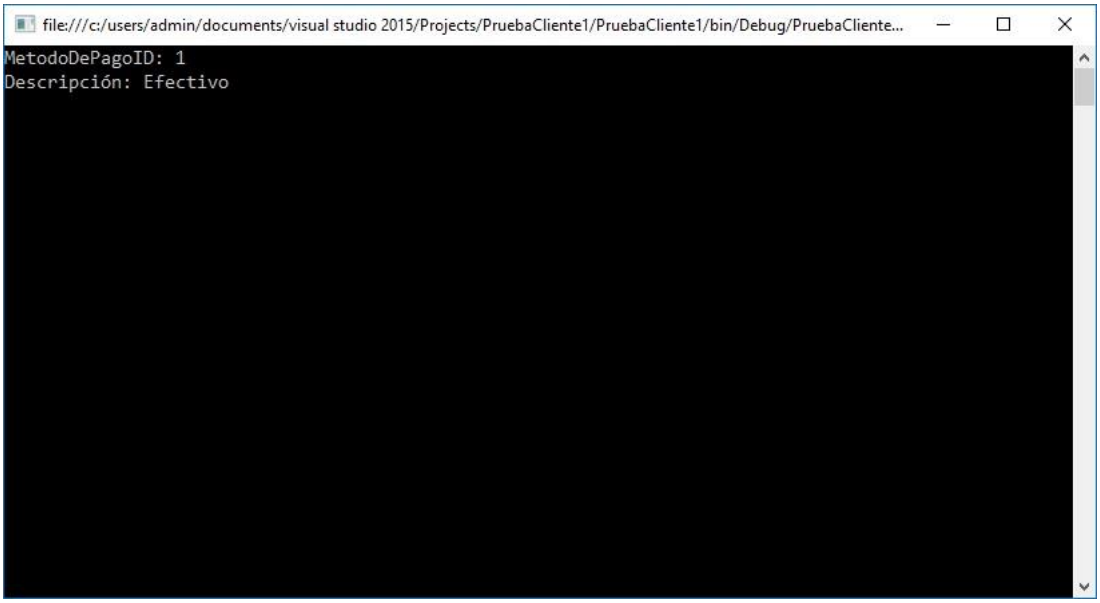

Figura 5.2 Interfaz de Aplicación Cliente 1

Podemos suponer que el servicio anterior es liberado a producción y las aplicaciones cliente que han consumido el servicio están funcionando bien. Posteriormente, una de las aplicaciones necesita contar con una clave que identifique el método de pago ante la autoridad fiscal, además de contar con un método que permita consultar los métodos de pago que se encuentra en la base de datos proporcionando el ID correspondiente.

En SQL en la tabla MetodoDePago se adiciona la columna “ClaveSAT” como campo opcional, de lo contrario causaría una excepción, como ya se mencionó anteriormente.

En contrato de datos ya modificado, después de pasar por el generador es el siguiente:

Cuadro 5.7 Modificación Automática al Contrato de Datos en CVA

```
[DataContract(Name = "MetodoDePago",
   Namespace="http://schemas.sdpoint.com/Gastos/2017/01")]
public class MetodoDePago
{
   [DataMember]
   public int MetodoDePagoID;
   [DataMember]
   public string Descripcion;
   [DataMember(IsRequired = false)]
   public string ClaveSAT;
}
```

El contrato de servicio modificado para Método de Pago es el siguiente:

Cuadro 5.8 Modificación Automática al Contrato de Servicios en CVA

```
[ServiceContract(Name = " MetodoPagoServ",
   Namespace = "http://servicios.sdpoint.com/Gastos/2017/01")]
public interface IMetodoDePagoService
{
   [OperationContract]
   MetodoDePago Update(MetodoDePago info);

   [OperationContract]
   MetodoDePago GetMetodoDePagoByID(int MetodoDePagoID);
}
```

Por último, la implementación del servicio también se ve afectada debido a que la capa de negocio también cambió y se incorporó un nuevo método:

Cuadro 5.9 Modificación Automática a la Implementación del Servicio en CVA

```
namespace Gastos.Services
{
    public class MetodoDePagoService: IMetodoDePagoService
    {
        public MetodoDePago Update(MetodoDePago metododepago)
        {
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.Update(metododepago);
        }
        public MetodoDePago GetMetodoDePagoByID(int MetodoDePagoID)
        {
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.GetMetodoDePagoByID(MetodoDePagoID);
        }
    }
}
```

Una vez publicado el servicio, si ejecutamos nuevamente la aplicación del cliente 1 podemos observar que sigue manteniendo una total compatibilidad con el servicio:

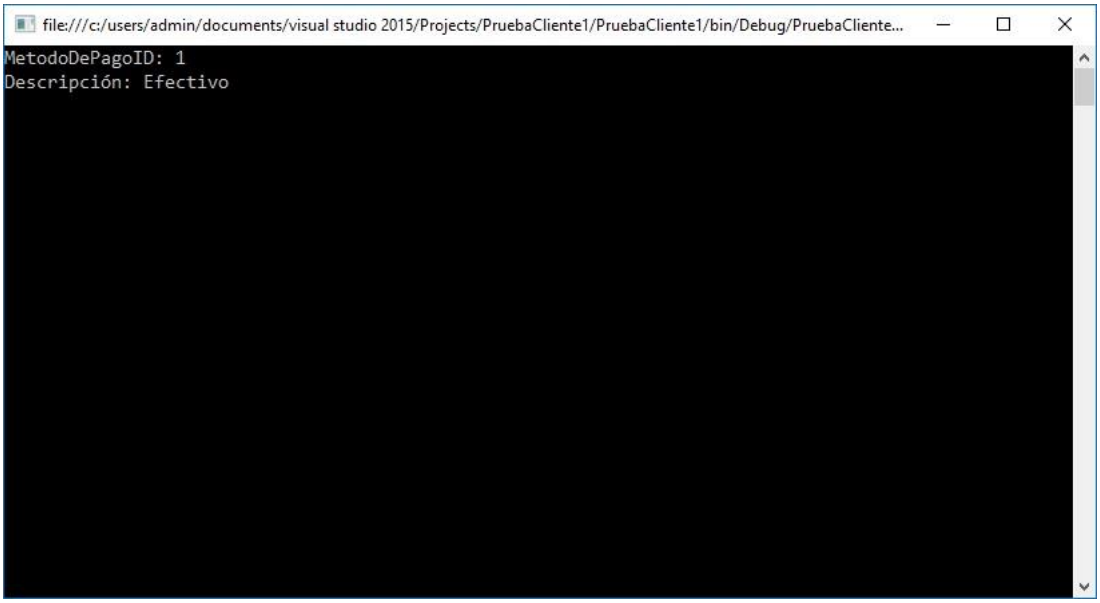

Figura 5.3 Interfaz de Aplicación Cliente 1 Posterior a Cambios

En la segunda aplicación es posible consumir el mismo servicio con los cambios solicitados y utilizar las nuevas propiedades del contrato de datos y métodos del contrato de servicio que se exponen en el esquema WSDL:

Cuadro 5.10 Consumo de Servicio en Cliente 2

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using WCFTestClient.MyService;
using System.Runtime.Serialization;

namespace PruebaCliente1
{
    class Program
    {
        static void Main(string[] args)
        {
            using (MetodoPagoServClient proxy = new MetodoPagoServClient ())
            {
                var sc = proxy.Update(new MetodoDePago { MetodoDePagoID = 1,
Descripcion = "Efectivo", ClaveSAT = "01" });
                Console.WriteLine("MetodoDePagoID: " + sc.MetodoDePagoID);
                Console.WriteLine("Descripción: " + sc.Descripcion);
                Console.WriteLine("ClaveSAT: " + sc.ClaveSAT);

                var sc = proxy.GetMetodoDePagoByID(1);
                Console.WriteLine("---Recuperando Información---");
                Console.WriteLine("MetodoDePagoID: " + sc.MetodoDePagoID);
                Console.WriteLine("Descripción: " + sc.Descripcion);
                Console.WriteLine("ClaveSAT: " + sc.ClaveSAT);
            }
            Console.ReadLine();
        }
    }
}
```

Al ejecutar la aplicación muestra la siguiente información:

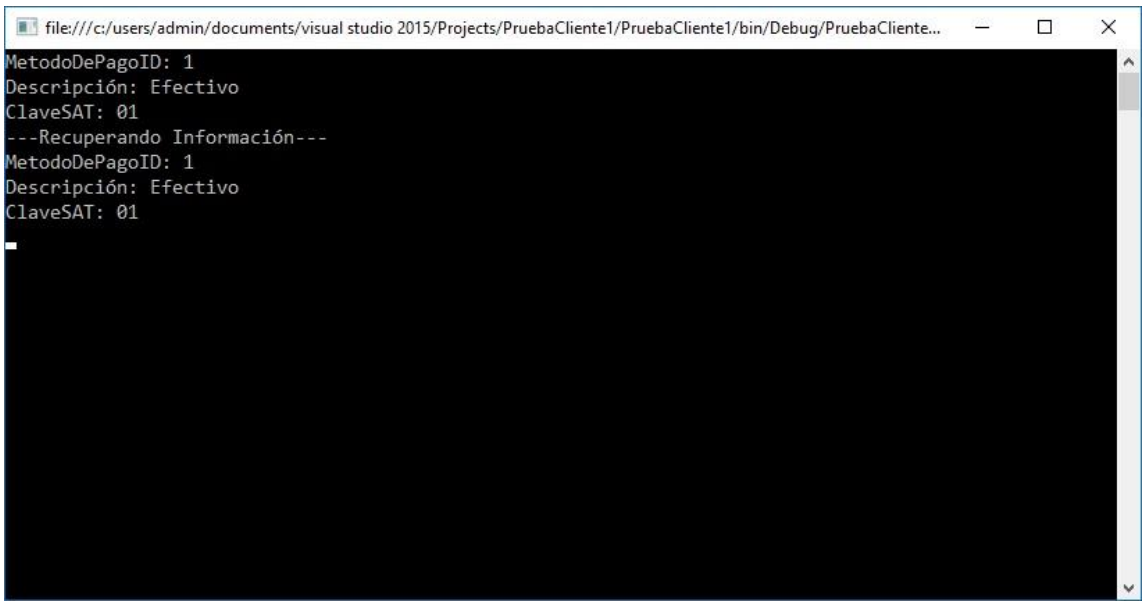

Figura 5.4 Interfaz de Aplicación Cliente 2

Así que podemos ver que ambos clientes están trabajando sin problemas después de la provisión de una nueva versión con cambios en contrato de servicio y contratos de datos en el extremo del servicio. Un cliente está utilizando las funcionalidades antiguas, mientras que otro utiliza las nuevas. Aunque los clientes existentes continúan trabajando con exito, existen los siguientes problemas que surgen con la tolerancia de versión:

- **Adición de nuevas operaciones en el contrato de servicio:** Si se agrega un nuevo método de servicio, se actualizará el WSDL. Los consumidores actuales del servicio no necesitan actualizar la referencia de servicio para referirse a la última versión del WSDL. Sólo aquellos consumidores que requieran las nuevas funcionalidades deben actualizar la referencia de servicio para consumirlos en el cliente. Por lo tanto, es imposible realizar un seguimiento de cuáles de los clientes existentes han actualizado su WSDL y cuales continuar en versiones anteriores.

- **Adición de miembros no necesarios al contrato de datos:** Si los miembros no necesarios se agregan a un contrato de datos y el código no se actualiza en el cliente, se pasan los valores predeterminados o se utilizan para el cliente.

Para inicializar los valores faltantes a algunos valores relevantes, puede ser necesario escribir código adicional.

- **Eliminación de miembros no requeridos de los contratos de datos**: En el artículo anterior, también discutimos, si un DataMember se quita del contrato de datos en el extremo del servicio y está presente en el código de stub, debido a la no actualización del servicio, el miembro enviado desde el cliente pasa por una ronda viaje. Debido a este problema, los datos pasados a los servicios o devueltos a los clientes se pierden.

El enfoque de la versión ágil sólo debe adoptarse después de considerar todos estos escenarios.

Además, es importante considerar que esta estrategia de versionado se rige por la utilización de un solo endpoint que es expuesto para ambos clientes. El endpoint se define en las especificaciones del archivo Web.Config como se muestra en la figura siguiente:

```
<services>
  <service name="FacmailGastos.Services.MetodoDePagoService" behaviorConfiguration="ServiceBehavior">
    <endpoint address="ep1" binding="basicHttpBinding" contract="FacmailGastos.Services.IMetodoDePagoService" />
  </service>
</services>
```

Figura 5.5 Endpoint en Control de Versiones Ágil

5.2.1.3 Control de Versiones Semi-Estricto

Este enfoque se encuentra en el punto intermedio entre versiones ágiles y estrictas. Los cambios se rastrean cuando se modifican los contratos o los endpoints (puntos finales). Permite agregar nuevas operaciones a un nuevo contrato de servicio de la Versión 2 que hereda el contrato de servicio de la Versión 1. El control de versiones se logra exponiendo un nuevo punto final para el contrato de servicio de la Versión 2. El espacio de nombres de las operaciones existentes sigue siendo el del contrato original, lo que significa que los clientes de Version1 pueden pasar al nuevo punto final sin impacto al código existente y actualizar sus proxys para reflejar nuevas operaciones a su elección. La siguiente figura ilustra el funcionamiento de esta estrategia:

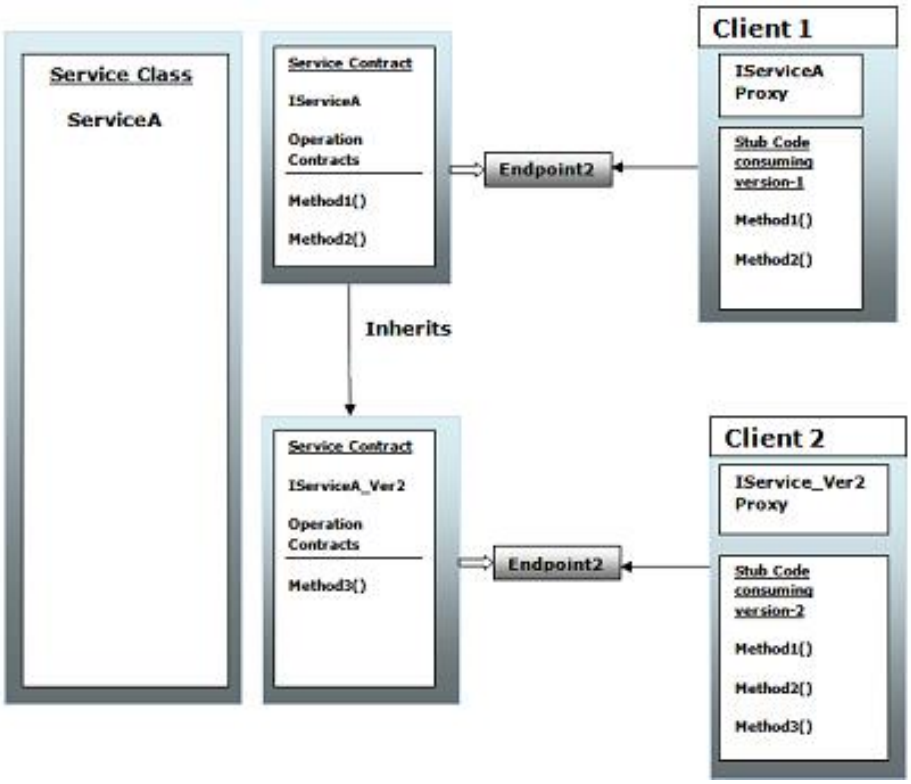

Figura 5.6 Estrategia de Control de Versiones Semi-Estricto

5.2.1.4 Pruebas Utilizando Estrategia de Control de Versiones Semi-Estricto

Utilizando el mismo modelo de datos se realiza una segunda prueba, utilizando un endpoint diferente para cada versión. Primeramente, se definen los contratos de servicio:

Cuadro 5.11 Contrato de Datos en Control de Versiones Semi-Estricto

```
     [DataContract(Name = "MetodoDePago", Namespace =
"http://schemas.sdpoint.com/Gastos/2017/01")]
    public class MetodoDePago
    {
        [DataMember]
        public int MetodoDePagoID;
        [DataMember]
        public string Descripcion;
        [DataMember(IsRequired = false)]
        public string ClaveSAT;
    }
```

Cuadro 5.12 Contrato de Servicio en Control de Versiones Semi-Estricto

```
    [ServiceContract(Name = " MetodoPagoServ", Namespace =
"http://servicios.sdpoint.com/Gastos/2017/01")]
    public interface IMetodoDePagoService
    {
        [OperationContract]
        MetodoDePago Update(MetodoDePago info);
    }

    [ServiceContract(Name = " MetodoPagoServ", Namespace =
"http://servicios.sdpoint.com/Gastos/2017/02")]
    public interface IMetodoDePagoService_V2 : IMetodoDePagoService
    {
        [OperationContract]
        MetodoDePago Update(MetodoDePago info);

        [OperationContract]
        MetodoDePago GetMetodoDePagoByID(int MetodoDePagoID);
    }
```

La principal modificación de esta estrategia es tener 2 interfaces para el contrato de servicio, de manera que en la implementación del servicio WCF se utiliza la herencia de la siguiente manera:

Cuadro 5.13 Implementación de Servicio en Control de Versiones Semi-Estricto

```
    public class MetodoDePagoService : IMetodoDePagoService_V2
    {
        public MetodoDePago Update(MetodoDePago metododepago)
        {
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.Update(metododepago);
        }

        public MetodoDePago GetMetodoDePagoByID(int MetodoDePagoID)
        {
```

```
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.GetMetodoDePagoByID(MetodoDePagoID);
        }
    }
```

En el archivo web.config se expone el nuevo endpoint:

```
<services>
  <service name="FacmailGastos.Services.MetodoDePagoService" behaviorConfiguration="ServiceBehavior">
    <endpoint address="ep1" binding="basicHttpBinding" contract="FacmailGastos.Services.IMetodoDePagoService" />
    <endpoint address="ep2" binding="basicHttpBinding" contract="FacmailGastos.Services.IMetodoDePagoService_V2" />
  </service>
</services>
```

Figura 5.7 Endpoints en Control de Versiones Semi-Estricto

Ahora, con este cambio en el extremo del servicio, el cliente existente puede seguir siendo compatible ya que el contrato de servicio IMetodoDePagoService contiene la antigua especificación de versiones. Los clientes que estén interesados en la nueva funcionalidad GetMetodoDePagoByID pueden actualizar su referencia para consumir el nuevo punto final.

A continuación, se muestra la implementación para la aplicación del cliente nuevo o un cliente existente que desea una funcionalidad adicional:

Cuadro 5.14 Consumo de Servicio en Cliente CVSE

```
using (MetodoPagoServClient proxy = new MetodoPagoServClient())
{
    var sc = proxy.Update(new MetodoDePago { MetodoDePagoID = 1, Descripcion =
"Efectivo" });
    Console.WriteLine("MetodoDePagoID: " + sc.MetodoDePagoID);
    Console.WriteLine("Descripción: " + sc.Descripcion);
    var sc = proxy.GetMetodoDePagoByID(1);
    Console.WriteLine("---Recuperando Información---");
    Console.WriteLine("MetodoDePagoID: " + sc.MetodoDePagoID);
    Console.WriteLine("Descripción: " + sc.Descripcion);
    Console.WriteLine("ClaveSAT: " + sc.ClaveSAT);
}
Console.ReadLine();
```

5.2.1.5 Control de Versiones Estricto

En la estrategia de control de versiones estricta, se requiere una especificación formal de versiones para todos y cada uno de los cambios en los contratos de servicio. En algunos escenarios, el versionado formal es una necesidad. Por ejemplo, si elimina operaciones de servicio. Al retrasar el control de versiones formales, ahorra en el esfuerzo de desarrollo por cambios insignificantes, pero pierde la capacidad de rastrear las versiones de los servicios que ha expuesto y seguir la migración de los clientes a la funcionalidad de servicio actualizada.

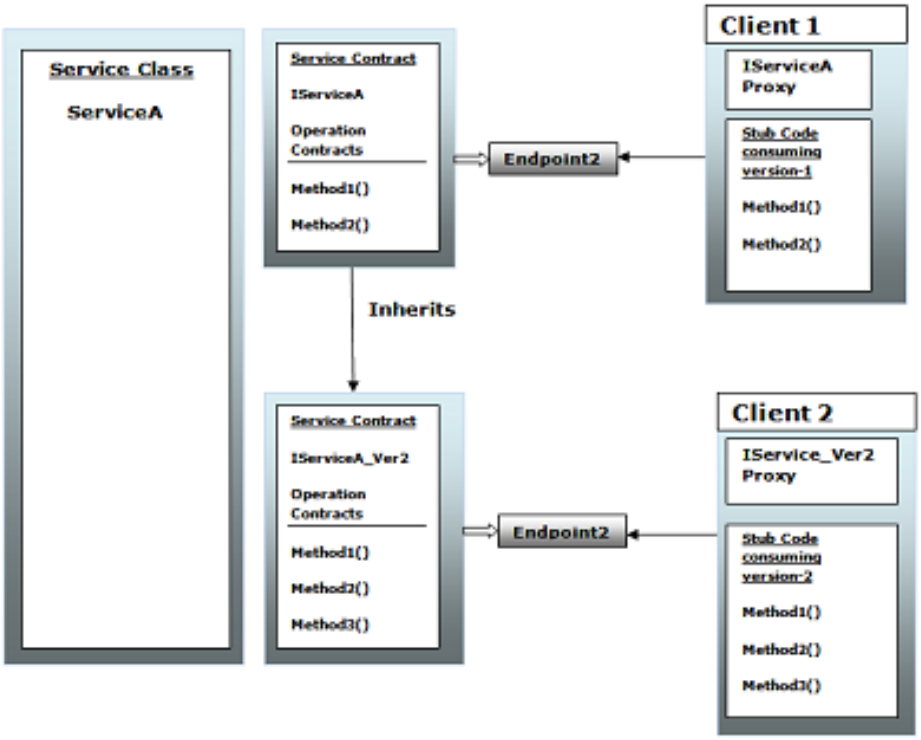

Figura 5.8 Estrategia de Control de Versiones Estricto

En este tipo de versiones, un nuevo punto final se expone cada vez que se elimina o agrega un método de servicio. Incluso si las operaciones existentes no se cambian, con un versionado estricto, el contrato de la versión 2 debe incluir todas las operaciones existentes bajo el nuevo espacio de nombres. La Figura 5.6 ilustra el funcionamiento de la estrategia control de versiones estricto.

5.2.1.6 Pruebas Utilizando Estrategia de Control de Versiones Estricto

Empleando el mismo escenario de las pruebas anteriores, tomamos la misma declaración del contrato de datos, pero se crea un nuevo service contract que contenga los ajustes necesarios y los métodos de la versión anterior, sin tener una herencia a nivel interfaz:

Cuadro 5.15 Contrato de Datos en Control de Versiones Estricto

```
    [ServiceContract(Name = "MetodoPagoServ", Namespace =
"http://servicios.sdpoint.com/Gastos/2017/02")]
    public interface IMetodoDePagoService_V2
    {
        [OperationContract]
        MetodoDePago Update(MetodoDePago info);

        [OperationContract]
        MetodoDePago GetMetodoDePagoByID(int MetodoDePagoID);
    }
```

Cuadro 5.16 Contrato de Servicio en Control de Versiones Estricto

```
    public class MetodoDePagoService : IMetodoDePagoService, IMetodoDePagoService_V2
    {
        public MetodoDePago Update(MetodoDePago metododepago)
        {
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.Update(metododepago);
        }
        public MetodoDePago GetMetodoDePagoByID(int MetodoDePagoID)
        {
            MetodoDePagoBL objMetodoDePago = new MetodoDePagoBL();
            return objMetodoDePago.GetMetodoDePagoByID(MetodoDePagoID);
        }
    }
```

Se puede observar que la principal diferencia es que en el control de versiones agiles el versionado y herencia se realiza en la interfaz del servicio mientras que en el estricto se realiza propiamente en la implementación del servicio, antes de ser expuesto. Tomando como

base la misma implementación que se realizó en el segundo cliente en el set de pruebas anterior se puede observar que la aplicación responde de la siguiente manera:

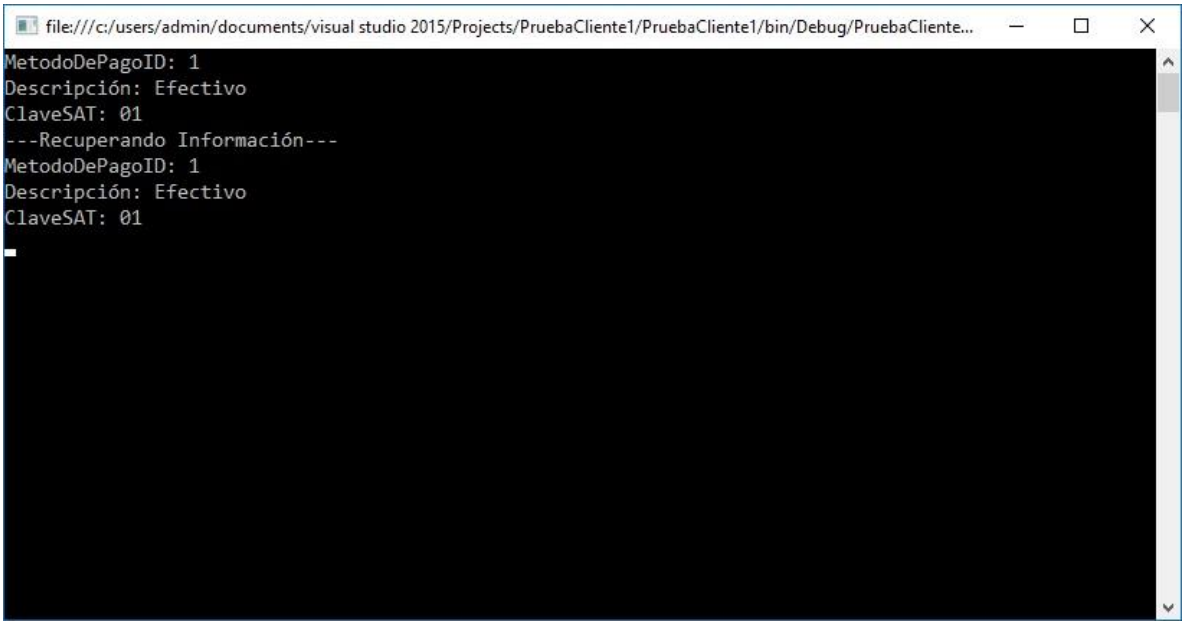

Figura 5.9 Interfaz de Aplicación Cliente CVE

A pesar de que los tres escenarios de pruebas en base a estrategias de control de versiones aparentemente nos muestran los mismos resultados, el uso de cada uno de ellos queda a criterio del programador que esté realizando la implementación. La herramienta descrita en capítulos anteriores que se incorpora al entorno de desarrollo de Visual Studio, proporciona al usuario compatibilidad con el control de versiones agiles, en base a la información contenida en el archivo WSDL.

Algunas otras conclusiones y recomendaciones relacionadas con las pruebas realizadas y los enfoques o estrategias de control de versiones se documentan en el sexto capítulo del presente documento.

5.3 Mediciones y Análisis de Uso

Con el fin de realizar un análisis sobre el uso de la herramienta que se describe en el presente documento, se realiza la siguiente comparativa entre tareas asignadas a miembros del departamento de desarrollo de software y programación relacionadas con la generación de servicios para ser consumidos entre plataformas web o móviles ajenas a .NET.

Los datos que se presentan a continuación muestran una relación entre la cantidad de entidades creadas o modificadas en una tarea en específico y el tiempo que fue requerido por el programador para la ejecución de la misma, realizando el trabajo de generación de servicios de manera manual. Esta medición contempla el tiempo desde que la tarea se inició, la programación y pruebas unitarias.

Cuadro 5.17 Relación de Tiempo Empleado Antes del Uso de la Herramienta

Descripción de Tarea	Estrategia de Control de Versiones	Entidades Creadas	Entidades Modificadas	Tiempo Empleado /Trabajado	Tiempo Promedio Por Entidad
Desarrollar servicios para la vinculación de ingresos (Consumos al generar corte)	No Aplica	2	0	21h	**10.5h**
Servicio para recibir pólizas de otros sistemas (PagoCheck)	No Aplica	2	0	16h	**8h**
Desarrollar servicios para la vinculación de ingresos (Clientes - Clientes, Estaciones - Sucursales)	No Aplica	3	0	18h	**6h**
Adaptación de servicios para la integración de almacenes ERP - SHC	Ágil	0	2	12h 20m	**6h 10m**

En el cuadro anterior se puede observar que, de un bloque de cuatro tareas, en promedio fueron creadas o modificadas dos entidades nuevas a nivel base de datos, de las cuales el tiempo promedio de ejecución oscila entre 6 y 10 horas aproximadamente. Solamente una tarea se trató de modificaciones en servicios existentes, de manera que se realizó un versionado utilizando la estrategia de control de versiones ágil. Es decir, los cambios requeridos solamente se trataban de añadir parámetros opcionales y nuevos métodos.

Estas tareas están documentadas en una plataforma utilizada por la organización para el seguimiento de la ejecución de las tareas que se ejecutan en sprints en base a la metodología Scrum que la empresa actualmente emplea en la parte operativa del desarrollo de software.

El principal objetivo del presente trabajo es la elaboración de una herramienta que permita reducir el tiempo destinado a tareas manuales y repetitivas en la generación, el versionado y mantenimiento de los servicios. Por lo cual, se realizó un conjunto de pruebas y experimentos para que desarrolladores que necesitaran realizar tareas de este tipo, pero con el uso de la herramienta propuesta. A continuación, se presentan los resultados obtenidos:

Cuadro 5.18 Relación de Tiempo Empleado Después del Uso de la Herramienta

Descripción de Tarea	Estrategia de Control de Versiones	Entidades Creadas	Entidades Modificadas	Tiempo Empleado /Trabajado	Tiempo Promedio Por Entidad
Backend de Grupos, Presupuestos y Gastos - Business, Data, Services (Interface Servicio WFC)	No Aplica	11	0	11h	**1h**
Crear Servicios para solicitudes de anticipos	Ágil	3	1	1h	**0.25h**
Crear Servicios para comprobaciones	No Aplica	5	0	1h	**0.20h**
Crear Servicios para operaciones posteriores (devoluciones y reembolso	No Aplica	3	0	3h	**1h**
Servicio de autentificación de usuario	Ágil	2	1	3h	**1h**
Preparar servicio para consulta de grupos desde plataforma WEB	Ágil	0	5	3h	**0.6h**
Crear servicios para plantillas de grupos de gasto	No Aplica	2	0	2h	**1h**

A pesar de que los escenarios son diversos, varían en complejidad y forman parte de diferentes proyectos, se puede observar una notable reducción de tiempo promedio que se dedica para llevar a cabo tareas relacionadas con la generación de servicios y la modificación de los mismos. Lo que beneficia para que lo que puede verse como una línea de productos de software sea más eficiente y mitigue los cuellos de botella que se pudieran presentar en el ámbito de los servicios y su consumo.

Capítulo 6

Conclusiones y Trabajo Futuro

En este último capítulo se presentan algunas conclusiones derivadas de la experiencia del desarrollo de la herramienta propuesta en el presente documento y de los ciclos de pruebas y experimentos realizados. De igual manera, se presentan algunas recomendaciones y oportunidades de mejora para el trabajo futuro.

6.1 Conclusiones

Al finalizar el desarrollo e implementación parcial de la propuesta descrita en el presente documento tenemos como resultado una herramienta que permite a los desarrolladores de la empresa Punto de Desarrollo de Software S.A. de C.V. optimizar el proceso de generación y mantenimiento de las capas de datos, negocios y servicios de cualquier proyecto que necesite que dichos servicios sean consumidos desde cualquier aplicación cliente independientemente de la plataforma en la que sea implementado. Del trabajo implementado podemos considerar tres grandes beneficios para la empresa:

1. **Automatización en Generación de Código:** La generación automática de la capa de datos, negocios y servicios reduce considerablemente el tiempo y esfuerzo que se dedica a este tipo de tareas. Además, representa una oportunidad para aumentar la velocidad que puede alcanzar un equipo de desarrollo de software.

2. **Optimización en Mantenimiento:** Se logra una mejora en facilitar la adopción de diferentes estrategias para controlar las versiones de los servicios generados.

3. **Integración:** El hecho de que la herramienta esté disponible para todos los desarrolladores en el mismo entorno de desarrollo propicia que se adopten fácilmente estándares en relación a una base de código existente.

El código de la herramienta también puede ser modificado en caso de que se deseen realizar mejoras o adaptaciones, ya que se encuentra disponible para la consulta de todos los miembros de la empresa en un repositorio privado.

6.2 Recomendaciones

Se recomienda que se capaciten a todos los desarrolladores que desean utilizar la herramienta propuesta para que identifiquen los diferentes escenarios en los que la herramienta les ayudará a mantener un control de versiones y aquellas situaciones en las que deben realizar pequeñas acciones manuales. A continuación, se describen las diferentes estrategias de control de versiones con los escenarios recomendados a aplicarse según sea el caso:

Cuadro 6.1 Escenarios Recomendados para Aplicación de Estrategias de Control de Versiones

Estrategia de Control de Versiones	**Escenario Recomendado para Aplicarse**
Control de Versiones Ágil	Adecuadas para un entorno que depende en gran medida de compatibilidad con versiones anteriores y debe tolerar cambios frecuentes.
Control de Versiones Estricto	Adecuada para un entorno que se actualiza con menos frecuencia o que requiere un estricto control de cambios.
Control de Versiones Semi-Estricto	Adecuado para un entorno que permite una política de versiones personalizadas que satisfaga las necesidades de la aplicación.

El mecanismo de generación de servicios permite seguir analizando posibles escenarios en los que es posible automatizar el control de versiones estricto y semi-estricto para continuar reduciendo las decisiones que debe tomar el programador, pero que por cuestiones de alcance no fue posible desarrollar en el presente proyecto. Se recomienda ampliamente considerar todos estos escenarios para ser incorporados en el código base de la herramienta propuesta.

6.3 Trabajo Futuro

Existen algunas posibilidades de mejora detectadas a lo largo del desarrollo de este proyecto. Primeramente, transmitir la idea de la optimización de líneas de productos de software para que otros miembros de la organización puedas realizar aportaciones de acuerdo a las dificultades que se les presentan en la ejecución de otro tipo de tareas.

Otra oportunidad de trabajo a futuro es la incorporación de otros modelos de capas de datos y de negocio como Entity Framework, OData y Web API, sin perder la esencia de la automatización en la generación y sobre todo el versionado de código, para poder mantener los servicios de la manera más transparente posible.

Por último, se propone realizar la implementación e institucionalización del uso de la herramienta para todos y cada uno de los desarrolladores involucrados en la generación de servicios. Para ello, se propone que la herramienta esté disponible en el servidor que se utiliza para las aplicaciones virtualizadas de desarrollo y también que la extensión que se incorpora con Visual Studio sea publicada en la página oficial de extensiones de Microsoft para que se pueda descargar e instalar en cualquier versión de dicho programa.

Bibliografía

[1] J. M. C and J. B. A, “Desarrollo de Software Empresarial.”

[2] D. H. H. Ingalls, “The Smalltalk-76 Programming Design and Implementation,” in *POPL ’78*, 1978, pp. 9–16.

[3] M. Lorenz and J. Albrecht, “Object - Relational Mapping Strategies revised – A comparison of Row - and Column - oriented Database Systems,” 2014.

[4] C. Ireland, D. Bowers, M. Newton, and K. Waugh, “A Classification of Object-Relational Impedance Mismatch,” *2009 First Int. Confernce Adv. Databases, Knowledge, Data Appl.*, pp. 36–43, 2009.

[5] S. Deelstra, M. Sinnema, and J. Bosch, “A Product Derivation Framework for Software Product Families,” *Softw. Prod. Eng.*, 2004.

[6] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. 2001.

[7] J. Rodriguez, “Líneas de Productos de Software,” 2007.

[8] L. Fuentes, C. Nebrera, and P. Sánchez, “Feature-oriented model-driven software product lines: The TENTE approach,” in *CEUR Workshop Proceedings*, 2009, vol. 453, pp. 67–72.

[9] M. Voelter and I. Groher, “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development,” *11th Int. Softw. Prod. Line Conf. (SPLC 2007)*, pp. 233–242, Sep. 2007.

[10] D. Ameller, X. Burgués, O. Collell, D. Costal, X. Franch, and M. P. Papazoglou, “Development of Service-Oriented Architectures using Model-Driven Development: A Mapping Study,” *Inf. Softw. Technol.*, vol. 62, pp. 42–66, 2015.

[11] J. I. Panach, N. Juristo, F. Valverde, and Ó. Pastor, “A framework to identify primitives that represent usability within Model-Driven Development methods,” *Inf. Softw. Technol.*, vol. 58, pp. 338–354, 2014.

[12] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Sci. Comput. Program.*, vol. 79, pp. 70–85, Jan. 2014.

[13] R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal, and A. Egyed, "From requirements to features: An exploratory study of feature-oriented refactoring," in *Proceedings - 15th International Software Product Line Conference, SPLC 2011*, 2011, pp. 181–190.

[14] S. Apel, C. Kaestner, and C. Lengauer, "Research challenges in the tension between features and services," in *Proceedings of the 2nd international workshop on Systems development in SOA environments - SDSOA '08*, 2008, p. 53.

[15] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology*, vol. 8. pp. 49–84, 2009.

[16] F. Wedyan, S. Ghosh, and L. R. Vijayasarathy, "An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs," *Inf. Softw. Technol.*, vol. 59, pp. 233–254, 2015.

[17] I. Jacobson and P.-W. Ng, "Aspect-Oriented Software Development with Use Cases," *Australian Family Physician*, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1062430.

[18] R. S. Pressman, *Software Engineering A Practitioner's Approach 7th Ed - Roger S. Pressman*. 2009.

[19] E. Tüzün, B. Tekinerdogan, M. E. Kalender, and S. Bilgen, "Empirical evaluation of a decision support model for adopting software product line engineering," *Inf. Softw. Technol.*, vol. 60, pp. 77–101, Apr. 2015.

[20] D. Cabrero, J. Garzás, and M. Piattini, "Understanding product lines through design patterns," in *ICSOFT 2007 - 2nd International Conference on Software and Data Technologies, Proceedings*, 2007, vol. SE, pp. 405–408.

[21] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed, "A Systematic Mapping Study of Search-Based Software Engineering for Software Product Lines," *Inf. Softw. Technol.*, vol. 61, pp. 33–51, Jan. 2015.

[22] C. Bauer, *Hibernate in Action*. 2005.

[23] S. W. Ambler, "Mapping objects to relational databases," *Softw. Dev.*, vol. 3, pp. 63–72, 1995.

[24] U. Z. Castro, M. Angel, R. Barros, and J. C. Nelson, *Guía de Arquitectura N-Capas orientada al Dominio con . NET*. 2010.

[25] J. H. Canós, P. Letelier, C. Penadés, and D. P. De Valencia, “Métodologías Ágiles en el Desarrollo de Software,” pp. 1–8.