UNIVERSIDAD NACIONAL DE LA PLATA

Facultad de Informática – Posgrado

Formalización con Metodologías MDD de una Propuesta de Framework Enfocada a Soluciones de Procesamiento Transaccional

Ing. Hernán Enrique **ZBUCKI**

Trabajo presentado a la **Facultad de Informática de la UNLP** como parte de los requisitos para la obtención del título de Maestría en Ingeniería de Software

Directora de la Tesis: Dra. Claudia Pons

Codirector de la Tesis: Mg. Javier Bazzocco

Buenos Aires, Febrero de 2016

Resumen

La *ingeniería de software* establece que la construcción de programas debe ser encarada de la misma forma que los ingenieros construyen otros sistemas complejos. Los sistemas de procesamiento transaccional no son la excepción. Para lidiar con algunos de los desafíos de programar estas soluciones, se introduce una propuesta de marco de trabajo, que propone la construcción de una base de conceptos comunes, obtenidos del análisis de soluciones preexistentes, y experiencias de los desarrolladores. Esta obtención de factores comunes se hace de forma iterativa, y se capitaliza en elementos del *framework* que aquí se introduce. En este trabajo de tesis se propone una formalización del *framework* en cuestión, por medio de la implementación de metodologías dirigidas por modelos (MDD). Específicamente se propone la construcción y adopción de un lenguaje de dominio específico (DSL) que contemple los elementos que conforman el marco de trabajo, permitiendo la generación automática de código. De esta forma, se pretende facilitar tanto la reusabilidad y manutención de los sistemas transaccionales que lo adopten, como así también la integración de la experiencia acumulada por los desarrolladores en el dominio.

Abstract

Software engineering establishes that the task of building programs must be addressed in the same way that other engineers build complex systems. Transaction processing systems are not the exception. In order to deal with some of the challenges of programming these solutions, this article introduce a framework, which proposes to build a base of *common concepts*, obtained from the analysis of pre-existent solutions, and experiences of the developers involved. This elicitation of *common concepts* is done iteratively, and its outputs are elements of the framework herein introduced. Therefore, it is given a formalization of this framework, through an implementation of a MDD technique (Model-Driven Development). Specifically it was implemented a DSM methodology which proposes to build a DSL (domain-specific language). This DSL is comprised by the domain elements found on the base framework, and it also enables semi-automatic code building through a translation specific tool. By these means, it is expected to foster reusability of components, decreasing maintenance costs, as well as summarizing collective knowledge of the domain spread through workforce's experience.

A Pablo Carballo por su hospitalidad, profesionalismo y amistad

Contenido

Lista de Figuras

Lista de Tablas

1 Introducción

1.1 Alcance, Motivación y Objetivos del Trabajo

Este trabajo es una propuesta para hacer ingeniería sobre un conjunto finito de soluciones transaccionales preexistentes, a través de la definición de un *marco de trabajo transaccional (framework)* y su respectiva formalización, que se propone realizar utilizando metodologías MDD (*Model-driven development),* específicamente las definidas en el enfoque conocido como DSM (*Domain Specific Modeling*). El *framework* intentará funcionar como *herramienta* para reducir los efectos adversos que se describirán en los capítulos posteriores de este trabajo, inherentes al proceso de creación y mantenimiento de servidores transaccionales. La empresa *PointPay Inc*. (www.pointpay.net), para la cual se construyó la primera versión del *framework,* valoró la definición de un marco de trabajo que se enfocara a la reusabilidad de la mayor cantidad de componentes posibles, la reducción de costos de mantenimiento, y la convergencia de criterios de diseño entre desarrolladores. Con estos tres objetivos en vista, se realizó un trabajo retrospectivo, que requirió la revisión y re-desarrollo iterativo de componentes preexistentes, el agregado de valor ingenieril a cada iteración de re-trabajo, la recolección de factores comunes del dominio, la capitalización de esos factores en elementos base del marco de trabajo, la refactorización a patrones conocidos, y la re-educación de los integrantes de los equipos de desarrollo, al trabajo cooperativo entre subsidiarias, a través del uso del marco de trabajo bajo definición. Este trabajo retrospectivo, que comenzó a mediados de Mayo de 2012, generó la primera versión de *TransactionKernel* (nombre del marco de trabajo). Actualmente esta revisión se sigue realizando de forma trimestral al día de la fecha.

El objetivo primario de esta tesis será formalizar el *framework TransactionKernel* a través de metodologías MDD - DSM. Así se intentará comprobar fehacientemente la siguiente hipótesis*: A partir de un conjunto de conceptos repetibles y pertenecientes al dominio del procesamiento electrónico de transacciones bancarias, de lealtad o similares, es posible generar sistemas de proceso multi-concurrentes de forma semi-automática, que a través de la transformación de modelos cooperativos (de mayor a menor nivel de abstracción), del armado de artefactos, y del soporte de metodologías* MDD*, mejoren tanto el grado de reusabilidad de componentes, como la disminución de los costes de mantenimiento, tasas de errores por línea de código, y los tiempos de entrega del producto (Time to Market)*. Este objetivo tendrá como producto directo la construcción de un DSL (*Domain Specific Language*) que conceptualice los factores comunes relevados de las soluciones preexistentes, dedicado al dominio de procesamiento de transacciones.

Como objetivo secundario se introducirá el marco de trabajo que se quiere formalizar, y se desarrollará un ejemplo de servidor transaccional que permita demostrar el uso y funcionamiento del lenguaje específico de dominio. Así también se describirán los pasos de la construcción, comenzando desde el diagrama vacío y contemplando las transformaciones y la generación semiautomática de artefactos de código reusables. De esta

forma se pretende validar los conceptos comunes relevados y conceptualizados en el DSL, tratando de justificar la utilidad de los mismos, y si realmente serían repetibles en cualquier otra solución de procesamiento electrónico.

El lenguaje de dominio específico será desarrollado en el IDE[1] *Visual Studio 2010* (extendido con el SDK[2] de Virtualización y Modelización llamado DSL Tools).

1.2 Organización del Trabajo

El trabajo se dividirá en siete capítulos, con un anexo complementario. Este capítulo titulado *Introducción* es el primero, donde ya se introdujeron los alcances, los objetivos y las motivaciones del trabajo. También se describirá una breve reseña histórica de los sistemas transaccionales y su importancia en el mundo cotidiano, como en el ámbito de los sistemas informáticos. Para ello se resumirán los puntos importantes recolectados de la bibliografía consultada.

En el segundo capítulo, titulado *Introducción al Dominio Transaccional* se introducen algunos elementos y conceptos del dominio referente al procesamiento electrónico de transacciones en cualquiera de sus variantes. Particularmente se describirán los sistemas y equipos participantes en el desarrollo de una transacción de punta a punta. También se describirán algunas secuencias de trabajo (*workflow*) para el procesamiento de una transacción en diferentes soluciones del mercado, desde el momento en que esta llega al procesador transaccional, hasta que se da por finalizado el proceso.

En el tercer capítulo, titulado *Estado del Arte y Revisión de la Bibliografía* se introducirán algunos trabajos consultados de la bibliografía, que fueron de importancia para la orientación y redacción de éste. Por otro lado, se analizarán conceptos teóricos de MDD, algunos de sus pilares y enfoques (MDA, DSM) y la existencia de marcos de trabajo orientados a facilitar la integración de componentes reutilizables en sistemas de procesamiento. También se verán las herramientas disponibles para implementar una propuesta MDD en un proyecto de desarrollo de sistema de procesamiento, enfocado desde la perspectiva de las herramientas disponibles en entornos *Visual Studio* y tecnologías relacionadas con el *framework Microsoft .NET*.

---

[1] *IDE:* Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación; es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI)

[2] *SDK:* Conjunto de herramientas de desarrollo de software que le permite al programador crear aplicaciones para un sistema concreto, por ejemplo ciertos paquetes de software, *frameworks*, plataformas de hardware, computadoras, videoconsolas, sistemas operativos, etc.

En el cuarto capítulo, titulado *Caso de estudio – Motivación y Desarrollo del Framework TransactionKernel*, se documentarán los puntos principales del *framework*, y se presentará todo el proceso de armado y definición del marco de trabajo en cuestión, haciendo un análisis de las ventajas y desventajas más visibles en términos de diseño y construcción.

En el quinto capítulo, titulado *Implementación de una Propuesta MDD Basada en el Framework TransactionKernel*, se implementará el enfoque DSM a través de un lenguaje específico de dominio (DSL). Este DSL contendrá los conceptos repetibles y característicos (que incluirán tanto a los factores comunes recolectados, como al modelo de clases refactorizado a patrones), para demostrar que es factible la implementación de MDD en el desarrollo de servidores concurrentes de procesamiento transaccional. Esta versión del DSL permitirá únicamente la transformación de instancias del metamodelo a código. El código generado implementará clases abstractas e interfaces de *TransactionKernel,* como se realizaba previamente a este trabajo de forma manual.

En el sexto capítulo, titulado *Evaluación del DSL*, se revisará un ejemplo de sistema de procesamiento transaccional desarrollado bajo *TransactionKernel* para recolectar las ventajas y desventajas de la implementación de esta metodología en el dominio. El objetivo es armar desde cero un sistema transaccional a través del DSL, dadas unas condiciones de negocio típicas en cualquier sistema enrutador de transacciones de crédito tiempo-aire.

En el último capítulo, titulado *Conclusiones y Línea de Trabajo Futuro*, se evaluarán los resultados de dos desarrollos realizados: el primero previo a la implementación de *TransactionKernel*, y el segundo posterior. Se analizarán las eventuales mejoras o desventajas, principalmente en reusabilidad, costos de mantenimiento, y homologación de criterios entre desarrolladores. También se resumirán las conclusiones y *feedbacks* obtenidos de los *stakeholders*. Como finalización, se redactarán las conclusiones pertinentes y los eventuales pasos a seguir en trabajos futuros.

Como complemento del trabajo se añadirá un anexo en donde se graficará el diagrama de flujo de la secuencia genérica de trabajo propuesta, que se describirá en el capítulo 4.

1.3 Temas de Investigación

El trabajo se realizará siguiendo una metodología de Investigación y Desarrollo (I+D). Los temas a investigar serán:

- Desarrollo de DSL con *DSL Tools* en *Visual Studio 2010 – Visualization and Modeling SDK*.
- Patrones de diseño: *Singleton*, *Facade*, *Strategy, Template Method, Factory Method, y Chain of Responsibility.*

- Transformaciones entre modelos, y propuestas MDD: MDA, DSM.
- Generación de código automática mediante *Text Templating* (.tt)
- Arquitectura y diseño de sistemas de procesamiento actuales.
- Desarrollo con técnicas *Multi-threading*.
- Metodologías TDD

1.4 Presentaciones en Congresos y Workshops

Los temas investigados y desarrollados que conforman a esta Tesis de Maestría fueron elaborados como artículos para ser presentados en CACIC 2014, en WICC 2015 y en CACIC 2015.

Para el CACIC 2014 se presentó el artículo *Una Propuesta de Marco de Trabajo Orientado al Dominio del Procesamiento Transaccional - Definición de una Secuencia de Proceso Genérica y Redefinible*. Luego para el WICC 2015 fue presentado otro artículo a modo de continuación del presentado en CACIC 2014, titulado *Una Propuesta de Marco de Trabajo Orientado al Dominio del Procesamiento Transaccional* dentro de la categoría *Ingeniería de Software*. Este *workshop* fue celebrado en la ciudad de Salta, Argentina, dentro de las inmediaciones de la Universidad Nacional de Salta (UNSa). Allí se presentó una cartulina en la reunión de expositores durante los días 16 y 17 de Abril de 2015. Por último, se presentó un artículo para el CACIC 2015 que fue aprobado por dos de los tres revisores (las notas de las evaluaciones fueron 5, 9 y 7 respectivamente). El artículo se llama *Una Propuesta de Implementación MDD y TDD en el Dominio de Sistemas de Procesamiento Transaccional.*

1.5 Reseña Histórica

Los sistemas de procesamiento de transacciones de autorizaciones de tarjeta de crédito / débito son unos de los más usados y de los más sofisticados dentro de los sistemas de información de gran escala. Un *CAS*[3] debe manejar varios tipos de transacciones simultáneas, y en gran volumen por unidad de tiempo, tales como compras, autenticaciones, transferencias, devoluciones, balances, promociones, etc. [1] Si bien estos sistemas también sirven para procesar transacciones de diversas índoles, el mayor de los usos está dado por el procesamiento financiero.

Durante la década de 1920, fueron puestas en el mercado las primeras tarjetas de crédito como tarjetas propietarias de algunas compañías petroleras que deseaban aumentar sus ventas dando créditos a sus

[3] *CAS:* Acrónimo inglés para el término *Card Authorization System*, o Sistema de Autorización de Tarjetas.

consumidores habituales. El procesamiento de esas tarjetas era realizado puramente en papel (sin ningún tipo de asistencia computarizada), dificultando el control y haciendo a la tarea propensa al fraude. A medida que los fraudes y las cuentas de crédito falsificadas se incrementaban, muchas de las empresas emisoras de tarjetas publicaban una lista periódica de los números de cuenta de tarjetas de crédito que serían "vedadas". Desafortunadamente este sistema de *lista negra primitivo* estaba plagado de errores y era tedioso para mantener al día, y para distribuirlo en tiempo y forma a los comercios. [2]

La adopción de sistemas de procesamiento y de tarjetas de crédito fue paulatinamente en aumento durante todo el resto del siglo XX. Durante la década de los 80', se expandió exponencialmente, particularmente porque las tecnologías avanzaron lo suficiente como para poder implementar de forma cada vez más automática el control de las cuentas de los clientes, en un sistema de crédito bancario. De esta forma las tarjetas de crédito comenzaron a surgir como un dispositivo de pago en reemplazo del dinero en efectivo o de cheques, tanto para millones de compras de rutina, como para muchas otras transacciones que de otro modo serían inconvenientes, o tal vez imposibles de realizar de forma manual. [2] El auge de las tarjetas surgió principalmente debido a la llegada de las terminales electrónicas de bajo costo, siendo este hito la piedra angular para lograr que el proceso de autorización se realizará electrónicamente en su totalidad. En esa década también se introdujo un protocolo estándar de comunicación para facilitar el logro de este propósito, la norma ISO 8583, si bien no es el único protocolo financiero de adopción generalizada. ISO 8583 es un estándar de la Organización Internacional de Normalización (ISO), que especifica el formato de los mensajes de intercambio para las transacciones electrónicas que involucran tarjetas de crédito / débito [3], realizadas desde los comercios, cajeros automáticos y cualquier otro equipo lector de tarjetas. Estas terminales basadas en micro-controladores se comenzaron a instalar en los comercios, y garantizaron el proceso seguro de todas las transacciones que fueran autorizadas a través de ellas.

Intrínsecamente, con el crecimiento del parque instalado de terminales, y de usuarios que preferían estos pagos digitales al pago en efectivo, comenzaron a desarrollarse una serie de problemáticas asociadas a la *performance* y la *seguridad*. Los sistemas de procesamiento siguen teniendo en la actualidad problemáticas asociadas con relación a estos dos tópicos, siendo las relacionadas a cuestiones técnicas las que interesan en este trabajo. Durante el desarrollo de estos capítulos, se irán enumerando para facilitar la comprensión del contexto al lector.

1.6 Resumen del Capítulo

En este capítulo se revisaron los siguientes temas:

- Se definieron los alcances, objetivos y motivaciones para la realización de este trabajo.
- Se realizó una descripción del formato y organización del mismo. Se definieron algunos de los temas de investigación para analizar y estudiar.

- Se introdujo una reseña histórica sobre el dominio del procesamiento transaccional, de modo de intentar dejar una noción al lector del contexto histórico del mismo.

2 Introducción al Dominio Transaccional

2.1 Introducción

En este capítulo se exponen algunos puntos de importancia en el dominio, principalmente las características de los elementos y de los equipos usados en el procesamiento de transacciones. También se describirán algunas problemáticas asociadas a la *performance* y la *seguridad* de los procesadores de transacciones. Por último se introducen dos secuencias de trabajo (*workflow*) de servidores transaccionales, una para el caso de procesamiento de solicitudes financieras (de crédito y débito), y la otra de recarga de tiempo-aire para crédito de telefonía celular.

2.2 Problemáticas Asociadas con los Sistemas de Procesamiento Electrónico

Más allá del dominio de aplicación, los puntos más delicados de un sistema de proceso de transacciones (particularmente los de tarjetas de crédito/débito), son la *performance* y la *seguridad*. Respecto a la *performance*, el factor primordial es el tiempo que lleva autorizar y completar una transacción de punta a punta, mientras que en el aspecto de *seguridad* el tema importante es la prevención del fraude y la confidencialidad de la información financiera. Ambos requerimientos están ligados: Los procesos de verificación, autenticación y criptografía, que sirven en pos de la seguridad de cualquier transacción, reducen el rendimiento y el tiempo de respuesta. Además se suma el efecto del aumento del volumen de transacciones a medida que se expande la red transaccional. Esto genera una situación donde se denota un consumo incremental de recursos computacionales, ya sea el porcentaje de uso de los procesadores de los servidores, o los tiempos de acceso a mecanismos de persistencia. Como regla general la *performance* de los sistemas de autorización se ve afectada cada vez que la cantidad de procesos de autorizaciones concurrentes aumenta, generalmente en horas pico, pero también en fechas festivas. Éstas, entre otras causas más, generan que un conjunto de transacciones simultáneas en un determinado punto singular en el tiempo no puedan resolverse en la ventana temporal prevista (muchas veces en el tiempo que el usuario desea esperar). En esos casos la transacción termina en estados categorizados como *Time-Out*[4] [1].

[4] *Time-Out:* Es un estado transaccional usado frecuentemente, que sucede cuando se vence la ventana de tiempo para esperar por una respuesta desde un nodo computacional remoto.

Otra problemática común en estas soluciones transaccionales es la *frecuencia de actualización del sistema* (nuevas reglas de negocio que generan cambios en los algoritmos), y la *capacidad de mantenimiento* de los componentes dentro de ellos. Los gobiernos, los bancos y otras entidades interesadas crean, y/o modifican normas y reglas de negocio recurrentemente, para satisfacer uno o varios objetivos corporativos. La motivación a estos cambios está ligada a la competencia entre compañías y bancos, de modo que son presionadas para ofrecer nuevos servicios o modificar los servicios existentes con frecuencia. [2] Estas situaciones causan constantes revisiones de los sistemas de autorización, aumentando la complejidad de mantenimiento de los mismos.

Volviendo a los *CAS,* la mayoría de los sistemas de autorización de tarjetas de crédito que están usando los bancos tienen más de 15 años de edad, con muchos parámetros *hard-coded*[5], orientados a configuraciones rígidas, y se requiere mucho tiempo para generar cambios en los códigos fuente. [1] Muchos de estos sistemas están en su capacidad máxima de procesamiento, lidiando para mantenerse en línea con el aumento en el volumen de pagos con tarjeta. Además suelen tener déficit de reglas de negocio empotradas y diagramas secuenciales de trabajo bien definidos, que resultan en operaciones ineficientes. [1]

A nivel arquitectura de software, muchos servidores concurrentes que todavía se construyen en el modelo de un solo subproceso (*single-threaded*) por transacción entrante, precisan de más tiempo para responder que a aquellos servidores que usan múltiples subprocesos (*multi-threaded*) por transacción entrante. [1] Poder usar una arquitectura o la otra, depende si es posible paralelizar las tareas necesarias para realizar una determinada transacción. Por lo contrario, esas tareas deberán realizarse de forma secuencial, y un solo subproceso por transacción sería suficiente, aunque el tiempo para generar una respuesta será mayor. Un sistema con subprocesos múltiples permite realizar tareas de procesamiento de forma simultánea, ahorrando *tiempos muertos*[6] mientras se espera respuesta de una tarea, o de lectoescrituras a dispositivos de E/S, ya que se reutilizan los recursos computacionales para realizar otras tareas en paralelo. De esta forma se incrementa la velocidad del caudal de salida del servidor, y también se reduce la cantidad de *Time Outs*. Por consecuencia, también terminan reduciendo los costos tarifarios de las comunicaciones. [1] Como ejemplo de lo expuesto recién, para realizar la autentificación de las cuentas de crédito en los *CAS*, se deben realizar tareas de validación de restricciones de tarjetas y validaciones de fraude en línea. Esas tareas básicamente realizan operaciones

---

[5] *Hard-Coded:* Es un término que describe la capacidad de parametrización de un sistema, sin necesidad de recompilar el código fuente para realizar cambios en esos parámetros.

[6] *Tiempos Muertos:* Describe todos aquellos tiempos donde un contexto de ejecución queda en estado "dormido" (el procesador lo quita de la cola de ejecución) esperando por el resultado de una operación computacional, generalmente de larga duración. Un ejemplo es la lecto-escritura de dispositivos de E/S.

criptográficas contra dispositivos de seguridad asociados al servidor (estos dispositivos criptográficos se llaman *HSM*), y que son costosas en tiempo. [3] Si estas tareas no tuvieran dependencia de datos, podrían paralelizarse en subprocesos hijos (*child threads*) para reducir la carga de trabajo en varios hilos de procesamiento. De todas formas los procesos basados en un solo subproceso por transacción (procesamiento secuencial) pueden resultar con mediciones de performance satisfactorias en función de la carga transaccional esperada. [1] Pero a medida que la carga se incremente, se comenzarán a ver problemas de rendimiento, obligando a los diseñadores a re-analizar cualquier tipo de optimización.

Los desarrolladores dedicados al dominio en cuestión están bajo un dilema importante ya que deben mantener un equilibrio delicado entre la calidad, la expectativa de vida del procesador transaccional y los costos de producción presupuestados. Para mantener el sistema funcionando, y a la vez, cumpliendo con las nuevas reglas de negocio o de servicio, sufren evoluciones constantes que muchas veces incurren negativamente en la calidad de la solución integral.

2.3 Elementos del Dominio Transaccional

En esta sección se introducen algunos de los elementos intervinientes en el procesamiento de transacciones y dos secuencias típicas usadas en sistemas de tarjetas de crédito / débito, y en sistemas para recargas de crédito tiempo-aire. Muchos de los elementos analizados a continuación podrían clasificarse de distintas formas; en este trabajo se destaca una distinción entre estos elementos en al menos tres capas: La *capa de adquisición* (*Acquirement*), la *capa de enlace* (*Gateway*) y la *capa de administración* (*Management*). Cabe acotar que otros autores pueden enumerarlas de distintas formas, sin embargo esto es tan solo una propuesta y se decide en este trabajo conceptualizarlas, agruparlas y nombrarlas de esta forma. Se pueden apreciar las capas y sus medios de interacción en la Figura 1:

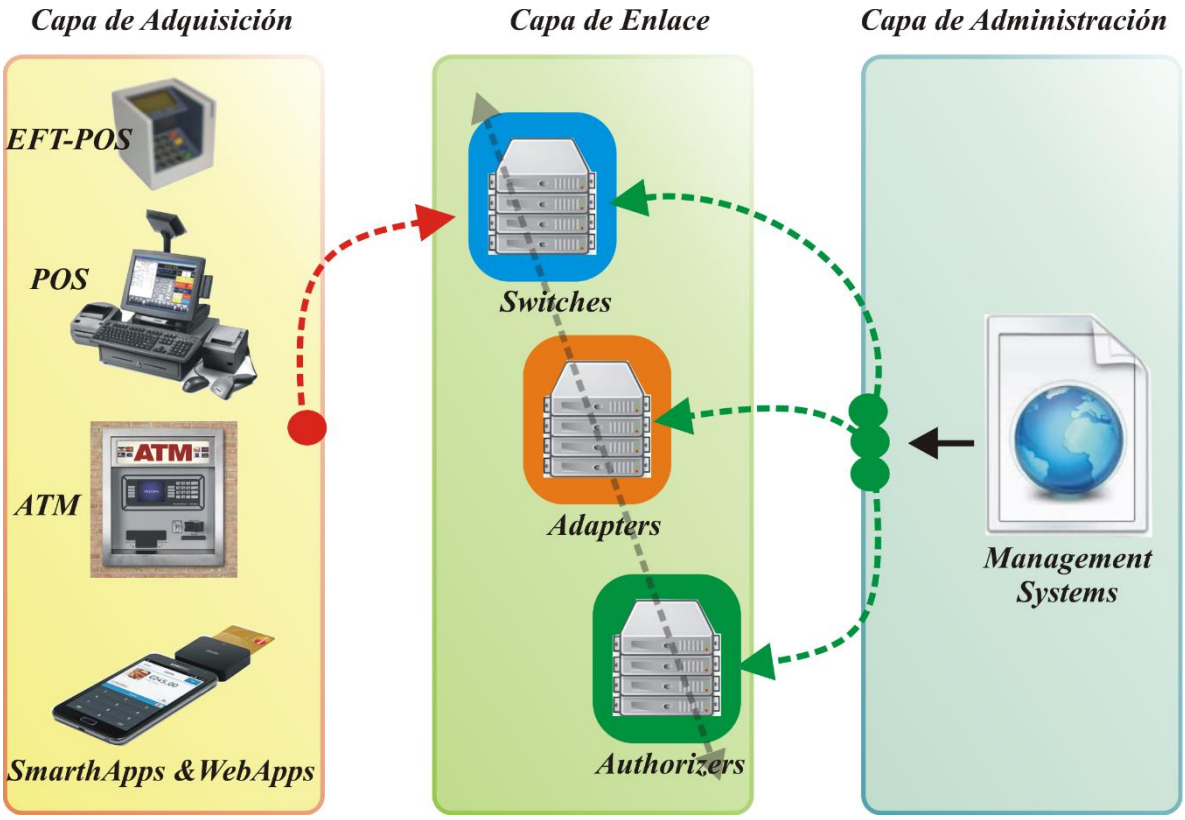

Figura 1- Clasificación de elementos del dominio transaccional según esta propuesta.

2.3.1 Capa de Adquisición

La capa de adquisición está constituida por todos los dispositivos de *hardware* y componentes de *software* que recolectan datos de un usuario, para luego digitalizarlos y enviarlos como una transacción hacia un nodo remoto. A continuación se describen algunos de ellos.

2.3.1.1 Punto de Venta – POS

Los POS, del inglés (*Point of Sale*) son los lugares donde las transacciones minoristas son realizadas. Es decir, es el punto en el que el cliente hace el pago al comerciante por el intercambio de bienes y servicios. También se llaman EPOS, dada la evolución a equipos electrónicos para facilitar el proceso de pago. A través de los puntos de venta, el comerciante minorista calculará el monto adeudado por el cliente, y le proveerá diferentes opciones de pago para poder saldarlo. Como comprobante de la operación, el comerciante generalmente imprime un ticket con los detalles de la misma.

Figura 2- Punto de venta típico en mostradores de comercios minoristas

Según los requerimientos del negocio, los POS también pueden conectarse con piezas de *hardware* específicas para determinados fines. Por ejemplo, los comerciantes pueden utilizar los POS conectados con balanzas, *scanners*, cajas registradoras, terminales de punto de venta (TPV / EFTPos), *touchscreens*, digitalizadores de firma de cliente, entre otros. De hecho un punto de venta minorista consiste típicamente de una caja registradora, de una CPU, monitor, caja de dinero, impresora de comprobantes, y lector de código de barras. Así también incluyen TPV integradas (EFT-POS) para incluir la lectura segura de tarjetas de débito / crédito. Los POS más modernos usan tecnologías *touchscreen* para los monitores, y los CPU vienen incorporados al chasis del monitor (*All-in-one*) para liberar la mayor cantidad de espacio sobre el mostrador del comercio. Las transacciones típicas son ventas, devoluciones, intercambios, emisión de *Gift Cards*, administración de programas de lealtad, promociones, descuentos, entre otras. Los POS ofrecen también diversos medios de pago, planes en cuotas, manejo de moneda extranjera, cuponeras de descuentos, entre otros *amenities*. Además el *software* de los POS puede ser personalizable según el comercio, de modo de facilitar a los cajeros o comerciantes el proceso de cobro según el rubro el negocio. La industria minorista es una de las usuarias predominantes de los *puntos de venta* de este tipo, y generalmente se encuentran en cadenas de farmacias, en cadenas de locales de ropa, y en cadenas de comida rápida.

2.3.1.2 Datáfonos / Terminales de Punto de Venta (TPV) / EFT POS

Una terminal de punto de venta (TPV) o datafono es un tipo de punto de venta (POS) que permite generar transacciones con tarjetas de crédito / débito, o cualquier otro tipo de transacción desde un comercio. Es un sistema electrónico de pago que involucra la transferencia de fondos de forma electrónica desde una cuenta cliente a la cuenta del comercio a través del uso de tarjetas de crédito / débito leídas desde estas terminales. En

Argentina son conocidas con el nombre de *POSNet*; el nombre en realidad pertenece a una de las empresas líderes de adquisición de transacciones, aunque terminó asociado de esta forma al vocabulario coloquial de la gente. En el mundo también se los conoce como EFT-POS dadas las siglas (*Electronic Funds Transfer Point Of Sale*). El objetivo de estos equipos es habilitar a un comercio la lectura de tarjetas bancarias (o en su defecto el ingreso manual de los datos necesarios), para transmitir la información del cliente a una entidad adquiriente de transacciones, y finalmente realizar la transferencia de fondos al comerciante como intercambio de un bien o un servicio prestado. A diferencia de los POS descritos en la sección anterior, los TPV fueron concebidos como *dispositivos seguros standalone,* de fácil distribución para comercios individuales. Estos POS están compuestos por una serie de dispositivos que vienen empotrados en un equipo relativamente pequeño, de bajo costo, de modo que permitan facilitar a cualquier comerciante la posibilidad de realizar cobros electrónicos. Además deben cumplir reglamentaciones para asegurar un camino fiable en el enrutamiento de transacciones financieras.

Los modelos más nuevos de terminales TPV no solo procesan tarjetas de crédito y débito, sino que también se utilizan en el proceso de *Gift Cards,* cheques, tarjetas de programas de lealtad, pagos de servicios, carga de crédito para sistemas de transporte, estacionamiento medido, carga de crédito para telefonía celular, loterías, entre otras variantes de procesamiento. La mayoría de estas terminales transmiten las transacciones a través de la línea telefónica, y las más modernas lo hacen a través de internet, ya sea por WIFI, redes cableadas o GPRS/EDGE/3G. Las terminales a su vez pueden ser portables (usan una batería recargable como fuente de alimentación), siendo ideales para comercios como restoranes o estaciones de servicio; en esos casos las comunicaciones obviamente son a través de interfaces inalámbricas.

La tecnología EFT-POS se originó en Estados Unidos en 1981, y se implementó durante 1982. Originalmente estas terminales estaban asociadas solamente a servicios financieros de un solo banco (es decir que solo procesaban tarjetas bancarias del banco que instalaba la terminal en el comercio), lo que produjo que la aceptación de estos sistemas en el público fuera a un ritmo inicialmente lento. Subsecuentemente, con el fomento de las redes interbancarias surgió un crecimiento en el consumo de los servicios financieros ofrecidos a través de las TPV. Actualmente el grado de adopción en el público del pago con tarjeta va en aumento, y en muchos países éste paso a ser el modo pago preferencial de los consumidores con respecto al efectivo.

Los TPV se componen de una componente de *hardware* (dispositivos físicos) y otra de *software* (sistema operativo y aplicaciones).

Figura 3 - Terminal de punto de venta del tipo *Xenta* (Marca *Atos WorldLine*)

La capa de *software* de estas terminales generalmente es un sistema hecho a medida, es decir realizado para un fin determinado y para una empresa en particular. No así su sistema operativo, ya que muchas suelen integrar *Linux* o *Android* (de todas formas algunos TPV vienen con sistemas operativos propietarios). Esa especificidad en la capa de aplicación viene acompañada de un costo más elevado que un componente de *software* propósito general. Así mismo las eventuales modificaciones o actualizaciones van siempre ligadas a la disponibilidad de la empresa que desarrolla ese software. Otro enfoque es el de generar aplicaciones base para distintos tipos de comercios, de modo de abarcar características comunes según el rubro o industria para el cual se piensa la aplicación. De este otro modo los costos pueden reducirse, con las ventajas y desventajas de comercializar aplicaciones *enlatadas* o *llave en mano*.

Con respecto al hardware, los TPV consisten de lectores de banda magnética y/o lectores de tarjeta CHIP / EMV, de una impresora térmica, de un teclado numérico seguro (más diversas teclas de función), de un *display* seguro, un sistema computacional de control (microcontrolador y memoria), de un cripto-procesador, y de diversas interfaces de comunicaciones (*RS-232, X25, Dial-up modem, Ethernet, USB, GPRS, WiFi*, *Bluetooth,* etc.). Las interfaces de comunicación permiten a los TPV expandir la gama de dispositivos externos que pueden interactuar con las terminales. El puerto RS-232 es de gran versatilidad y es el más usado para la interacción con otros dispositivos de forma local. Este puerto permite a los TPV interactuar con una gran variedad de sistemas específicos, desde cajas registradoras hasta transformadores diferenciales RS-485 para administrar surtidores de GNC y combustibles líquidos, o por ejemplo pistolas lectoras de código de barras para loterías.

Muchas veces por incompatibilidad de normas de seguridad, los TPV deben conectarse con PINPADs de modo de permitir el ingreso seguro de datos del usuario (como pines de tarjetas) por afuera del TPV.

Como conclusión, estos terminales de punto de venta resultan ser equipos muy versátiles para cualquier dominio de negocio que requiera la implementación de transacciones electrónicas. La personalización de los desarrollos a nivel aplicativo, y la amplia gama de dispositivos que pueden interactuar con estas terminales fomentan esta situación, por lo que no solo quedan ancladas a soluciones de índole financiera, sino a aplicaciones multi-rubro.

2.3.1.3 ATM (Cajeros Automáticos)

Los cajeros automáticos (en inglés ATM por *Automated Teller Machine*) son dispositivos de telecomunicación electrónicos que facilitan la realización de transacciones financieras a usuarios finales, en general con la condición de negocio de estar afiliados a una entidad bancaria por medio de una cuenta. La inclusión de estos equipos produjo un cambio beneficioso para los usuarios en el proceso bancario, ya que estos permitían emitir transacciones sin la necesidad de un cajero humano y/o contador. Esto trajo aparejado una reducción de tiempos para el proceso realizado, desde el punto de vista del usuario final. Así mismo redujo los costos para las entidades bancarias. Se estima que actualmente existen en el mundo 2.2 millones de cajeros automáticos activos.

En los cajeros actuales, los clientes son identificados mediante el ingreso de una tarjeta plástica, ya sea por el lector de banda magnética, o el lector CHIP. La tarjeta contiene un código único de identificación de tarjeta (el número de tarjeta), y además contiene información de seguridad como la fecha de vencimiento o el CVVC (CVV). La autenticación de usuario se realiza mediante un código PIN que el mismo debe ingresar previo a iniciar cualquier operación con el cajero.

La gran mayoría de los cajeros están conectados a redes interbancarias, permitiendo a los usuarios retirar y/o depositar dinero desde equipos que no pertenezcan al banco donde la cuenta está radicada. Inclusive el usuario puede realizar transacciones desde el exterior y retirar / depositar dinero en moneda local. Algunos ejemplos de estas redes interbancarias en Argentina son *Cirrus, Link* y *Banelco*. Así mismo, los clientes pueden acceder a sus cuentas bancarias de modo de poder realizar transacciones, como por ejemplo consultas de saldo, retiros de dinero, depósitos de dinero, administrar claves de acceso vía teléfonos inteligentes o *Home Banking*, y cargar crédito en las líneas de sus celulares, entre otras.

Los cajeros delegan las autorizaciones de las transacciones financieras a las entidades emisoras de las tarjetas. El cajero puede acceder a los emisores y adquirientes gracias a la red interbancaria, y el intercambio de mensajes se realiza generalmente bajo el protocolo ISO8583. Generalmente, previo al ruteo de la transacción hacia esos destinos, los cajeros se conectan contra un equipo *host* o un *ATM Controller* vía ADSL o por *dial-up* sobre línea telefónica. Los protocolos de bajo nivel utilizados para este enlace son generalmente *TCP/IP* sobre *Ethernet, X.25 y SNA* sobre *SDLC*. Además de las técnicas de seguridad y secreto que se implementan sobre las

transacciones, la comunicación entre el cajero y el procesador final puede estar total o parcialmente encriptada por métodos y/o algoritmos como *SSL, DES, 3DES*, *RSA,* etc.

Un cajero está generalmente constituido por estos dispositivos:

- CPU (para controlar la interfaz de usuario y los dispositivos de E/S)
- Lectora de banda magnética o lectora CHIP.
- Un PIN PAD para el ingreso de datos sensibles. Debe ser fabricado para que no pueda ser manipulado desde el exterior, y debe cumplir con normas de seguridad PCI. Puede implementarse en pantallas táctiles o con teclados piezo-eléctricos.
- Un cripto-procesador, para el manejo de algoritmos criptográficos. Se usa por ejemplo en los dispositivos de entrada que requieren seguridad, como el PIN PAD, o las lectoras de tarjeta.
- Un *display*.
- Una impresora de comprobantes.
- Sensores en general, para aumentar la seguridad.
- Una bóveda para contener los billetes, o cualquier otra nota bancaria.

Debido a la demanda computacional creciente de procesamiento, y a la reducción de precios de las arquitecturas de tipo PC, los cajeros han pasado de estar armados con *hardware* personalizado (con microcontroladores y circuitería de aplicación específica), a adoptar *hardware* de computadoras hogareñas (por ejemplo se adoptó el protocolo USB para la conexión con algunos periféricos, o incluso Ethernet e IP como protocolos de comunicación). Más aún, se han implementado sistemas operativos de PC dentro de cajeros. Como desventaja a este enfoque, los cajeros se volvieron potencialmente susceptibles a errores típicos de PC, ya sean provenientes del *hardware* o del *software*.

Actualmente se estima que la mayoría de los cajeros a nivel mundial usan sistemas operativos de la familia *Microsoft Windows*, mayoritariamente *Windows XP Professional* o *Windows XP Embedded*, aunque históricamente se han usado *Windows NT, Windows CE* o *Windows 2000*.

2.3.1.4 Soluciones *Web* y *Smartphones Apps (e-commerce)*

Dentro de esta categoría se encuentran los sistemas de comercio electrónico, o como se los conoce en inglés con el vocablo *e-commerce*. Un *e-commerce* es un sistema de *software* que facilita la aceptación de pagos para transacciones en línea. Los sistemas de pagos basados en *e-commerce* se volvieron populares principalmente por el uso extensivo y creciente de compras por Internet, y por el uso de los sistemas de *home banking*.

Es un hecho que hoy en día se puedan realizar pagos por Internet, y se convirtió en una moda gracias al uso extensivo de celulares inteligentes. Estos fomentaron aún más el uso de estos sistemas, implementados como

aplicaciones de pago, ya sea de bancos o de sitios de compra-venta de bienes y servicios. Una de las mayores ventajas que la *World Wide Web* puede ofrecer a sus usuarios es la habilidad de poder transportar el negocio de cualquiera, a cualquier parte del mundo. Y es por esto que los sistemas *e-commerce* son de tal importancia: a través de ellos se puede realizar cobros desde cualquier latitud y longitud. Actualmente no solo es fácil hacer pagos, sino que también es un medio considerado seguro, si bien al principio no eran ampliamente aceptados por el público por miedo a perder el dinero, o por ser simplemente una novedad. Los medios de pago mayormente adoptados por los *e-commerce* son las tarjetas de crédito, las tarjetas de débito y las transferencias bancarias.

A nivel implementación, para este trabajo se categorizan los sistemas *e-commerce* en dos grandes familias: las aplicaciones web y las aplicaciones para teléfonos inteligentes, tabletas, PC's etc. Para ambos, el requisito esencial es la seguridad de la transacción que procesarán, y la facilidad para realizar las transacciones desde el punto de vista del usuario del aplicativo.

Por el lado de la seguridad, los sistemas deben contemplar el uso de criptografía para codificar el medio (adopción de SSL por ejemplo), o para codificar el mensaje (por ejemplo con claves asimétricas RSA públicas y privadas). Así mismo deben realizar certificaciones para homologar los requerimientos pedidos por los bancos y/o adquirientes con los que trabajarán para el procesamiento de una transacción. Muchos de estos sistemas pueden adoptar protocolos de mensajería financieros, como ISO8583.

Así también se han desarrollado dispositivos con el fin de habilitar a una gama de comerciantes la posibilidad de procesar tarjetas de crédito y débito, sin necesidad de contar con un *EFT-POS* propiamente dicho. Estos comerciantes muchas veces no llegan a poder alquilar un servicio de pago electrónico que les incluya un TPV, ya sea por razones de costo-beneficio, o porque trabajan ambulatoriamente. Uno de los dispositivos que se está comenzando a usar es el lector portátil de tarjetas bancarias, compatible con teléfonos celulares inteligentes. De este modo, utilizando un software desarrollado para sistemas operativos *Android*, *IOS*, o cualquier otro, el usuario puede, de forma segura, cobrar a sus clientes por medios electrónicos a un costo mucho menor que el de los servicios de procesamiento convencionales.

Figura 4 - Lector de tarjetas CHIP USB para *smartphones*

2.3.2 Capa de Enlace (*Gateway*)

Dentro de esta categoría se encuentran todos los sistemas transaccionales que se caracterizan por recibir transacciones desde un nodo externo (emisor). A su vez, estos pueden reenviar las transacciones entrantes a otros nodos externos (receptor), o por lo contrario pueden tomar la decisión de aprobar o rechazar la solicitud recibida. Cabe acotar que si se reenvía la solicitud hacia otro nodo externo, ese nodo externo también será un sistema transaccional de esta categoría, que podrá resolver la solicitud por sus propios medios, o volverá a reenviarla creando una cadena de procesamiento transaccional. Según la clasificación propuesta para este trabajo, los elementos de esta capa se pueden categorizar en estos tres tipos: *concentradores, autorizadores* y *adaptadores/ puentes*.

2.3.2.1 Concentradores (*Switches*)

Los concentradores o *switches* son sistemas transaccionales, generalmente implementados como servidores concurrentes, que se caracterizan por escuchar peticiones (transacciones) desde un *handler* (algún puerto físico o lógico dentro del equipo donde están instalados, por ejemplo *sockets, pipes, named pipes, message queues*) para el correspondiente procesamiento transaccional de la solicitud entrante. Llevan el nombre de *concentradores* porque los puntos de venta (POS, EFT-POS, etc.) dentro de una red transaccional envían las peticiones directamente a estos sistemas, y consecuentemente, los *switches* las concentran para poder resolverlas adecuadamente. Es por esto que los *switches* generalmente son considerados puntos de convergencia dentro de una red de POS distribuidos.

Si bien todos los sistemas transaccionales dentro de la *capa de enlace* son *concentradores* (es decir, todos escuchan peticiones multipunto), un *switch* tiene la responsabilidad pura de concentrar peticiones desde una red de puntos de venta, realizar pre-validaciones y post-validaciones sobre las transacciones entrantes desde ellos y redirigir convenientemente las solicitudes a los autorizadores para el producto o servicio que se quiere adquirir con la transacción en curso.

Para describir mejor las funcionalidades de los *concentradores*, se introduce al lector a un ejemplo real de un *switch* de recarga de crédito para telefonía celular. Primero el servidor concurrente escucha por un puerto TCP las conexiones entrantes y concentra las solicitudes provenientes desde los puntos de venta distribuidos. Luego realiza pre-validaciones para poder descartar aquellas transacciones que no cumplan con las reglas de negocio mínimas para comenzar con el proceso de autorización; dentro de esas pre-validaciones se pueden encontrar la verificación del comercio (que este habilitado o que exista), la verificación de la terminal, que el comercio tenga saldo en su *bolsa de saldo* para poder cargar el crédito al celular del usuario final, etc. Luego, si cumple con las pre-validaciones, tratará de redirigir la solicitud al autorizador de la operadora de telefonía del cliente para la compra del crédito en cuestión. En algunos casos, además de redirigir solicitudes hacia otros autorizadores (nodos externos), el *switch* puede aprobar/rechazar algunas transacciones por sus propios medios, es decir sin redirigirla a ningún otro nodo. Este caso se da por ejemplo cuando el punto de venta solicita una consulta de saldo: en esos casos el *switch* conoce esa información accediendo a su propia base de datos sin necesidad de recurrir a ningún autorizador. Finalmente, con el resultado de la operación (ya sea obtenida desde un autorizador externo o de sí mismo), el *switch* le responde al punto de venta a través del puerto TCP, y realiza eventuales post-validaciones (mantenimientos en la cuenta del comercio, actualización de saldos, etc.).

2.3.2.2 Autorizadores (*Authorizers*)

Los autorizadores son también servidores concurrentes que escuchan peticiones y solicitudes por un *handler* habilitado del equipo (probablemente puertos TCP / UDP, pero pueden ser también *pipes, named pipes* o cualquier otra interfaz de E/S). De hecho también concentran transacciones, pero se caracterizan por ser las instancias finales de resolución de una transacción. Es decir que la gran mayoría de las peticiones entrantes no serán redirigidas (salvo casos específicos) porque el autorizador tiene toda la información necesaria para poder aprobar o rechazar una transacción.

Dentro de esta categoría se encuentran los servidores concurrentes usados para procesar transacciones de crédito y débito. Si bien la red transaccional puede ser ampliamente distribuida en nodos, el aprobar o rechazar las transacciones financieras requiere que el nodo responsable de esta última tarea cuente con requisitos operativos preferenciales con respecto a los otros nodos de la cadena transaccional. Los autorizadores entonces deben

cumplimentar normas de seguridad estrictas, deben cumplir con *SLA* [7]estrictos en materia de calidad de servicio (que corra bajo configuración de redundancia 24 horas, los 7 días de la semana, los 365 días del año), y generalmente están situados en centros de cómputos importantes y sofisticados. Los responsables de estos sistemas generalmente son las empresas *bancarias* o las empresas *adquirientes* que directamente brindan el servicio al público. Por ejemplo, en el caso de las redes de recarga de tiempo-aire de telefonía celular, estos autorizadores finales pertenecen a las operadoras de telefonía, mientras que *aguas abajo,* se encuentran *switches* y adaptadores (*bridges*) para ir adaptando distintos puntos de distribución desde el operador principal.

2.3.2.3 Adaptadores / Puentes (*Bridges*)

Los adaptadores no difieren mucho de los *switches* en la tarea de servir concurrentemente a un conjunto de puntos de ventas distribuidos: estos también escuchan por un *handler* disponible del sistema en el que son ejecutados. Sin embargo, el objetivo principal de estos *bridges* es adaptar las distintas redes de recarga aisladas, para que sean compatibles con un protocolo específico, y así sumarse a una red más amplia de puntos de venta. En otras palabras, los adaptadores transforman el protocolo brindado por un proveedor para la adquisición de un producto determinado. De este modo la red de recarga que implementa el adaptador pueda vender tal producto y sumarse ampliando la red de recarga del proveedor del producto.

Generalmente los adaptadores no trabajan con puntos de venta directamente, ya que no es la responsabilidad de los adaptadores realizar pre-validaciones o post-validaciones sobre las solicitudes entrantes. En cambio trabajan en conexiones H2H (*Host to Host*), es decir entre un *switch* y otro de un proveedor, o entre un *switch* y un autorizador. También podría darse el caso de tener adaptadores en cascada, transformando protocolos de forma serializada.

En la práctica se ha analizado que muchos *switches* tienen incorporado como parte de su lógica la adaptación de protocolos para la adquisición de productos, responsabilidad que debería ser de los *bridges*. Si bien en esta propuesta se considera que para vender un único producto, el *switch* y el *bridge* podrían ser el mismo módulo lógico, lo más conveniente es adoptar el desacople de responsabilidades entre estas dos entidades, siempre y cuando sea la intención de los *stakeholders* agregar cada vez más productos sobre una red transaccional. Es decir, el *switch* concentrará a todos sus puntos de venta y los pre/post validará. Además podrá autorizar algunas transacciones de inicialización de POS, *echoes,* o cualquier otra transacción a nivel administrativo. Pero debería *conversar en un solo idioma* para la adquisición de varios productos, y que los adaptadores sean los encargados

[7] *SLA* es el acrónimo en inglés de *Service Level Agreement*. Es un acuerdo sobre los requisitos y expectativas del modo de funcionamiento que mínimamente debe brindar un servicio bajo este contrato.

de entender este único idioma y readaptar los protocolos contra ellos. Además deberían encargarse de enmascarar al *switch* de toda la lógica necesaria para mantener activas las conexiones con los proveedores.

2.3.3 Capa de Administración (*Management*)

Dentro de esta categoría se ubican para este trabajo todos los sistemas que permiten dar visibilidad de las transacciones realizadas a través de una *solución* de procesamiento. En este caso, la palabra *solución* hace referencia tanto a los equipos adquisidores de datos (EFT-POS, POS, Aplicaciones Web, *Smartphones*, etc.), como a los concentradores, adaptadores y autorizadores finales. Estos sistemas son herramientas administrativas de gran envergadura, que principalmente traducen el término *dar visibilidad de las transacciones,* como reportes, informes, y plantillas de control para las cuentas de los clientes, los productos que se están vendiendo u ofreciendo, etc.

Estas plataformas pueden ser parte de dos grandes grupos conocidos: las aplicaciones de *front-office*, y las de *back-office*. Las primeras son aplicaciones de *software* que tienen interacción directa con los clientes. Proveen funcionalidades e información para la administración de órdenes de compra, configurar los productos que se comercializan y brindar soporte especializado a los usuarios finales. Dentro de este subgrupo, los más conocidos son los *home-banking*. En cambio, los sistemas de *back-office* se enfocan a proveer funcionalidades para operaciones internas del negocio en cuestión. Dentro de esta subcategoría se incluyen los ERP (del acrónimo inglés *Enterprise Resource Planning*), los de control de inventario, o cualquier otro sistema de administración que esté relacionado con el control del negocio, o con sus materias primas o con sus elementos de trabajo, por ejemplo.

Los sistemas de administración (tanto *front-office* como *back-office*) son actualmente implementados como *aplicaciones web* o como plataformas orientadas a *cloud computing*, enfocadas al trabajo colaborativo y a brindar mayor cantidad de información útil al negocio. Aunque la tendencia actual es que los sistemas faciliten la interacción entre usuarios y el intercambio de datos, muchos de ellos son sistemas *legacy* y pueden estar realizados bajo cualquier plataforma, por ejemplo aplicaciones distribuibles (*stand-alone*) para PC. Más allá de la implementación final, el objetivo (en referencia al dominio específico de este trabajo) es hacer uso de los datos de las transacciones para poder obtener información procesada de cualquier tipo, que permita facilitar la toma de decisiones en las gerencias comerciales, de ventas, de productos / servicios, etc. Para ello, estos sistemas generan reportes de todo tipo, informes, e implementan *dashboards* interactivos que facilitan la comprensión de la información al usuario final. También muchos permiten controlar el estado y la actividad de los elementos transaccionales. Por elementos transaccionales se consideran tanto los POS de una determinada red (para saber si están activos, dónde están geográficamente, etc.), como los *switches*, *bridges* o *authorizers*, según corresponda en cada caso en particular. En estos últimos casos, se puede administrar por dónde debe redirigirse una transacción, cuáles son los protocolos que deben aceptarse, qué productos están habilitados, qué

rangos de tarjetas se validarán o no, cuáles son los montos mínimos y máximos para un determinado producto, etc.

2.3.4 Secuencia Transaccional Típica

La secuencia de trabajo de los servidores transaccionales es relativamente similar entre los distintos tipos de procesamiento que se puedan analizar. Es importante aclarar que durante este trabajo, el término *transacción* hará referencia a la ejecución de una secuencia predefinida de procesamiento. A continuación se introducirán cómo son algunas secuencias típicas de varios sub-dominios de trabajo, y cómo es la distribución de los elementos de conectividad involucrados (servicios, autorizadores, puentes, concentradores, puntos de venta, etc.).

2.3.4.1 Solución Financiera

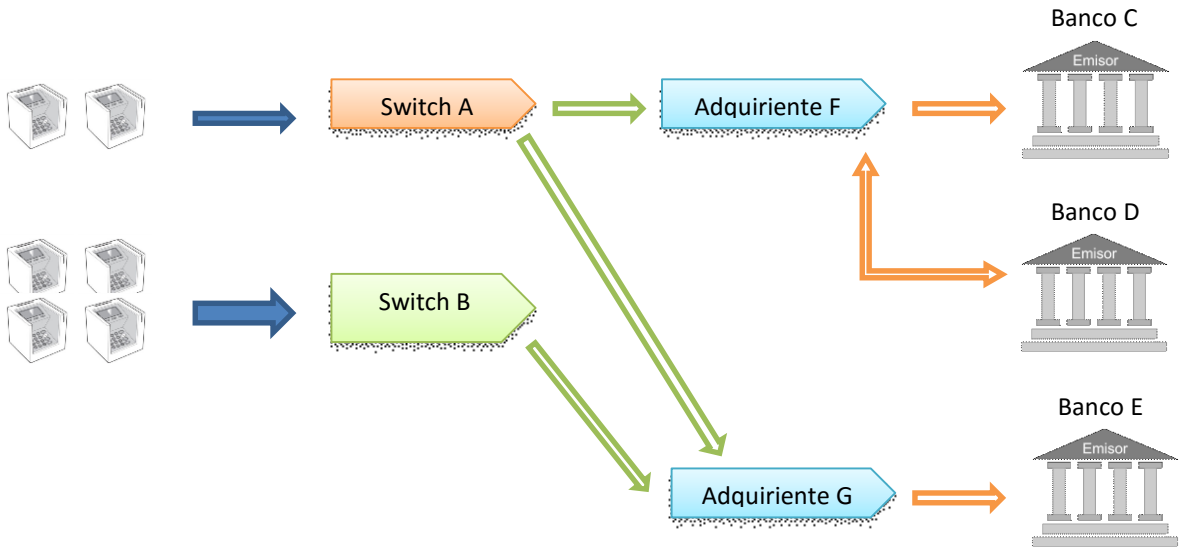

Figura 5 - Diagrama de distribución de elementos en una solución financiera típica

Al igual que muchos otros sub-dominios, el inicio de una secuencia de trabajo se da por la llegada de una trama de datos al puerto de escucha del servicio. Generalmente, el protocolo de transporte usado es TCP-IP; sobre ese, luego se implementarán otros protocolos donde la información pueda ir cifrada, o formateada, en concordancia con las partes en juego. Las peticiones generalmente salen de *cajeros automáticos* (*ATM*), *puntos de venta* (*POS*), *celulares, pinpads*, u otros sistemas informáticos, como *páginas web y web services*, entre otros.

Por cada transacción entrante, se genera un *hilo de ejecución* o *thread* que se encargará de resolver el procesamiento de esa transacción, y luego formar la respuesta para el cliente que está esperando respuesta a través del socket. Por lo que, en un estado permanente y sin transacciones dentro del sistema, solo existe como mínimo un *hilo de ejecución* principal que se encargará de escuchar por un socket TCP-IP. Generalmente, y por facilidades otorgadas por los lenguajes de programación, se utiliza un *thread pool,* para la administración de todos los *threads* que eventualmente se generen. Dentro del *thread pool* hay un conjunto de *worker threads* en estado *idle,* disponibles y a la espera que la cola de trabajo les asigne alguna una nueva tarea. La cola de trabajo enviará una señal a los *threads* en espera dentro de la cola para que cobren vida y tomen el trabajo de procesamiento de ese socket conectado.

La *vida* de una transacción comienza, para los fines de este trabajo, justo en el momento en que el hilo de ejecución es asignado para su proceso. Ese hilo tiene conocimiento del socket con el cual el cliente tuvo acceso al servidor, y a través de él, comienza a leer datos crudos que se hayan enviado desde el cliente. Generalmente, estos datos están formateados en el protocolo ISO8583, por lo que en cuanto se obtiene una lectura, se procede al análisis y procesamiento (*parsing*) del *array de bytes* obtenido, con el objetivo de armar una estructura de entrada (requerimiento). Esta tarea es similar a una deserialización de datos, desde un *array* a una clase. Cabe acotar que si bien ISO8583 es ampliamente usado, no es el único protocolo financiero usado en los *CAS* (HPDH por ejemplo).

La primera tarea a realizar dentro del hilo, y con los datos ya deserializados, es la validación de tarjeta, y luego la validación anti fraude con los elementos presentes en el mensaje. Particularmente en este tipo de procesamiento, la validación anti fraude se hace mediante operaciones criptográficas que suelen ser costosas en tiempo de recursos. Es por eso que para optimizar los tiempos de procesamiento, estas tareas criptográficas no se realizan en el microprocesador del servidor físico, sino en hardware dedicado para tal fin. Los *HSM* son estas piezas de hardware dedicado, que se conectan con los servidores a través de la red (sobre TCP-IP), con protocolos propietarios de los distintos fabricantes. A su vez, las tareas de validación de tarjeta, y de validación anti fraude son independientes una de otra, por lo que pueden ser paralelizadas. Eso por eso que los *CAS* en particular suelen usar un segundo *pool de threads*, para realizar las tareas de validación anti fraude a través de hilos separados del de la transacción, cada uno interactuando con el *HSM* para resolver las operaciones criptográficas en función de los elementos encriptados en el mensaje recibido desde el cliente. Entre las validaciones anti fraude se pueden enumerar la validación del código de seguridad de la tarjeta, la validación del número de identificación de la tarjeta, validación del *PIN* (si se hubiese ingresado), y validaciones de criptogramas de la aplicación *EMV*.

Simultáneamente a las validaciones anti fraude llevadas a cabo por estos *hilos hijos*, el *thread* principal de la transacción comienza a procesar en paralelo las validaciones de restricción de tarjeta. Éstas incluyen

validaciones de existencia de la tarjeta, validaciones del estado de la tarjeta, validaciones de fechas de vencimiento, validaciones del uso de tarjeta, y validaciones de saldos de la cuenta. [4]

Una vez que el *thread* principal termina de hacer todas estas validaciones, espera por la terminación de los *hilos hijos*, convergiendo la ejecución en el hilo principal. Una vez que los *hilos hijos* convergen en el *thread* principal, pasan a disponibilidad del *thread pool*, para eventualmente procesar tareas encoladas o próximas a delegarse al *pool*. En paralelo, el hilo principal de la transacción debe generar una respuesta para ser enviada al cliente, ya sea aprobándola o rechazándola, a través de todos los resultados de las validaciones de tarjeta realizadas, y de las operaciones criptográficas anti fraude.

Para generar el código de respuesta al cliente, se analizan todas las respuestas obtenidas de las validaciones. Si dentro de estas validaciones, existe alguna (o más de una) evaluación que desapruebe las condiciones presentadas por el ente financiero, el código de respuesta estará asociado a la validación más gruesa o a la primera en orden de evaluación. En el caso contrario, si la transacción es aprobada se genera un número único aleatorio de autorización, para referencia del cliente sobre la aprobación de la reciente venta. [4]

Cualquiera sea el estado final de la transacción (aprobada o rechazada), el *thread* principal tiene como responsabilidad generar el mensaje ISO8583 de respuesta en una estructura, que terminará siendo serializada para enviarse. Por lo que una vez armada esa estructura, la misma se ensambla (serializa) y se escribe en el socket abierto en el cliente. Es esperable que el cliente reciba la respuesta y genere un proceso final a nivel local, por ejemplo la actualización del *batch de transacciones*, impresión del ticket, etc. Es importante destacar que el proceso descrito en los párrafos anteriores tiene que durar una cantidad de tiempo dentro de una ventana temporal prevista tanto por el cliente como por el servidor, sino, existe la posibilidad que el equipo del otro lado del socket haya cerrado la conexión al haberse superado el tiempo máximo de espera. Es en esa situación en la que el cliente marca la transacción de su lado como *Time-Out.*

Luego del envío de la respuesta, el servidor supone que el cliente recibió los datos a través del socket, por lo que procede a cerrarlo, desecharlo, y continúa con procesos internos y/o tareas de mantenimiento. Dentro de estas tareas finales, se pueden encontrar el guardado de los mensajes de entrada/salida (para fines de auditoría y/o control interno), y la actualización del saldo de la cuenta asociada a la tarjeta. En general, cualquier proceso que no determine el resultado de la transacción podrá realizarse luego de enviar la respuesta al cliente, de modo de ahorrar la mayor cantidad de tiempo de proceso dentro de la ventana donde el cliente está esperando la respuesta. Finalizando toda la lista de tareas de mantenimiento, el *thread* principal termina y queda disponible dentro del *thread pool* para poder ser reasignado en una próxima transacción entrante.

2.3.4.2 Solución de Recarga de Crédito Tiempo-Aire

Los servidores concurrentes de recarga de tiempo-aire, por ejemplo, tienen la ventaja de ser más flexibles en muchos aspectos de seguridad, conectividad y de arquitectura del servidor, con respecto a los que procesan crédito y débito. La mayor flexibilidad se da por la importancia de la información que se está procesando: no es lo mismo procesar recargas de tiempo aire (en promedio de $30 - $50) que transacciones de débito/crédito. Como el negocio está en conseguir una comisión dentro de la cadena de distribución de un producto dado, muchas veces se requieren construir puentes (*bridges*) y concentradores (*switches*) que permitan involucrar a una compañía en esa cadena. Luego, *caminos distintos de distribución* pueden tener porcentajes diferentes de ganancia, y el objetivo está en poder adaptarse al camino donde la ganancia es mayor. A continuación se expresa un diagrama de conexión típico con los elementos mencionados, involucrando a la distribución de un producto en particular desde el autorizador (generalmente la operadora), hasta las redes de puntos de venta.

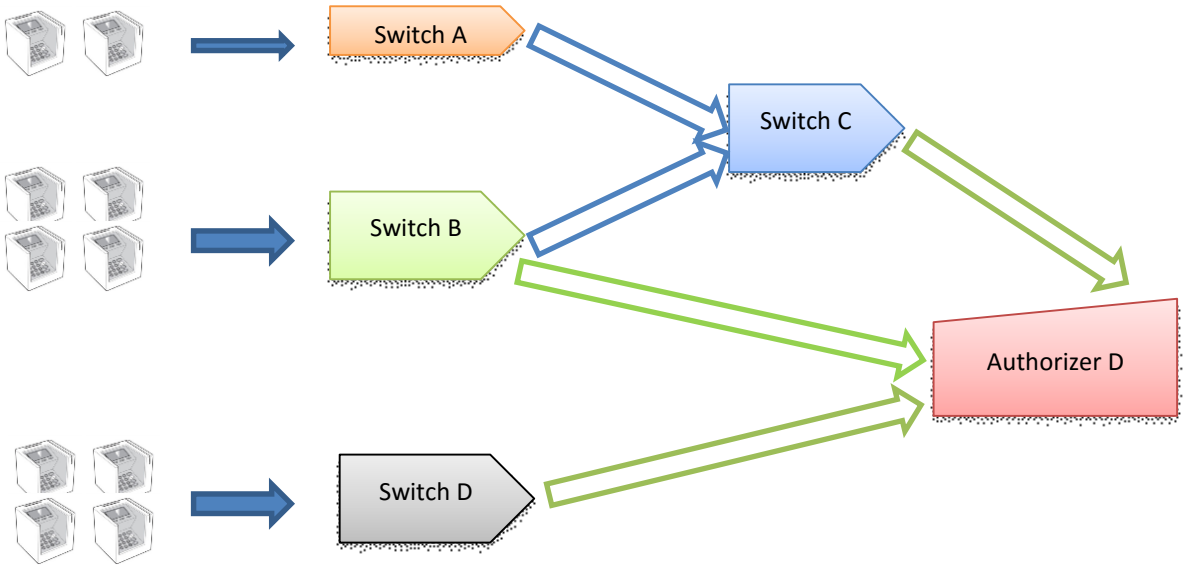

Figura 6 - Ejemplo de diagrama de distribución de componentes de una solución de recarga tiempo-aire

El diagrama de componentes muestra un esquema típico de distribución de *switches*, *bridges* y *authorizers* para una solución de recarga de tiempo-aire. Las transacciones se generan desde un parque de puntos de venta, al igual que en la solución financiera. Este tipo de soluciones requiere de menor seguridad en la transmisión de datos (comparada con aquellas de procesamiento de transacciones financieras), y es por eso que abundan los sistemas de dispositivos portables, o basados en sistemas informáticos cuya seguridad es menor a la provista por ejemplo, por los cajeros automáticos (*ATM*). De todas formas, el procesamiento de recargas de celular, como se lo conoce habitualmente, es un valor agregado a los servicios de pago por lo general, por lo que hoy en día también es posible recargar desde un *ATM*.

Desde la perspectiva de un *switch* como los de la Figura 6, las transacciones se inician de forma similar a un *CAS*: Nuevamente el inicio de la secuencia de trabajo comienza por la llegada de una trama de datos a un puerto

de escucha configurado. Generalmente, el protocolo usado también es TCP-IP; y sobre ese, luego se pueden implementar otros donde la información puede ir cifrada, o formateada en concordancia con las partes involucradas en el proceso de la transacción. Las solicitudes pueden llegar desde *cajeros automáticos* (*ATM*), *puntos de venta* (*POS*)*, celulares, páginas web y web services*, entre otros.

Al igual que en los sistema CAS, por cada transacción entrante se genera un *hilo de ejecución* o *thread* que se encargará de resolver el procesamiento de esa transacción. Mientras que el servicio esté en estado permanente y sin transacciones, existirá al menos un *hilo de ejecución* principal que se encargará de escuchar por un socket TCP-IP. Se observó en muchos servicios la implementación de *Thread Pools* para la asignación de hilos por cada solicitud recibida. Es decir, por cada transacción escuchada en el hilo principal, se encolará la transacción entrante de modo que un *worker thread* se encargue de procesarla. Estas transacciones quedan encoladas hasta que el *Thread Pool* envié una señal a los *threads* en espera, para que cobren vida y tomen el trabajo de procesar la petición.

En este tipo de procesador, la *vida* de una transacción también comienza justo en el momento en que el *thread* es asignado para su proceso. Ese hilo tiene conocimiento del socket con el cual el cliente tuvo acceso al servidor, y a través de él, comienza a leer datos crudos que se hayan enviado desde el cliente. Generalmente, estos datos están formateados en el protocolo ISO8583 u cualquier otro protocolo que cumpla con las expectativas. En cuanto se obtiene una lectura, se procede al análisis y procesamiento (*parsing*) del *array de bytes* obtenido, con el objetivo de armar una estructura de requerimiento.

También puede darse el caso que el *switch* no presente como opción un *"motor"* con un *thread principal* escuchando a un socket TCP-IP. En ese caso los concentradores ofrecen generalmente una interfaz del tipo *Web Service*. A diferencia de lo descrito con anterioridad, las transacciones *cobran vida* sobre un hilo de ejecución que no es administrado por el *switch* por sí mismo. Desde el punto de vista del desarrollador, la transacción comienza en la primera línea de código asociada a un *Web Method o Data Operation.* La asociación entre el hilo y la transacción es realizada por el servicio administrador de *Web Services* del sistema operativo, por ejemplo *Internet Information Services* (IIS)*,* en plataformas *Microsoft*. En otras palabras, cuando se usan *Web Services*, se facilita la interacción con otros *switches* aguas abajo en la red de distribución, y se reducen los *time to market* de conexión *HTH* entre compañías procesadoras. Como ventaja a nivel programación, no existe la necesidad de lectoescritura de datos crudos desde/hacia un socket, ni de la serialización/deserialización de *arrays* a estructuras. Los datos entre los puntos de venta y el concentrador, (o entre concentradores) se transmiten a través del protocolo *SOAP, XML, JSON* u otro similar. Desde la perspectiva del desarrollador, un enfoque práctico es tener tantos *Web Methods*, como transacciones procese el concentrador, por lo que los datos de cada transacción son pasados como parámetros de entrada en el método, y los resultados como un objeto, o una colección de objetos de respuesta del método.

Al igual que en el procesamiento de crédito/débito, una vez obtenidos los datos del cliente la transacción se puede dividir en tres etapas: una de *pre-procesamiento*, una segunda de *procesamiento*, y otra final de *post-procesamiento*. En la primera etapa se realizan validaciones sobre el estado de la terminal, el estado del comercio, la validez del par comercio/terminal, si el distribuidor es válido, si hay saldo en la bolsa del distribuidor, si el comercio tiene saldo disponible, si el comercio/terminal puede recargar o vender el producto que solicita, etc. Con el resultado de estas validaciones, luego se genera un movimiento en una base de datos, o en un mecanismo de persistencia, para marcar la existencia de esa transacción en ese momento. A la par de estos procesos de validación, se ejecutan algoritmos de ruteo, que permiten identificar para un producto solicitado en un momento dado, cuál es la mejor alternativa para adquirir la recarga aguas arriba en la cadena de la distribución de ese producto. La elección de la mejor alternativa se puede realizar por varios criterios, pero el más importante es el criterio de mayor utilidad económica, que busca la mayor comisión de los concentradores mayoristas habilitados en un momento dado. Pero se pueden tomar en cuenta el estado de la red, el estado de los proveedores (que estén activos en la última media hora, por ejemplo) o cualquier otro criterio que considere la salud de la red de distribución, asegurando siempre que fuera posible, la realización en tiempo y forma de la transacción.

En la etapa de procesamiento, la transacción se autoriza o se rechaza de forma local (cuando la operación no requiere el reenvío de la solicitud), o se retransmite al siguiente nodo de la cadena que pueda procesar la solicitud. Suponiendo el caso que sea necesario la retransmisión a un nodo externo en la red de distribución, se realiza una conexión hacia éste, ya sea a través de un socket TCP-IP, a través de un *Web Service*, o algún otro medio. Una vez realizada la conexión, el concentrador debe reenviar los datos de la solicitud formateados según el protocolo acordado entre ambos *hosts*. Consecuentemente esperará por una respuesta. De forma subyacente deberán existir los mecanismos para mantener la conexión, si fuera necesario. Como la variedad de los protocolos entre concentradores es muy amplia, se usan *adaptadores* (servidores concurrentes con la finalidad de adaptar los protocolos entre los dos *hosts*), para desacoplar al concentrador de la lógica de mantenimiento de la conexión con otros nodos aguas arriba. De este modo, el concentrador se comunica con todos los adaptadores a través de un protocolo simple, convenido de forma interna por la compañía, facilitando el mantenimiento del concentrador a medida que se suman nuevos canales de distribución (nuevos proveedores, con nuevos *hosts*, con protocolos y secuencias de conexión distintas). Para facilitar la comprensión de esta etapa, se ofrece un ejemplo a continuación: Suponiendo que se pretende realizar una venta (siguiendo la Figura 6 como ejemplo), si analizamos el flujo transaccional desde el *Switch B*, podemos ver que se puede obtener un producto *P* reenviando la petición a un concentrador intermedio *Switch C*, o directamente al autorizador de ese producto (*Authorizer D*). En la mayoría de los casos el camino más directo es el que ofrece mejor comisión, por lo que en principio la mejor opción es reenviar la solicitud a *D*. Pero si el algoritmo detecta que en la última hora estuvo el enlace caído, el algoritmo decidirá ir por C. Tal vez, para ese producto P en particular el camino *B-C* tenga mejor comisión (por ejemplo *C* prefiere ir a pérdida como contraparte de ganar puntos de venta en

su subred) y en ese caso, y solo para P, el camino elegido es C. La idea es que este direccionamiento es dinámico y puede modificarse por múltiples razones, desde cuestiones técnicas hasta comerciales.

Por último en la etapa de post-procesamiento, se analiza la respuesta obtenida del nodo externo (si la hubiera), y se almacena en la base de datos, o en el mecanismo de persistencia usado por la solución. En el caso de no tener respuesta, o cualquier otro error inesperado, debería indicarse la situación en el histórico de transacciones, con un error descriptivo para cada situación. Por último, se genera una estructura de respuesta con los datos obtenidos tanto para una aprobación o un rechazo de la transacción. Esos datos se serializan en un *array* o *stream* de *bytes* que son devueltos al cliente a través del *socket* conectado. Una vez emitida la respuesta, el *socket* se cierra y se pueden realizar tareas de mantenimiento (actualización de los saldos en la cuenta del comercio, difusión de mensajes a los POS, consultas de saldo en la bolsa del proveedor usado, etc.).

2.4 Resumen del Capítulo

Durante este capítulo se trataron los siguientes temas:

- Se introdujeron algunas de las problemáticas asociadas con el dominio, particularmente el juego de compromiso entre los requerimientos de seguridad y de rendimiento en los sistemas transaccionales.
- Se introdujeron los elementos principales del dominio transaccional, a través de una categorización de tres capas. La *capa de adquisición* agrupa a todos los dispositivos para el *input* de los datos de una solicitud transaccional. La *capa de enlace* conglomera a todos los servidores que rutean una transacción hasta el autorizador (un autorizador también es un servidor transaccional y se incluye dentro de esta capa). Por último la *capa de administración* engloba a los sistemas de *back-office* y *front-office* que hacen uso de los datos obtenidos por medio de las transacciones.
- Se describieron algunas de las secuencias transaccionales más comunes, con el objetivo de generar la noción al lector de todos los pasos necesarios para aprobar o rechazar una solicitud entrante en un servidor concurrente. Se desarrollaron dos ejemplos: el primero consistió en una secuencia transaccional financiera genérica, y el segundo en una secuencia genérica en un servidor transaccional de recargas de tiempo aire para telefonía celular.

3 Estado del Arte y Revisión de la Bibliografía

3.1 Introducción

Este capítulo hará una introducción teórica sobre los pilares de la metodología de desarrollo MDD. Además se investigará bibliografía disponible para tratar de encontrar otros casos donde se hayan implementado sistemas transaccionales *dirigidos por modelos*, y eventualmente revisar los resultados de las experiencias.

Así mismo se analizará si existen *frameworks* para facilitar la integración de componentes enfocados al procesamiento de transacciones. Se le recuerda al lector que uno de los puntos principales de este trabajo es la construcción de un *marco de trabajo* que pueda soportar componentes comunes y reutilizables, y que facilite la implementación de transformaciones automáticas desde un modelo de dominio específico a código funcional. Es por esto que, cualquier trabajo preexistente relacionado al tema servirá sin dudas para aprender de las cualidades y defectos de éstos, en pos de mejorar el *marco de trabajo* que se propondrá en capítulos posteriores.

Debido a las virtudes que presenta TDD para comprobar el funcionamiento de un servidor transaccional, se introducirá esta metodología dirigida por pruebas. Lógicamente, se orientará el tema TDD hacia el desarrollo y evaluación de soluciones de procesamiento transaccional. Además se presentará cómo podría ayudar el uso de pruebas unitarias en el mantenimiento de este tipo de sistemas, en especial cuando se tienen que hacer desarrollos evolutivos y/o correctivos tratando de asegurar el correcto funcionamiento de la aplicación.

3.2 Trabajos Investigados Relacionados

Del conjunto de artículos recolectados durante la primera fase del proyecto, se destacaron los siguientes artículos y/o desarrollos:

El primer trabajo (*Multi-threading technique for authorization of credit card system using .NET and JAVA*) del autor W. Y. Ming [1] propone un servidor transaccional concurrente orientado a la paralelización de tareas (*multi-threading*) para mejorar los tiempos de respuesta en autorizadores financieros. El análisis del autor recae en qué tareas típicas de estos servidores transaccionales concurrentes pueden paralelizarse dentro de cada hilo de ejecución de cada transacción, por ejemplo, las operaciones criptográficas. Como ya se adelantó en los primeros capítulos, muchas de estas operaciones criptográficas y de validación se hacen con dispositivos de E/S (por ejemplo, un HSM conectado al servidor), y la interacción contra éstos puede llegar a ser costosa en tiempos de respuesta y en uso de recursos computacionales. El autor propone el desarrollo de un servidor transaccional concurrente, mostrando distintas técnicas de paralelización para diferentes etapas de procesamiento. Estos ejemplos de código son presentados tanto para lenguaje Java como para C# (.NET Framework). Este trabajo fue de gran utilidad por el material formal aportado, dada la coincidencia en el dominio de ambas

investigaciones. Así mismo hace una gran introducción y reseña histórica, y describió varios procesos del dominio de forma precisa. Como desventaja este trabajo no hace hincapié en la formalización de las técnicas de paralelización. Tampoco hace referencia a *marcos de trabajo* u algún otro tipo de artefacto que propongan una formalización de las técnicas que describe.

En el segundo trabajo (*Lagniappe: Multi-* programming made simple*) [5] el autor hace una propuesta de formalización MDD de un conjunto de elementos de dominio sobre arquitecturas *multi-procesador* o *multi-threading* (*multi-**). El artículo está orientado particularmente a sistemas de alto tráfico como enrutadores GENI, sistemas de consola y sistemas transaccionales en general. No define *elementos de dominio complejos* (como se presentará en este trabajo más adelante, de modo de representar conceptos comunes), en cambio define un conjunto de *elementos de dominio reducidos*, que pueden conectarse entre sí para formar elementos de dominio más complejos. Si bien el dominio no es parecido al tratado en este trabajo, este artículo aportó una perspectiva para describir y presentar el que aquí se redacta, y el concepto de *elementos de dominio reducidos* es interesante. En el trabajo de tesis que en estos capítulos se presenta se prefirió usar un *conjunto de elementos de dominio complejos* en contraste con los *elementos de dominio reducidos*. Es decir, en el escenario de elementos de dominio complejos, se generan tantos elementos de dominio distintos como objetos de un marco de trabajo se desean representar. En cambio en el escenario de elementos de dominio reducidos, la gama de estos elementos es pequeña, pero se define la interconexión entre ellos para poder armar otros más complejos.

En el tercer trabajo (*Domain-Specific Languages for Enterprise Systems*) [6] los autores documentan una implementación basada en una arquitectura llamada POETS (*Process-Oriented Event-driven Transaction Systems*), para la construcción de sistemas del tipo ERP orientada a eventos. POETS divide el diseño de los sistemas ERP en tres capas: *datos transaccionales* (las acciones/transacciones que ya ocurrieron en el sistema), *reportes* (los resultados de procesar los *datos transaccionales*) y *contratos* (las transacciones esperadas). En este artículo se hace una propuesta de modelización MDD sobre esta implementación POETS que se documenta. Es decir, se presentan elementos de dominio para modelar con un DSL las capas de *datos transaccionales*, los *reportes* y los *contratos*. Si bien el trabajo está orientado a un dominio diferente al que se presenta en éste, la forma de documentar la implementación y de exponer el DSL fueron de gran utilidad para diagramar las secciones y definir la compaginación de este trabajo. Comienza introduciendo la arquitectura POETS, luego introduce un caso de estudio de un ERP, y luego define formalmente (a través de lógica textual) los elementos de dominio del DSL. Por último se presenta la definición de los elementos de dominio para ese caso de estudio.

3.3 Desarrollo Dirigido por Modelos (MDD)

3.3.1 Introducción

Los desarrolladores de software están bajo un dilema importante a la hora de construir sistemas de cualquier tipo, ya que deben mantener un equilibrio delicado entre la calidad, la expectativa de vida de un producto y los costos de producción de los mismos. [7] La construcción de sistemas de procesamiento transaccional no queda exenta a esta circunstancia, debido a que la expectativa de vida del producto al momento de desarrollarse es generalmente grande: como se mencionó, algunos bancos mantienen sus sistemas hasta llegar a la obsolescencia (magnitudes de 15 años por ejemplo). Para mantener el sistema funcionando, y a la vez, cumpliendo con las nuevas reglas de negocio o de servicio, sufren evoluciones constantemente, muchas veces incurriendo negativamente en la calidad del *autorizador*, *switch*, o del *adaptador* por ejemplo.

La *ingeniería de software* establece que el problema de construir *software* debe ser encarado de la misma forma en que los ingenieros construyen otros sistemas complejos, como puentes, edificios, barcos y aviones. La idea básica consiste en observar el sistema de *software* a construir como un producto complejo y a su proceso de construcción como un trabajo ingenieril. Es decir, un proceso planificado basado en metodologías formales apoyadas por el uso de herramientas. Hacia finales de los años 70, *Tom Demarco* en su libro "*Structured Analysis and System Specification*" introdujo el concepto de *desarrollo de software basado en modelos* o *MBD* (por sus siglas en inglés *Model Based Development*). *DeMarco* destacó que la construcción de un sistema de *software* debe ser precedida por la construcción de un modelo, tal como se realiza en otros sistemas ingenieriles. El modelo del sistema es una conceptualización del dominio del problema y de su solución. El modelo se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal. La abstracción es una de las principales técnicas con la que la mente humana se enfrenta a la complejidad. Ocultando lo que es irrelevante, un sistema complejo se puede reducir a algo comprensible y manejable. Cuando se trata de *software*, es sumamente útil abstraerse de los detalles tecnológicos de implementación y tratar con los conceptos del dominio de la forma más directa posible. De esta forma, el modelo de un sistema provee un medio de comunicación y negociación entre usuarios, analistas y desarrolladores que oculta o minimiza los aspectos relacionados con la tecnología de implementación. [8]

Sin embargo, los enfoques actuales de construcción de *software* no son suficientes para tratar los inconvenientes relacionados a los efectos de cambios de tecnología, a los efectos de ambientes de trabajo con plataformas múltiples, y a los efectos que surgen de la necesidad de cambios en los requerimientos de forma recurrente (grado de facilidad de adaptación de un diseño/sistema). [7] Si bien el *desarrollo de software basado en modelos* ha representado un paso importante en la *ingeniería de software*, el proceso sigue careciendo de características que nos permitan asegurar la calidad y corrección del producto final. El proceso de desarrollo de *software*, como

se utiliza hoy en día, es conducido por el diseño de bajo nivel y por el código. En las primeras fases se construyen distintos modelos tales como los modelos de los requisitos (generalmente escritos en lenguaje natural), los modelos de análisis y los modelos de diseño (frecuentemente expresados mediante diagramas). Estas fases pueden realizarse de una manera iterativa-incremental o en forma de cascada. Pero en la práctica cotidiana, los modelos quedan rápidamente desactualizados debido al atajo que suelen tomar los desarrolladores en el ciclo de vida de este proceso. [8]

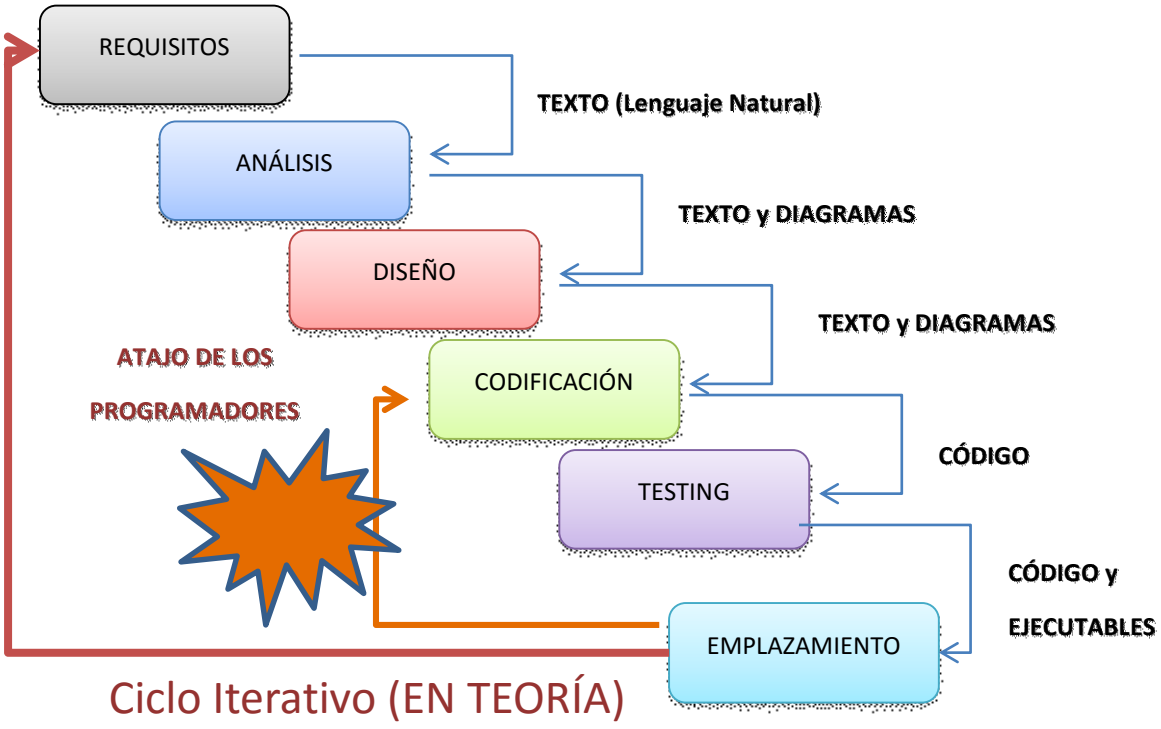

Figura 7 - Ciclo de vida en cascada, enfocado al desarrollo basado en modelos (MBD), donde se muestra los atajos al código para la evolución o reparación de sistemas

A continuación se detallan otros inconvenientes relacionados con la metodología de desarrollo basada en modelos, referidos en [8]. Uno de ellos es el efecto típico que se observa durante las tareas de mantenimiento de un sistema y el grado de actualización a nivel documentación del producto, tanto para otros desarrolladores en el equipo, como para la visibilidad hacia el exterior: En *MBD*, la conexión entre los diagramas y el código se va perdiendo gradualmente mientras se progresa en la fase de codificación. Los programadores suelen hacer los cambios sólo en el código, porque no hay tiempo disponible para actualizar los diagramas y otros documentos de alto nivel. Además, el valor agregado de diagramas actualizados y los documentos es cuestionable, porque cualquier nuevo cambio se encuentra en el código de todos modos. Muchas veces se considera la tarea de la documentación como una sobrecarga adicional. Por lo contrario, la opción de desechar

los modelos y considerar únicamente al código puede complicar extremadamente la tarea de mantenimiento del sistema, especialmente luego de transcurrido un tiempo considerable desde la construcción del mismo [8], como suele suceder en el caso de los sistemas autorizadores de tarjetas de crédito / débito. Debido a la complejidad inherente de un sistema de procesamiento transaccional determinado (cualquiera sea su finalidad), resulta imprescindible contar con documentación en distintos niveles de abstracción.

Otra consecuencia que se observa en la construcción directa en código fuente es la poca flexibilidad a cambios tecnológicos: La industria del *software* tiene una característica especial que la diferencia de las otras industrias. Cada año (o en menos tiempo), aparecen nuevas tecnologías que rápidamente llegan a ser populares. Muchas veces son los clientes los que exigen esta nueva tecnología, Por lo tanto, tienen que pasar a utilizarlas cuanto antes. Como consecuencia del cambio, las inversiones en tecnologías anteriores pierden valor. Por consiguiente, el *software* existente debe migrar a una nueva tecnología, o a una versión nueva y diferente de la usada en su construcción. [8]

En el dominio del procesamiento de transacciones, las tecnologías tanto de software como de hardware se renuevan a un ritmo mucho menor que en otros dominios. Cabe acotar esto, de modo que el lector comprenda que por temas económicos y de amortización de productos, los sistemas, las tecnologías y equipos usados para armar una red de procesamiento generalmente se mantienen por muchos años. Principalmente por el tamaño de las inversiones realizadas, o porque no desean re-invertir en la evolución de sistemas *legacy* que funcionan desde hace años, y que terminan siendo confiables con el pasar del tiempo. Solamente cuando surgen nuevas normativas, restricciones legales o cuando surgen nuevos negocios, se considera la evolución de un sistema transaccional. Este hecho no significa que las tecnologías no existan, sino que se trata de alargar la vida útil de las mismas. Además el hecho que los equipos y elementos transaccionales no estén comercializados al público en general, (como si sucede con un *smartphone*, una *tablet,* o una *notebook* de última generación, etc.) genera una menor necesidad de renovar constantemente las líneas de producto por parte de las empresas fabricantes de equipos, de *software* y de tecnologías. En otras palabras, el efecto de la menor flexibilidad a cambios tecnológicos en implementaciones directas a código causa menor impacto en este dominio, que en una *Software Factory,* por ejemplo, dedicada al desarrollo de aplicaciones *web* o *mobile.*

3.3.2 Paradigma MDD

El *Desarrollo de Software Dirigido por Modelos* MDD (por sus siglas en inglés: *Model Driven software Development*) se ha convertido en un nuevo paradigma de desarrollo de *software*. MDD promete mejorar el proceso de construcción de *software* basándose en un proceso guiado por modelos y soportado por potentes herramientas. El adjetivo “dirigido” (*driven*) en MDD, a diferencia de “basado” (*based*), enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos a través de pasos de transformación

y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor de MDD. [8]

Los puntos claves de esta metodología son:

- **Abstracción:** El enfoque de MDD para incrementar los niveles de abstracción es definir lenguajes de modelado específicos de dominio cuyos conceptos reflejen estrechamente los conceptos del dominio del problema, mientras se ocultan o minimizan los aspectos relacionados con las tecnologías de implementación. [8]
- **Automatización**: La automatización es el método más eficaz para aumentar la productividad y la calidad. En MDD la idea es utilizar a las computadoras para automatizar tareas repetitivas que se puedan mecanizar, tareas que los seres humanos no realizan con particular eficacia. [8]
- **Estandarización**: MDD debe ser implementado mediante una serie de estándares industriales abiertos facilitando, por ejemplo, la capacidad para intercambiar especificaciones entre herramientas complementarias, o entre herramientas equivalentes de diferentes proveedores. [8]

3.3.3 Ventajas de la Metodología desde la Perspectiva de los Sistemas de Procesamiento

3.3.3.1 Incremento en la Productividad y Re-Uso de Código

MDD reduce los costos de desarrollo de *software* mediante la generación automática del código y otros artefactos a partir de los modelos, lo cual incrementa la productividad de los desarrolladores. De este modo se invierte esfuerzo en el desarrollo de modelos y transformaciones: a su vez, ésta inversión se va amortizando a medida que los modelos y las transformaciones son re-usados. Por otra parte el re-uso de artefactos ya probados incrementa la confianza en el desarrollo de nuevas funcionalidades y reduce los riesgos ya que los temas técnicos han sido previamente resueltos. [8]

La confianza en un artefacto / transformación ya probada con anterioridad, es un *requerimiento no funcional* muy importante para cualquier dominio de aplicación, y es particularmente importante en el de sistemas de autorización y procesamiento. Eso se debe principalmente a que el procesamiento de transacciones se ejecuta generalmente sin interacción humana y cualquier error es difícil de ver, ya sea durante la puesta en marcha en ambientes productivos, o en un sistema que está en marcha hace tiempo y funcionando en estado permanente. Generalmente son procesos que corren como *servicios* en sistemas operativos *Windows*, como *daemon's* en plataformas *UNIX-like*, o en general como procesos en *background.* La visibilidad hacia el exterior se da a través de archivos de *log,* mensajes por consola, a través de procesos que generen alarmas, envíen mails, o

concilien transacciones. En el mejor de los casos, cualquier error grave podría manifestarse en las etapas iniciales de la ejecución del servidor concurrente, pero en el peor de los casos el error puede quedar latente, generando problemas sin ser descubierto por mucho tiempo. Además los *inputs* que pueden causar errores o excepciones muchas veces se dan eventualmente, cuando las transacciones llegan al sistema. Es por eso que se considera de gran utilidad contar con generadores automáticos de código, que permitan repetir una y otra vez artefactos ya probados en otros sistemas de procesamiento, que han pasado por situaciones de carga, inestabilidad de la red, etc.

3.3.3.2 Adaptación a los Cambios Tecnológicos

En muchos dominios informatizados, el progreso de la tecnología hace que los componentes de *software* se vuelvan obsoletos rápidamente. MDD ayuda a solucionar este problema a través de modelos de alto nivel, que están libres de detalles de la implementación. Este detalle facilita la adaptación a los cambios que pueda sufrir la plataforma tecnológica subyacente o la arquitectura de implementación. Por ejemplo, los cambios se realizan modificando la transformación del PIM (*Platform-Independent Model*) al PSM (*Platform-Specific Model*). La nueva transformación es re-aplicada sobre los modelos originales para producir artefactos de implementación actualizados. Esta flexibilidad permite probar diferentes ideas antes de tomar una decisión final. Y además permite que una mala decisión pueda fácilmente ser enmendada. [8]

Particularmente las tecnologías informáticas involucradas en los sistemas de procesamiento suelen permanecer implementadas por mucho más tiempo, que en otros dominios. Lógicamente, las bondades de las nuevas tecnologías y metodologías sirven para muchos tipos de negocio y sistemas, pero no todas logran implementarse en ambientes productivos a la misma velocidad con la que aparecen en el mercado. Por ejemplo, aun muchas terminales POS siguen siendo programadas en lenguaje C, *Assembler*, y otros lenguajes no orientados a objetos. Esto se debe generalmente a temas que exceden cuestiones técnicas, involucrando temas de políticas de negocio, y económicos. Las empresas procesadoras tratan de aumentar el valor generado por cualquier punto de venta y servidor concurrente al máximo. Si los equipos usados (tal vez por años) comprueban seguir siendo eficaces para los fines del negocio, los mismos suelen mantenerse hasta que terminan su vida útil, más allá de la tecnología de *software* involucrada. Como se comentó en las características de los sistemas de autorización de tarjetas, la mayoría de los bancos poseen sistemas de procesamiento, que en promedio, rondan los 15 años de antigüedad, generalmente *rígidos* a los cambios y de gran presupuesto para modificarlos. [4]

Según la apreciación del autor, esta característica de la metodología MDD no deja de ser útil en el negocio del procesamiento de transacciones, ya que tarde o temprano siempre será necesario migrar a nuevas plataformas de *software*. Pero debido al ritmo de cambio, de seguro no será la mayor ventaja de aplicar MDD en el dominio.

3.3.3.3 Adaptación a los Cambios en los Requisitos

Poder adaptarse a los cambios es un requerimiento clave para los negocios, y los sistemas informáticos deben ser capaces de soportarlos. Cuando usamos un proceso MDD, agregar o modificar una funcionalidad de negocios es una tarea bastante sencilla, ya que el trabajo de automatización ya está hecho. Cuando agregamos una nueva función, sólo necesitamos desarrollar el modelo específico para esa nueva función. El resto de la información necesaria para generar los artefactos de implementación ya ha sido capturada en las transformaciones y puede ser re-usada. [8]

Esta característica es una de las principales bondades aprovechables de MDD en los sistemas de procesamiento: Como se describe en [2] por ejemplo, una de las características de los *CAS* es la frecuencia de actualización de reglas de negocio, la modificación de transacciones y de las secuencias de procesamiento. Generalmente se debe a que los gobiernos generan y refuerzan leyes con alcance a las compañías de tarjetas de crédito, produciendo cambios en el *software* de las plataformas. Así también se debe a cambios producidos por la competencia entre compañías. De esta forma, la implementación de la metodología podrá seguramente aportar a la reducción de costos de mantenimiento y evolución del programa, y a la convergencia de criterios entre los programadores. Además, la aplicación manual de las prácticas de codificación y diseño es una tarea propensa a errores, por lo que la automatización MDD favorecerá la generación consistente de los artefactos. [8]

3.3.3.4 Mejoras en la Comunicación

Uno de los puntos importantes del nivel de abstracción de los modelos con respecto al código, es que los primeros omiten detalles de implementación que no son relevantes para entender el comportamiento lógico del sistema. Los modelos están más cerca del dominio del problema, reduciendo la brecha semántica entre los conceptos que son entendidos por los usuarios y el lenguaje en el cual se expresa la solución. Esta mejora en la comunicación influye favorablemente en la producción de *software*. Así también, los modelos facilitan el entendimiento del sistema por parte de los distintos desarrolladores. Esto da origen a discusiones más productivas y permite mejorar los diseños. Además, el hecho de que los modelos son parte del sistema y no sólo documentación, hace que éstos siempre permanezcan actualizados y confiables. [8]

Si bien los sistemas de procesamiento terminan siendo usados por millones de usuarios de forma indirecta, los usuarios directos son generalmente las compañías que los adquieren y/o los desarrollan. Y aun así, no tienen casi interacción humana, tal vez para configurar eventualmente su comportamiento. Por lo que la mejora en la comunicación se puede aprovechar sobre todo en la convergencia de criterios de los programadores involucrados, e indirectamente, como medida de cumplimiento de los requerimientos elicitados.

3.3.3.5 Captura de la Experiencia

Como se describe claramente en [8], las organizaciones y los proyectos frecuentemente dependen de expertos clave quienes toman las decisiones respecto al sistema. En el negocio del procesamiento de transacciones, la disponibilidad de conseguir recursos humanos para tareas de desarrollo con experiencia previa en el dominio es generalmente inferior a la disponibilidad de recursos de otros rubros informáticos. Por consecuencia, hay menor cantidad de candidatos con experiencia en este dominio, siendo difícil de reemplazar a aquellos que se van a otras compañías. De hecho, la mayoría de los involucrados en el desarrollo para POS, y sistemas B24, son casi siempre las mismas personas que van rotando de empresa a empresa por una u otra razón. Muchos se conocen de experiencias laborales anteriores, y se recomiendan a medida que surgen nuevas oportunidades laborales. Sin embargo muchos otros trabajan por períodos que pueden rondar desde los 3 a los 8 años en un mismo proyecto[8].

Cualquiera sea el caso, la cantidad de personal disponible que esté experimentado en el mundo transaccional es relativamente menor al de otros rubros, haciéndolos valiosos para cualquier compañía que disponga de sus servicios. Si eventualmente dejan el puesto que ocupan, muchos proyectos quedan generalmente truncos, mal documentados, con baja calidad de seguimiento, y los reemplazantes deben consumir tiempo y esfuerzo en entender cómo funcionaban esos sistemas preexistentes. Además las primeras acciones realizadas por los reemplazantes que terminan siendo puestas en producción son propensas a tener errores, en función del grado de entendimiento que el individuo haya logrado del sistema, mientras que, simultáneamente, generaba el código correctivo y/o evolutivo.

Es por eso que, la implementación de MDD puede ayudar a reducir esfuerzo de capacitación de personal. Al capturar la experiencia de los expertos claves en los modelos y en las transformaciones, otros miembros del equipo pueden aprovecharla sin requerir su presencia. Este conocimiento se mantiene, aún si eventualmente los expertos se alejan de la organización. [8]

3.3.3.6 Duración de los Modelos

En MDD los modelos son productos importantes que capturan lo que el sistema informático de la organización hace. Los modelos de alto nivel son resistentes a los cambios a nivel plataforma y sólo sufren cambios cuando lo hacen los requisitos del negocio. [8]

---

[8] Las declaraciones se hacen en basadas en la experiencia vivida por el autor en más de ocho años dentro del rubro, y son a nivel informal para contar una perspectiva percibida por el mismo, con respecto al ambiente laboral de aquellos involucrados en el negocio.

3.3.3.7 Posibilidad de Demorar las Decisiones Tecnológicas

Según se expresa en [8], cuando aplicamos MDD, las primeras etapas del desarrollo se focalizan en las actividades de modelado. Esto significa que es posible demorar la elección de una plataforma tecnológica específica o una versión de producto hasta más adelante cuando se disponga de información que permita realizar una elección más adecuada.

En los sistemas transaccionales esta ventaja no es de gran utilidad práctica, ya que los desarrollos que son aprobados a niveles gerenciales generalmente tienen definidos de antemano los entornos que serán usados. De hecho la aprobación de realizar un proyecto o no, depende de tener definidas una cantidad de variables, no solo de nivel técnico, pero que probablemente incluyen a la plataforma donde se ejecutará el sistema. En función de la importancia de la transacción que se procesará, será necesario contar con más o menos definiciones de antemano: Por ejemplo, no es lo mismo procesar *puntos* de una cuenta de un socio adherido a un programa de lealtad de clientes, que procesar *dinero* de una compra con tarjeta de crédito.

3.3.4 Propuestas Concretas de MDD

3.3.4.1 Arquitectura Dirigida por Modelos (MDA)

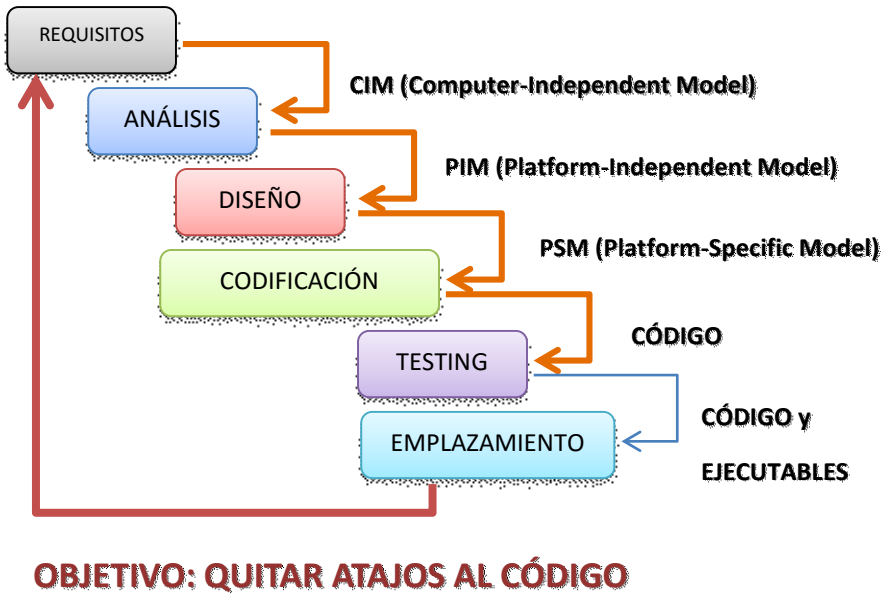

Figura 8 - Ciclo de vida en cascada, enfocado a la transformación automática de modelos de mayor a menor nivel de abstracción en MDA

La propuesta *Model Driven Architecture* (Arquitectura Dirigida por Modelos o MDA), es una de las iniciativas más conocida y extendida dentro del ámbito de MDD. MDA es un concepto promovido por el *Object Management Group* a partir de 2000, con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos, siguiendo el proceso MDD. En MDA, la funcionalidad del sistema es definida en primer lugar como un modelo independiente de la plataforma (*Platform-Independent Model* o PIM) a través de un lenguaje específico para el dominio del que se trate. En este punto aparece además un tipo de modelo existente en MDA, no mencionado por MDD: el modelo de definición de la plataforma (*Platform Definition Model* o PDM). Entonces, dado un PDM correspondiente a *CORBA, .NET, Web*, etc., el modelo *PIM* puede traducirse a uno o más modelos específicos de la plataforma (*Platform-Specific Models* o PSMs) para la implementación correspondiente, usando diferentes lenguajes específicos del dominio, o lenguajes de propósito general como Java, C#, Python, etc. [8]

MDA no se limita al desarrollo de sistemas de *software*, sino que también se adapta para el desarrollo de otros tipos de sistemas. Por ejemplo, MDA puede ser aplicado en el desarrollo de sistemas que incluyen a personas como participantes junto con el *software* y el soporte físico. Asimismo, MDA está bien adaptado para el modelado del negocio y empresas. La propuesta MDA está relacionada con múltiples estándares, tales como el

Unified Modeling Language (UML), el *Meta-Object Facility* (*MOF*), XML *Metadata Interchange* (*XMI*), *Enterprise Distributed Object Computing* (EDOC), el *Software Process Engineering Metamodel* (SPEM) y el *Common Warehouse Metamodel* (CWM). [8]

Cumpliendo con las directivas del OMG, las dos principales motivaciones de MDA son la interoperabilidad (independencia de los fabricantes a través de estandarizaciones) y la portabilidad (independencia de la plataforma) de los sistemas de *software*; las mismas motivaciones que llevaron al desarrollo de CORBA. Además, el OMG postula como objetivo de MDA separar el diseño del sistema tanto de la arquitectura como de las tecnologías de construcción, facilitando así que el diseño y la arquitectura puedan ser alterados independientemente. El diseño alberga los requisitos funcionales (casos de uso, por ejemplo) mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos los requisitos no funcionales como la escalabilidad, fiabilidad o rendimiento. MDA asegura que el *modelo independiente de la plataforma* (PIM), el cual representa un diseño conceptual que plasma los requisitos funcionales, sobreviva a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas de *software*. Por supuesto, la noción de transformación de modelos en MDA es central. La traducción entre modelos se realiza normalmente utilizando herramientas automatizadas, es decir herramientas de transformación de modelos que soportan *MDA*. Algunas de ellas permiten al usuario definir sus propias transformaciones. [8]

3.3.4.2 Modelado Específico de Dominio (DSM)

La iniciativa DSM (*Domain-Specific Modeling*) es principalmente conocida como la idea de crear modelos para un dominio, utilizando un lenguaje focalizado y especializado para dicho dominio. Estos lenguajes se denominan DSL (por su nombre en inglés: *Domain-Specific Language*) y permiten especificar la solución usando directamente conceptos del dominio del problema. Los productos finales son luego generados desde estas especificaciones de alto nivel. Esta automatización es posible si ambos, el lenguaje y los generadores, se ajustan a los requisitos de un único dominio. Definimos como dominio a un área de interés para un esfuerzo de desarrollo en particular. En la práctica, cada solución DSM se enfoca en dominios pequeños porque el foco reductor habilita mejores posibilidades para su automatización y estos dominios pequeños son más fáciles de definir. Usualmente, las soluciones en DSM son usadas en relación a un producto particular, una línea de producción, un ambiente específico, o una plataforma. El desafío de los desarrolladores y empresas se centra en los siguientes elementos claves dentro de una solución DSM: [8]

- La definición de los lenguajes de modelado.
- La creación de los generadores de código.
- La implementación de *frameworks* específicos del dominio

Estos elementos no se encuentran demasiado distantes de los elementos de modelado de MDD. En general, DSM usa los conceptos dominio, modelo, metamodelo, meta-metamodelo como MDD, sin mayores cambios y propone la automatización en el ciclo de vida del software. Los *lenguajes de dominio* son usados para construir modelos. Estos lenguajes son frecuentemente (pero no necesariamente) gráficos. Los DSL no utilizan ningún estándar del *Object Management Group* (OMG) para su infraestructura, es decir no están basados en UML, y los metamodelos no son instancias de MOF *(Meta-Object Facility)* (a diferencia usan por ejemplo MDF, el *framework* de metadatos para este propósito). Finalmente, existe una familia importante de herramientas para crear soluciones en DSM que ayudan en la labor de automatización. *Visual Studio* provee herramientas para definir los metamodelos así como su sintaxis concreta y editores. [8] *DSL Tools* es la herramienta modular de *Microsoft* para su entorno *Visual Studio*, que sirve en la creación de metamodelos y lenguajes de dominio específico. *DSL Tools* fue el componente utilizado para construir el lenguaje específico y concreto de formalización, para el marco de trabajo en cuestión dentro de esta tesis.

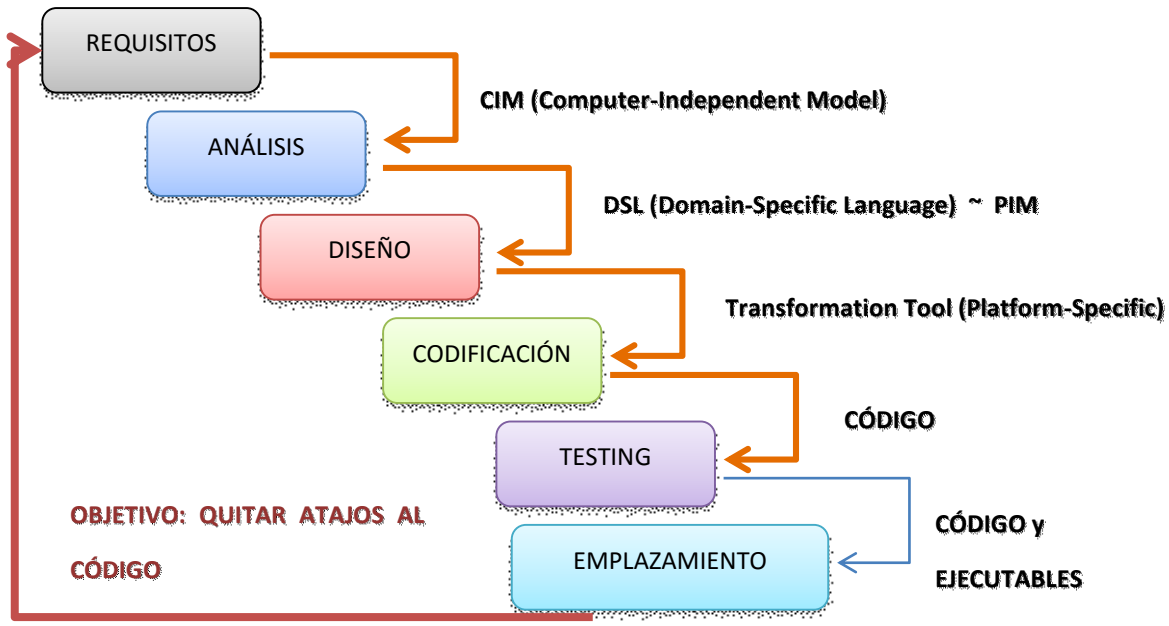

Figura 9 - Ciclo de vida DSM

3.4 Desarrollo Dirigido a Pruebas (TDD)

3.4.1 Introducción

*Test-driven development (*TDD*)*, o *desarrollo dirigido a pruebas*, es un enfoque evolutivo para el desarrollo de *software* que combina dos técnicas: el enfoque *test-first development* o TFD (que fomenta el desarrollo de las pruebas antes que el propio código), y el enfoque de *refactoring*. TDD se resume entonces como una técnica de programación, cuya meta primaria es hacer foco sobre la construcción de una especificación mediante el diseño de casos de prueba unitarios, en vez de solo utilizar estas pruebas para la validación una vez que se termina el desarrollo. Es decir, es una forma de pensar sobre los requerimientos o el diseño antes de empezar a escribir el código funcional del programa en sí mismo. [9]

El primer paso en el armado de un proyecto enfocado a TDD comienza con la construcción rápida de un test unitario, básico, y de modo tal que las pruebas unitarias que se escriban fallen en primera instancia. De este modo se deberían desarrollar tantas pruebas unitarias como especificaciones se hayan definido, y que se pretenden terminar validando durante la construcción del código funcional. Luego, se debe actualizar el código del sistema gradualmente, tratando de ir cumpliendo cada uno de los test unitarios predefinidos al comienzo del desarrollo. Durante el proceso, se va refactorizando el código de modo que el sistema cumpla con las especificaciones de la mejor forma posible, a nivel diseño y arquitectura. Así se describe este proceso, resumiendo a la metodología TDD como una adición de técnicas, tal que se cumpla la fórmula *TDD = Refactoring + TFD*. [9]

TDD da vuelta al revés el proceso tradicional de construcción de programas. Por ejemplo, cuando se requiere implementar una nueva funcionalidad, una de las primeras preguntas que surgen es si el diseño actual permite la inclusión de esta funcionalidad de la mejor manera posible. Si el equipo desarrollador así lo considera, se procede a escribir test unitarios para cumplir con el enfoque TFD *(test-first)*. Si se considera lo contrario, se debería realizar una tarea de refactorización sobre el diseño actual afectado por la funcionalidad a incorporar, de modo de realizar la implementación de forma fácil. Como resultado de este proceso, se mejora de forma continua la calidad del diseño, haciéndolo más fácil para mantener y /o evolucionar en el futuro. [9]

En vez de escribir código funcional, y luego código de pruebas, se debería escribir primero las pruebas unitarias antes que el código funcional. De hecho debería realizarse de a pequeños pasos: es decir, primero se escribe una fracción de casos de prueba para un sub-aspecto de la funcionalidad a incorporar, y luego se escribe el código funcional para hacer cumplir las pruebas unitarias de ese sub-aspecto. Un programador que incorpora la metodología TDD termina prefiriendo el tener un caso unitario que falle para una funcionalidad dada, previo a tener el código escrito para tal fin. Una vez puesto el caso unitario se realiza el trabajo para que el set de pruebas resulte correcto, si bien el código que se escribe puede hacer fallar casos de unidad preexistentes, que antes de

la implementación pasaban de forma correcta. De esta forma se pueden detectar errores generados por el ingreso de código de esta nueva funcionalidad, refactorizando o corrigiendo según corresponda para desacoplar la inferencia entre los componentes afectados. Si bien este proceso suena simple, cuando se comienza con la adopción de la metodología es muy fácil *caer en la tentación* de escribir el código funcional primero antes que el código. Muchas veces la técnica de *pair programming* ayuda a que el compañero mantenga la disciplina de escribir siempre el caso unitario primero. [9]

3.4.2 TDD y Testing Tradicional

TDD es mayoritariamente una técnica de especificación con un efecto colateral de asegurar que el código fuente es validado fehacientemente. Pero de todos modos, los casos unitarios no son los únicos a considerar en un proceso de desarrollo. Por ejemplo se deberían considerar técnicas de pruebas de integración previas a la puesta en marcha, entre otros de gran importancia para cumplir con las especificaciones y expectativas. [9]

Con técnicas de test tradicionales, un test es exitoso cuando encuentra una o más cosas para corregir. Es lo mismo en TDD; cuando una prueba unitaria falla se denota un progreso ya que queda confirmado que es necesario hacer una corrección sobre el código. Adicionalmente, en TDD se tiene una medición clara y cuantitativa de progreso en cuanto la prueba unitaria deja de fallar. De esta forma, esta metodología ayuda a aumentar la confianza en que el sistema cumple con los requerimientos pautados. [9]

Como sucede en procesos de validación tradicionales, cuanto más grande es el perfil de riesgo del sistema, más amplias deberían ser las pruebas a realizar. Un hecho interesante de TDD es que se puede cubrir potencialmente el 100% del sistema a fines de validación, ya que cada línea de código ingresada puede ser testeada automáticamente. Esto no puede lograrse tan fácilmente con técnicas de test tradicionales. [9]

3.4.3 TDD y la Documentación

Generalmente los programadores prefieren trabajar directamente con el código, en vez de leer por completo (previo a comenzar a trabajar) la especificación de requisitos dada para un sistema. Cabe acotar que esta práctica no tiene nada de malo, siempre y cuando el código generado sea el de pruebas unitarias. De hecho, el enfoque TDD ofrece la posibilidad de escribir una especificación funcional bien definida del código del programa, a través de casos de prueba unitarios. Como resultado, los casos de prueba se convierten efectivamente en una parte importante de la documentación técnica del proyecto. Así también, los *test de aceptación* se convierten en parte de la especificación de requerimientos, ya que definen lo que los *stakeholders* esperan del sistema. Por ende, definen los requisitos críticos. Ahora bien, los distintos set de pruebas (regresión, test de integración, test de unidad, test de aceptación, etc.) no logran conformar toda la documentación necesaria para un proyecto, pero si una gran parte del conjunto total. [9]

3.5 Comparación de Enfoques: TDD y MDD

A continuación se enumeran algunas comparaciones entre ambos enfoques:

- Las metodologías TDD acortan los ciclos de retroalimentación durante la etapa de programación, mientras que las metodologías MDD acortan los ciclos de retroalimentación durante el modelado y el diseño de la solución.
- TDD provee una forma de especificación detallada a través de casos unitarios, mientras que MDD es más adecuado para la abstracción de problemas grandes a través de modelos que van de menor a mayor complejidad.
- TDD y MDD promueven el desarrollo de alta calidad de código y la mejora de calidad en la comunicación con los *stakeholders* y los otros integrantes del equipo de desarrollo.
- TDD provee evidencia concreta que el programa funciona correctamente según los casos de prueba definidos, mientras que MDD soporta al equipo de desarrollo y a los *stakeholders* a un mejor entendimiento de la solución.
- TDD *habla* a los programadores mientras que MDD *dialoga* no solo con los desarrolladores, sino también con los analistas de negocio, los *stakeholders*, y los profesionales IT a nivel de capa de datos, arquitectura y diseño.
- TDD no está orientado a *outputs* visuales (reportes, modelos, etc.) mientras que MDD si lo hace, con la salvedad que TDD puede brindar reportes de cantidades de pruebas unitarias satisfactorias o fallidas.
- Ambas metodologías son relativamente nuevas y para *programadores tradicionales* pueden resultar amenazantes, o *chocantes* en un principio.
- Ambas metodologías promueven el desarrollo evolutivo.

Ahora bien, ¿cuál de estos dos enfoques debería considerarse para un desarrollo? La respuesta depende de los integrantes del equipo de trabajo en cuestión, y sus preferencias cognitivas. Algunas personas tienen mayor facilidad para la *pensar visualmente*, y por ende pueden preferir expresar sus ideas a través de dibujos o modelos. En cambio otras personas pueden preferir pensar de una forma no tan gráfica y así preferir una metodología puramente TDD. Por supuesto, esto no debe ser considerado como una decisión estrictamente binaria, y deberían implementarse las dos metodologías en distintos grados de adopción para aprovechar de la forma más cómoda para el equipo de desarrollo, las ventajas de cada una de ellas. [9]

En pos de combinar ambas metodologías, MDD podría usarse para crear modelos en conjunto con los *stakeholders* del proyecto, y de este modo poder comprender sus requerimientos a través de modelos de diseño y arquitectura. Por otro lado, TDD debería usarse como una parte crítica del esfuerzo de construcción que asegure que el desarrollo se está haciendo de forma ordenada, y con código funcional validado

cuantitativamente. Como resultado, se debería obtener una solución de alta calidad técnica, que cumple con los requerimientos de las partes interesadas en el proyecto. [9]

3.6 Resumen del Capítulo

Durante este capítulo se trataron los siguientes temas:

- Se presentaron tres reseñas de trabajos relacionados al tema de investigación de esta tesis. Estos trabajos sirvieron para orientar la redacción de todas estas líneas, y así también como material académico complementario sobre la metodología MDD, y sobre el dominio del procesamiento de transacciones electrónicas financieras.
- Se introdujeron las problemáticas inherentes al desarrollo de *software*, dada la naturaleza invisible y compleja de cualquier programa informático. Se remarcó el concepto fomentado por la *Ingeniería de Software*, que recomienda considerar una solución informática como cualquier otro producto complejo de otras ingenierías.
- Se trató el *atajo de los programadores*, dentro del ciclo de desarrollo tradicional de *software,* denotando algunas desventajas entre la correlación entre la documentación, los modelos primitivos y el código funcional de una solución.
- Se introdujo el paradigma MDD, analizando algunas de sus ventajas con respecto al paradigma MBD. Así mismo se presentaron dos propuestas de implementación MDD: *Model Driven Architecture* (MDA) y *Domain Specific Modeling* (DSM). La que más se adapta a las necesidades de este trabajo es probablemente DSM, ya que avala la creación de lenguajes específicos de dominio (DSL) conjugados con marcos de trabajo (*frameworks*) propietarios. Estos *frameworks* a su vez capitalizan la lógica de reglas del negocio en cuestión para fines de reutilización.
- Se introdujo la metodología TDD*,* y se comparó en algunos puntos de importancia para este trabajo contra las metodologías MDD presentadas. La intención es poder combinarlas convenientemente para sacar provecho en esta propuesta de formalización.

4 Caso de Estudio – Motivación y Desarrollo del Framework TransactionKernel

4.1 Introducción

En este capítulo se introducirá el trabajo de análisis realizado sobre códigos preexistentes, que terminó derivando en las tareas de reingeniería y refactorización a través de revisiones retrospectivas periódicas. Durante las primeras líneas de este capítulo se describirán las experiencias obtenidas de dichos análisis, para que se puedan fundamentar los conceptos transaccionales repetibles encontrados, que iban apareciendo entre proyectos de forma común. Estos conceptos son muy importantes, porque son la clave para poder armar un DSL que habilite una implementación concreta y útil MDD en el dominio.

En la primera parte de este capítulo se hará foco en dos experiencias laborales del autor. Son dos experiencias separadas, vividas en dos empresas diferentes, donde los equipos de desarrollo involucrados tenían como objetivos evolucionar y mantener los códigos de dos servidores transaccionales. El producto de esta experiencia se capitaliza en una serie de observaciones que contienen defectos, anti-patrones, y también aciertos. Todos los puntos débiles fueron la base motivacional para comenzar a analizar de qué forma se podían mejorar.

La segunda parte del capítulo hará foco en el desarrollo del *framework TransactionKernel*. Con la información obtenida en la primera parte, se generó una capa de *software* que contiene los elementos y conceptos que se repitieron de forma común en la mayoría de las soluciones. Esta capa tiene como objetivo homogeneizar el desarrollo de servidores transaccionales a través de la concentración de aspectos comunes y repetibles, probados y desarrollados para fomentar la reusabilidad y fácil manutención de soluciones que la utilicen. A la vez, esta capa intenta minimizar los *vicios* del código, que se generan cuando los programadores reutilizan código de forma descontrolada. Y por último, intenta aprovechar siempre que fuera posible, la mayor cantidad de conceptos de la *ingeniería de software*, maximizando el uso de los recursos del sistema, y aumentando el nivel de profesionalidad y calidad del producto.

Este marco de trabajo es la base para la hipótesis de formalización que se propone en este trabajo de tesis.

4.2 Situación Pre-Framework

En esta sección se introducen en detalle dos experiencias laborales del autor, donde se hará foco a las observaciones realizadas. Estas observaciones también están sustentadas con otras experiencias de trabajos hechos sobre varias soluciones de procesamiento transaccional. Por cuestiones de privacidad, no se harán referencias a nombres propios de entidades ni de personas, y por temas de confidencialidad se tratará de evitar cualquier nombre o código de los sistemas involucrados que se describen a continuación. A continuación se introduce en forma de historia, los sucesos laborales y técnicos que desencadenan directa o indirectamente la motivación al cambio propuesto.

El autor de este trabajo formó parte de dos equipos de desarrollo en los últimos ocho años, para dos compañías distintas. Para la primera compañía, el equipo de desarrollo tenía a cargo mantener una solución única de un programa de lealtad, para una petrolera muy importante en Argentina. El sistema a desarrollar y mantener era un *switch* concentrador, que atendía alrededor de 100 POS, y que originalmente estaba basado en un simulador de *switch*. Como en cualquier sistema que nace pensado para ser de un alcance más pequeño del que luego resulta abarcar, los *bugs* y errores comienzan a brotar en el código acompañando al crecimiento del mismo. Sumado a esta condición, estaba presente la poca experiencia del equipo en ese momento para poder hacer evolucionar al sistema de forma ordenada. Este conjunto de condiciones hacía que cualquier evolución fuera agregada casi sin valor ingenieril, sobre un sistema base sub dimensionado.

Las principales observaciones de este sistema de procesamiento de transacciones de lealtad eran las siguientes:

- Estaba escrito en C++, haciendo al sistema más eficiente en uso de recursos de CPU con respecto a otros idiomas (como por ejemplo *.NET* ni *Java,* al ser código nativo). Potencialmente por esta razón era un sistema rápido para procesar transacciones en cantidad. Sin embargo esta ventaja en eficiencia se hace notoria cuando la cantidad de transacciones que pasan por el sistema es importante (por ejemplo, por arriba de las 10 millones de transacciones mensuales). En el caso particular de esta solución de lealtad, la cantidad de transacciones mensuales estaba entre 8000 y 10000, y no se hacían notorias las mejoras debido al uso de código nativo por sobre otro lenguaje basado en código interpretado. Por lo que una de las mayores ventajas del lenguaje estaba desaprovechada.
- A pesar de haber estado escrito en C++, el sistema no estaba orientado a objetos, desaprovechando el potencial semántico del lenguaje. Solo había algunas clases del simulador original. Las subsiguientes

evoluciones se hicieron con agregados de código a estas clases, y al final el sistema terminó con una gran clase que contenía toda la lógica (*God object*[9]).

- El sistema corría bajo *Windows*, en modo de servicio. Se ejecutaba tanto por consola como por la consola de servicios del sistema operativo. Era bastante útil el uso de la consola visual, ya que con un parámetro por línea de comando, se podía ver en tiempo real las líneas que el sistema escribía.
- Inicialmente, el simulador (la base fundacional del *switch*) no estaba pensado para trabajar con ningún medio de persistencia. Luego fue necesario interactuar contra una base de datos que estaba desarrollada por un *partner* de la compañía, pero de la cual el equipo de desarrollo no tenía acceso. Solo se disponía de los parámetros de entrada y salida de los *Stored Procedures* que serían llamados. La capa de datos se armó con una herramienta de generación automática de clases que a partir de estos *Stored Procedures*, armaba subclases concretas heredadas de *CRecordSet*. El *God Object* llamaba e instanciaba oportunamente cada una de estas subclases en función de la necesidad de interactuar con la base de datos. El mayor de los problemas era el *driver* ODBC, que producía pérdidas de memoria (*memory leaks*) por cada conexión abierta. Si bien esta era solo una de las tantas causas que generaba pérdidas de memoria, el sistema terminaba acumulando suficiente pérdida como para que fuera necesario reiniciarlo. En caso contrario, éste era susceptible de sufrir un error en tiempo de ejecución que terminaba frenándolo a la fuerza. La interacción con la base de datos era bastante complicada para el equipo de desarrollo y nunca se migró a OLEDB, o a otro tipo de conector más adecuado.
- El *God Object* contenía un único método que hacía toda la transacción, desde que la recibía hasta que la respondía. Aproximadamente el método medía unas 14000 *LOC*[10] y a medida que fue creciendo se hizo más complicado mantenerlo por la cantidad de sentencias condicionales presentes.
- El sistema, al estar escrito en C++, no contaba con un administrador automático de memoria. No tenía *Garbage Collector*[11] de ningún tipo, y la correcta disposición de la memoria y uso de los punteros quedaban a responsabilidad de la labor de los desarrolladores. Debido a la cantidad de líneas del único

---

[9] Un *God Object* es un antipatrón conocido en la Ingeniería de Software, que describe a un objeto que *sabe mucho* o *tiene muchas responsabilidades*. Generalmente son de gran tamaño, no tienen una división coherente de responsabilidades, y disminuyen la mantenibilidad.

[10] *LOC* es el acrónimo de *líneas de código,* del inglés *Lines Of Code.* Se usa como medida cuantitativa aproximada de la extensión de un programa.

[11] Un recolector de basura (del inglés *Garbage Collector*) es un mecanismo implícito de gestión de memoria implementado en algunos lenguajes de programación de tipo interpretado o semi interpretado.

método del *God Object,* se hacía imposible encontrar aquellos puntos del programa donde la memoria no se liberaba correctamente, generando un *bug* conocido pero costoso de arreglar.

- Para generar registro de los pasos que se iban realizando durante la vida de una transacción, se usaba un método de *logging* propio, que era útil para los fines de la solución aunque conocido solo por los desarrolladores del equipo.
- Originalmente el sistema usaba variables de contexto transaccionales *globales,* generando un caos cuando caían transacciones de forma simultánea (los punteros a estos contextos apuntaban no solo al contexto propio sino al de otras transacciones simultáneas). Se logró controlar esto migrando los contextos a variables locales y utilizando inyección de dependencias, pero manteniendo el paradigma de programación secuencial, y desaprovechando cualquier patrón posible para desacoplar el código y hacerlo *thread-safe*.
- Las transacciones tenían unas con otras varios pasos en común: leer datos de un socket, desarmar el paquete en campos, validar los campos, llamar a los *Stored Procedures* que fueran necesarios, procesar la respuesta de la base de datos, armar la respuesta hacia el POS, serializar los datos, mandarlos a través del socket y finalizar la conexión. Más allá de la finalidad de cada una, siempre se cumplían estos pasos.

A mediados de Julio de 2011, y luego de varios años de aprendizaje, el autor comenzó a trabajar en la compañía actual donde ahora desarrolla su labor. Esta empresa, ya de mayor envergadura, contaba con cuatro subsidiarias en América Latina y atendían varios *switches transaccionales* por oficina, dedicándose exclusivamente a las recargas de tiempo-aire. Estos *switches* procesaban mayores cantidades de transacciones y eran más sofisticados que el de lealtad mantenido previamente. También presentaban ventajas y tenían defectos por resolver, algunos similares a los vistos con anterioridad. Las observaciones obtenidas del trabajo de análisis de estos sistemas es la siguiente:

- También estaban escritos en C++, haciendo a estos sistemas bastante eficientes en uso de recursos de CPU. La cantidad de transacciones era mayor que la trabajada en la primera compañía; solo en Argentina había alrededor de un millón de transacciones mensuales (medición aproximada durante mediados de 2011) divididas por el procesamiento de seis *switches*, que redirigían transacciones a otros nodos fuera de la compañía, a través de sockets TCP.
- Además de redirigir (*routing*) las transacciones hacia otros servidores concurrentes (los de las operadoras, u otros *GRE*'s), los *switches* debían validar los datos de terminal, comercio, monto, producto, comisiones, etc. Estos datos se resguardaban en una base de datos con motor *SQLServer 2000*. El *driver* de comunicación con la base de datos era *ODBC*, produciendo las mismas pérdidas de memoria (*memory leaks*) observados con anterioridad. Aunque en estos casos, el inconveniente se hacía más visible por la cantidad de transacciones entrantes: no solo se generaban errores en tiempo

de ejecución debido al tamaño de la memoria consumida, sino que surgían problemas con los tiempos de acceso a la base. En horas *pico* de recarga, la base se saturaba y el *driver* respondía con error de *Time-Out*, teniendo que rechazar la transacción en el mejor de los casos, si no causaba otro error más grave antes.

- El sistema no permitía imprimir líneas por consola, a modo de *debug*, pero usaba una librería más compleja de *logging* que trabajaba escribiendo en archivos. Ventajosamente, la librería estaba separada en un archivo de extensión *.dll* que se referenciaba en los distintos proyectos de servidores concurrentes. La librería a su vez era propietaria, por lo que era conocida solo por los equipos de desarrollo de los países que la usaban. Cualquier modificación o mejora sobre ella era percibida solo por el equipo que la modificaba, y debido a que no estaban coordinados los cambios entre las subsidiarias, la librería era poco interoperable entre una solución de un país y la de otro.
- Las transacciones también tenían pasos en común entre un proyecto y otro: De las observaciones resultaban que cualquier servidor como mínimo tenía que leer de un socket, desarmar el paquete en campos, validar los campos, decidir si la transacción se podía autorizar localmente, si la transacción había que reenviarla se enviaba por otro socket, se esperaba la respuesta del proveedor, se procesaba la respuesta de la base de datos / proveedor (según fuera el caso), se armaba la respuesta hacia el POS, se serializaban los datos, se mandaban a través del socket y se finalizaba la conexión. Los pasos eran bastante similares en los *switches* de recarga observados, como en la solución de lealtad que se había mantenido anteriormente.
- A nivel arquitectura, para el manejo de la base de datos y de la clase principal de la solución se usó un *Singleton*. También se utilizaron las ventajas semánticas del lenguaje, aprovechando el uso de clases y de interfaces. Sin embargo, una vez que se generaba el *thread* de atención de la transacción, se llamaba a un único método que empezó a crecer en cantidad de sentencias condicionales a medida que se iban agregando nuevas transacciones. Esto causó que el seguimiento, mantenimiento y evolución del código se complicara a medida que el *switch* crecía de tamaño.
- A favor de la arquitectura utilizada, el método que resolvía la transacción llamaba internamente a varios métodos dentro de él, dividiendo la carga de trabajo y las responsabilidades. Pero cada *switch* hacía llamadas a métodos propios definidos exclusivamente para cada solución. Esta divergencia se hacía más profunda si se comparaban *switches* de otras subsidiarias. La diferencia de criterios tomados entre los programadores era cada vez mayor a medida que surgían nuevas evoluciones.
- Más allá de los problemas de memoria mencionados, había fallas de memoria debido a objetos inicializados y nunca desechados correctamente, o punteros a clases de excepción que no se liberaban, lo que causaba que el incremento de memoria de forma descontrolada ocurriera no de forma sistemática, sino cuando había condiciones de error en el ambiente. Si bien esto se detectó, se logró descubrir la causa ya con el problema muy avanzado, y fue muy complicado erradicar estos *bugs*.

- Debido a la cantidad de fallas en el ambiente productivo, las puestas en producción consumían mucho tiempo en *revisiones* del trabajo realizado, controlando muchas variables que pudieran causar un problema. El *poco nivel de reutilización de código*, es decir, la baja adopción de componentes cerrados, probados y utilizados satisfactoriamente causó que el nivel de confianza en cada versión instalada sea baja. Los días donde se hacían puestas productivas eran algo traumáticas para los responsables, debido a que ocurría con frecuencia que, una vez instalado el nuevo *parche* o versión, algún error sucediera.
- En aspectos de conexión contra proveedores, era común el uso de protocolos propietarios que viajaban sobre TCP-IP. Pero luego muchos proveedores comenzaron a exponer *Web Services,* por lo que la integración de los métodos *Web*, y del seguimiento de cambios en los *WSDL* en C++ empezó a ser un problema. Se hacía necesario contar con algún lenguaje que fuera nativamente compatible con estas tecnologías. Para poder salir adelante, se usaron librerías *GNU,* como *gSOAP*[12]. Sin embargo las implementaciones con *gSOAP* no fueron totalmente satisfactorias a nivel de arquitectura, si bien funcionaban. Por ejemplo en la subsidiaria de Colombia, cada vez que se modificaba un solo *WSDL*, o se agregaba uno nuevo, se debían regenerar todos los *WSDL* de las demás conexiones. Esto generaba errores de versiones de *WSDL's* y muchas veces si los proveedores habían hecho cambios causaban errores en la plataforma.
- A nivel responsabilidades los *switches* empezaron a tener problemas graves cada vez que se requería conectarse con un nuevo proveedor. Las lógicas de los proveedores se iban sumando al gran bloque de código monolítico del *switch,* aumentando el acoplamiento de datos y de código. Visto como una *caja negra*, los *switches* comenzaron a ser programas de gran extensión que escuchaban a través de un socket solicitudes formateadas con el protocolo *ISO8583* y que reenviaban pedidos en múltiples protocolos propietarios y/o hacia *Web Services* de proveedores. No existía la división entre *switches* y *bridges* tal como se introdujo oportunamente durante el capítulo 2 de este trabajo.
- A nivel de *testing,* no había herramientas eficientes para probar un sistema como estos. Estos *switches* funcionan bajo eventos (conexiones entrantes o temporizadores). Se usaban herramientas propietarias que simulaban POS, donde se cargaban tramas de solicitudes de recarga, y que eran enviadas al puerto donde el *switch* escuchaba. Aun así, el *testing* no resultaba siempre de la misma manera para un mismo estímulo, ya que la transacción podía aprobarse o declinarse por falta de saldo, porque el comercio momentáneamente estaba deshabilitado, porque la base de datos tenía un *Connection String* distinto y apuntaba a otra instancia *SQL*, etc.

---

[12] *gSOAP* es un conjunto de herramientas para lenguaje C y C++ con la finalidad de crear interfaces *SOAP/XML* de *Web Services*, y otros *XML* genéricos. Estas herramientas analizan los *WSDLs* y los esquemas *XML* (por separado o como un conjunto) y mapean los mensajes *SOAP* / esquemas *XML* en código C o C++. [15]

- A nivel documentación, no solo la del diseño de la solución estaba sin actualizar, sino que tampoco se mantenía actualizada la documentación de los protocolos usados para comunicar puntos de venta con los *switches*, y los *switches* con los proveedores. En los sistemas transaccionales, la documentación del protocolo transaccional (los campos usados para cada transacción) es inclusive más importante que la del diseño, porque las modificaciones y evoluciones muchas veces involucran cambios en los datos que se envían y se reciben. Esto dificulta la recreación de pruebas, y el seguimiento de versiones de los puntos de venta, al no lograr definir qué versión enviaba cuál campo en sus solicitudes.

4.3 Desarrollo del Framework "TransactionKernel"

A mediados de Mayo de 2012, y luego de una larga tarea de observación y de recolección de ventajas y desventajas de los sistemas preexistentes, se trató de diseñar una capa de *software* que corriera sobre un lenguaje interpretado (*.NET Framework o Java)*. Los macro-objetivos de esta capa fueron: aumentar la reusabilidad de código, bajar los costos de mantenimiento, aumentar el grado de confiabilidad del código, desacoplar responsabilidades a través del reúso de componentes ya probados, y converger los criterios de los programadores. Se decidió en principio hacer la capa compatible con *.NET Framework* debido a la facilidad de integración con base de datos *SQLServer,* y con *WebServices,* como así también debido al manejo administrado de memoria. Sin embargo *TransactionKernel* fue diseñado para correr en cualquier lenguaje orientado a objetos, debido a que el aporte central está en la conceptualización de los factores típicos del procesamiento transaccional, a través de una semántica de clases y herencia. *TransactionKernel* se convierte en una capa por debajo de las soluciones, como una suerte de herramienta para facilitar los objetivos planteados, y que no está cerrado: todos los integrantes del equipo de desarrollo colaboraron aportando ideas en función de sus observaciones a nivel local.

Partiendo de la situación previa a este marco de trabajo, descrita en las secciones anteriores, los objetivos a nivel técnico de *TransactionKernel* fueron:

- **Encontrar los conceptos primarios repetidos y capitalizarlos en una propuesta de clases re-definibles por herencia**: Principalmente reconocer que el modelo de procesamiento transaccional se concentra en dos *actores*: *motores de entrada/salida*, y *transacciones*.
- **Encontrar todos los conceptos secundarios repetidos y capitalizarlos en una propuesta de clases re-definibles por herencia**: Son todos aquellos actores que contribuyen a los *motores* y a las *transacciones*. Se verán en las secciones siguientes.
- **Creación de una propuesta de secuencia transaccional única y re-definible**: Las transacciones analizadas contenían pasos similares entre unas y otras, y por lo tanto se pretende lograr una *secuencia única* que permita converger criterios de programación dentro de un equipo distribuido de desarrollo, y a la vez con capacidad para especificar y redefinir cada paso de la secuencia para cumplir con los requerimientos del sistema que se propone construir.
- **Refactorización a patrones:** En la medida de lo posible, las piezas de código que integrarán el *framework* deberían asemejarse a soluciones y recetas ampliamente utilizadas en el mundo, y que la Ingeniería de Software avala para la resolución de problemas comunes. De este modo, cualquier integrante que se sume a los equipos de desarrollo, y que utilice el *framework,* podrá aportar ideas y colaborar más rápidamente dada la tendencia a usar diseños conocidos globalmente.

- **Desacoplar el núcleo monolítico que era el *switch principal*, en *switches* y *bridges*:** Los *switches* solo se encargarían de concentrar transacciones de los distintos puntos de venta, y de validarlas contra la base de datos, comunicándose con los proveedores a través de *bridges*. Los *bridges* a su vez también son *switches*, pero a diferencia de éstos, tienen mínima interacción con la base de datos, y deben cumplir con el objetivo de adaptar las solicitudes a los canales de los proveedores.
- **Estandarizar el mecanismo de *testing*, implementando metodologías TDD[13]:** A partir de este *framework* se fomentaría el uso de casos unitarios de prueba, ejecutados a través de *NUnit*[14], o en caso de implementarse en otros idiomas, *JUnit*, *CUnit*, *SUnit*[15], etc.
- **Estandarizar el mecanismo de *logging*, con herramientas preexistentes y globalmente usadas:** No solo se pretendía converger el uso de herramientas no desarrolladas dentro de la compañía, sino que se quería aprovechar librerías que permiten escribir *logs* tanto en archivos, como en base de datos, o por un socket, por mail, etc.
- **Desacoplar lógicas en librerías separadas:** El *framework*, la solución, el *logger* y el *framework* TDD, *y* los protocolos usados para comunicarse, separados en librerías (en *.NET* separados en archivos *.dll* distintos).

En las siguientes sub-secciones, se describen en detalle los conceptos recolectados que se proponen como producto del armado de este *framework*.

---

[13] *TDD* es una práctica de programación que involucra otras dos prácticas: Escribir las pruebas primero (*Test First Development*) y Refactorización (*Refactoring*). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (*unit test* en inglés).

[14] *NUnit* es un marco de trabajo de pruebas de unidad para todos los lenguajes .NET.

[15] *JUnit, CUnit, SUnit:* son marcos de trabajo de pruebas de unidad para lenguajes Java, C/C++ y Smalltalk respectivamente. *NUnit* se desarrolló tomando como base *JUnit*.

4.4 Creación de una Secuencia de Procesamiento Redefinible

La secuencia que se propone en este *framework* está implementada por un método llamado *DoTransaction(),* que tendrá como objetivo llevar a cabo la transacción. Este método será el principal dentro de la clase base para modelar cualquier transacción; la clase abstracta que contiene al método se llama *AbstractTransactionHandler* y es uno de los conceptos comunes observados y capitalizados dentro de este *framework.*

De las observaciones realizadas se determinó una *secuencia de tres pasos* como mínimo: el primero de *Pre-procesamiento*, el segundo de *Procesamiento,* y el tercero de *Post-Procesamiento*. Para la propuesta que se hace en el *framework* se supone que estos tres pasos deberían ser suficientes. Cada una de las tres fases está compuesta por sub-pasos secuenciales opcionales, implementados como delegados a métodos de interfaz. Este mecanismo está soportado desde la teoría por el patrón de diseño *Template Method*, que habilita a un algoritmo a diferir en ciertos pasos en función de la subclase instanciada. La estructura del algoritmo (su secuencia) no cambia, pero algunos de sus pasos son redefinidos y atendidos de forma diferente en cada subclase [10]. La implementación con delegados en vez de métodos directamente permite que sólo puedan implementarse aquellos sub-pasos que le interesen al diseñador de la transacción, obviando aquellos que no se deseen. En el último caso, esos delegados quedarán *desconectados*. De hecho, los delegados que no estén apuntando a un método serán considerados por defecto con valor de retorno *true. DoTransaction()* ejecuta estas tres fases llamando a tres métodos. Además la clase *AbstractTransactionHandler* contiene un campo privado que sirve para habilitar o deshabilitar la ejecución de cualquiera de las tres fases, si por algún motivo durante la secuencia se decide no ejecutar una de las etapas (ver Anexo A).

4.4.1 Etapa de Pre-Procesamiento (DoFirstStage())

La fase de *pre-procesamiento* se capitaliza en un método privado llamado *DoFirstStage()*, cuyo flujo de datos se muestra en la Figura10.

El paso (1) tiene la responsabilidad de recolectar los datos de interés que la transacción usará, de la trama de entrada enviada por el cliente. El método se llama *GetRequirement()*, que deberá estar apuntado oportunamente por el delegado *GetRequirementMethod()*, del tipo *GetRequirementDelegate*. Dentro de ese proceso de recolección, se deberían validar que todos los campos esperados estén dentro de la estructura de requerimiento, y en el caso contrario tomar una decisión de proceder o no con la transacción. El método debería devolver *true* si se recuperaron todos los datos esperados de forma correcta, *false* en caso contrario.

El paso (2) se diseñó para albergar todas aquellas tareas que se encarguen de ejecutar las validaciones contra un mecanismo de persistencia. Los datos entrantes de una transacción casi siempre deben ser validados: por ejemplo, hay que validar si la terminal de dónde provino la transacción está habilitada, si el comercio está

habilitado, si el comercio tiene saldo, etc. Además, en esta etapa se debería inscribir la transacción como un movimiento en un histórico de transacciones. Se debe guardar el resultado de las validaciones realizadas durante esta misma etapa. Este paso se implementa en el método *PreProcessTransaction()*, y la secuencia llama al método a través del delegado *PreProcessTransactionMethod()* del tipo *PreProcessTransactionDelegate*, siempre que el paso *GetRequirementMethod()* haya devuelto previamente *true*.

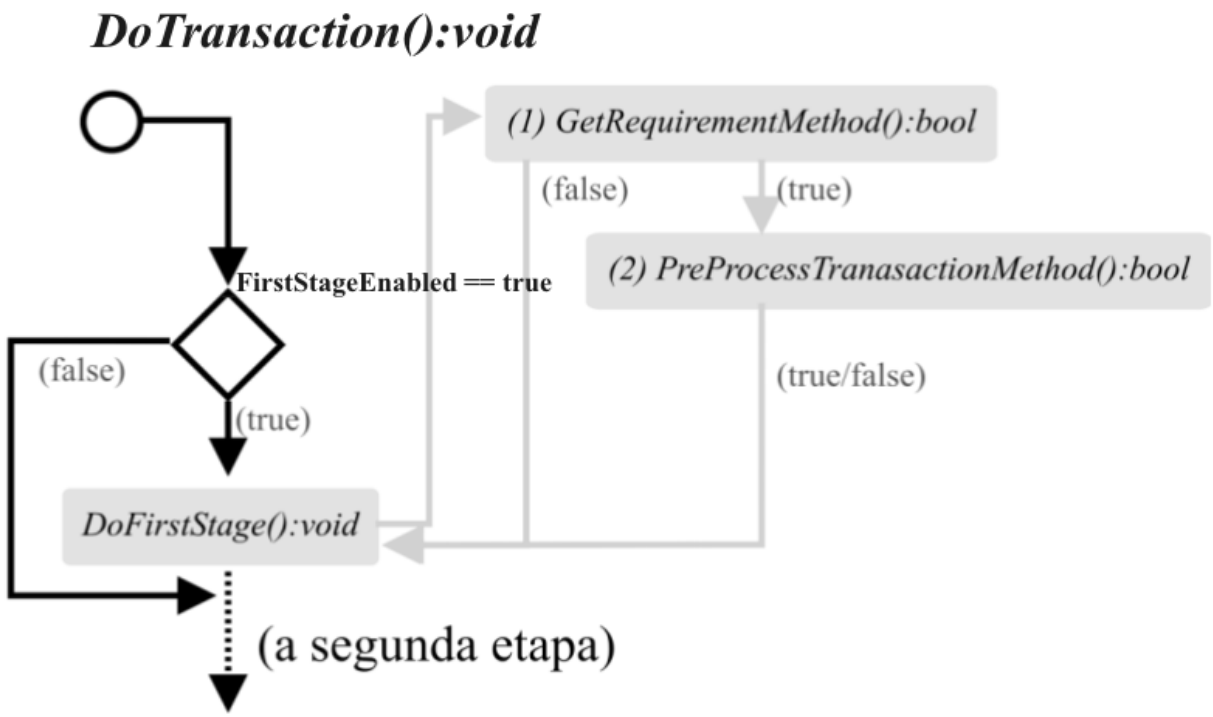

Figura 10 - Secuencia de la primera etapa

4.4.2 Etapa de Procesamiento (DoSecondStage())

La segunda fase (*Procesamiento*) se capitaliza en un método privado, llamado *DoSecondStage()*, cuyo flujo de datos se muestra en la siguiente Figura 11.

El paso (1) tiene la responsabilidad de decidir de forma dinámica si los siguientes pasos de la transacción se seguirán ejecutando dentro del objeto actual, o si la ejecución se delegará hacia otro objeto heredado de *AbstractTransactionHandler*, dentro del mismo sistema. Este mecanismo permite interconectar distintos objetos del tipo *AbstractTransactionHandler* entre sí, para crear una cadena de *handlers* que representen a la transacción a lo largo de toda su vida. Para eso, la secuencia ejecuta un *delegate* llamado *ForwardHandlerFactoryMethod()*, del tipo *ForwardHandlerFactoryDelegate*. El método es del tipo *fábrica*:

debe retornar una instancia válida de *AbstractTransactionHandler* (si la decisión es continuar la ejecución en este nuevo objeto), o *null* si la decisión es continuar los pasos dentro del mismo objeto. En el primer caso, *this* llamará como próximo paso al método *DoTransaction()* (paso 2b) de ese nuevo objeto, generando una cadena de responsabilidad para la resolución de la solicitud (en este caso, se refactorizó para implementar un patrón *Chain of Responsibility*). Este patrón trabaja con una cadena de *handlers* para un requerimiento dado, tal que si un objeto no puede resolver el pedido, se lo pasa al siguiente objeto de la cadena de responsabilidad. En el segundo caso, *this* contiene la implementación tanto para resolver la transacción de forma local, como para re-enviarla por un *socket*, y por lo tanto continua la ejecución de sub pasos dentro del mismo objeto.

Si el caso efectivamente requiere que la trama sea reenviada hacia un nodo externo, el próximo paso (paso 2a) tendrá la responsabilidad de armar el requerimiento que terminará siendo re-enviado hacia el exterior. Entonces, la secuencia llamará al delegado *BuildRequirementMethod()* del tipo *BuildRequirementDelegate*, que debería estar apuntando a *BuildRequirement()*. La implementación debe retornar *true* si pudo crear un requerimiento, y en caso contrario *false*.

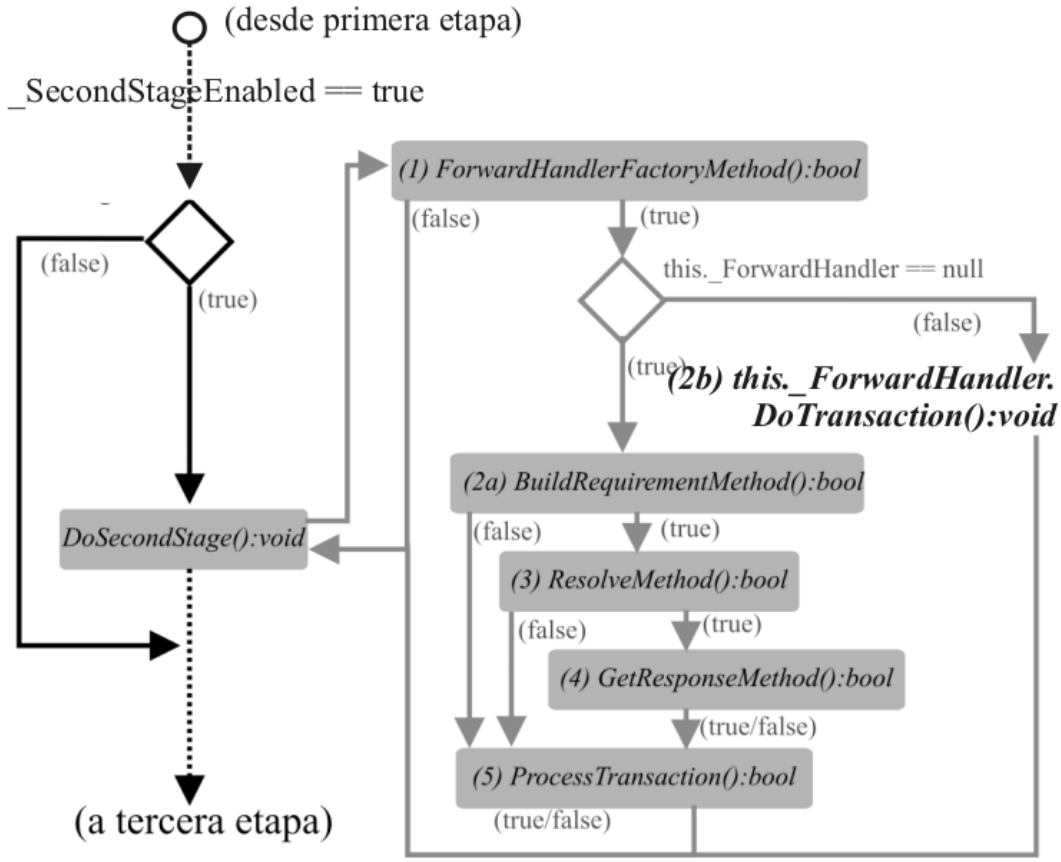

Figura 11 - Secuencia de la segunda etapa

El paso (3) tiene la responsabilidad de resolver la transacción, entendiendo como *resolver* a conectarse, enviar, recibir, y desconectarse del nodo externo. Como las formas de conexión contra esos nodos pueden variar, o tal vez no trabajen con protocolos orientados a la conexión, queda libre para los objetos concretos la implementación particular de todo este mecanismo. Este método en realidad terminará llamando a los puntos de entrada de los conceptos nomenclados como *motores de salida*, que son un tipo de elemento del *framework* para capitalizar aquellas entidades responsables de las comunicaciones hacia el exterior. Esos *motores de salida* son los que desacoplan a la transacción de toda la lógica de comunicación contra un nodo externo (otro autorizador por ejemplo). La secuencia llamará entonces a un delegado *ResolveMethod()*, del tipo *ResolveDelegate*, que oportunamente estará apuntando a un método *Resolve()* que cumpla con el prototipo propuesto por ese delegado. La implementación del método deberá retornar *true* si la transacción contra el nodo externo terminó de forma esperada, en caso contrario *false*.

El paso (4) es un delegado llamado *GetResponseMethod()*, del tipo *GetResponseDelegate*, que deberá estar apuntando a un método *GetResponse()*. Este método solo se ejecuta cuando el paso (3) (*ResolveMethod()*) retorna *true*, indicando que existe una respuesta fehaciente del nodo externo. Esa respuesta puede ser aprobada o desaprobada, y puede contener datos de utilidad como el código de autorización, un código de referencia, etc. El objetivo es evaluar el resultado de la transacción en función de esos datos, y definir el resultado dentro de la estructura de contexto. Debe retornar *true* si la respuesta se pudo procesar sin problemas, más allá del resultado propio de la transacción. En caso que algún dato recibido del nodo externo no sea correcto, o no cumpla con las especificaciones, debería retornar *false*.

Por último, el paso (5) tiene como objetivo definir el estado de la transacción, en cualquier caso inesperado dentro de *DoSecondStage()*. Por ejemplo, resulta útil para cuando hay problemas de conexión durante el paso (3) (un *Time-Out*, un corte de conexión inesperado, etc.). En este paso se puede tomar una decisión de rechazar la transacción, reintentarla, reversarla, etc. *ProcessTransactionMethod()* es un delegado del tipo *ProcessTransactionDelegate*, que deberá estar apuntando a un método *ProcessTransaction()*, que cumpla con el prototipo de ese delegado.

4.4.3 Etapa de Post-Procesamiento (DoThirdStage())

La tercera fase (post-procesamiento) se capitaliza en un método privado de *AbstractTransactionHandler*, llamado *DoThirdStage()*, cuyo flujo de datos se muestra en la siguiente Figura 12.

El paso (1) tiene la responsabilidad de crear la respuesta para el cliente, usando todos los datos recolectados durante la transacción. Para eso debe completar una estructura de respuesta con los datos correspondientes, y serializarlos en un *array* de *bytes*. El paso concreto es *BuildResponseMethod()*, del tipo *BuildResponseDelegate*.

El método debe responder *true* si pudo generar una estructura de respuesta, y su correspondiente transformación a un *array* de *bytes* válido. En caso contrario, deberá responder *false*.

El paso (2) tiene la responsabilidad de decidir si envía el arreglo armado en el paso (1). El método *Reply()* debe verificar, por ejemplo, si por el estado de la transacción debe efectivamente enviar la respuesta, o por lo contrario forzar una *no respuesta* al cliente. El paso concreto en la secuencia se implementa como un delegado llamado *ReplyMethod()*, del tipo *ReplyDelegate*, siempre que el paso (1) responda *true* (es decir que hay disponible un *array* de bytes de respuesta). Como valor de retorno devuelve *true* si pudo enviar la respuesta, o *false* en caso contrario.

El paso (3) tiene la responsabilidad de realizar la tarea de post-procesamiento propiamente dicha sobre la base de datos, usando todos los datos obtenidos durante la transacción. Es precisamente en este punto donde se debería actualizar el *histórico de transacciones*, de modo de guardar tanto el estado final de la transacción, como todos los datos complementarios que fueron surgiendo durante el proceso de la misma. El paso concreto se implementa como un delegado llamado *PostProcessTransactionMethod()*, del tipo *PostProcessTransactionDelegate*. Cabe acotar que la llamada a este delegado se hace después de *Reply()* para que el post-procesamiento no aumente el tiempo de vida de la transacción del lado del cliente, reduciendo el tiempo de comunicación. El delegado responde *true* si no hubo ningún problema para realizar las tareas eventuales de actualización de datos. Por lo contrario deberá devolver *false*.

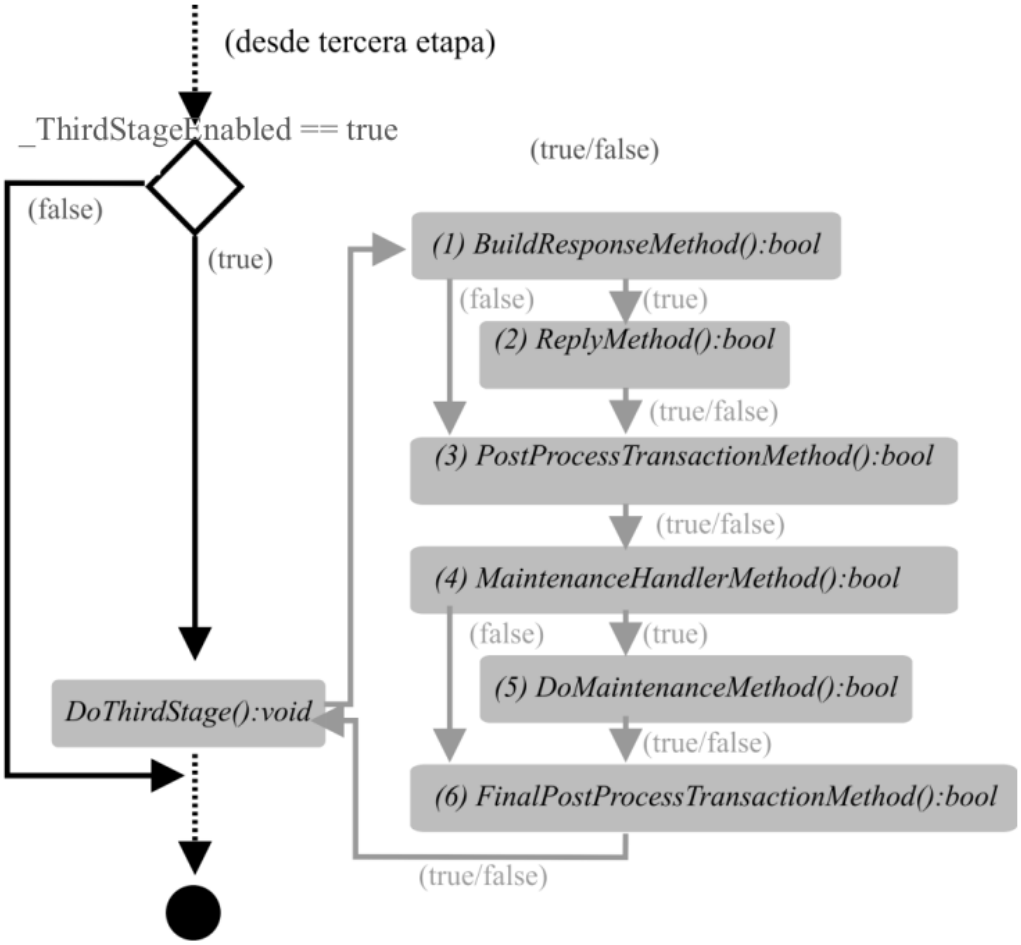

Figura 12 - Secuencia de la tercera etapa

El paso (4) tiene la responsabilidad de analizar si es necesario crear una instancia del tipo *AbstractTransactionHandler*, de modo de configurar una transacción encadenada de mantenimiento. Por ejemplo, en recargas de tiempo aire, se aprovecha este paso para hacer el mantenimiento de consultas de saldo contra proveedores. Otro uso muy común es el de actualizar los saldos de cuentas corrientes de las tarjetas de débito / crédito que fueron procesadas durante la transacción. En definitiva, todas las tareas catalogadas como *mantenimiento* tienen la característica que el cliente (POS, ATM, etc.) no debe estar conectado para esperar a la finalización de estas tareas. Por lo tanto, todas aquellas que no requieran del cliente conectado (dado que el cliente no precisa en tiempo real el resultado devuelto por las tareas), deben ser ejecutadas posteriormente a la respuesta enviada por el sistema. Este paso deberá implementar una fábrica de instancias (patrón *Factory Method*), similar a *ForwardHandlerFactory()*. Es decir, el paso debe retornar eventualmente una instancia válida del tipo *AbstractTransactionHandler* que configure todas las tareas de mantenimiento. Este paso deberá

retornar *false* en caso de no necesitar hacer mantenimiento alguno, y *true* si efectivamente es necesario. En esta última situación, se ejecutará el paso (5), que tiene la responsabilidad de llamar al *DoTransaction()* de la instancia creada en el paso anterior. *DoMaintenanceHandlerMethod()* hace concreto este paso y es un delegado del tipo *DoMaintenanceHandlerDelegate.*

Por último, el paso (6) llama al delegado *FinalPostProcessTransactionMethod(),* del tipo *FinalPostProcessTransactionDelegate*, que tiene como responsabilidad actualizar y/o resguardar la información obtenida como resultado, luego de realizar las tareas del paso anterior (5). Debería responder *true*, si pudo realizar el trabajo de actualización, o *false* en caso contrario.

4.5 Conceptos Primarios y Secundarios del Dominio Transaccional

A continuación se describen aquellos conceptos recolectados de las observaciones de casos preexistentes. Estos conceptos surgen del análisis realizado, enfocado a todos aquellas características que se hacen comunes entre una solución de procesamiento y otra. Algunas tendrán mayor protagonismo que otras, y lo más importante es su característica de *propuesta*: Como tal, es un conjunto abierto de conceptos que podrían modificarse para sumar nuevos elementos en el futuro, quitar aquellos que no fueron de utilidad, y readaptar aquellos que fueron concebidos y que luego fueron más eficaces enfocados de una u otra forma.

4.5.1 Bitácora(*Loggers*)

El concepto de *Bitácora* está constituido por la necesidad de contar con una herramienta / librería de carácter auxiliar en el sistema de procesamiento, que pueda registrar todos los pasos que se consideren importantes durante la vida de una transacción, o del servidor concurrente en sí mismo. Gracias a esta característica, los *logs* son testimonios de utilidad para evidenciar en dónde puede haber un problema cuando se presentan fallas en el procesamiento de una eventual transacción, y de última, poder hacer los reclamos correspondientes entre los interesados. La evidencia construida por un *log* no solo afecta los *stakeholders* técnicos, sino a los involucrados en otros departamentos, como el equipo comercial y ejecutivo (cuando tienen que hacer reclamos por dinero mal cobrado, puntos mal asignados o crédito consumido/recargado erróneamente).

Dentro de esta propuesta de *framework* se declara que es de suma importancia contar con un mecanismo de *logging* para cualquier sistema transaccional que se desee construir. Cualquier sistema de *bitácora* es una herramienta valiosa para poder hacer un seguimiento cronológico de lo que sucede durante el desarrollo de una transacción. De la experiencia de todas las soluciones mantenidas y/o desarrolladas por el autor, se puede afirmar que todos los sistemas transaccionales han presentado algún mecanismo de *log*. La gran mayoría utilizaba *mecanismos de logs propietarios*. Si bien desarrollar un generador de *log* que escriba en un archivo es

relativamente fácil, las tecnologías disponibles y las herramientas mundialmente usadas presentan características y opciones mucho más amplias que las de un mecanismo propietario. Es por eso que, se recomienda que cualquier sistema transaccional desarrollado bajo este *framework* utilice alguna herramienta de uso globalizado, y que se depreque cuanto antes el uso de mecanismos propietarios de *logging*. Particularmente, la versión construida de *TransactionKernel* soporta compatibilidad con *log4net.* Para esta propuesta de *framework*, se eligió *log4net* porque este mecanismo contaba con una serie de características interesantes, particularmente a la hora de desacoplar las responsabilidades principales de procesamiento con las de *logging,* y porque facilitaba la integración y uso de estas librerías en los equipos de desarrollo distribuidos geográficamente.

Algunas de las características interesantes que se fijaron como requisito en esta propuesta, para un sistema de *log* son:

- **Soporte para múltiples tecnologías**: Si bien *TransactionKernel* está diseñado para .NET, los conceptos utilizados son transferibles más allá de la tecnología / plataforma de implementación. Para hacer viable una eventual migración a otra plataforma, es necesario contar con librerías auxiliares independientes de la tecnología.
- **Posibilidad de salida multimedio**: Es importante poder *loguear* en varios medios, y no sólo en archivos dentro del sistema donde corre el servicio. Por ejemplo, ha resultado de gran utilidad contar con un *logger* TCP, accesible vía *Telnet* desde una PC remota. De este modo, por ejemplo, es posible observar cronológicamente las transacciones entrantes, en tiempo real. Así mismo, un *logger* con escritura hacia base de datos, ayuda mucho para poder filtrar transacciones entre muchas otras simultáneas (a través de las operaciones CRUD). Por último, un *logger* a consola es muy útil durante las etapas de desarrollo de un servicio transaccional.
- **Arquitectura de log jerárquica:** La mayor ventaja de esta característica es filtrar líneas de *log* a través de niveles de criticidad de los mensajes que se expresan en ellas. De esta forma, se pueden clasificar los mensajes de *log* escritos y así se puede decidir escribir aquellos cuya importancia supere el umbral de criticidad configurado, en un momento dado sin necesidad de recompilar la librería. En la mayoría de los sistemas de *log*, estos niveles de criticidad son (FATAL, ERROR, WARNING, INFO, DEBUG).
- **Configuración por XML o por algún otro medio de forma dinámica:** Es una ventaja deseada para un sistema de *logging*. Principalmente es importante si se compara con la configuración vía líneas de código, que ante una eventual reconfiguración del *logger*, inevitablemente se debe recompilar todo el sistema.
- **Arquitectura funcionalmente comprobada, y alto rendimiento de escritura en el medio de persistencia:** Poder reutilizar código con un funcionamiento estable y con una arquitectura

comprobada a nivel global es importante y beneficioso, principalmente por la confiabilidad que brinda la librería. Particularmente, las escrituras en el *log* (que de por cierto son muchas en un sistema transaccional con concurrencia alta), pueden afectar al rendimiento transaccional del sistema si se menosprecia el mecanismo y/o la ingeniería de una librería de *logging* desarrollada de forma interna. Por esto es importante usar herramientas de *logging* que sean confiables.

- **Disponibilidad de la librería:** Para esta propuesta es importante el foco en la reutilización y la reducción de los costos de mantenimiento del código. Uno de los caminos para llegar a la reutilización modular de código está relacionado con la capacidad de homologar criterios en los equipos de desarrollo, y el uso acordado de librerías y metodologías de trabajo. El *grado de disponibilidad* de las librerías a reutilizar es sumamente importante para poder reutilizarlas fácilmente, múltiples veces, en distintas situaciones y en diversos equipos de trabajo distribuidos. Las librerías usadas por la comunidad generalmente son de fácil acceso a través del mismo IDE. En cambio, las librerías creadas de forma interna tienden a ser más personales y menos accesibles a los demás colegas, particularmente cuando el desarrollador comienza a generar nuevas versiones de la misma.

4.5.2 Contextos (*Context*)

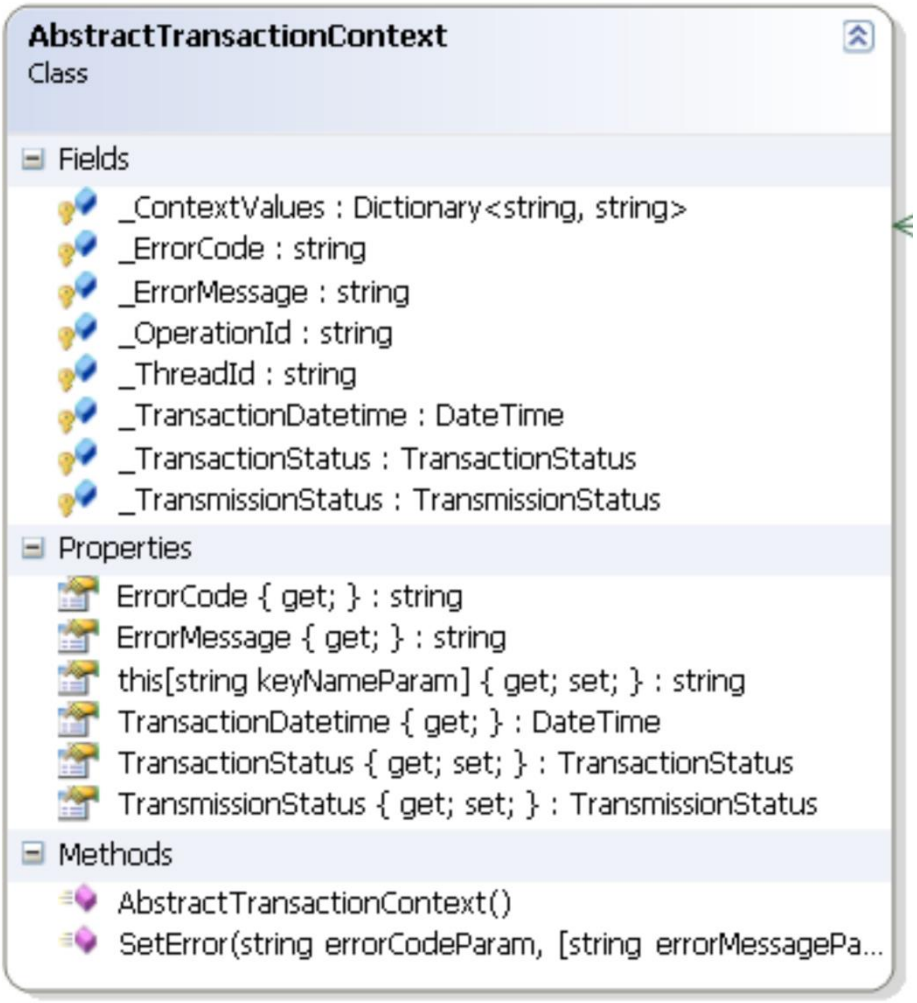

Figura 13 - Clase abstracta de un contexto base

El concepto de *Context* (contexto transaccional) está constituido por un objeto que cobra vida cuando se inicia una transacción y que se deshecha cuando finaliza la misma. Tiene el objetivo de almacenar todos los datos que sean interesantes para la transacción en curso, y que en algún momento participarán directa o indirectamente en el procesamiento. Comparativamente, hace las veces de *pila* durante la ejecución de una función de lenguaje C, (las pilas son los espacios de memoria donde se guardan los parámetros usados por la función de forma local). De hecho, la idea de un contexto transaccional surge de la idea de las *pilas* usadas en funciones.

El *contexto transaccional* pertenece solo a un hilo de ejecución (solo a una transacción) y cada transacción debe generar el suyo. También debe ser conocido por los objetos involucrados en la resolución de la transacción, de modo de poder interactuar con el contexto. Es por eso, que el contexto generalmente se pasa de constructor a constructor, en un esquema típico de *inyección de dependencias*, siempre que el objeto a construir tenga la

necesidad de hacer algo con esos datos. Habrá en el sistema tantos objetos *Context*, como transacciones simultáneas haya.

De las observaciones realizadas de sistemas preexistentes se decidió enmarcar este concepto en una clase abstracta llamada *AbstractTransactionContext,* de modo que sea extensible para una solución dada. Sin embargo, se detectaron algunos campos que son de utilidad en casi todas las soluciones, por lo que quedaron elegidos en esta conceptualización y dentro de la clase abstracta. A continuación se describen estos campos y métodos, y sus objetivos:

Conceptos	**Objetivo**
_ContextValues	Es una colección de pares *clave valor*, para almacenar datos específicos dentro del contexto, sin la necesidad de escribir campos específicos en las subclases. De este modo, se evita que el usuario del *framework* tenga que definir todos los campos en la subclase que se usaran en el sistema.
_ErrorCode, _ErrorMessage, SetError()	Son dos propiedades para manejar un eventual código de error, y un mensaje opcional respectivamente. *SetError()* es un método de acceso (solo escritura) para establecer un código de error. De forma obligatoria requiere como parámetro el código de error a establecer, y opcionalmente el mensaje de error asociado. Todas las transacciones analizadas en los sistemas observados para la construcción de esta propuesta usaron una variable o un campo para resguardar un eventual código de error. Es por eso que se decidió incorporarlo a la clase base del concepto *Context*.
_OperationId	Es un código que identifica el tipo de transacción en curso, por ejemplo una notificación, una consulta de saldo, una consulta de puntos, una compra con tarjeta de crédito, etc. Este código que describe el tipo de transacción debe viajar obligatoriamente en el protocolo de comunicación, sino el sistema no tendrá forma de entender qué debe hacer con la transacción. Es por eso que, como todas las transacciones tienen la obligación de identificarse desde el momento en que se transmiten por un medio, se

	considera para la propuesta que este campo debe estar en la clase base del concepto *Context*
_ThreadId	La relación entre hilos de ejecución y transacciones en un sistema es, para esta propuesta, uno a uno. Cada transacción entrante tendrá asignado un hilo de ejecución (*thread*), por lo que se decidió resguardar el valor de identificación del hilo que brinda el sistema operativo, en un campo dentro de la clase base del concepto *Context.* En rigor este valor lo puede brindar el sistema operativo, o el *framework* encargado de manejar los hilos, por ejemplo .NET *o Java.*
_TransactionDatetime	Todas las transacciones analizadas en los sistemas bajo observación utilizan la fecha para diferentes fines; el uso principal surge cuando la transacción se guarda como movimiento en una tabla histórica de movimientos y/o transacciones.
_TransactionStatus, _TransmissionStatus	Son dos enumeraciones usadas para resguardar el estado de la transacción (_TransactionStatus), y el estado de una eventual retransmisión del pedido a otro nodo externo (_TransmissionStatus). Los valores con los que se forman estas enumeraciones se obtuvieron de observaciones y de análisis de los sistemas observados, y de los estados comunes establecidos en las transacciones procesadas por estos sistemas. Si bien, en esta propuesta no se pueden predecir los estados necesarios por un usuario específico, se acumulan aquellos comunes, intentando que éstos sean de utilidad. Por ello se agregan como propiedades dentro de la clase base del concepto *_Context*.

Tabla 1 - Descripción de los campos en la clase base del concepto *Context*

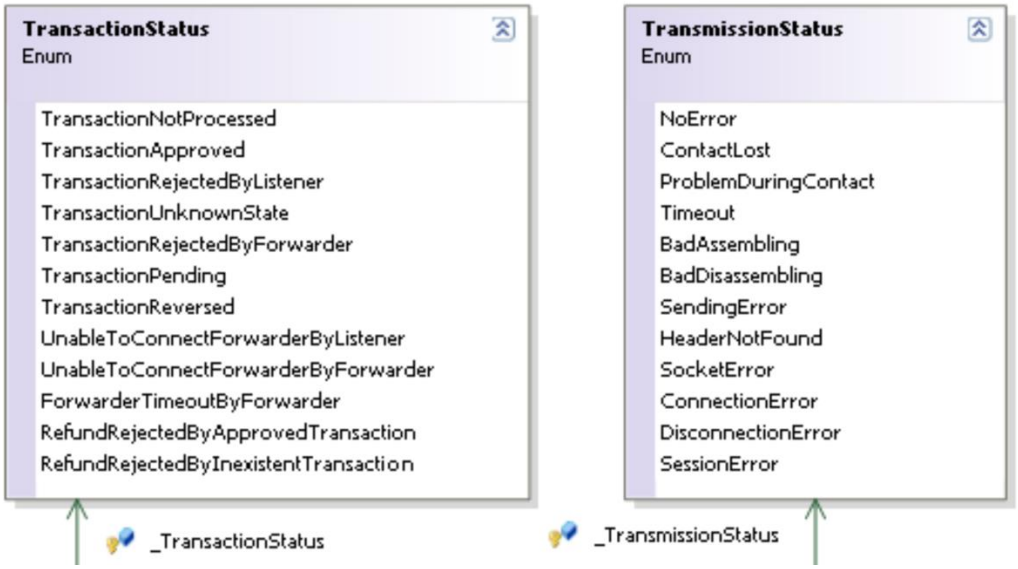

Figura 14 - Estados típicos recolectados tanto para una transacción como para la transmisión subyacente

4.5.3 Analizadores (*Parsers*)

4.5.3.1 Analizador de Protocolo (*Parser*)

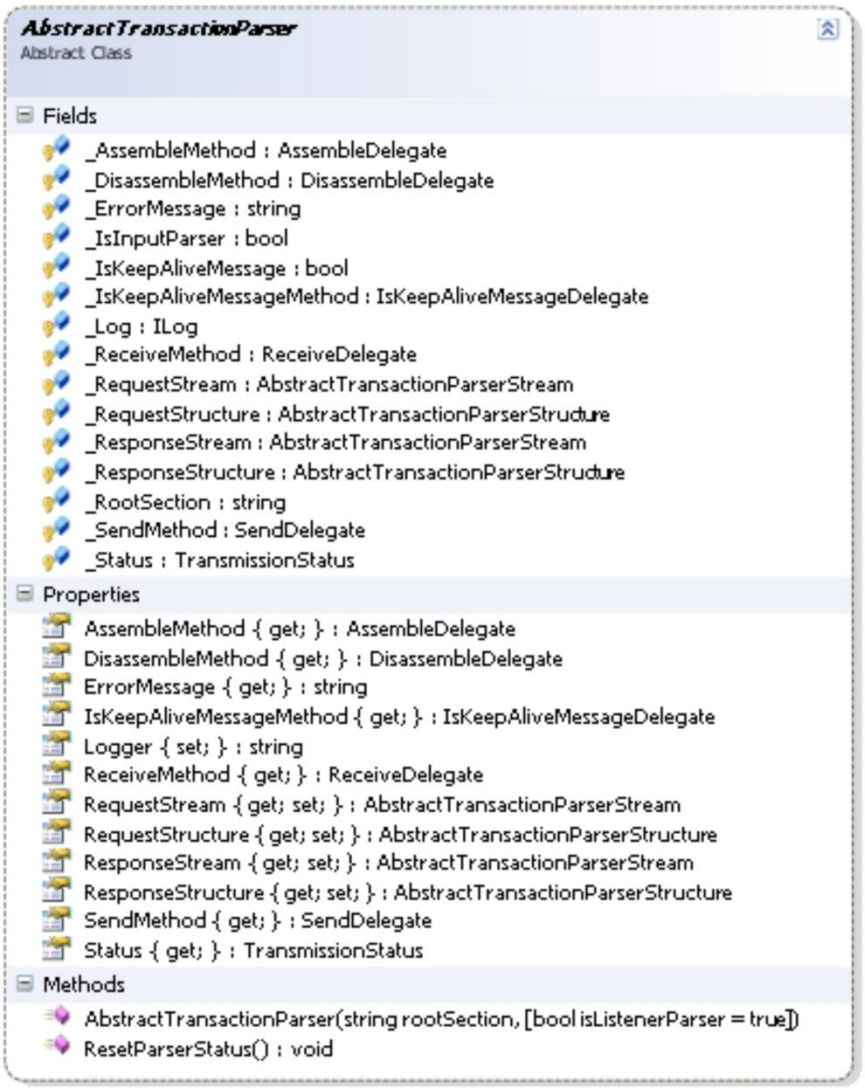

Figura 15 - Clase abstracta conceptual de un analizador base

El concepto *analizador de protocolo* (*Parser*) surge de la necesidad de interpretar la información que viaja entre dos puntos. Esa información viaja como una *corriente de bytes*, formateada por un protocolo acordado entre un cliente y un servidor transaccional, o entre dos servidores transaccionales (nodos). Los objetivos principales de este concepto son dos:

- Implementar la lógica de serialización de información (desde una estructura de analizador (*ParserStructure*) a una corriente de analizador (*ParserStream*))

- Implementar la lógica de deserialización de información (desde una corriente de analizador (*ParserStream*) a una estructura de analizador (*ParserStructure*))

Un *parser* es una pieza fundamental en un servidor concurrente para los fines de esta propuesta de *framework* transaccional, porque los *motores de entrada* los usarán para capturar la información que llega desde un cliente remoto como una *corriente de bytes*, y luego para responderle a ese cliente de la misma forma. Sin embargo, la transacción generada por el motor (con el objetivo de procesar la información), no verá la información como una *corriente de bytes*, sino como dos estructuras: una estructura de requerimiento con los datos entrantes al servidor (deserializados), y otra de respuesta, con los datos que el servidor debe llenar para armar la respuesta, que luego se enviará. Además un *parser* puede conocer la forma de enviar y de recibir información a través de un objeto de conexión (como un *socket*).

Los *parser* son un concepto central para coordinar otros conceptos que se explican en los párrafos siguientes, como las *estructuras de analizador*, las *corrientes de analizador*, los *campos de analizador* y los *subcampos de analizador*. A continuación se describen los campos, propiedades, métodos y otros elementos que son contenidos dentro de la clase base de este concepto, de modo de agrupar las características comunes observadas en campo de varias soluciones transaccionales.

Conceptos	**Objetivo**
_RequestStructure, _ResponseStructure	Son dos objetos del tipo *Parser Structure*, de modo que la instancia del analizador conozca y administre las estructuras asignadas para resguardar el requerimiento y el armado de la respuesta. Lógicamente las instancias de estas estructuras deben ser de una subclase concreta de *Parser Structure* válida.
_RequestStream, _ResponseStream	Son dos objetos del tipo *Parser Stream*, de modo que la instancia del analizador pueda resguardar las *corrientes de bytes* entrantes y salientes. Cuando se realiza una lectura de una *corriente de bytes* entrantes, el *stream* resultante debería almacenarse en **_RequestStream**. En cambio, cuando se ensambla una respuesta, se utilizan los datos almacenados en **_ResponseStructure,** que terminan siendo transformados en una *corriente de bytes* lista para ser enviada al cliente que inició el requerimiento. Esa corriente de bytes resultante debería resguardarse en **_ResponseStream.**

_AssembleMethod, **_DisassembleMethod**	Son dos objetos del tipo **AssembleDelegate** y **DisassembleDelegate** respectivamente. Se utilizan para apuntar a los métodos que tendrán las implementaciones para ensamblar una respuesta, y desarmar un requerimiento. Estos métodos deben implementarse, si el analizador hereda la interfaz **IAssembleable.** Si no lo hereda (aunque no tiene mucho sentido desde el punto de vista práctico) ambos delegados deberán asignarse con valor *null*.
_SendMethod, **_ReceiveMethod,** **_IsKeepAliveMessageMethod,**	Son tres objetos del tipo, **SendDelegate, ReceiveDelegate e IsKeepAliveMessageDelegate** respectivamente. Se utilizan para apuntar a los métodos que tendrán las implementaciones para enviar *streams*, recibir *streams,* y detectar eventuales mensajes de *Keep Alive*. Los métodos a los cuales deberían apuntar son aquellos que se obtienen como implementación obligatoria cuando el analizador hereda la interfaz **ICommunicable.** La detección temprana de mensajes de *Keep Alive* sirve, si así se deseara, para filtrar aquellos mensajes que no son transaccionales por sí mismos, sino para verificar la disponibilidad de conexión desde un cliente remoto. Si el contenido de los mensajes de *Keep Alive* es inentendible por el protocolo transaccional, el *logging* de mensajes puede evidenciar errores a la hora de entenderlos como mensajes dentro del protocolo establecido. Debido a esto, para evitar agregar líneas de *log* innecesarias cuando se capturan estos mensajes, se propone este delegado, que debe apuntar a un método con la implementación para detectar por fuera del protocolo, los mensajes de *Keep Alive,* y responder el mensaje en la forma acordada entre las partes. .
_Log	Es un campo que se carga con la referencia del objeto del *logger* configurado para la instancia de ejecución del sistema.
_IsInputParser	Es un campo del tipo *bool* que solo es accesible desde adentro del analizador. Debe asignarse con el valor *true* para indicar que el *parser*

	es parte de un *motor de entrada* (*Input Transactional Engine*), y *false* si es parte de un *motor de salida* (*Output Transactional Engine*). La razón para hacer esta diferencia reside en si el protocolo tiene estructuras diferentes para manejar los mensajes entrantes y salientes. Según la convención a la que se adhiere esta propuesta, si el analizador pertenece a un *motor de entrada*, los mensajes entrantes son requerimientos, y los mensajes salientes son respuestas. Si el analizador pertenece a un motor de salida, los mensajes salientes (un requerimiento hacia un nodo externo) son manejados dentro de la implementación como respuestas, y los mensajes entrantes (respuestas desde un nodo externo) como requerimientos. En conclusión, todos los mensajes salientes del servidor son respuestas, y todos los entrantes son requerimientos más allá de donde esté situado el analizador.
_IsKeepAliveMessage	Es un campo del tipo *bool* que indica si el mensaje leído es o no un mensaje de *Keep Alive*. Esta propiedad es interna al analizador instanciado, y es asignada en el método de lectura apuntado por **_ReceiveMethod**. La visibilidad al exterior se da a través del método apuntado por **_IsKeepAliveMessageMethod**
_Status	Es un campo del tipo *TransmissionStatus*, que originalmente guardaba el estado de la conexión. Esta propiedad era utilizada solo cuando el analizador heredaba la habilidad *comunicable* (heredaba de la interfaz **ICommunicable**). El estado era útil para detectar si había un error en el envío, o en la recepción de una *corriente de bytes*. Actualmente se extendió para evidenciar inconvenientes durante el proceso de ensamblado y desensamblado de datos; de esta forma pasó a ser un objeto que resguarda el estado interno del analizador.
_ErrorMessage	Es un campo interno del analizador del tipo *string*, con el objetivo de resguardar un eventual mensaje de error asociado a alguno de los estados internos del mismo, almacenado en _Status.
_RootSection	Es un campo interno del analizador del tipo *string*, que opcionalmente almacena la clave del nodo XML donde se guarda la configuración específica de los campos de un protocolo multiuso, según una

	implementación de negocio específica. Es decir, la configuración del protocolo se define en un archivo XML, donde se definen los campos utilizados por transacción. Luego, el analizador lee este archivo para entender el formato de los mensajes entrantes. Por ejemplo, los analizadores de ISO8583 de esta propuesta, pueden configurarse para interpretar qué significa el contenido de los campos que transporta durante una transacción específica, para una solución determinada. Ese significado puede servir luego para escribir una *biografía transaccional* de esa solución de forma automática, y de esta forma se intenta achicar la brecha entre la documentación del protocolo y el código que la implementa.
ResetParserStatus()	Es un método para reiniciar el estado de un *parser* a su estado inicial, reasignando tanto al campo **_Status** como a **_ErrorMessage** a sus valores por defecto.

Tabla 2 - Elementos comunes definidos dentro de la clase base del concepto *Parser*

4.5.3.2 Estructura, Campo y Sub-Campo de Analizador (*Parser Structure, Parser Field, Parser Subfield*)

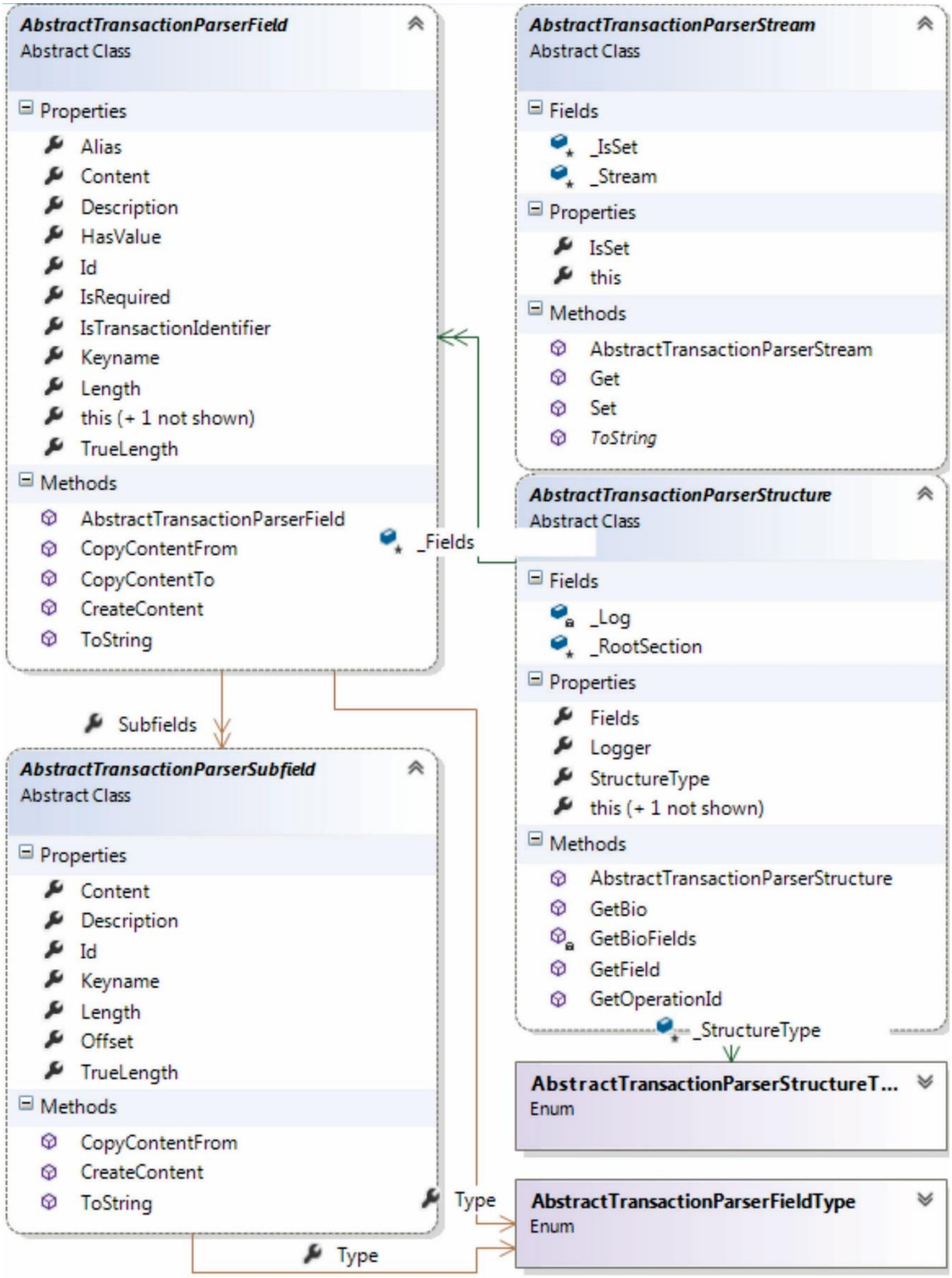

Figura 16 - Diagrama de Clases involucradas en las estructuras, campos y sub-campos de los analizadores.

Estos tres conceptos son funcionales exclusivamente al concepto *Parser* como un todo. El concepto de *Parser Structure* se implementa como un objeto que contenga todos los campos y métodos para almacenar los datos obtenidos desde una *corriente de bytes*, o para guardar los datos que se serializarán a una *corriente de bytes*. Una *estructura de analizador* puede especificarse por herencia para almacenar información específica de un determinado protocolo. Es decir, que las clases hijas que hereden de *Parser Structure* pueden especificarse para representar las características de un protocolo de comunicación dado. Sin embargo, resultó conveniente armar dentro de la clase base una colección de objetos del tipo *Parser Field,* para almacenar la información de forma genérica (sin conocer de antemano la especificación de una solución). Se puede resumir que las *estructuras de analizador* en sí mismas son colecciones de *campos de analizador* (*Parser Field*) configurables por un archivo XML, donde se guardará toda la información descriptiva del protocolo para una solución dada. Cualquier otro dato deducible de la información dentro de estos campos (por ejemplo, los prefijos de longitud que preceden a algunos campos, los separadores entre campos, etc.) se resguarda dentro de la estructura de forma automática, sin necesidad de definir un campo para tal fin.

A su vez, el concepto de *Parser Field*, es el de administrar la información contenida en un *Parser Structure* en campos individuales, que en conjunto forman un *todo* equivalente a la estructura que los contiene. De este modo, cada *Parser Field* se compone en una porción llamada *carga* (*payload*), y en otra porción que enmascara los datos de coordinación de esa *carga* (*overhead*). Los *campos de analizador* pueden especificarse por herencia para conseguir representar la realidad de un campo de un protocolo dado. Así también pueden incluir funciones de copia y asignación de datos que facilite la interacción con otros objetos. Los *campos de analizador* contienen una colección de *subcampos de analizador* como parte de su definición. Es decir que la información almacenada dentro del campo (*carga*) puede expresarse también como una suma de *subcampos de analizador* con significados diferentes.

De allí surge el concepto de *Parser SubField,* cuya definición es muy similar a la de un *campo de analizador*, pero se diferencia en que el contenido que resguarda es una porción de la *carga* del campo al cual pertenece. Los *subcampos de analizador* son útiles solo en aquellos casos donde la información lógica dentro de un campo pueda dividirse es más de un significado, y que por motivos de visualización hacia el exterior, o de administración de la información, sea conveniente tratarla así. Si el contenido de un campo es igual a un campo lógico, los *subcampos* no tienen sentido, y la colección de *Parser SubFields* dentro del campo estará vacía.

Previo a describir los campos definidos para estos conceptos, es clave acotar que se diseñó la arquitectura de los *parsers* para que los mismos puedan tener una cantidad dinámica de campos, definiéndose esa cantidad vía XML. También puede configurarse el significado de cada campo para una solución específica. La ventaja de esta habilidad es poder reconfigurar los campos que serán usados en una implementación específica, dado un protocolo multi-propósito, sin necesidad de recompilar la solución. Los significados que puedan tener en una implementación y en otra dada sirven para generar documentación HTML automatizada del protocolo acordado

con un cliente, de modo de acortar la brecha entre el código y la información técnica disponible sobre las capas de comunicación entre las partes.

En la definición de campos dentro del XML también se puede configurar qué campo (o concatenación de dos o más campos) hacen a la identificación de la transacción. Por ejemplo para ISO8583 (formato binario) se utilizan los campos MSG_TYPE y FIELD_3 para determinar cuál es la transacción entrante, y de ahí identificar la clase que habría que instanciar para atender la transacción.

Conceptos	**Objetivo**
_Fields	Es una colección de objetos con tipo heredado de *Parser Field,* que es visible de forma interna a la clase base de la estructura de analizador (*AbstractTransactionParserStructure*). En esta colección se deben almacenar todos los campos usados para conformar un *Parser Structure*.
this[int id], this[string keyname]	Son dos propiedades de acceso rápido a la colección *_Fields* visibles desde el exterior de la clase base *AbstractTransactionParserStructure*. Se puede acceder tanto a través de un índice numérico configurado para un campo determinado (ese valor se configura en un XML externo), como así también a través de una cadena (*string*) *keyname* que represente el significado configurado para un campo en una solución específica (ese valor también se configura en ese mismo XML).
_StructureType	Es un campo del tipo enumeración, que puede almacenar los valores *Request* o *Response*, indicando si el objeto pertenece a un *Parser Structure* de requerimiento o respuesta respectivamente. Este campo pertenece a la clase base *AbstractTransactionParserStructure*, y tiene valor cuando en un protocolo determinado, las estructuras de datos son distintas en la recepción (requerimiento) con respecto al envío de la respuesta.
GetOperationId()	Devuelve un objeto que contiene la información concatenada de los campos marcados en la configuración XML con el atributo *IsTransactionIdentifier*. Es un método definido virtualmente dentro de la clase base *AbstractTransactionParserStructure*. Cuando en la lista de campos se marca(n) alguno(s) con el atributo mencionado, el valor

	concatenado es usado por el método fábrica de transacciones (*TransactionHandlerFactory()*) del *motor de entrada*, para identificar qué clase debe instanciarse para atender a la transacción. El identificador de la transacción se obtiene haciendo una llamada a este método, generalmente sobre el objeto *Parser Structure* usado para almacenar el requerimiento.
GetField()	Es un método deprecado de la clase base *AbstractTransactionParserStructure* para obtener un campo a través de un índice dentro de la definición de campos en el XML. (Debe usarse this[int id], o this[string keyname])
GetBio(), GetBioFields()	Son dos métodos virtuales definidos en la clase base *AbstractTransactionParserStructure*, que devuelven una cadena con código HTML con la información almacenada dentro del objeto *Parser Structure*. Esa cadena sirve para crear documentación automática y dar visualización del protocolo enviado y recibido entre las partes. La documentación HTML muestra la información de cada campo tanto en formato de *corriente de bytes* como en campos desensamblados, y sus respectivos significados para la implementación. Estos métodos utilizan la información configurada en los atributos XML de la definición del protocolo para una solución dada, de modo de achicar la brecha entre el código y la documentación del proyecto a nivel de capa de comunicación.
_Log	Es un campo que se carga con la referencia del objeto del *logger* configurado para la instancia de ejecución del sistema.
_RootSection	Es el nombre del *tag* XML cuyo nodo contiene toda la definición referente para el protocolo en la implementación dada. De existir más de un protocolo, los **_RootSection** deberán ser diferentes.

Tabla 3 - Elementos comunes definidos en la clase base para el concepto *Parser Structure*

Conceptos	**Objetivo**
Id, Keyname	Son dos propiedades que sirven para identificar a un campo y/o un subcampo dentro de una colección. Por diseño se usaron dos propiedades de tipos diferentes, para poder identificar con un entero vía la propiedad *Id*, o por una cadena vía la propiedad *Keyname*. La propiedad *Id* fue pensada para poder acceder a través de un índice

	numérico, mientras que la propiedad *Keyname* fue pensada para acceder a través de una palabra que simbolice al campo en referencia.
Alias	Es una propiedad adicional para poder referenciar a un campo dentro de una colección de campos. El objetivo es poder acceder al mismo a través de una palabra clave definida en el XML de configuración, particularmente para el ámbito de una solución dada. Esto quiere decir que se puede acceder al campo a través del *Alias*, con una palabra clave con sentido para el alcance de la transacción en curso, detectada a nivel *Parser Structure* con el método *GetOperationId()*.
Content, Description	Son dos propiedades clave para el concepto de *campo y subcampo de analizador*. *Content* es la cadena de bytes que almacena la porción de trama desarmada como un *campo (o subcampo) de analizador* por el método *Disassemble()*. En cambio, *Description* es una propiedad que se obtiene de la configuración XML, y que sirve como descripción del contenido almacenado esperado. Con esa descripción, los métodos *GetBio()* y *GetBioFields()* pueden armar un documento HTML con la explicación de los campos usados por transacción.
Subfields, this[int id], this[string keyname]	Son propiedades de la clase base *AbstractTransactionParserField*, y sirven para almacenar una colección de subcampos, y para accederlos desde afuera del objeto *campo de analizador (Parser Field)*. *Subfields* es la propiedad interna al objeto *campo de analizador* que contiene una eventual colección de subcampos del tipo *AbstractTransactionParserSubField*. Se debe recordar que un *campo de analizador* puede opcionalmente estar formado por un conjunto de subcampos, o por sí mismo como unidad mínima de información durante el proceso de desensamblado de la *corriente entrante de bytes*. Las propiedades de acceso *this[int id]* y *this[string keyname]* permiten acceder a un subcampo dentro de la colección almacenada en *Subfields* desde un objeto externo, ya sea por un índice numérico, o por una cadena de caracteres (*string*) que haga alusión a su significado.

HasValue, IsRequired, IsTransactionIdentifier	Son propiedades del tipo *bool* que simbolizan diferentes características del campo (o el subcampo en cuestión) configurables desde un XML. Con la propiedad *HasValue* se puede evaluar sin acceder a *Content* si el campo (o subcampo) tiene un valor asignado. Con la propiedad *IsRequired* se puede evaluar si el campo (o subcampo) es requisito obligatorio dentro de la estructura de la transacción en curso. Esta propiedad se obtiene del archivo XML de configuración. Por último *IsTransactionIdentifier* es una propiedad configurable también por XML y se utiliza para que un objeto *Parser Structure* reconozca cuál es el tipo de la transacción entrante a través de la *corriente de bytes* recibida de un cliente. El método *GetOperationId()* devuelve una concatenación de todos los campos marcados como *IsTransactionIdentifier=true*.
Length, TrueLength	Con estas dos propiedades se expresa la longitud de un campo o de un subcampo determinado. La diferencia entre *Length* y *TrueLength* reside cuando el campo analizado tiene en su contenido un prefijo/sufijo de longitud, o de cualquier otro tipo, que no hace a la información que se pretende almacenar. Por ejemplo, algunos campos del protocolo ISO8583 son del tipo LLVAR, indicando que cada campo comienza con dos bytes, expresando en ellos la longitud del contenido (*payload*) dentro de ese campo. En ese caso *Length* será la cantidad total de bytes del campo, y *TrueLength* deberá entender que por el tipo de campo la *longitud verdadera* son dos bytes menos que la cantidad almacenada en *Content.*
Offset	Es una propiedad exclusiva de los *subcampos de analizador*, y allí se almacena la posición en donde comienza un subcampo dentro de la cadena de bytes del *campo de analizador*.
CopyContentFrom(), CopyContentTo()	Son dos métodos tanto de los *campos* como de los *subcampos de analizador,* que sirven para copiar contenido desde una cadena de

	bytes fuente al campo *Content*, o copiar desde *Content* el contenido hacia una cadena de bytes destino.

Tabla 4 - Elementos comunes definidos en la clase base del concepto *Parser Field, y Parser SubField*

4.5.3.3 Corriente de Analizador (*Parser Stream*)

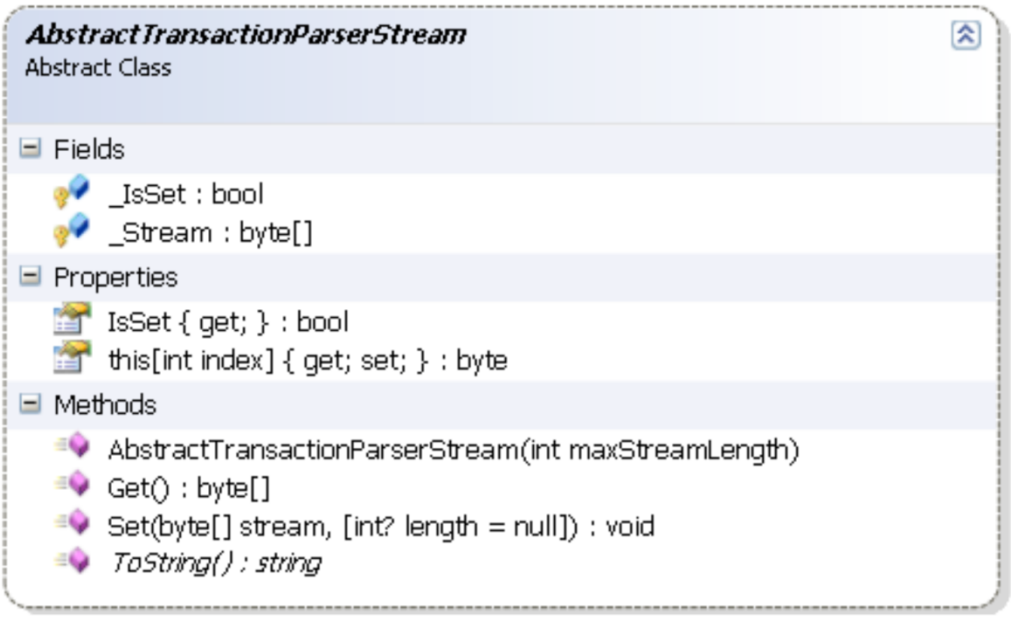

Figura 17 - Clase abstracta de una corriente de analizador base

El concepto de *Parser Stream* surge de la necesidad de almacenar las *corrientes de bytes entrantes y salientes* al momento de desarrollar un nodo transaccional. El método de recepción, apuntado oportunamente por el delegado *ReceiveMethod* del analizador, tiene por responsabilidad almacenar los bytes recibidos desde un cliente externo dentro de un objeto de este tipo. Por su parte, el método de ensamblado (apuntado oportunamente por el delegado *AssembleMethod* del analizador) debe almacenar en una *corriente de bytes* el resultado de serializar los campos de la estructura de respuesta (que eventualmente se enviarán como respuesta al cliente externo). El uso de esta clase tiene la ventaja de poder dar visibilidad al exterior de las cadenas almacenadas (según se especifique en las subclases). Esta clase hace de envoltorio para un *array de bytes* dado, con la ventaja de tener métodos de carga, lectura y de impresión de los valores representados en estos *arrays*.

Conceptos	**Objetivo**
_Stream, this[int index]	*_Stream* es un campo privado de la clase *AbstractTransactionParserStream*, y se diseñó para almacenar la *cadena de bytes* recibida de un cliente y así también la *cadena de bytes* armada como eventual respuesta hacia el nodo que generó la transacción. *this[int index]* en cambio es una propiedad pública con el objetivo de dar visibilidad de cada byte dentro la cadena de bytes

	almacenada en *_Stream*. El valor del tipo entero pasado como parámetro es el índice dentro de la cadena del byte que se pretende obtener.
Get(), Set()	Son dos métodos virtuales que por defecto tienen las responsabilidades de devolver la cadena de bytes, y de copiar una cadena de bytes destino a *_Stream*, respectivamente. Si así se deseara, una subclase puede redefinir estos comportamientos para extender las funcionalidades de obtención y copiado según los requisitos de una solución o analizador específico.
_IsSet	Es un campo privado de la clase base en referencia, que conoce si el objeto fue o no asignado con una cadena de valores de bytes. Se diseñó para evitar tener que evaluar la cadena con valores en *null*, o sin haberse inicializado la misma.
ToString()	Es un método abstracto, que si bien tiene un comportamiento por defecto en la clase base *Object*, se sobrecargó para hacerlo de implementación obligatoria en la jerarquía de clases. De este modo se propone quitarle ese comportamiento por defecto para que el usuario de las *corrientes de analizador* redefina una implementación por cada tipo heredado de *Parser Stream*.

Tabla 5 - Elementos comunes definidos en la clase base del concepto *Parser Stream*

4.5.3.4 Habilidades de los Analizadores

Los analizadores de protocolo (*Parser*) pueden heredar una serie de habilidades, agrupadas según las características de los métodos que las forman. Específicamente, esta herencia se da a través del uso de *interfaces* del *framework* .NET. Un *parser* determinado puede opcionalmente heredar cualquiera de las interfaces que definen las habilidades: en la primera versión de la propuesta existen dos interfaces de habilidades. Por lo que un *Parser* dado puede implementar cualquiera de las dos de forma independiente. En la práctica, la *habilidad ensamblable* es la característica principal de un *parser*, por lo que no se han presentado casos en que no haya herencia de esa habilidad, pero, puede darse el caso que *no sepa como enviarse ni recibirse* un *parser* a través de un *socket*. En ese caso, el Parser heredaría solo una interface.

Para las implementaciones de los métodos definidos en las interfaces se utilizan delegados. Estos delegados relacionan las llamadas a métodos con sus implementaciones, de modo de poder apuntar a código fuente cuando éstos estén funcionales. A continuación se describen las dos interfaces que representan a las dos series de habilidades que un *parser* puede adquirir: las habilidades *ensamblables (Assembleable)* y las habilidades *comunicables (Communicable)*.

4.5.3.4.1 Ensamblables (*Assembleable*)

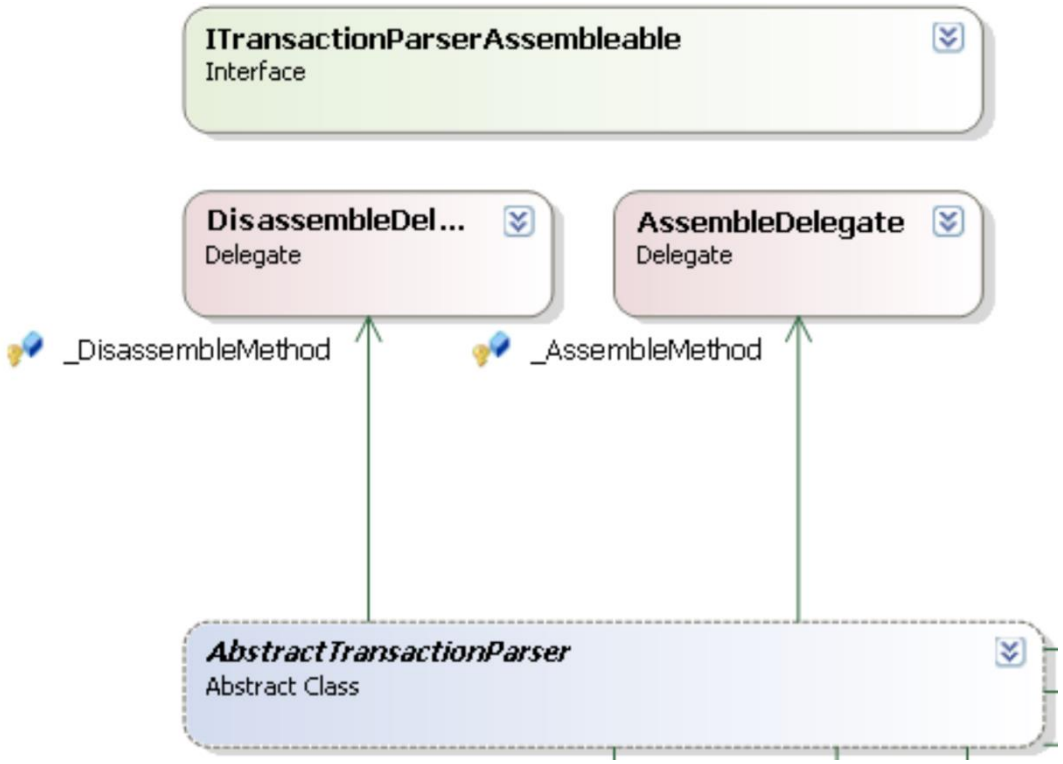

Figura 18 - Diagrama de Clases de la habilidad Ensamblable

Las *habilidades ensamblables* definen los métodos mínimos para que un *parser* determinado pueda desensamblar *array de bytes* a estructuras (*Parser Structure*) y ensamblar estructuras a *array de bytes*. Los dos métodos definidos en la interfaz *IAssembleable* son:

- **Assemble()**: Debe implementarse para que el *parser* pueda serializar datos desde una estructura de analizador (*parser structure*), a una corriente de analizador (*parser stream*). El delegado asociado a este método es *AssembleDelegate.*
- **Disassemble()**: Debe implementarse para que el *parser* pueda deserializar datos desde una corriente de analizador (*parser stream*) a una estructura de analizador (*parser structure*). El delegado asociado a este método es *DisassembleDelegate.*

4.5.3.4.2 Comunicables (*Communicable*)

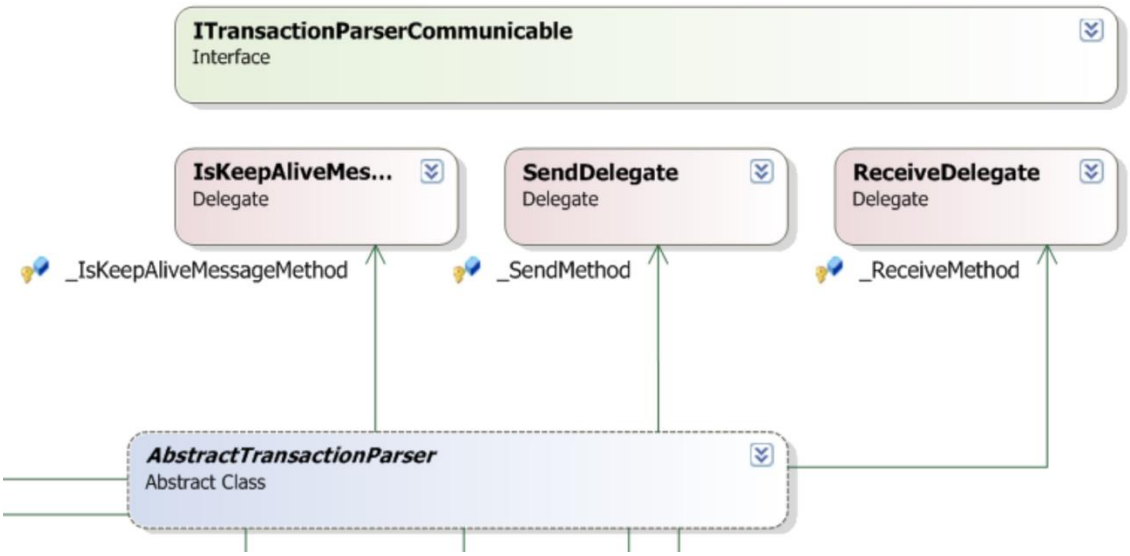

Figura 19 - Diagrama de Clases de la habilidad Comunicable

Las *habilidades comunicables* definen los métodos mínimos para implementar el envío y recepción de las *corrientes de analizador* (*Parser Stream*) de salida y entrada respectivamente. Los tres métodos definidos en la interfaz *ICommunicable* son:

- **Send()**: Debe implementarse para que el *parser* pueda enviar los datos en el *parser stream* de salida a través de un *handler* o *file descriptor*, por defecto un *socket TCP*. El delegado asociado a este método es *SendDelegate.*

- **Receive()**: Debe implementarse para que el *parser* pueda recibir los datos en el *parser stream* de entrada a través de un *handler* o *file descriptor*, por defecto un *socket TCP*. El delegado asociado a este método es *ReceiveDelegate.*
- **IsKeepAliveMessage()**: Debe implementarse para que el *parser* retorne el resultado del análisis realizado sobre la trama entrante, y saber si la trama entrante es válida o de *KeepAlive* (mensaje de control de la red). El delegado asociado es *IsKeepAliveMessageDelegate*.

4.5.4 Manejadores de Transacciones (*Handlers*)

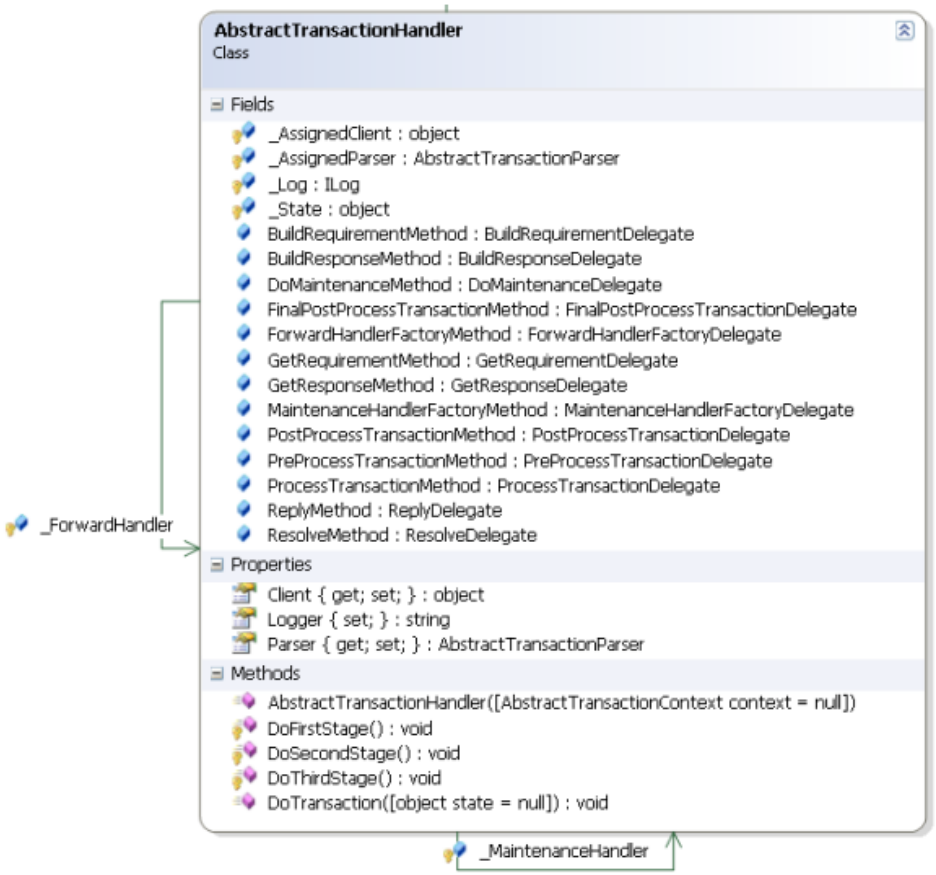

Figura 20 - Clase abstracta de una transacción base

El concepto *Transaction Handler* surge como la idea principal de esta propuesta, *desarrollar un framework transaccional orientado a la transacción*. Para ello, cada vez que se detecta un requerimiento entrante, se tratará de instanciar una o más clases del tipo *AbstractTransactionHandler* que representen a una única transacción en varios estados del proceso.

AbstractTransactionHandler es la clase base para implementar el concepto de transacción, con su secuencia de trabajo ya descripta, y para conectar a una transacción con *motores de entrada / salida* (se explicará más adelante), o inclusive otros *handlers* de transacciones.

Los *motores de entrada* instanciarán un objeto de tipo subclase de *AbstractTransactionHandler*, en función de la información obtenida por el *Parser* asociado a ese motor. De hecho, la operación *GetOperationId()* del *Parser Structure* donde se almacenó el requerimiento entrante, devolverá el identificador de transacción, permitiendo seleccionar dinámicamente la subclase de *Transaction Handler* a usar.

Las transacciones podrán comunicarse con otras transacciones, de modo de desacoplar responsabilidades distintas a medida del proceso. Por ejemplo, para una transacción que debe reenviar una trama de datos a un nodo externo probablemente use dos *transaction handlers*: uno para la etapa de recepción y envío de respuesta contra el cliente que inicia la petición, y otra para el intercambio de datos contra ese nodo externo. Las transacciones que deben encarar la tarea de reenviar un pedido hacia un nodo externo tendrán como finalidad conectarse con un *motor de salida*, que consecuentemente tratarán de enviar y recibir una eventual respuesta, para luego desconectarse.

La espina vertebral de la clase *AbstractTransactionHandler* es el método *DoTransaction()* analizado con anterioridad. En este método se articula la *secuencia genérica de trabajo*, que orquesta cada paso dentro del procesamiento transaccional. Algunos pasos podrán implementarse y otros no. También algunos pasos podrán delegarse para que otra instancia de *AbstractTransactionHandler* los resuelva, armando una cadena de objetos de este tipo. Esa cadena funciona como un *todo* que permite la representar la transacción real en etapas.

A continuación se describen los campos y propiedades más importantes de este concepto.

Conceptos	**Objetivo**
DoTransaction(…)	Es el método principal de la clase base *AbstractTransactionHandler*, ya que es el que articula las llamadas a los delegados según la secuencia de proceso ya descrita de esta propuesta. Puede tener un parámetro opcional llamado *State,* de modo que la transacción eventualmente conozca un estado anterior que sirva para tomar decisiones de forma interna.
DoFirstStage(…), DoSecondStage(…), DoThirdStage(…)	Son tres métodos internos dentro de la clase base *AbstractTransactionHandler*, que son llamados en ese orden para resolver el proceso de una transacción. La *secuencia genérica de trabajo* se puede agrupar en tres

	etapas: cada uno de estos métodos representa a cada una de las etapas.
_MaintenanceHandler	Es un campo interno a la clase base que es particularmente del mismo tipo (*AbstractTransactionHandler*). De esta forma, se pretende que el método fábrica para *handlers* de mantenimiento (*MaintenanceHandlerFactory()*) pueda generar un objeto heredado de la clase *AbstractTransactionHandler*. Este objeto puede atender cualquier tarea de mantenimiento de la transacción, una vez que se haya procesado el *handler* inicial.
_ForwardHandler	Es un campo interno a la clase base que es particularmente del mismo tipo (*AbstractTransactionHandler*). De esta forma, se pretende que el método fábrica para *handlers* de reenvío (*ForwardHandlerFactory()*) pueda generar un objeto heredado de la clase *AbstractTransactionHandler* que pueda atender cualquier tarea de reenvío de peticiones a un nodo remoto, o a otra capa del modelo de la solución.
_AssignedClient	Es un campo interno a la clase base del tipo *object*, que almacena el *handler* abierto por el cliente que generó la petición. Por ejemplo, *_AssignedClient* puede ser el *socket* de conexión con el cliente en curso (objeto del tipo *TcpClient*)
_AssignedParser	Similar a *_AssignedClient*, pero almacena el *parser* asignado por el *motor de entrada*, si la transacción está conectada a un motor de este tipo. Si la transacción está conectada a un *motor de salida*, el objeto *handler* de transacción debería configurar el *parser* que usará el motor al que apunta. Los objetos *Transaction Handler* que estén involucrados en etapas intermedias del proceso (que no estén conectados a ningún motor) deberían tener este campo en *null*.
GetRequirementMethod, BuildResponseMethod, ReplyMethod	Son tres campos de tipos delegados (*GetRequirementDelegate, BuildResponseDelegate* y *ReplyDelegate*), usados para expresar un conjunto de habilidades heredables vía una interface

	(*ITransactionHandlerListenable*), por un objeto del tipo *AbstractTransactionHandler*. Las habilidades asociadas con estos delegados se explicarán en la sección *Habilidades de las Transacciones*.
PreProcessTransactionMethod, PostProcessTransactionMethod FinalPostProcessTransactionMethod	Son tres campos de tipos delegados (*PreProcessTransactionDelegate*, *PostProcessTransactionDelegate* y *FinalPostProcessTransactionDelegate*), usados para expresar un conjunto de habilidades heredables vía una interface (*ITransactionHandlerPersistable*), por un objeto del tipo *AbstractTransactionHandler*. Las habilidades asociadas con estos delegados se explicarán en la sección *Habilidades de las Transacciones*.
DoMaintenanceMethod, MaintenanceHandlerFactoryMethod	Son dos campos de tipos delegados (*DoMaintenanceDelegate* y *MaintenanceHandleFactoryDelegate*), usados para expresar un conjunto de habilidades heredables vía una interface (*ITransactionHandlerMaintenanceable*), por un objeto del tipo *AbstractTransactionHandler*. Las habilidades asociadas con estos delegados se explicarán en la sección *Habilidades de las Transacciones*.
BuildRequirementMethod, ForwardHandlerFactoryMethod, ProcessTransactionMethod, ResolveMethod, GetResponseMethod	Son cinco campos de tipos delegados (*BuildRequirementDelegate*, *ForwardHandlerFactoryDelegate, ProcessTransactionDelegate, ResolveDelegate* y *GetResponseDelegate*), usados para expresar un conjunto de habilidades heredables vía una interface (*ITransactionHandlerForwardable*), por un objeto del tipo *AbstractTransactionHandler*. Las habilidades asociadas con estos delegados se explicarán en la sección *Habilidades de las Transacciones*.

Tabla 6 - Elementos comunes definidos en la clase base del concepto *Transaction Handler*

4.5.4.1 Habilidades de las Transacciones

Al igual que los *parsers*, los manejadores (*handlers*) de transacciones también pueden heredar un conjunto de habilidades, a través de la herencia de interfaces. Estas habilidades se agruparon arbitrariamente, tratando de concentrar una serie de métodos que tuvieran características en común en su objetivo colectivo, y que fueran parte de los pasos comunes para la resolución de una transacción.

Los métodos definidos en estos contratos (interfaces) forman en su totalidad todos los pasos de la *secuencia de genérica de trabajo* propuesta en este *framework* para la resolución de cualquier tipo de transacción. Como no todas las transacciones realizarán todos los pasos propuestos aquí, solo aquellos que si se realicen deben estar implementados en los métodos correspondientes, y deben ser apuntados con los delegados de la clase *AbstractTransactionHandler*. Y por el contrario, los delegados restantes deben estar apuntando a *null*. Como se explicó en secciones anteriores, la secuencia de trabajo evalúa por cada paso si cada delegado está apuntando a un método real, y si lo está, llama a ese método. En el caso que no estén apuntando a nada (*null*), la evaluación de esos pasos se dará como verdadero (true) y la ejecución de la secuencia continuará.

4.5.4.1.1 Escuchables (*Listenable*)

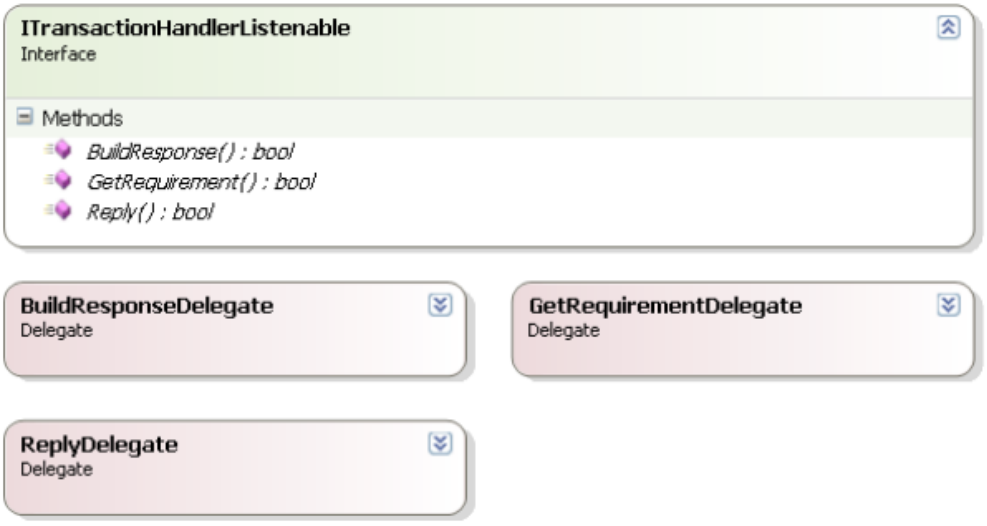

Figura 21 - Diagrama de clases de las entidades que hacen a la habilidad "escuchable"

Las *habilidades escuchables* definen los métodos mínimos para implementar el proceso del requerimiento del cliente, el armado de la respuesta hacia el cliente, y el envío condicional de la respuesta. Los tres métodos definidos en la interfaz *ITransactionHandlerListenable* son:

- **GetRequirement()**: Este método debe implementarse para obtener los datos útiles provenientes del cliente y del sistema transaccional, en pos de la resolución de la transacción. Todos los datos de

entrada provenientes del cliente quedan almacenados en el *Parser Structure* asociado al *Parser* del manejador de la transacción (requerimiento). Este método deberá además validar los datos que se extraen de esa estructura para poder decidir si se cuenta con toda la información necesaria para realizar el procesamiento del requerimiento. Debe retornar *true* si la extracción y las validaciones resultan *ok,* o *false* en caso contrario.

- **BuildResponse():** Debe implementarse para poder armar una estructura de analizador (*Parser Structure*) con todos los datos que formarán la respuesta a enviar al cliente. Este método debe además implementar, o hacer la llamada al método de ensamblado (*Assemble()*) del *parser* asociado al *motor de entrada*. De este modo también estaría encargado de hacer la serialización de la estructura de forma directa o indirecta, que luego se enviará al cliente en forma de *corriente de bytes* (*Parser Stream*). Deberá retornar *true* si *BuildResponse* puede armar la estructura de respuesta sin problemas, y además, si el resultado de *Assemble()* es *true.* En caso contrario deberá retornar *false*.
- **Reply():** Debe contener la lógica para enviar la respuesta a través del *handler* de conexión con el cliente (*socket, file descriptor, TcpClient, etc.*). Además puede contener lógica adicional para decidir por los resultados de la transacción, si el cliente debe recibir o no una respuesta. Por ejemplo, si el servicio pierde contacto con un nodo externo por cualquier motivo, puede ser deseable forzar una consulta de transacción (no respondiendo) por parte del cliente conectado al servicio. Para ello, *Reply()* puede evaluar el código de error del proceso realizado y decidir si envía o no la respuesta. Debe retornar *true* si pudo enviar la respuesta, o *false* en caso contrario.

4.5.4.1.2 Reenviables (*Forwardable*)

Las habilidades *reenviables* definen los métodos mínimos para implementar el reenvío de una transacción a un nodo externo para delegar su procesamiento. Los cinco métodos definidos en la interfaz *ITransactionHandlerForwardable* son:

- **ForwardHandlerFactory():** Es un método fábrica que debe decidir si construye una instancia de *AbstractTransactionHandler* para delegar el reenvío del requerimiento. Si no se construye ninguna instancia, el objeto del tipo *AbstractTransactionHandler* actual será el encargado de procesar la transacción de forma interna (tiene todos los datos para aprobarla o rechazarla) o externa (reenviándola a un nodo externo). Si en cambio se construye una instancia en el método fábrica, el objeto del tipo *AbstractTransactionHandler* actual delegará el proceso a ese nuevo objeto creado. Retorna *true* si pudo crear la instancia o si todo resultó correctamente, *false* en caso contrario.
- **BuildRequirement()**: Debe implementarse para armar una estructura o para llenar los parámetros que deben retransmitirse a un nodo externo. Si se da el caso que es necesario el armado de una

estructura de analizador (*Parser Structure*), también tiene la responsabilidad de llamar al método *Assemble()* del *parser*. Retorna *true* si pudo armar el requerimiento a reenviar (y en todo caso si *Assemble()* también retornó *true*). Por lo contrario debe retornar *false*.

- **Resolve()**: Es el método principal de este conjunto de habilidades, ya que es el responsable de conectarse, enviar, recibir y desconectarse contra un nodo externo. La lógica debe implementarse directamente en este método (en el caso de no poder utilizar ningún *motor de salida*), o hacer la llamada al método *Resolve()* de los *motores de salida*, que se explicarán más adelante. Retorna *true* si todo el proceso se realizó sin problemas, *false* en caso contrario.
- **GetResponse():** Debe implementarse para validar los datos recibidos en *Resolve()*. Aquí debe implementarse la obtención de los datos útiles para los fines de procesar la transacción (que llegaron en la respuesta del nodo externo). También debe decidirse el resultado lógico de la transacción, tomando en cuenta los resultados y valores recibidos. Retorna *true* si se pudo realizar la validación sin inconvenientes, y *false* en caso contrario.
- **ProcessTransaction():** Es un método que debe contener una serie de validaciones sobre el estado de la transmisión y/o de los datos, en caso que *Resolve()* retorne *false*. De este modo debe asignar el resultado lógico de la transacción, cuando hubo una eventualidad, y no se obtuvieron datos para que *GetResponse()* procese. Retorna *true* si pudo realizar las validaciones correctamente y si pudo asignar un resultado lógico a la transacción, *false* en caso contrario.

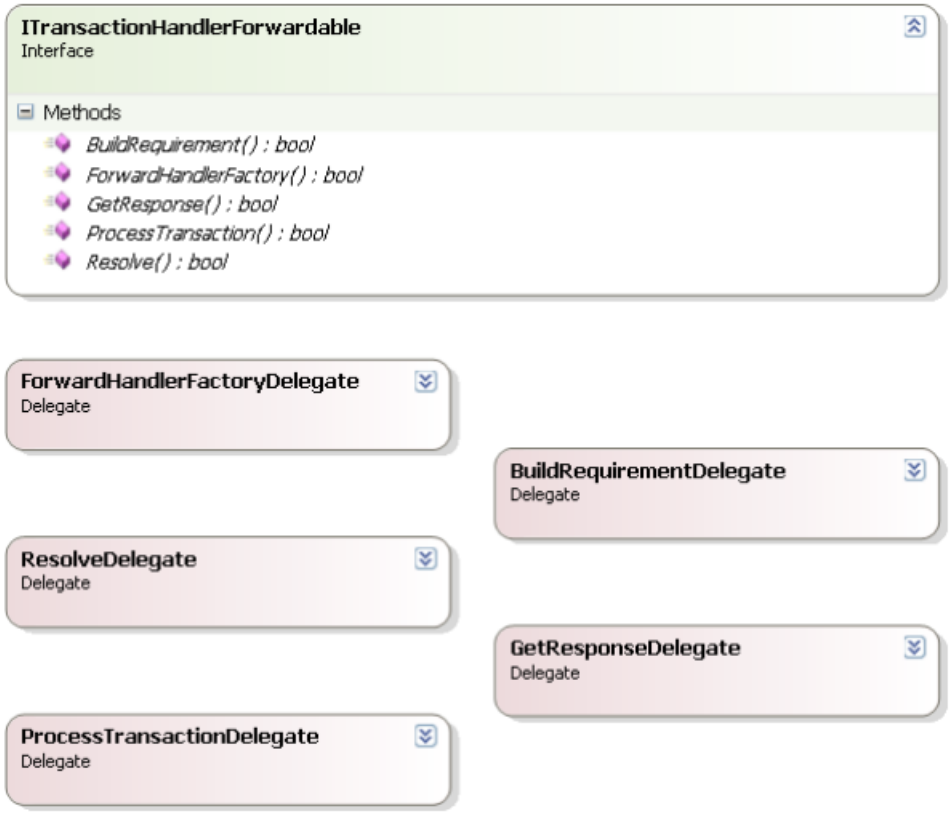

Figura 22 - Diagrama de clases las entidades que hacen a la habilidad "reenviable"

4.5.4.1.3 Persistentes (*Persistable*)

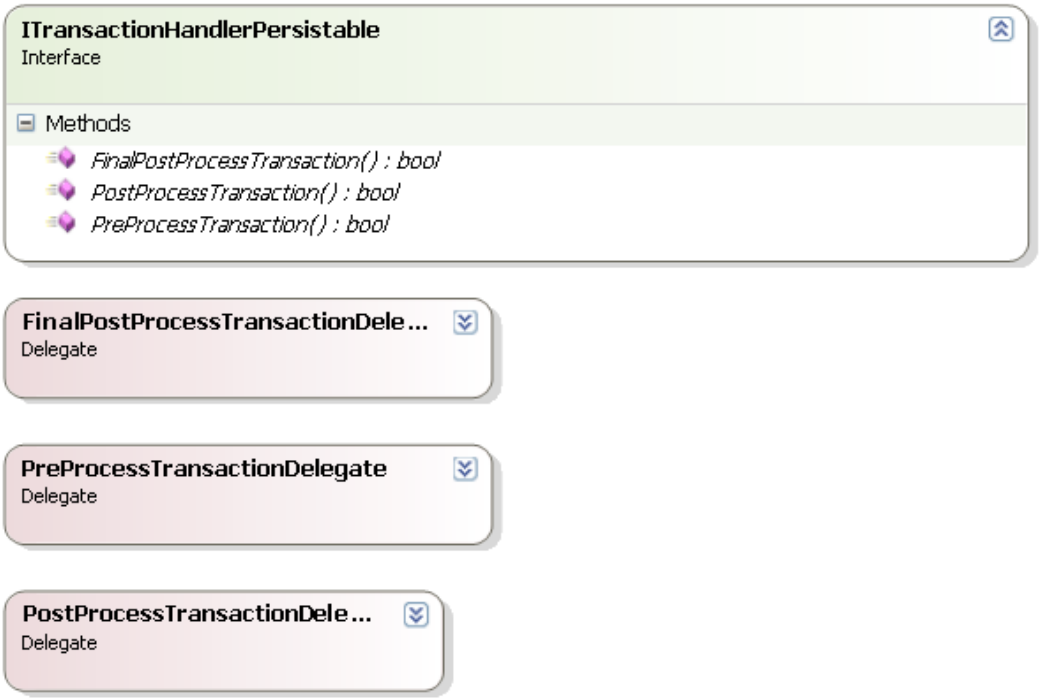

Figura 23 - Diagrama de clases de las entidades que hacen a la habilidad "persistente"

Las habilidades *persistibles* definen los métodos mínimos para implementar la persistencia de la transacción en diferentes momentos de la secuencia de trabajo propuesta. Los tres métodos definidos en la interfaz *ITransactionHandlerPersistable* son:

- **PreProcessTransaction():** En este método debería desarrollarse la lógica para almacenar la transacción entrante en un histórico de transacciones, y ejecutar todas las pre-validaciones que se deban hacer, contrastando los datos de la transacción contra los valores que estén en la fuente de almacenamiento. De esta serie de pre-validaciones, se deberá decidir si la transacción se aprueba, si se rechaza, si el monto es suficiente para procesarla, por dónde deberá reenviarse (si es necesario reenviarla), etc. El resultado debería guardarse (si es que hubiera una definición para la transacción en este momento), en el movimiento creado en el histórico. Si no lo hubiera, solo deberían guardarse los datos entrantes y esperar a la resolución en una etapa más tardía, donde oportunamente deberá guardarse el resultado final (por ejemplo en *PostProcessTransaction()*). El método debe retornar *true* siempre que pueda realizar el pre-proceso sin problemas, más allá del resultado del conjunto de pre-validaciones. *False* en caso contrario.
- **PostProcessTransaction():** En este otro método debe desarrollarse la lógica para guardar el resultado final de la transacción. Este método es llamado en la etapa tres, una vez que se ha contestado al cliente una respuesta con el resultado del proceso. Además se deberían implementar

todas las tareas de actualización de datos y mantenimiento de las cuentas, balances, etc. que hubieran sido afectadas por la transacción. El método debe retornar *true* siempre que pueda realizar el post-proceso sin problemas, más allá del resultado del conjunto de post-validaciones. *False* en caso contrario.

- **FinalPostProcessTransaction():** Por último, este método debe implementarse para realizar la actualización de datos y mantenimiento de las cuentas, balances, etc., si hubiera sido necesario crear un *handler* de transacción de mantenimiento (ver sección siguiente). Si por algún motivo este *handler* modifica datos, o agrega más información a la recolectada durante el proceso de la transacción original, tal vez sea necesario salvaguardar esos cambios, y es aquí donde debería realizarse. La llamada a este método es el último paso de la secuencia de trabajo propuesta por el *framework*, y es la última oportunidad en la *vida de la transacción* para guardar datos en una fuente almacenable. Debe retornar *true* siempre que pueda realizar el post-proceso final sin problemas. *False* en caso contrario.

4.5.4.1.4 De manutención (*Maintenanceable*)

Las habilidades *de manutención* definen los métodos mínimos para implementar una instancia del tipo *AbstractTransactionHandler* que pueda atender el mantenimiento de la transacción actual, una vez que ésta última terminó de realizar su proceso, y que el cliente pudo haber recibido una respuesta.

Por ejemplo, este caso se da cuando se debe realizar una consulta de saldo hacia un nodo externo, después de realizada una venta contra éste. Como el saldo del proveedor le es indiferente al cliente final, el *handler* de mantenimiento es ejecutado posterior al envío de la respuesta hacia cliente respecto a su requerimiento original (una venta en este caso). De este modo, se reduce la cantidad de tiempo que el servidor mantiene conectado a un cliente que está en espera por la respuesta.

Los dos métodos definidos en la interfaz *ITransactionHandlerPersistable* son:

- **MaintenanceHandlerFactory():** Es un método fábrica que debería tener la lógica para saber qué clase de mantenimiento instanciar en función de los datos contextuales de la transacción original, y bajo qué condiciones debe instanciarse alguna clase para tal fin. Debe retornar *true* si pudo generar una instancia válida sobre el campo *_MaintenanceHandler* de un tipo que herede de *AbstractTransactionHandler*, *false* en caso contrario.
- **DoMaintenance():** Este método debería tener implementada la lógica para hacer una llamada a *DoTransaction()* sobre el objeto instanciado en *_MaintenanceHandler*. A su vez este objeto podría tener otro *handler* de mantenimiento, generando una cadena de mantenimiento con los *handlers*

como eslabones. Debe retornar *true* si pudo ejecutar *DoTransaction()* sin problemas, *false* en caso contrario

Figura 24 - Diagrama de clases de las entidades que hacen a la habilidad "de manutención"

4.5.5 Motores Transaccionales (*Engines*)

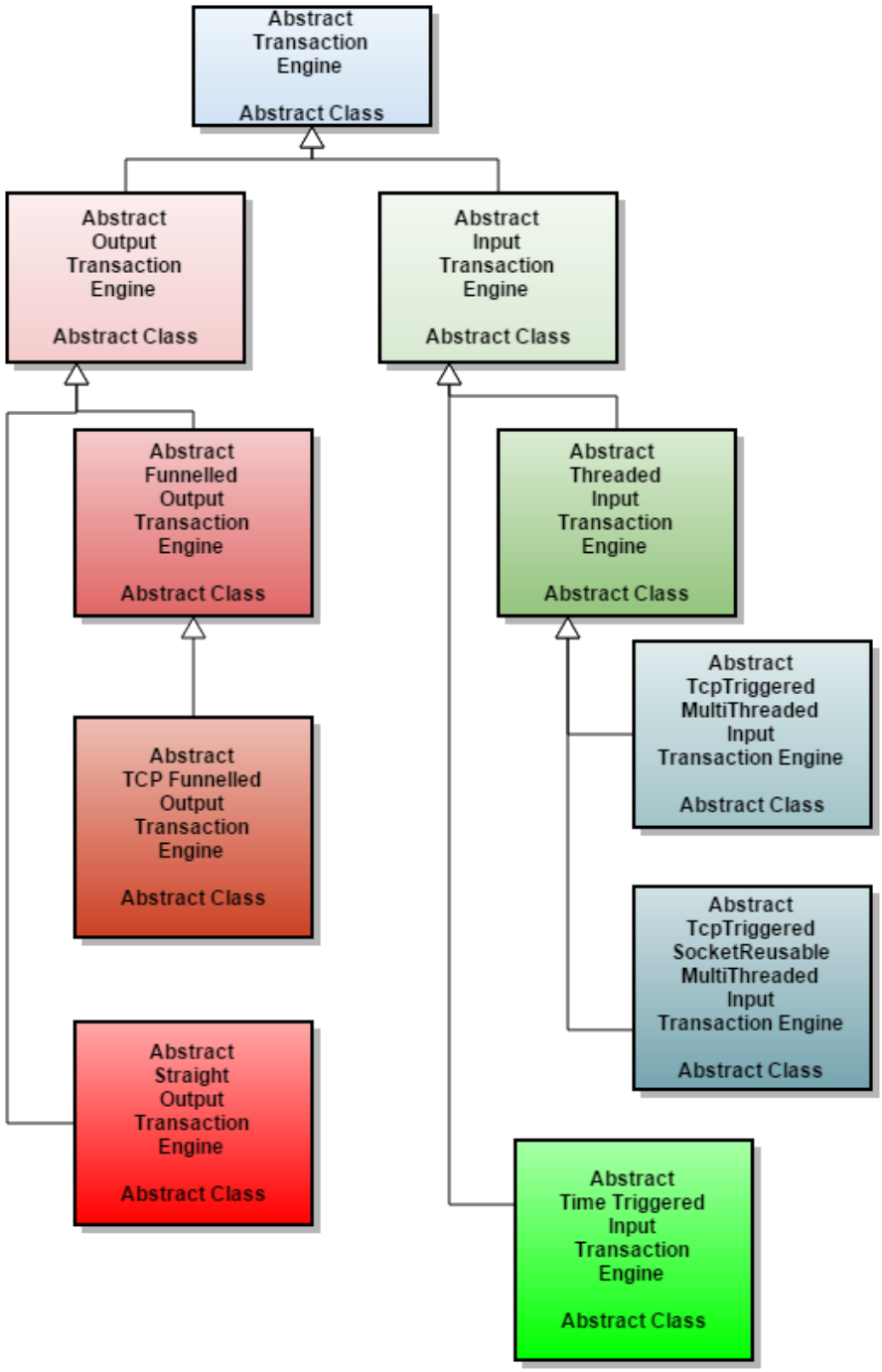

Figura 25 - Diagrama de Clases y jerarquía de herencia de una serie de motores transaccionales típicos

El concepto de *Transaction Engine* es clave para esta propuesta. Los *motores* tendrán la responsabilidad principal de coordinar todos los conceptos vistos anteriormente para poder realizar el proceso transaccional en tiempo y forma. Además es el concepto de la propuesta que primero se ejecuta apenas arrancado el sistema de procesamiento, y que se mantiene en ejecución hasta el momento en que éste se apaga. Es por esta razón que los motores, específicamente los que llamaremos *motores de entrada*, son los corazones de cualquier sistema de procesamiento desarrollado por esta propuesta de *framework*.

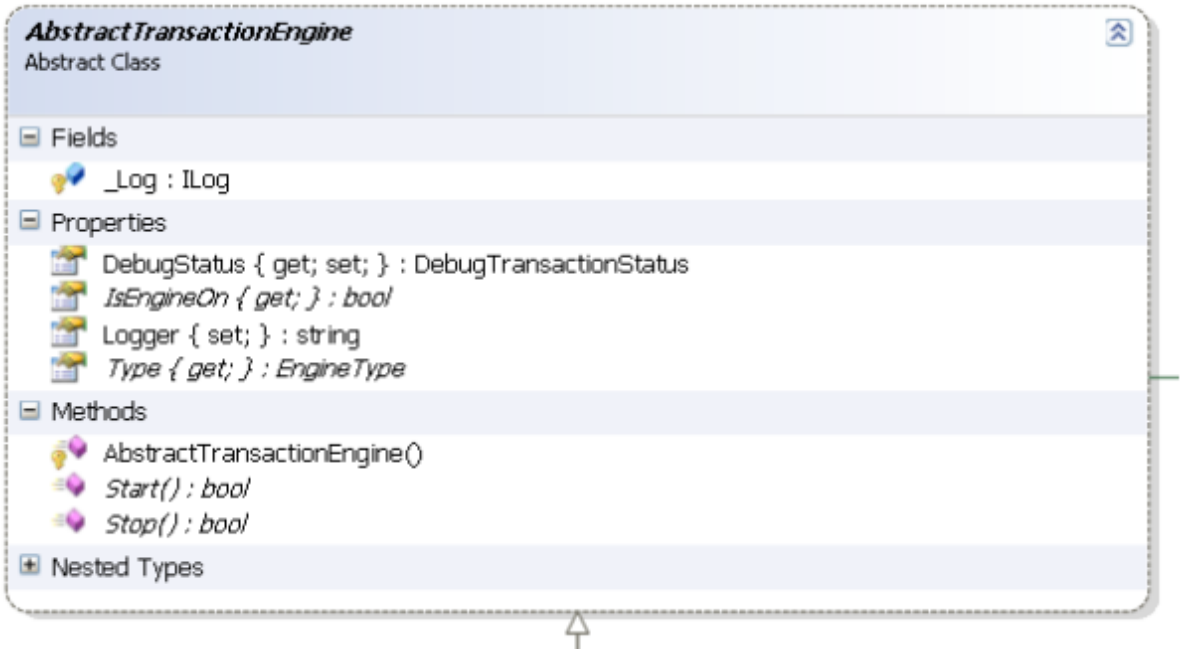

Figura 26 - Clase abstracta de un motor base

Para los fines de este trabajo se ha decidido clasificar a los motores en dos grandes familias, los *motores de entrada* y *los motores de salida*. Los *motores de entrada* serán los encargados de obtener / generar los *inputs* de datos que deberán procesarse en el sistema y de coordinar las eventuales respuestas, mientras que los *motores de salida* conocerán las distintas formas de conectarse a un nodo externo / remoto para poder hacer intercambio de información (reenvío de las peticiones) si fuera necesario. De todos modos estas familias de motores heredan de una clase abstracta llamada *AbstractTransactionEngine*, que almacena los factores comunes de ambas. Así también cada familia tiene su propia jerarquía de clases abstractas, cada una especificando a través del árbol de herencia características de las distintas formas de intercambiar información para los fines de la propuesta, desde/hacia un cliente, o desde/hacia un nodo externo. A continuación se detallan las propiedades, campos y métodos de las clases más importantes dentro de la jerarquía propuesta.

Conceptos	**Objetivo**

Start(), Stop()	Son dos métodos abstractos, cuya finalidad es la de arrancar un motor, y pararlo respectivamente. Estos métodos fueron pensados para que sean llamados desde los *handlers* de los eventos *OnStart()* y *OnStop()* dentro de un controlador de *servicio de Windows*. De esta forma, al arrancar el servicio se deberían arrancar en un orden dado por el desarrollador todos los *motores de entrada*, y luego los *motores de salida*. La secuencia contraria debería darse en el caso de finalizar la ejecución del servicio. Las clases abstractas dentro del árbol de herencia de los motores irán implementando comportamientos por defecto según el grado de especificidad de cada una.
Type	Es una propiedad abstracta de solo lectura, que debe implementarse en las subclases para responder el tipo del motor instanciado. Hasta el momento de este documento, el tipo de motor tiene solo dos valores definidos: *InputEngine* o *OutputEngine.* Sin embargo está pensado para poder retornar múltiples tipos de motores distintos en el futuro, si fuera necesario.
IsEngineOn	Es una propiedad abstracta de solo lectura, que debe sobrecargarse en las subclases para indicar si un motor está prendido o no, en función de la lógica específica dentro de cada subclase.
DebugStatus	Es un campo de enumeración del tipo *Flags,* con fines de evaluación del comportamiento del motor para el desarrollador. Este campo solo tiene sentido en compilaciones en versión *Debug,* y almacena en forma de *bitmap* los distintos puntos atravesados por una transacción dentro de la secuencia de trabajo de la propuesta. De este modo, los métodos de *Unit Test* pueden evaluar si una transacción pasó por todos los puntos de la secuencia esperados, y determinar si una prueba de unidad resulta satisfactoria o fallida.
_Log	Es un campo que se carga con la referencia del objeto del *logger* configurado para la instancia de ejecución del sistema. Con este campo los motores pueden generar su propia información de log.

Tabla 7 - Elementos comunes definidos en la clase base del concepto *Transaction Engine*

4.5.5.1 Motores de Entrada (*Input Engines*)

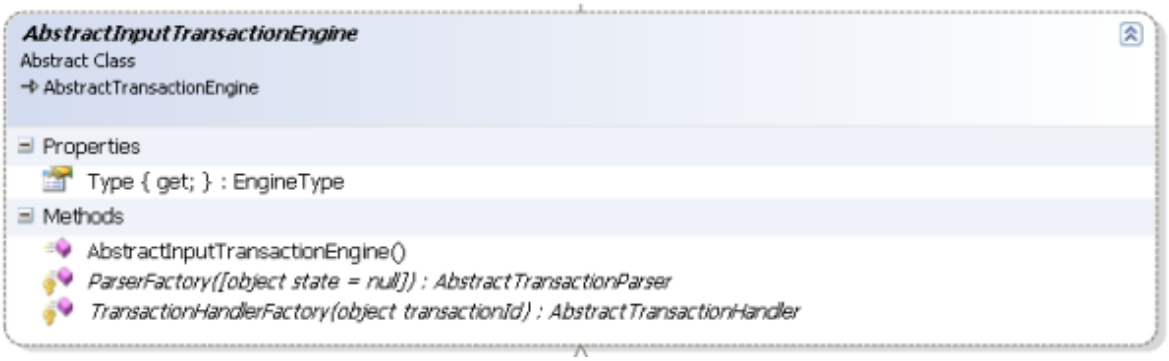

Figura 27 - Clase abstracta de un motor de entrada base

Los *motores de entrada* forman una de las dos familias de motores usados en este *framework*. De las dos familias, la formada por los *motores de entrada* es la más importante debido a que sin estos motores no habría ningún estímulo de entrada para el sistema transaccional. La familia de *motores de entrada* tiene un árbol de subclases en donde cada una de ellas implementará un tipo distinto de motor, categorizados según la forma de conexión, la cantidad de subprocesos disponibles para la atención de las transacciones, etc. Es por eso que cada subclase dentro de este árbol fue desarrollada tratando de capturar las cualidades y características de los motores transaccionales usados en soluciones preexistentes, que fueron observados y analizados para construir este *framework*.

Los *motores de entrada* tienen la responsabilidad de coordinar los conceptos de *Parser* y de *Transaction Handler:*

- En primer lugar, deberán configurar un *analizador* (*parser*) para poder entender un protocolo de comunicación determinado. Además, si el *parser* conoce como intercambiar información con un cliente, el motor podrá hacer uso de esa lógica para intercambiar información con ese cliente, con el *parser* de intermediario.
- En segundo lugar, por cada requerimiento entrante se deberá instanciar una clase de atención de la transacción, según el tipo del requerimiento (tipo de transacción). Para eso, el motor se basa en la información entregada por el *parser*, una vez que desensambló la *corriente de bytes* obtenida en la etapa de recepción / lectura.

A continuación se detallan los campos, propiedades y métodos más importantes de la clase *AbstractInputTransactionEngine.*

Conceptos	**Objetivo**
ParserFactory()	Es un método abstracto del tipo *fábrica*, cuya responsabilidad es la de instanciar el *parser* configurado para ese motor. Esta tarea debería

	realizarse en la implementación de una subclase de *AbstractInputTransactionEngine,* que conozca efectivamente el protocolo a usarse entre las partes. Las secuencias de trabajo de los motores no requieren saber cuál será el *parser* a instanciar, sino que llamarán a este método para que de forma indistinta puedan tener una instancia válida de *AbstractTransactionParser* con la cual puedan ensamblar y desensamblar información intercambiada entre el servidor y un cliente. Así también, el motor puede usar eventualmente el *parser* para enviar y recibir la información según la lógica del protocolo entre las partes. Si bien la mayoría de las observaciones hechas en soluciones transaccionales coincidieron en que el tipo de *analizador* a instanciar se mantiene constante en una interfaz cliente-servidor dada una solución, *ParserFactory()* cuenta con una parámetro opcional del tipo *object*, donde se puede inyectar dependencias para que este método pueda decidir de forma dinámica qué tipo de *parser* debería instanciar. Este método debe devolver una instancia válida de alguna subclase de *AbstractTransactionParser.*
TransactionHandlerFactory()	Es un método abstracto del tipo *fábrica*, cuya responsabilidad es la de instanciar un *handler de transacción* que pueda atender el requerimiento. La clase de la instancia debe heredar de *AbstractTransactionHandler*. Las secuencias de trabajo de los motores no tienen por qué conocer de antemano las tareas de procesamiento de cada tipo de transacción, sino que deben conocer el punto de entrada de todos los tipos de transacciones (*DoTransaction()*), de modo de desacoplar al motor del tipo de solicitud entrante. Es así que dentro de la secuencia de trabajo del motor, se realizará una llamada a este método para crear una instancia adecuada para el procesamiento. *TransactionHandlerFactory()* debe conocer al menos un valor de la *corriente de bytes* entrante que le permita conocer el tipo de transacción a instanciar. Para ello, es necesario pasar por parámetro la

	concatenación de los campos de protocolo que sean usados como identificadores (marcados en el XML del protocolo como *IsTransactionIdentifier = true*). Estos valores son se obtienen como resultado del método *GetOperationId(),* perteneciente a *la estructura de analizador* donde está el requerimiento almacenado (la estructura debe ser de algún subtipo de *AbstractTransactionParserStructure* para heredar *GetOperationId()*). *TransactionHandlerFactory()* debe devolver una instancia válida de alguna subclase de *AbstractTransactionHandler.*

Tabla 8 - Elementos comunes definidos en la clase base del concepto *Input Transaction Engine*

Como ejemplo de algunos *motores de entrada* utilizados, a continuación se describirán los más importantes, con sus características principales.

4.5.5.1.1 Ejemplo A: Motor de Entrada Multi-Hilos Disparado por Conexión TCP (*Tcp Triggered Multi Threaded Input Engine*)

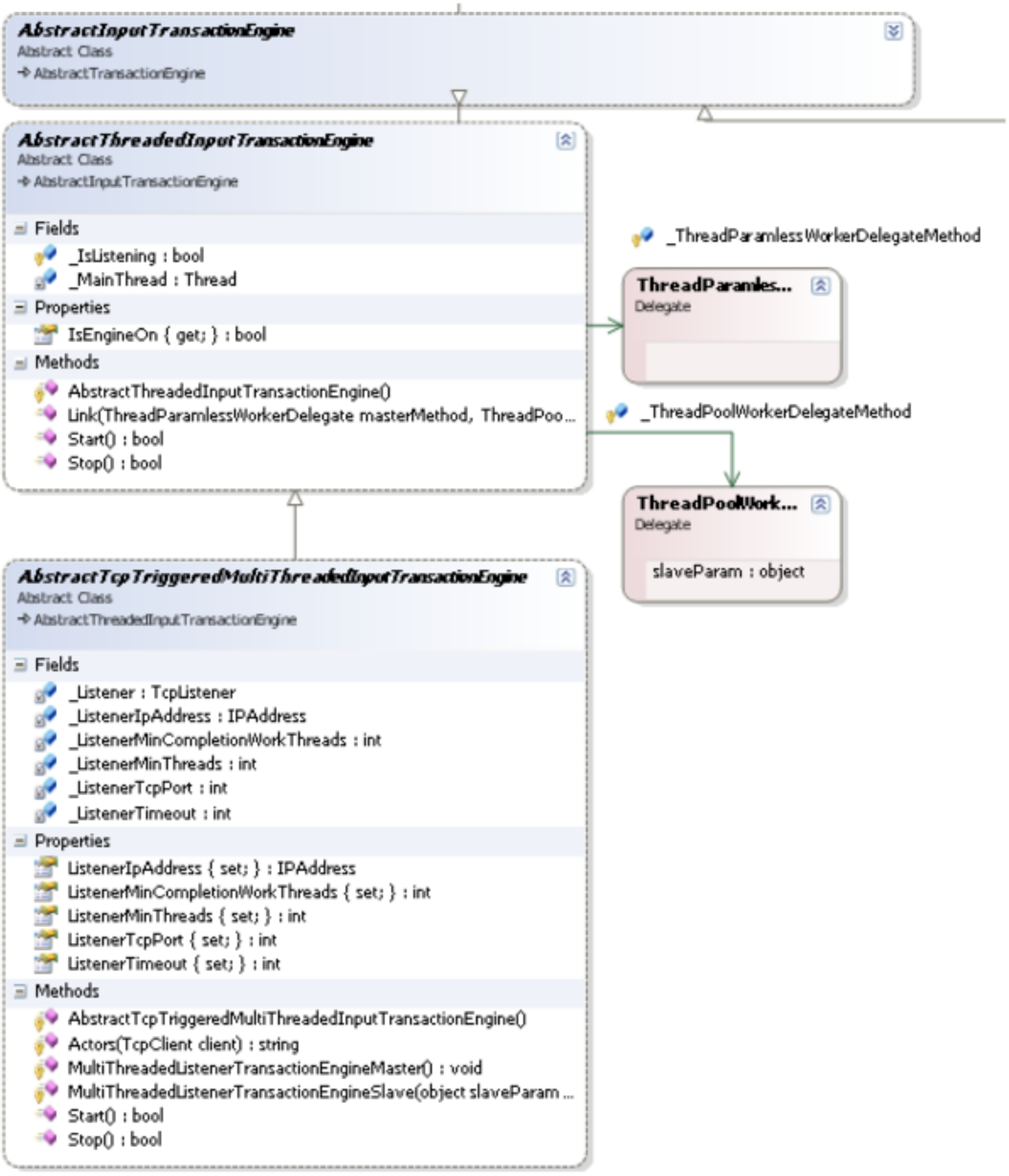

Figura 28 - Diagrama de clases de un motor multi-hilo disparado por conexiones TCP entrantes

Los *motores de entrada* multi-hilos disparados por conexión TCP son los más fáciles de encontrar a la hora de hacer estudios de observación sobre las arquitecturas de servidores concurrentes. Debido a las características

propias del protocolo TCP (protocolo orientado a la conexión), la mayoría de las soluciones transaccionales lo utilizan como medio de transporte de la información. Este *motor de entrada* tiene como base un proceso que se encarga de escuchar conexiones entrantes a través de un puerto TCP. Debido a que las transacciones entre múltiples clientes son independientes entre sí (es decir, la emisión de una no depende de otra), los procesamientos pueden paralelizarse. Por lo que es necesario generar algún mecanismo para que el procesamiento se pueda delegar, en vez de ser atendido por el proceso de escucha de nuevas conexiones. Es por esto que es muy común encontrar que los servidores concurrentes tienen una arquitectura formada por un hilo de ejecución o proceso principal, y muchos subprocesos (*threads*), o procesos hijos (*child processes*) que son generados arbitrariamente para atender las transacciones. Entonces, estos tipos de motores pueden describirse por dos secuencias de trabajo características, una *secuencia de escucha y asignación*, y otra *secuencia de lectura u obtención de datos*: la primera secuencia deberá ser ejecutada por el hilo principal, y la secuencia de lectura por los *hilos de fondo*.

AbstractTcpTriggeredMultiThreadedInputTransactionEngine es la clase abstracta que almacena la mayoría de las características comunes observadas durante el estudio preparativo para este trabajo. Fue diseñada para cumplir con este patrón de procesamiento de transacciones, en donde existirá un hilo de ejecución principal que se encargará de escuchar de forma cíclica por una conexión TCP entrante, y delegará la tarea de procesamiento a tantos *subprocesos de fondo* (*background thread workers*) como transacciones entrantes hayan. A su vez la clase hereda (dentro de la jerarquía de *motores de entrada*), de las clases *AbstractThreadedInputTransactionEngine,* y esta última de *AbstractInputTransactionEngine*. A continuación se describen los campos, propiedades y métodos principales de estas clases, para entender la implementación realizada para cumplimentar con los conceptos observados.

Conceptos	**Objetivo**
_MainThread	Es un campo privado del tipo *Thread* que el motor usará para administrar el hilo principal encargado de escuchar las transacciones entrantes. Cuando el motor reciba el mensaje *Start(),* deberá iniciar la ejecución del *thread* en *_MainThread,* y cuando reciba el mensaje *Stop()* deberá finalizar la ejecución. El campo *_MainThread* albergará al método que será ejecutado por el hilo principal. En esta propuesta se asocia un delegado, que correspondientemente termina apuntando al método. Por defecto, el motor usará dos delegados para apuntar tanto al método de escucha (el

	que estará asociado con *_MainThread*) como al método de procesamiento (el que estará asociada con cada *background thread worker*). Para el caso de *_MainThread* se asociará el delegado *_ThreadParamlessWorkerDelegateMethod,* que a su vez estará apuntando por defecto al método *MultiThreadedListenerTransactionEngineMaster* que tiene la secuencia de escucha de clientes también por defecto. Si así se deseará, el delegado puede re-apuntarse a otra secuencia de trabajo implementada por un método de usuario, generando un comportamiento específico.
_Listener, _ListenerIpAddress, _ListenerTcpPort, _ListenerTimeout	Son los campos privados de la clase *AbstractTcpTriggeredMultiThreadedInputTransactionEngine* que almacenan la información necesaria para configurar un puerto TCP de escucha. *_ListenerIpAddress* del tipo *IPAddress* almacena la dirección IP del servidor concurrente. *_ListenerTcpPort* de tipo *int* almacena el puerto de escucha del motor. *_ListenerTimeout*, almacena la cantidad de tiempo de espera, medido en segundos, para las recepciones entrantes y las emisiones de información hacia un cliente (por defecto 60s). Por último, el campo *_Listener* es el socket TCP configurado para escuchar por el puerto *_ListenerTcpPort*, en la dirección *_ListenerIPAddress* con un *Time-Out* de *_ListenerTimeout* segundos.
_ListenerMinThreads, _ListenerCompletionWorkThreads	Son dos campos privados para configurar el *Thread Pool* asignado al contexto de ejecución del servicio. Con estos valores se puede modificar la cantidad de procesos simultáneos que pueden ser atendidos por el *pool.* Estos dos campos son utilizados directamente contra el método *SetMinThreads()* de la clase *ThreadPool* del *framework* .NET

_IsListening, IsEngineOn	*_IsListening* es un campo de *AbstractThreadedInputTransactionEngine,* en donde el motor debe asignarle *true* cuando esté encendido, o *false* cuando no lo esté. De este modo, se sobrecarga la propiedad *IsEngineOn* de solo lectura para que pueda evaluar *_IsListening* y dar visibilidad hacia afuera (o a otros objetos) sobre el estado funcional del motor.
_ThreadParamlessWorkerDelegateMethod, _ThreadPoolWorkerDelegateMethod, MultiThreadedListenerTransactionEngineMaster, MultiThreadedListenerTransactionEngineSlave, Link()	Para poder personalizar las dos secuencias de trabajo de este motor (la *secuencia de escucha*, y la de *secuencia de lectura*) se utilizaron delegados que apunten a los métodos con la implementación de cada una de estas secuencias. Estos delegados son *_ThreadParamlessWorkerDelegateMethod* (del tipo *ThreadParamlessWorkerDelegate*) que deberá apuntar al método con la *secuencia de escucha*, y *_ThreadPoolWorkerDelegateMethod* (del tipo *ThreadPoolWorkerDelegate*) que deberá apuntar al método con la *secuencia de lectura*. El motor tiene comportamientos por defecto para ambas secuencias, implementadas en los métodos *MultiThreadedListenerTransactionEngineMaster* (para la *secuencia de escucha*) y *MultiThreadedListenerTransactionEngineSlave* (para la *secuencia de lectura*). Los dos delegados apuntan respectivamente a estos métodos, y pueden re-apuntarse a métodos desarrollados por el usuario, a través del método *Link()*. *Link()* requiere dos parámetros, siendo el primero el método con la implementación de usuario para la *secuencia de escucha*, y el último el método de usuario para la *secuencia de lectura*.

Tabla 9 - Elementos comunes principales, definidos en las clases bases de los conceptos *Threaded Input Transaction Engine* y *Tcp Triggered Multi Threaded Input Transaction Engine.*

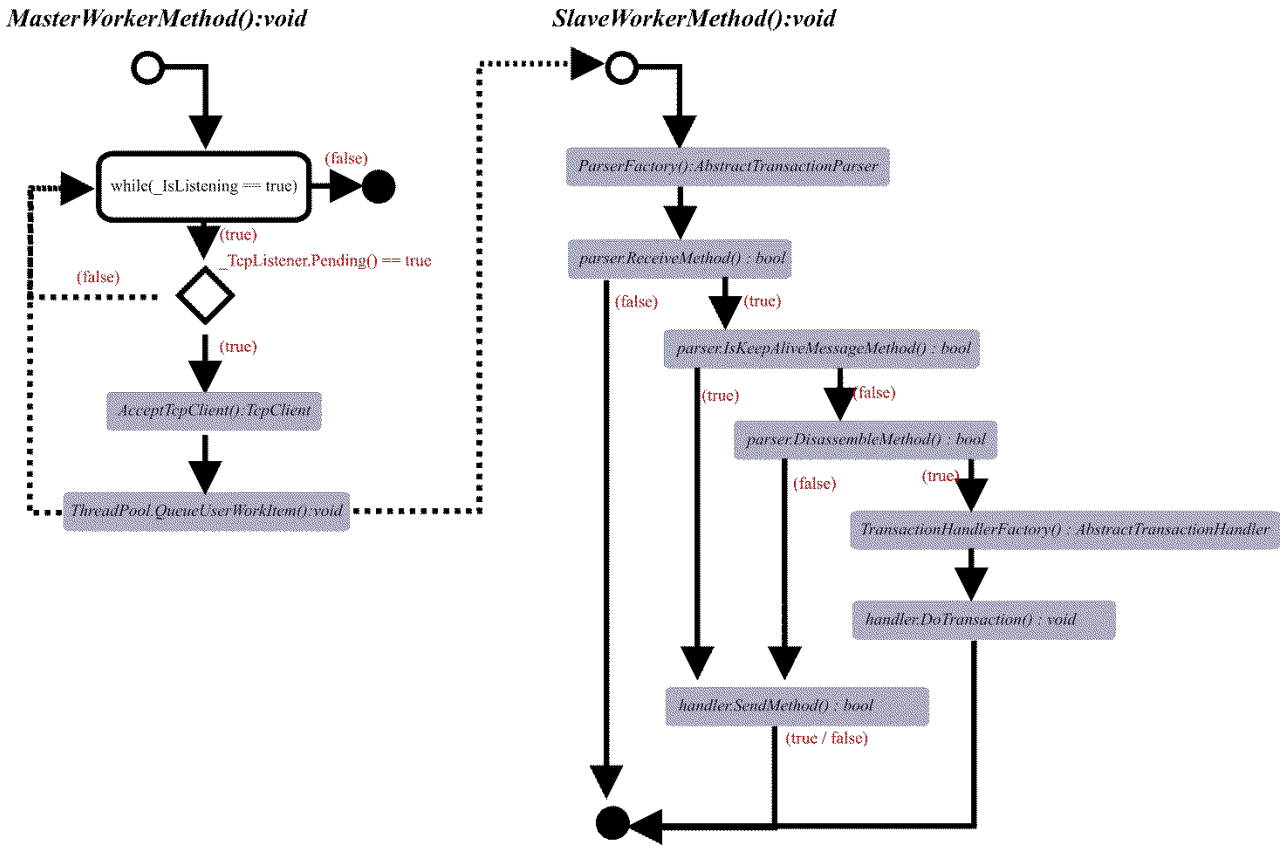

Figura 29 – Secuencias de trabajo propuestas para los motores de entrada multihilo disparados por clientes TCP. La secuencia de la izquierda corresponde a la secuencia de escucha y asignación por defecto, mientras que la de la derecha corresponde a la secuencia de lectura de datos provenientes del cliente conectado.

4.5.5.1.2 Ejemplo B: Motor de Entrada Multi-Hilos Disparado por Conexión TCP, con Reutilización de Socket (*Reusable Socket Tcp Triggered Multi Threaded Input Engine*)

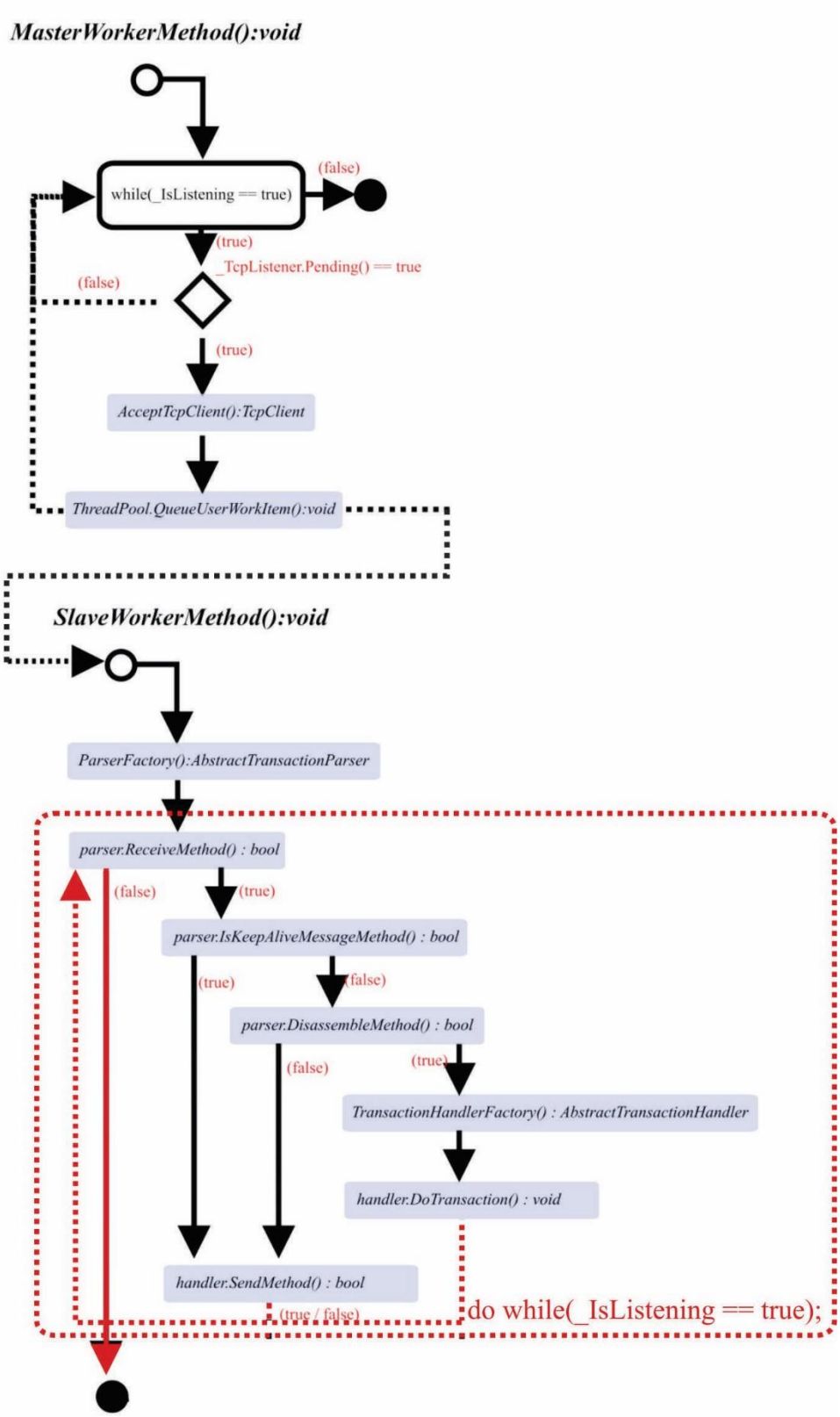
MasterWorkerMethod():void
while(_IsListening == true)
(false)
(true)
_TcpListener.Pending() == true
(false)
(true)
AcceptTcpClient():TcpClient
ThreadPool.QueueUserWorkItem():void
SlaveWorkerMethod():void
ParserFactory():AbstractTransactionParser
parser.ReceiveMethod() : bool
(false)
(true)
parser.IsKeepAliveMessageMethod() : bool
(true)
(false)
parser.DisassembleMethod() : bool
(false)
(true)
TransactionHandlerFactory() : AbstractTransactionHandler
handler.DoTransaction() : void
handler.SendMethod() : bool
(true / false)
do while(_IsListening == true);

Figura 30 - Secuencias de trabajo propuestas para los motores de entrada multihilo disparados por clientes TCP con conexiones reusables. La secuencia de la izquierda corresponde a la secuencia de escucha y asignación por defecto, mientras que la de la derecha corresponde a la secuencia de lectura de datos provenientes del cliente conectado.

El motor que se presenta en la Figura 30 es muy similar al de *entrada con multi-hilos disparados por conexión TCP*. La única diferencia es la reutilización de las conexiones entrantes. Por ende, este motor es una variante del motor anterior, donde la conexión no es descartada luego de procesar toda la transacción en la secuencia de trabajo de lectura. En vez de ser descartada, la secuencia de escucha se encuentra enmarcada por un lazo que hace que se reinicie el ciclo, en el método de lectura *parser.ReceiveMethod()*. Si el cliente vuelve a enviar información, la misma podrá ser leída y procesada como un segundo requerimiento con la misma conexión. Este proceso puede repetirse hasta que el cliente se desconecte, hasta que el motor detecte un *Time-Out* en el método de recepción, o si se apaga el servicio (*_IsListening = false*).

El concepto de este motor surge de los POS usados en el mercado financiero, que utilizan la línea telefónica para comunicarse con los servidores de procesamiento. Si el motor descartase la conexión al finalizar el proceso del primer requerimiento, el POS debería recomunicarse por cada pedido adicional que envíe, haciendo llamadas telefónicas de más. En cambio, con este motor se puede reutilizar la comunicación para reducir costos de llamadas telefónicas y tiempos de conexión. Solo cuando los POS se comunican directamente a Internet (WiFi, GPRS, EDGE, 3G, 4G, etc) ambos motores son ambivalentes, ya que los costos y tiempos no se incrementarían sustancialmente entre uno y otro.

4.5.5.1.3 Ejemplo C: Motor de Entrada Disparado por Eventos Temporales (*Time Triggered Input Engine*)

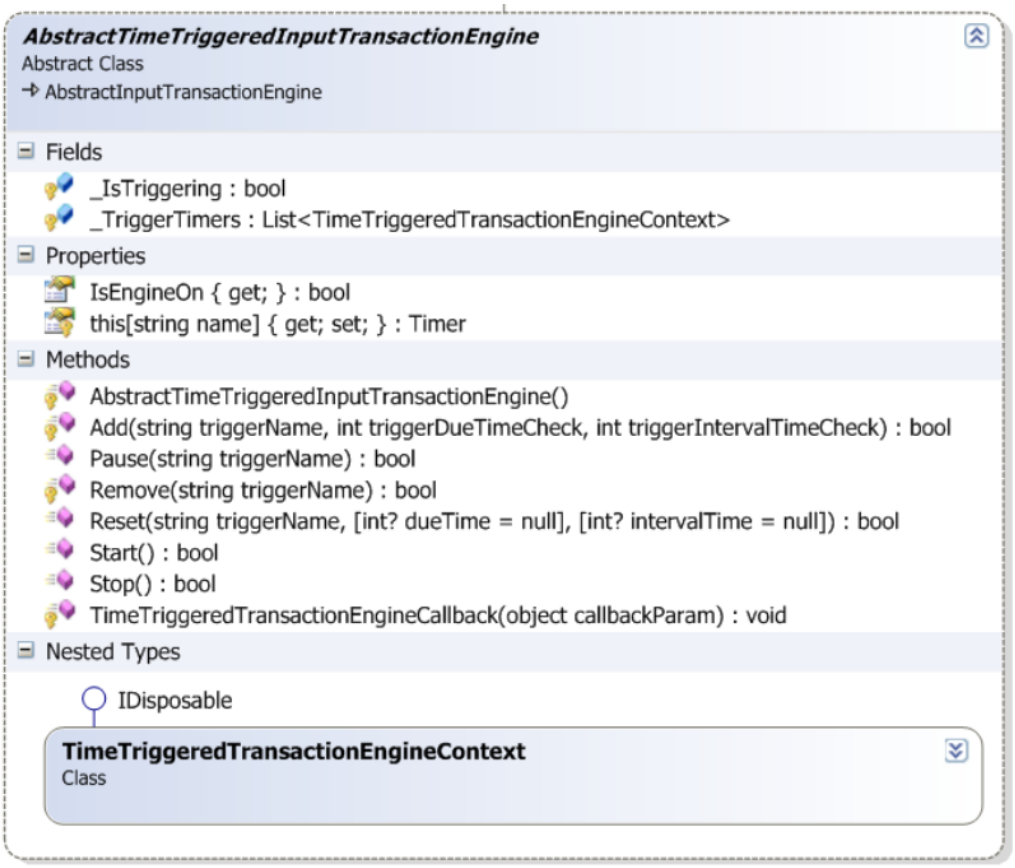

Figura 31 - Clase base de un motor disparado por tiempo

Este motor surge de la necesidad de *autogenerar* transacciones ante eventos internos del sistema de procesamiento. No todas las transacciones deben generarse por solicitudes entrantes: Por ejemplo, los mensajes de administración de red son muy útiles en redes *host to host* interbancarias para conocer el estado de la conexión, o manejar sesiones lógicas. Un ejemplo concreto es el de las transacciones de *echo*. A través de éstas, un sistema puede verificar que el nodo al cual desea conectarse se mantiene en línea, y en el caso que se detecte un corte en la comunicación (el *echo* no es respondido), permite generar una alarma, o evitar el reenvío de transacciones por ese camino. En este caso, los *echoes* no deberían ejecutarse cuando llega una transacción, sino a través de intervalos de tiempo preprogramados, de modo de evitar enviar una transacción real a un nodo caído.

Es por esto que este motor es muy útil para todas aquellas transacciones que deben programarse para ser ejecutadas a través de una base de tiempo, ya sea por única vez, o de forma cíclica. Cabe acotar que en esta propuesta se hizo incapié solamente en un motor disparado por eventos temporales, pero el concepto es extensible a cualquier otro tipo de evento interno del sistema, y se deja como línea de trabajo futura la implementación de otros motores que puedan atender algunos de estos eventos.

Conceptos	Objetivo
_TriggerTimers, this[string name] y TimeTriggeredTransactionEngineContext.	*_TriggerTimers* es una colección de contextos, en donde cada uno de éstos guarda información esencial para configurar un *timer* de forma unívoca dentro del motor. Los campos usados para describir un *timer* son: nombre descriptivo del *timer*, función de *callback*, cuenta del *timer*, cuenta del *timer* en modo cíclico y *handler* del *timer*. Todos estos datos están agrupados en la clase contextual en referencia, del tipo *TimeTriggeredTransactionEngineContext*. *this[string name]* es una propiedad de acceso de solo lectura, para obtener un contexto de *timer* dentro de la colección, a través de su nombre descriptivo.
Add(…), Remove(…)	Permite agregar y quitar respectivamente contextos de *timer*, de modo de manejar externamente los elementos pertenecientes a la colección de contextos interna al motor. La idea de estos métodos es que puedan ser utilizados para poder configurar inicialmente al motor, en algún constructor administrador de los motores, o en un método de inicio de un objeto *Facade* del sistema.
Pause(…), Reset(…)	El método *Pause()* saca de ejecución al *timer* referenciado por el parámetro *triggerName*. Lógicamente el *timer* debe existir previamente dentro de la colección de contextos interna al motor. Para volverlo a poner en cola de ejecución al *timer*, hay que hacer una llamada al método *Reset()*, especificando por parámetro el nombre del *timer* que se desea reactivar. Estos métodos son de utilidad para *prender y apagar timers* de transacciones que solo deben generarse en el motor bajo ciertas circusntancias, como por ejemplo la transacción de *Log On*, que solo debería reactivarse cada vez que se detecta un corte de conexión, y una posterior reconexión con un banco o procesador.

_IsTriggering, IsEngineOn	*_IsTriggering* es un campo privado del motor que debe setearse en *True* si los *timers* configurados están efectivamente iniciados, y por ende, generando transacciones según el comportamiento esperado. *IsEngineOn* debe ser sobrecargado para devolver el valor del campo *_IsTriggering*.
TimeTriggeredTransactionEngineCallback	Las funciones de *callback* de cada *timer* (los métodos que se ejecutarán cuando un *timer* finalice su cuenta regresiva) deben ser del tipo *TimeTriggeredTransactionEngineCallback*. Las funciones de *callback* no podrán ser métodos definidos por el usuario. Sin embargo será un método común a todos los motores de este tipo, en donde se configura una secuencia de trabajo predefinida. El usuario será el responsable de sobrecargar éstos métodos para hacer específica la secuencia de un motor (subclase) que herede de este tipo.

Tabla 10 - Elementos comunes principales, definidos en la clase base del concepto *Time Triggered Input Transaction Engine*

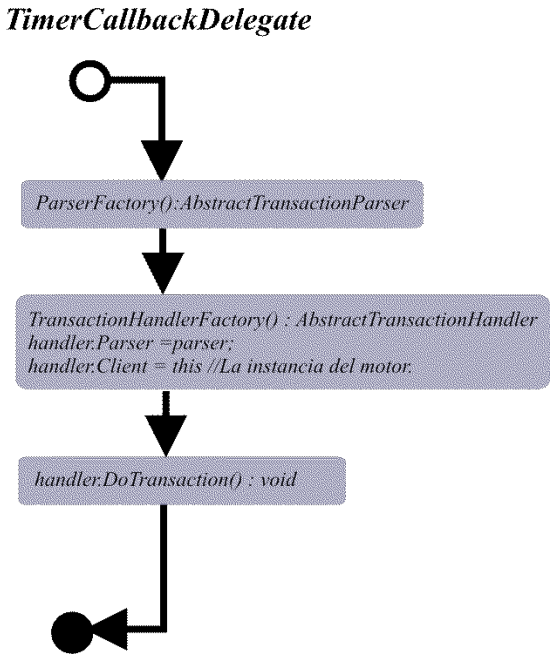

Figura 32 - Secuencia de trabajo de un motor disparado por eventos temporales programados

En la Figura 32, se describen los pasos que caracterizan la secuencia común para los *motores disparados por eventos temporales*. Cuando cualquiera de los *timers* termina de realizar su cuenta regresiva, ejecutará el método de *callback* asociado. Este método contiene una propuesta de secuencia, conceptualizada luego de la observación y análisis de otros motores, y de soluciones preexistentes que utilizaban bases de tiempo para generar transacciones. El método de *callback* se asignará a cualquier *timer*, por lo que la especificación se dará por la sobrecarga de algunos de los métodos que son parte de los pasos sugeridos en la secuencia.

Durante el primer paso, *ParserFactory()* debe ser sobrecargado para que devuelva un objeto del tipo *AbstractTransactionParser*. En los *motores de entrada* vistos hasta ahora, el objeto *Parser* sería el encargado de recibir y desarmar la trama proveniente de un cliente externo, que origina la solicitud. Como en este caso no hay cliente externo, *ParserFactory()* no solamente deberá retornar una instancia de *Parser* válida, sino que también deberá armar de cero una trama que será necesaria para ejecutar la transacción. Para cargar esa trama (que no llega de ningún lado, sino que se genera a propósito dentro del método), *ParserFactory()* deberá conocer cuál fue el *timer* que originó el evento. Para ello, el parámetro del método de *callback* tendrá cargado el nombre descriptivo del *timer* responsable de la ejecución del evento (ver prototipo *TimeTriggeredTransactionEngineCallback*). De este modo, un mismo motor podrá generar tantas transacciones distintas como *timers* tenga asociados.

En los pasos subsiguientes de la secuencia, el usuario deberá sobrecargar el método *TransactionHandlerFactory(),* para devolver una instancia de *AbstractTransactionHandler*, que pueda atender la transacción propiamente dicha. Dentro de ese método se deberá conocer el tipo de operación de la trama armada para poder instanciar la clase correcta. Este método, de hecho, tiene la misma forma de implementarse que en otros *motores de entrada*, ya que no distingue si la trama fue recibida realmente o generada internamente por el motor.

4.5.5.2 Motores de Salida (*Output Engines*)

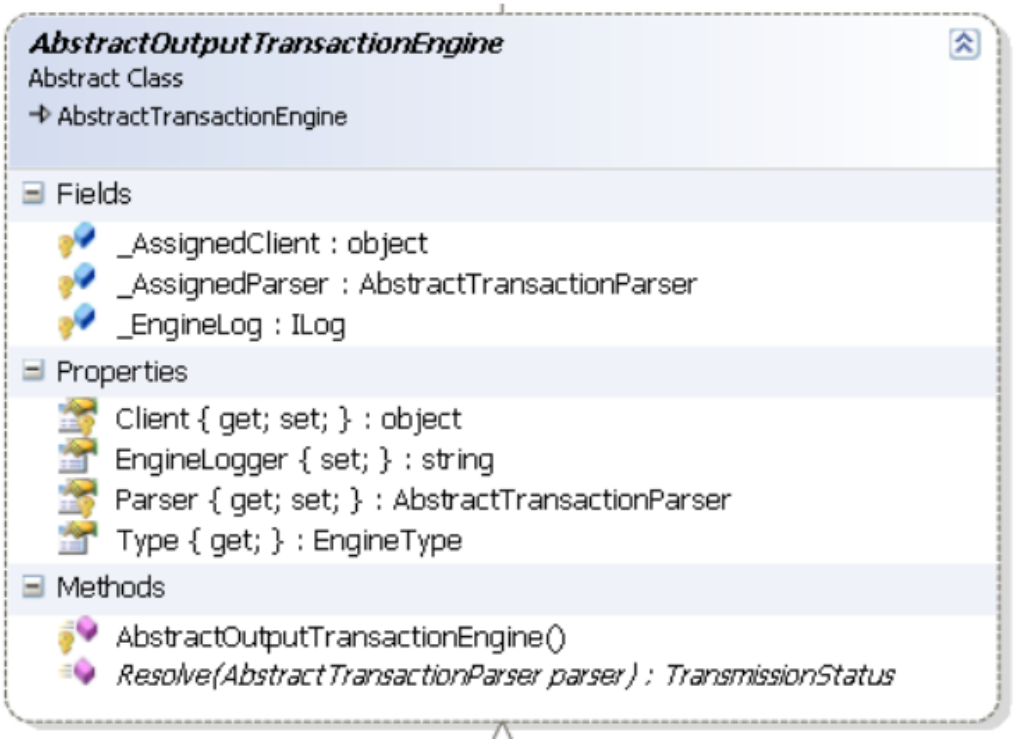

Figura 33 - Clase abstracta de un motor de salida

Los *motores de salida* surgen como antítesis de los de entrada, ya que muchas transacciones deberán ser eventualmente reenviadas hacia un nodo externo con motivos de lograr su procesamiento. Según las observaciones de campo y el análisis de soluciones ya en ambientes productivos, existen muchas formas de reenviar datos hacia un nodo externo en el mundo transaccional, y es la intención de esta familia de motores agrupar todas las características comunes recolectadas para tal fin, en una jerarquía de clases. La clase *AbstractOutputTransactionEngine* será la base de características comunes para todos los tipos de *motores de salida* descriptos en esta propuesta. Esta clase, al igual que *AbstractInputTransactionEngine,* hereda de *AbstractTransactionEngine.* A continuación se describen los elementos que han sido propuestos para ser parte de todos los *motores de salida* (elementos de las clase base *AbstractOutputTransactionEngine*), y luego se describirán algunos ejemplos de *motores de salida* concretos, que son utilizados en implementaciones reales.

Conceptos	**Objetivo**
_AssignedClient	Es un campo privado del tipo *object*, que deberá almacenar un objeto de conexión contra un nodo externo. Este objeto generalmente es del tipo *TcpClient* (por ejemplo, para conectarse contra un nodo externo vía TCP), pero puede ser de cualquier otro tipo que represente un *handler* de conexión (UDP*, Named pipes*, etc.).

_AssignedParser	Es otro campo privado de la clase *AbstractOutputTransactionEngine* que debe almacenar algún *Parser*. Ese *Parser* tiene que poder ensamblar y desensamblar paquetes y tramas, según el protocolo acordado con el nodo externo. Además puede eventualmente tener el conocimiento de cómo enviar y recibir datos formateados con el protocolo acordado entre las partes, por lo que el *motor de salida* podría intercambiar información con un nodo externo a través de este objeto.
Resolve(…)	Es un método virtual que debe ser redefinido en subclases dentro de la jerarquía de *motores de salida*, para que cada motor pueda desarrollar las tareas básicas de intercambio de información entre sí mismo y un nodo externo, según las especificaciones. Este método es el principal de cualquier *motor de salida*, ya que tiene la responsabilidad de realizar estas cuatro tareas, siempre que correspondan según la forma de intercambio de información: ➢ Conectarse con el nodo externo ➢ Enviar la información ➢ Recibir eventualmente una respuesta ➢ Desconectarse del nodo externo. Lógicamente, un usuario del *framework* podrá armar una implementación de *motor de salida* que pueda tener más o menos responsabilidades dentro de *Resolve()* según la funcionalidad que se quiera obtener para una solución específica, pero al menos debería cumplir con estas cuatro tareas básicas. Los *handlers* de transacción (del tipo *AbstractTransactionHandler*) que tengan la responsabilidad de reenviar la información hacia un nodo externo, (es decir, heredan la habilidad *ITransactionHandlerForwardable*) deberán hacer dentro del método de interfaz *Resolve(),* el correspondiente llamado al método *Resolve()* del *motor de salida* que usarán para el reenvío. Como parámetro de entrada recibirá un *Parser*, que a su vez contendrá las tramas y estructuras a enviar al nodo externo.
_EngineLog	Es un campo del tipo *ILog* (*Log4Net*) que se carga con una referencia a un *logger* válido configurado para escribir líneas propias del motor. Además del campo *_Log*, *_EngineLog* permite separar líneas de *log* exclusivas del motor con respecto a las líneas de *log* propias de las transacción (*_Log*). En esta propuesta se decidió generar

	el campo, ya que algunos de los *motores de salida* desarrollados eran bastante complejos y emitían muchas líneas de *log*. De este modo se permite separarlas en archivos diferentes para que no interfieran en el archivo principal de *log* de transacción. El objetivo entonces es reducir la carga de líneas de *log* que entorpecen el entendimiento de los sucesos propios de la transacción, más allá de la mecánica del motor usado.

Tabla 11 - Elementos comunes definidos en la clase base del concepto *Output Transaction Engine*

4.5.5.2.1 Ejemplo A: Motor de Salida Directo (*Straight Output Engine*)

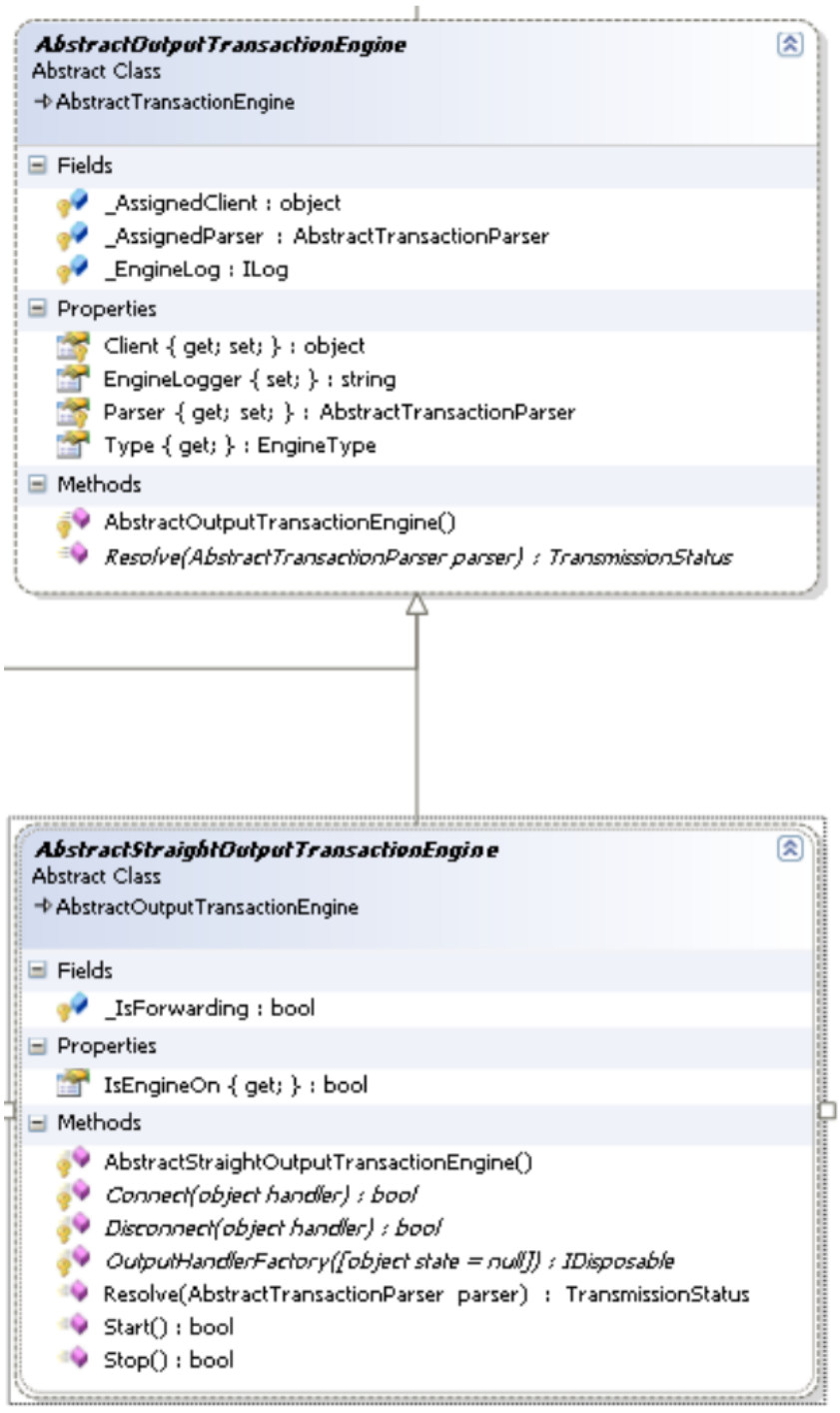

Figura 34 - Diagrama de Clases de un motor de salida directo base

El *motor de salida* directo está pensado para mantener una conexión con un nodo por cada transacción en curso. Es decir, la relación *Transacción por handlers de conexión* es uno a uno. Si bien hasta antes de esta propuesta, la mayoría de los hilos de ejecución se comunicaban de esta forma con tan solo definir un *handler* de conexión (por ejemplo un *handler* del tipo *TcpClient*), la idea de estos objetos es formalizar las conexiones de este tipo bajo la jerarquía de motores de esta propuesta. En otras palabras, no hay ninguna ventaja técnica entre usar este motor y un simple *handler* de conexión, salvo por que estos motores sirven para encapsular la lógica dentro de elementos reutilizables del *framework.* Además estas clases pueden ser útiles como base de un lenguaje de dominio específico. Con esta meta en frente, la idea es poder agrupar a todas estas conexiones salientes (con relación uno a uno entre *thread* y *handler*) en *motores de salida*, que puedan luego ser autogenerados por herramientas intérpretes de un eventual DSL.

Los elementos más importantes de la clase base de este tipo de motor (*AbstractStraightOutputTransactionEngine*) son los siguientes:

Conceptos	**Objetivo**
_IsForwarding, IsEngineOn	*_IsForwarding* es un campo privado que debe configurarse a *True* si el motor puede reenviar transacciones hacia un nodo externo. Más allá de la necesidad intrínseca de estar conectado para reenviar el pedido, muchos motores usan esta propiedad para diferenciar estados de sesión lógicos, habilitando o deshabilitando el reenvío de peticiones a un nodo, en un momento dado. Es responsabilidad del motor configurar internamente este campo en función de la mecánica del mismo. *IsEngineOn* es una propiedad pública de solo lectura, para poder exponer si el motor está habilitado para reenviar solicitudes; en otras palabras debería implementarse para retornar *_IsForwarding*.
Connect(…), Disconnect(…)	Son dos métodos virtuales que deben sobrecargarse obligatoriamente en las subclases que hereden de *AbstractStraightOutputTransactionEngine*. Estos métodos son llamados desde la secuencia del motor en cuestión, y lógicamente deben tener las implementaciones para poder conectarse y desconectarse satisfactoriamente de un nodo externo al sistema.
OutputHandlerFactory(…)	Es otro de los métodos virtuales de esta clase que debe sobrecargarse obligatoriamente para el correcto funcionamiento del motor. El método también es llamado desde la secuencia redefinida en la

	sobrecarga de *Resolve()*, dentro de la implementación de *AbstractStraightOutputTransactionEngine*. El objetivo de este método *fábrica* es devolver un *handler* de conexión válido, sin importar el tipo del mismo, pero con la única condición que ese tipo debe heredar de la interfaz *IDisposable*. De esta forma, la secuencia llama a este método dentro de un bloque *using,* asegurando el correcto tratamiento del objeto en memoria. La mayoría de los tipos de *handler* usados para conexiones entre nodos (según las observaciones realizadas en sistemas preexistentes), terminan conteniendo en su árbol de herencia una referencia a *IDisposable*, probablemente porque en su mayoría contienen objetos heredados del tipo *stream* para manejar el intercambio de datos. Por esta razón se propuso que ésta sea la única condición para manejar una referencia a un *handler* de forma impersonalizada dentro del motor. La responsabilidad de definir el tipo exacto del *handler* a usar, debe ser de la subclase que herede de este motor.

Tabla 12 - Elementos comunes definidos en la clase base para el concepto *Straight Output Transaction Engine*

Como se adelantó en la explicación de los métodos *Connect()*, *Disconnect()* y *OutputHandlerFactory()*, *Resolve()* deja de ser un método virtual puro (como estaba en *AbstractOutputTransactionEngine*) para contener un comportamiento por defecto. Ese comportamiento es la secuencia estándar que se propone para este tipo de motor directo.

El primer paso de la secuencia será averiguar si el motor efectivamente está en condiciones para retransmitir solicitudes; para ello evalúa el campo *_IsForwarding*. Si el motor está disponible, usará el *parser* pasado como parámetro para ensamblar el requerimiento a reenviar. Es decir, la estructura cargada durante *BuildRequirement* (ver habilidad *ITransactionHandlerForwardable*) es ensamblada a un *parser stream* de forma automática en este paso, sin necesidad que la transacción tenga esa responsabilidad.

El segundo paso consiste en generar una instancia de un *handler* válido para la conexión, a través de *OutputHandlerFactory*.

Por último, se deberá realizar la secuencia de conexión, envío, recepción y desconexión. La conexión y desconexión se realiza a través de los métodos virtuales *Connect()* y *Disconnect()*, mientras que el envío y la recepción son realizados por los delegados (*SendMethod* y *ReceiveMethod* respectivamente) del *parser* pasado como parámetro. Para la versión del motor que se propone en este trabajo, se hace obligatorio que el *parser* pasado como parámetro cuente con las implementaciones para poder intercambiar información según su propio

protocolo. En caso contrario, el motor que herede de *AbstractStraightOutputTransactionEngine* deberá sobrecargar *Resolve()* para modificar la secuencia propuesta, e incorporar llamadas a métodos de envío y recepción propios.

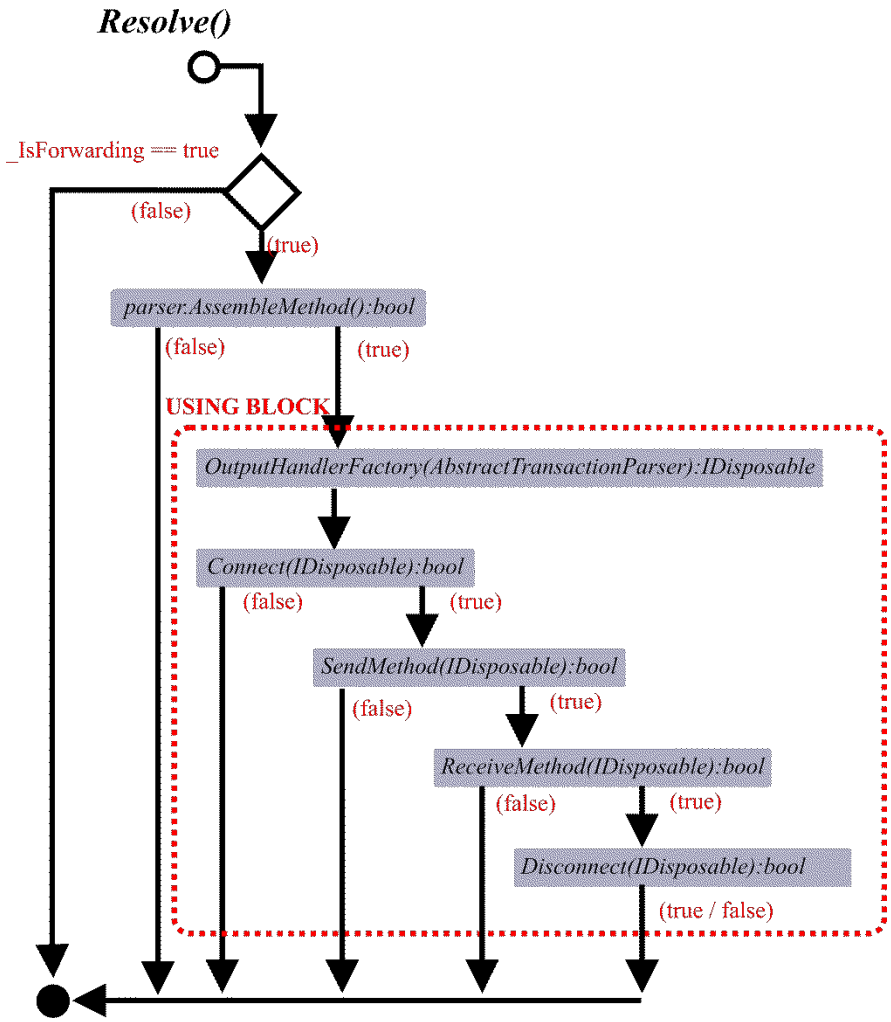

Figura 35 - Secuencia de trabajo de un motor de salida directo

4.5.5.2.2 Ejemplo B: Motor de Salida Mono-Punto (*Tcp Funneled Output Engine*)

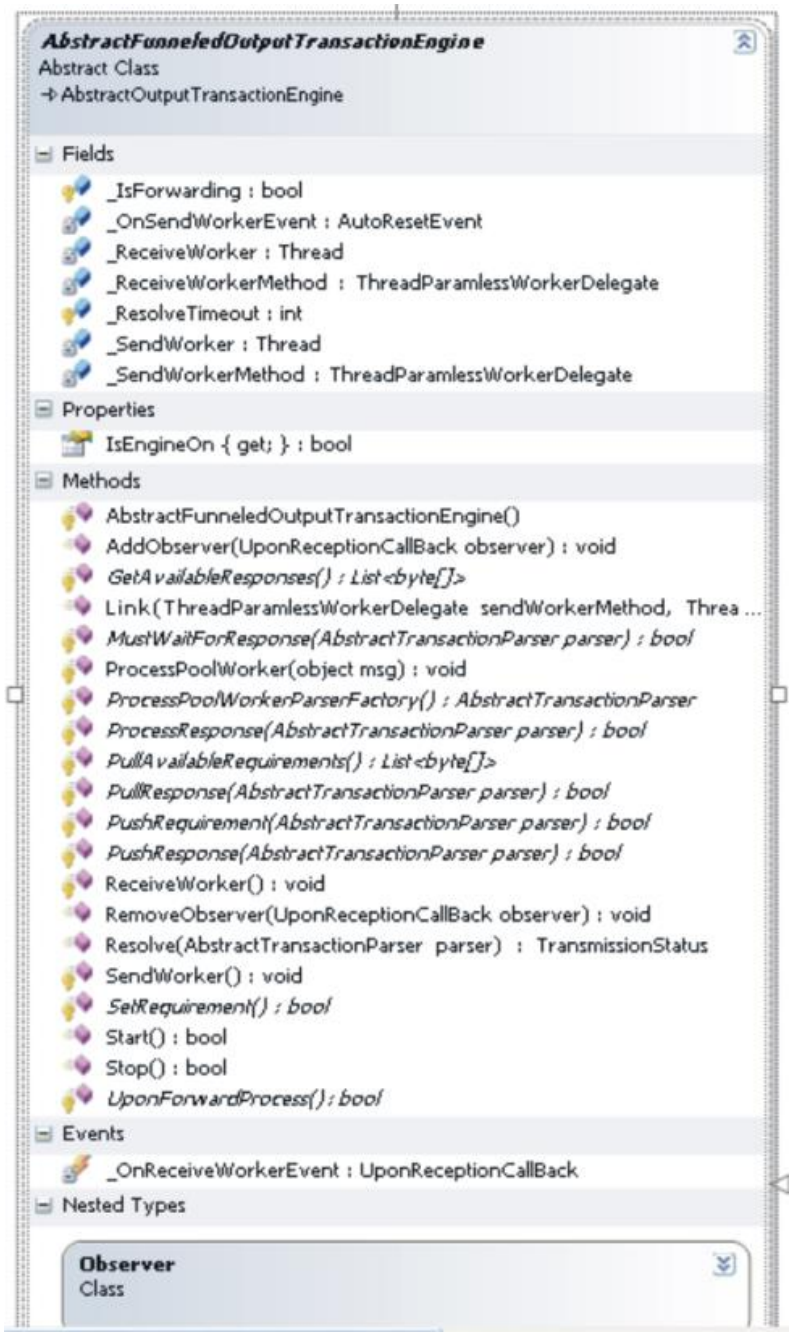

Figura 36 – Diagrama de clases de un motor mono-punto (solo una conexión TCP para múltiples hilos de ejecución)

El *motor de salida mono-punto* surge de la necesidad de usar un único medio de conexión contra un nodo externo, y que este medio a su vez pueda ser compartido por todas las transacciones simultáneas dentro de un

sistema transaccional. Esto quiere decir que, para este motor, la relación *handler de conexión por hilo de ejecución* es uno a muchos. Este motor es sumamente complejo comparado con el directo de salida descrito en la sección anterior, ya que requiere más lógica de control para poder mantener esta relación *uno a muchos*. Según las observaciones realizadas sobre sistemas transaccionales, este tipo de conexiones mono punto se usan habitualmente entre sistemas de procesamiento bancarios, para transacciones de crédito y débito. Dado que el negocio de procesamiento de transacciones monetarias es grande, este motor se convierte de gran utilidad para facilitar la integración con otros sistemas de procesamiento. Esta propuesta de *framework* intenta capturar los mecanismos necesarios para hacer factible esta facilitación en un conjunto de campos, propiedades, eventos y métodos dentro de la clase *AbstractFunneledOutputTransactionEngine*. El nombre de esta clase hace referencia a la palabra inglesa *Funnel*, que significa *embudo,* de modo de describir la característica principal de converger todas las peticiones en una única conexión para todas las transacciones que requieran reenviar una solicitud hacia el exterior del sistema.

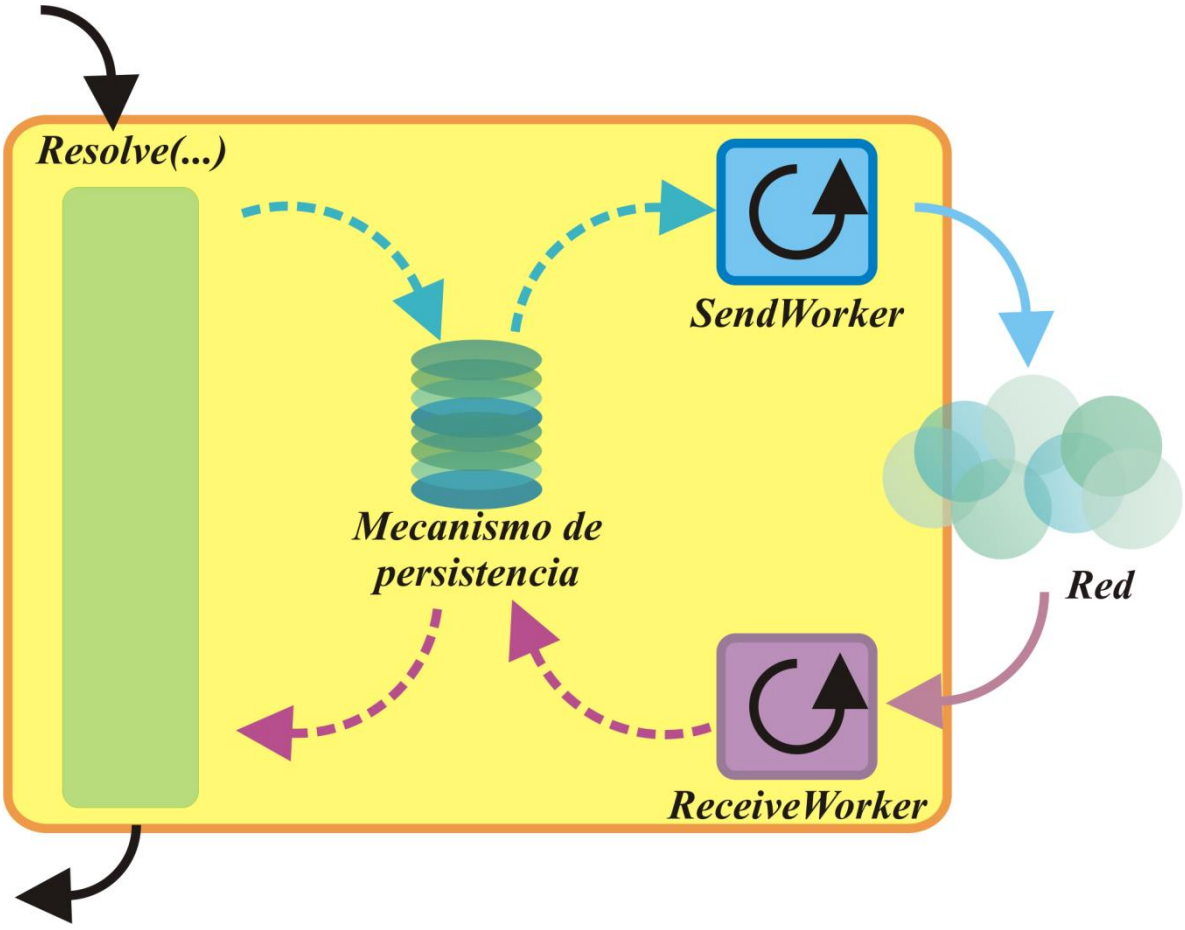

Figura 37 - Elementos principales del motor mono-punto de salida.

La mejor forma de entender este motor es comprendiendo su secuencia de trabajo, y sus dos hilos de ejecución principales. Así mismo es necesario comprender que el motor deberá hacer dos tareas importantes, más allá de enviar y recibir: Deberá *concentrar* las solicitudes a enviar, y *distribuir* a los hilos de ejecución las eventuales respuestas recibidas, asociando cada solicitud con su respectiva respuesta. Se decidió que el motor debía contar entonces con dos *threads* simultáneos, que deberían correr todo el tiempo, mientras el motor esté encendido: Un hilo de ejecución dedicado a enviar las solicitudes concentradas, y otro hilo de ejecución dedicado a escuchar por el *handler* de conexión, recibir datos por este, y distribuir la información entrante. Por último, existirán contemporáneamente tantos hilos ejecutando el método *Resolve()*, como transacciones simultáneas existan. El motor se debe apoyar sobre un mecanismo de persistencia para poder hacer la tarea de concentración, almacenaje de solicitudes, y el emparejamiento de respuestas con sus respectivas solicitudes.

La secuencia de trabajo se inicia con al menos una sola transacción haciendo una llamada a *Resolve()*. A continuación se describen los pasos internos, para facilitar la comprensión de su mecanismo:

1. Se declara una instancia de un objeto del tipo *Observer* que pertenecerá al contexto de ejecución de cada *thread* (es decir, de cada transacción). La clase *Observer* es un tipo interno definido por el motor, que permite implementar el patrón *Observer (*ver sección *Refactorización de Patrones - Observer)*. Para los fines del motor, los *Observers* sirven en la distribución de mensajes entrantes, desde el hilo de ejecución *ReceiveWorker,* a los hilos de las transacciones que estarán esperando una eventual respuesta.
2. Ahora debe evaluarse el campo privado *_IsForwardingEngine*. Como en otros *motores de salida*, este campo debe contener *True* si el motor puede reenviar solicitudes. Principalmente en este motor el campo gana importancia, ya que para mantener una conexión se debe controlar que la misma se encuentre activa, inclusive con alguna sesión lógica iniciada previo a estar apta para el reenvío de solicitudes. *IsEngineOn* debería retornar si el motor se encuentra operativo.
3. Luego, se hace la llamada a *AssembleMethod* sobre el objeto *parser* pasado como parámetro. El objetivo es automatizar la serialización del requerimiento a reenviarse (de *Parser Structure* a *Parser Stream*). De este modo, el usuario del motor solo debe preocuparse de llenar la estructura con los datos a enviar. Retorna *True* si pudo serializarse la estructura, o *False* en caso contrario.
4. Con los datos serializados, se hace una llamada a un método abstracto del motor, llamado *PushRequirement()*. Este método requiere que se pase como parámetro el objeto *parser* que contiene los datos serializados. El objetivo de este método es persistir en algún mecanismo de almacenamiento la *corriente de bytes* que deberá enviarse a través del *thread SendWorker.* Dentro de la secuencia de *SendWorker,* se hace una búsqueda de los mensajes encolados que estén en estado *no enviado*; luego estos se capturan de la fuente de almacenamiento y se envían a través del *handler* de conexión interno del motor. Lógicamente, es responsabilidad de la subclase que herede de

AbstractFunneledOutputTransactionEngine que se sobrecargue este método para personalizar el acceso y el guardado de los datos, en función del mecanismo de persistencia seleccionado para la implementación concreta del motor (por ejemplo una base de datos). Este método deberá retornar *True*, si pudo encolar el mensaje, en caso contrario *False*. En conclusión, en este paso se logra la *concentración de datos* de muchos hilos de ejecución a uno.

5. Si el mensaje pudo ser efectivamente encolado, se debe avisar al *thread SendWorker* que uno nuevo se encuentra disponible para ser enviado por el *handler* de conexión interno del motor. Esto se logra a través del manejo de eventos asincrónicos: *SendWorker* está en espera de ser señalizado por el objeto privado *_OnSendWorkerEvent* del tipo *AutoResetEvent* (este tipo es propio de .NET, y sirve para implementar un *handler* de atención de eventos *llave en mano*). Para enviar señales de un hilo a otro (en este caso el receptor de la señal sería *SendWorker*), el *thread* que ejecuta *Resolve()* debe hacer la llamada al método *Set()* del objeto *_OnSendWorkerEvent.* Como en realidad simultáneamente puede haber muchos *threads* ejecutando *Resolve()* (potencialmente tantos como el máximo configurado para el *Thread Pool*), la llamada a *_OnSendWorkerEvent.Set()* se haría tantas veces como transacciones haya en un momento dado. Entonces se notificará a *SendWorker* todas esas veces. Este evento desata la recuperación del (o los) mensajes de la fuente de persistencia. Si *SendWorker* pudo ser señalizado correctamente, *Set()* devolverá *True*, y *False* en caso contrario.
6. El próximo paso en la secuencia es el llamado al método *AddObserver()* del motor. Esta llamada requiere que se pase por parámetro un método *callback*, que para la secuencia por defecto del motor será un método interno del objeto *Observer* (definido en el paso 1). Este objeto *observer* tiene como finalidad adherirse al evento *_OnReceiveWorkerEvent,* de modo que cuando haya una respuesta disponible, el motor pueda señalizar a todos los hilos de transacciones con *observers* válidos que existen datos nuevos, y que deberán intentar emparejarse con las solicitudes reenviadas en curso. El método que se señalará eventualmente en los *observers* (como consecuencia del evento *_OnReceiveWorkerEvent*) es *OnReceive*() (definido dentro de la clase *Observer*).

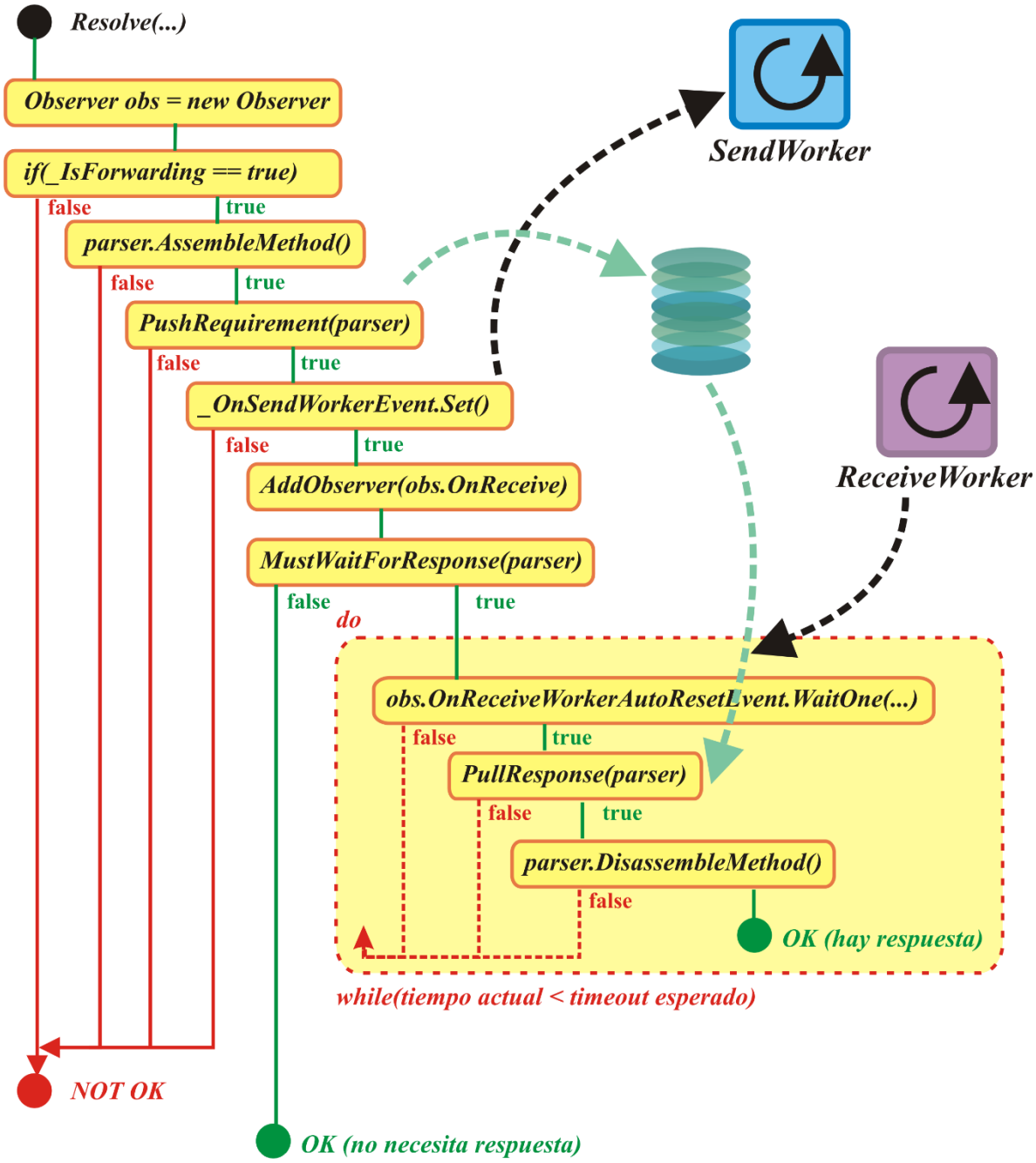

Figura 38 - Secuencia de trabajo propuesta para el método *Resolve()* usado por los hilos de ejecución de las transacciones, de modo de interactuar con el motor mono-punto de salida

7. Si bien los *motores de salida* son pensados para reenviar una solicitud, y luego esperar una respuesta, en el caso del *motor de salida mono-punto* surge una excepción a la regla. No todos los pedidos enviados por el *handler* de conexión deben esperar respuesta. Debido a que la conexión contra un nodo externo debe permanecer la mayor parte del tiempo en línea, los requerimientos pueden ser enviados no solo desde el *motor de salida*, sino que desde los bancos, los adquirientes o cualquier otro nodo externo conectado del otro lado. Es decir, la comunicación puede originarse de cualquiera de los dos lados. Por ejemplo, este *motor de salida* suele reenviar muchos mensajes de administración de red (*echos, log on, log off*, etc.) porque son necesarios para mantener activa la conexión. Pero también se pueden recibir estas transacciones sin necesidad de haber emitido un requerimiento previo, ya que los nodos externos pueden estar igual de interesados en mantener la conexión y cualquier sesión lógica iniciada entre las partes. Por este motivo, la secuencia llama al método abstracto *MustWaitForResponse(),* de modo que la subclase sobrecargue este método para analizar si el mensaje que se acaba de encolar en el punto 6 requiere o no una respuesta. Si no requiriera una respuesta, el método debe retornar *False*, y significa que el mensaje encolado era en sí mismo una respuesta a un pedido asincrónico recibo por *ReceiveWorker*. En cambio, si el mensaje encolado requiere respuesta, debe retornar *True*, y así la secuencia continua con la espera por la señalización del objeto *observer*.
8. El próximo paso es la espera por un potencial mensaje de respuesta. La secuencia hace el llamado al método *WaitOne()*, del campo público del objeto *observer* llamado *OnReceiveWorkerAutoResetEvent*. El mecanismo de disparo funciona de la siguiente manera: Cuando *ReceiveWorker* recibe un mensaje de respuesta, el evento *OnReceiveWorkerEvent* se dispara y alerta a todos los *observers* que se hayan subscripto al evento, de modo que cada hilo de transacción quede alertado. El método de *callback* que tiene definido por defecto cada *observador* (el método *OnReceive()*) implementa un llamado a *Set()* sobre el campo *OnReceiveWorkerAutoResetEvent* (que es del tipo *AutoResetEvent*). La propiedad más importante del *AutoResetEvent* es que puede esperar por un tiempo configurable, y si sobrepasa ese tiempo, el *observer* puede ser invalidado, respondiendo a las clases de la transacción en curso que el requerimiento salió por *Time-Out*. La única forma que se invalide un *Observer* es que desde el momento del envío pase más tiempo que el admitido por la ventana de espera de respuesta. De esta forma se realiza la segunda en importancia del motor: la *distribución de potenciales respuestas* entro los múltiples hilos que puedan estar esperando por una.
9. Si bien en el paso anterior cada hilo fue señalado a través de su instancia *observer,* cada hilo tendrá que realizar la tarea de emparejamiento de su solicitud con la potencial respuesta. Para ello la secuencia hace un llamada a *PullResponse()*. Este método se debe sobrecargar con la implementación para realizar esta tarea de emparejamiento según la lógica de cada subclase que herede de *AbstractFunneledOutputTransactionEngine.* Debe recuperar el mensaje recién llegado desde la fuente de persistencia, hacer el emparejamiento y retornar *True* si el mensaje anunciado en el paso anterior

pertenece al hilo de ejecución (transacción), y *False* en caso contrario (cuando no haya podido recuperarlo o emparejarlo).

10. Por último, si el mensaje pudo ser emparejado, resta desensamblarlo en una estructura de analizador (*Parser Structure*) para entregar la respuesta en este formato a la transacción que hizo la llamada a *Resolve().* Como en otras ocasiones, si se pudo desensamblar el mensaje (el mensaje era coherente) el método retornará *True*, en caso contrario *False*.

4.6 Refactorización a Patrones

4.6.1 Strategy

El patrón *Strategy* involucra la remoción de un algoritmo desde una clase principal, para ponerlo en una clase aparte. Puede haber diferentes algoritmos (estrategias) que son aplicables a un mismo problema. Si estos algoritmos se guardan todos en la clase principal, se genera una situación donde el código es muy extenso, de baja manutención, y con muchos condicionales que hacen difícil el seguimiento visual del código, o la intervención del mismo. Este patrón permite a un cliente elegir qué algoritmo usar, dentro de una familia de algoritmos posibles, y le otorga acceso al mismo. Así mismo, los algoritmos pueden ser expresados de forma independiente de los datos que usan. [10]

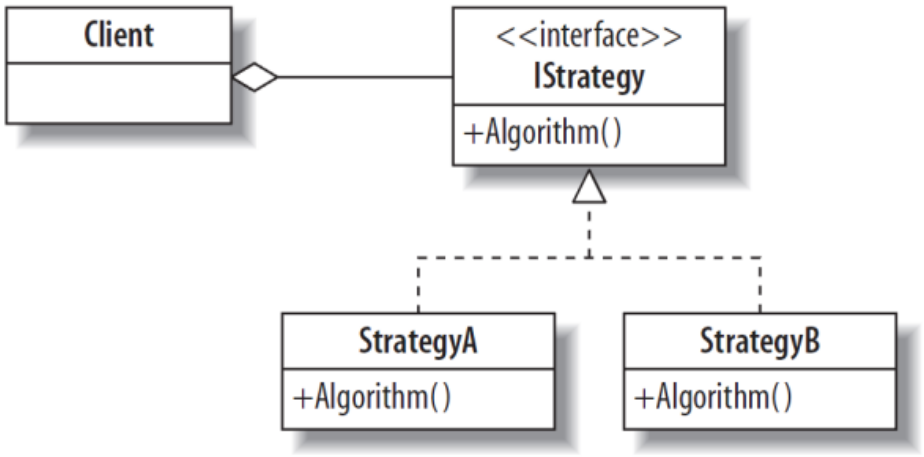

Figura 39 - Diagrama de clases del patrón Strategy [10]

Este patrón fue implementado en el *framework* que aquí se propone para poder reducir la cantidad de sentencias condicionales que existían en el algoritmo de proceso de una transacción. Como se comentó con anterioridad, las secuencias de las transacciones en las soluciones previas a este trabajo estaban implementadas en una sola clase. En el caso particular que se citó del *switch* de recarga de tiempo-aire para la oficina de Argentina, esa clase se llamaba *CIso*, debido a que implementaba la interpretación del protocolo ISO8583. Por lo que todas las transacciones terminaban siendo procesadas por un algoritmo que residía en el método principal de esta clase (*DoTransaction()*), pero que a su vez la resolución no era idéntica para todas las transacciones por igual.

A medida que se requería la intervención del código, ya sea porque se agregaba una transacción, o porque se debía corregir un *bug*, se fueron agregando comportamientos en este método, o en métodos auxiliares que eran llamados como pasos internos de *DoTransaction()*. Por consecuencia el código creció, y también creció la cantidad de sentencias condicionales, para poder mantener las diferencias de algoritmos entre distintos tipos de

transacciones. La probabilidad de modificar indirectamente el proceso de una transacción, que no era objeto de la modificación deseada, crecía a medida que se agregaban más líneas de código. Es decir, el código quedaba cada vez más acoplado. Así mismo, la capacidad de mantener el código bajaba, y los costos se incrementaron, ya sea por el tiempo que requería hacer una modificación adentro de ese método, o por el tiempo que costaba probar que no se afectaban otras transacciones que estaban funcionando correctamente.

La refactorización principal que se hizo a partir de este *framework* fue la de desacoplar los algoritmos de esta clase principal a otras clases. Pero, ¿cuáles debían ser estas clases? Para ello se observaron y se identificaron las principales causas que llevaban a hacer las modificaciones frecuentes en el código de los *switches*. En la mayoría de las ocasiones, las intervenciones se hacían por agregados de transacciones nuevas, o por modificaciones en los pasos de las secuencias de procesamiento de una determinada transacción preexistente.

De este modo, el patrón *Strategy* modifica el enfoque de los servidores concurrentes realizados, desde la orientación al *flujo de transacción,* hacia la orientación al *tipo de transacción.* En otras palabras, el *motor de entrada* tenía que entender de la trama entrante cuál era el tipo de transacción (venta, consulta de saldo, depósito, consulta de puntos, etc.) y en función de ese tipo, instanciar la clase responsable del algoritmo de proceso correspondiente. Luego, el motor solo tiene que llamar al algoritmo de forma genérica, siendo implementado en cada subclase según las necesidades de cada transacción.

A continuación se muestra un diagrama de clases, obtenido de una implementación real que consta de cuatro transacciones: *Venta en línea* (*SaleOn*), *Venta fuera de línea* (*SaleOff*), *Venta prepaga en línea* (*SalePreOn*) y *Consulta de Saldo* (*BalanceQuery*). El punto principal es mantener la relación *una clase, una transacción.* Si bien es posible implementar el patrón con una interface, en *TransactionKernel* se utiliza la clase base de todas las transacciones (*AbstractTransactionHandler*). Como ya se explicó en las secciones anteriores, en dicha clase se define el único punto de entrada a una transacción, el método *DoTransaction().* Este método es llamado desde todos los *motores de entrada*, indiferentemente del tipo de transacción. Luego *DoTransaction()* llamará convenientemente a los métodos privados *DoFirstStage(), DoSecondStage() y DoThirdStage()* según la secuencia de procesos de tres etapas que se adoptó para este *framework*.

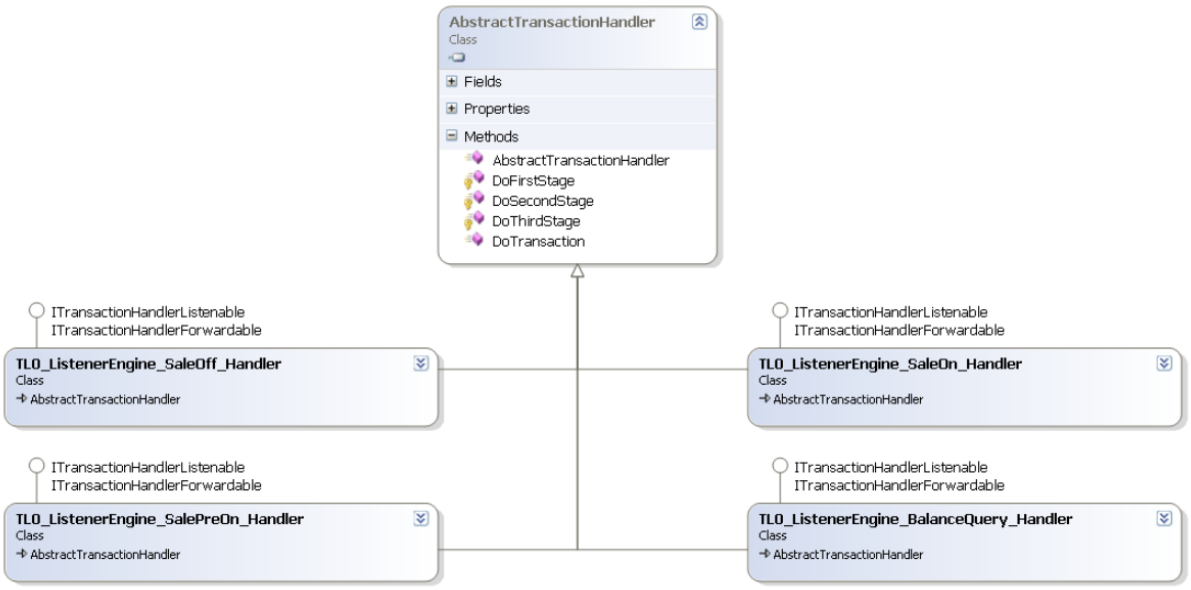

Figura 40 - Diagrama de clases de una implementación de ejemplo de cuatro transacciones.

4.6.2 Template Method

El patrón *Template Method* habilita que un algoritmo difiera en ciertos pasos en función de la subclase instanciada. La estructura del algoritmo (su secuencia) no cambia, pero algunos de sus pasos son redefinidos y atendidos de forma diferente en cada subclase. [10]

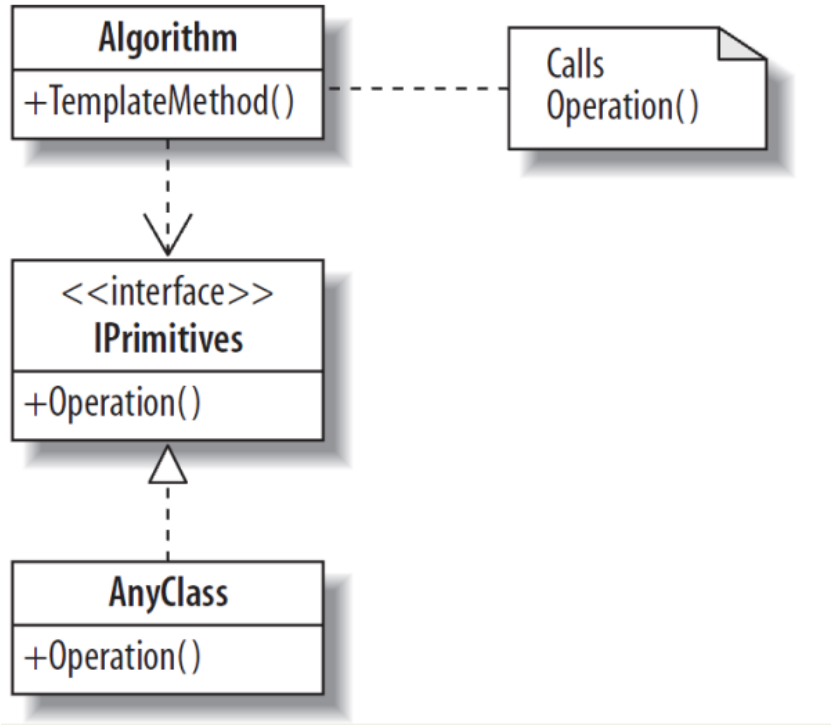

Figura 41 - Diagrama de clases del patrón Template Method [10]

Template Method es de gran utilidad para diferenciar un mismo algoritmo según la definición de sus pasos, en cada una de las subclases interesadas. En el diseño del *framework* que se presenta en este trabajo se implementó este patrón para acoplarse a la refactorización realizada con el patrón anterior (*Strategy*). Cada transacción puede personalizar el proceso realizado en cada uno de los pasos de la secuencia definida en *AbstractTransactionHandler*. El orden de los pasos de esta secuencia es fijo, y se armó de la observación de secuencias comunes en soluciones transaccionales preexistentes.

Según se explicó anteriormente, la secuencia esta armada a partir de un esqueleto definido en *DoTransaction()*. A su vez, este método *DoTransaction()* llama a las tres etapas en orden (*DoFirstStage()*, *DoSecondStage()* y *DoThirdStage()*). Los pasos propiamente dichos de la secuencia están definidos dentro de estos tres métodos privados. La ejecución de cada paso se realiza a través de llamadas a *delegados*[16], que están definidos como conjuntos de habilidades comunes, y que se heredan a través de interfaces. Con el fin de recordarlas, el conjunto de habilidades dentro de la interfaz *ITransactionHandlerListenable* permite heredar los métodos de

[16] Los delegados son mecanismos en .NET Framework para manejar de manera indirecta los mensajes entre objetos. Es la versión análoga al concepto de punteros a función de C.

conectividad con *motores de entrada*. Por su parte, *ITransactionHandlerForwardable* permite heredar habilidades de conectividad con otras transacciones en capas siguientes y/o *motores de salida*, *ITransactionHandlerPersistable* permite heredar habilidades de persistencia, e *ITransactionHandlerMaintenanceable* permite heredar habilidades de mantenimiento post-proceso.

Por ejemplo, en el caso del armado de un simulador (que solo responde a transacciones entrantes con tramas predefinidas para fines de *testing*) se implementó una transacción *objetivo* (*target*), que solo hereda *ITransactionHandlerListenable,* y que está conectado a un *motor de entrada* del tipo *TcpTriggeredMultiThreadedInputEngine*. La transacción deberá implementar *BuildReponse()*, *GetRequirement()*, y *Reply()* debido al contrato impuesto por la interfaz. Luego la subclase asociada a la transacción deberá estar implementada con código acorde al procesamiento deseado en cada paso: En *GetRequirement()* se deberán capturar los datos que le resulten relevantes a la transacción. Estos datos se obtienen de la estructura armada por el *motor de entrada* al deserializar los datos del cliente. En *BuildResponse()* se deberá armar la estructura de respuesta y serializar los datos, y en *Reply()* se deberá decidir si corresponde enviar una respuesta al cliente. Cuando una conexión de un cliente arribe, se instanciará la subclase de la transacción a simular, y el motor llamará al método *DoTransaction()*, perteneciente a la clase abstracta *AbstractTransactionHandler*. Debido a esa abstracción, las tres etapas se ejecutarán encontrando que solo tres delegados del total de delegados disponibles están instanciados, los cuales serán llamados de forma indiferente (sin precisar adonde realmente están apuntando, mientras cumplan con el prototipo). En caso de heredar más habilidades, la cantidad de delegados instanciados y apuntados a métodos será mayor, y los pasos podrán diferenciarse aún más entre una transacción y otra. Si una transacción no cuenta con todas las habilidades, esos pasos apuntarán a *NULL*, indicando que no hay método ni redefinición alguna disponible. De ese modo la secuencia a pesar de ser fija, toma flexibilidad y permite ser personalizada.

La personalización de la secuencia es fruto de este patrón, que nos permite mantener de forma fija una propuesta de pasos consecutivos que cumplen con la mayoría de los sistemas observados de campo, pero que a su vez permite variar la acción realizada según cada subclase.

4.6.3 Singleton Façade

Este es un patrón compuesto, que hace uso de dos de ellos conocidos individualmente, pero que en la práctica funcionan en conjunto casi de forma implícita. Uno de ellos es el patrón *Singleton*, cuyo propósito es asegurar que solo habrá una instancia de una clase en todo un contexto de ejecución. Este patrón también asegura una única forma de acceder globalmente a ese objeto, es decir la clase es instanciada una sola vez, y todos los mensajes al objeto son redirigidos a esa única instancia. Además el objeto no debería crearse hasta que no es necesario su uso. [10]

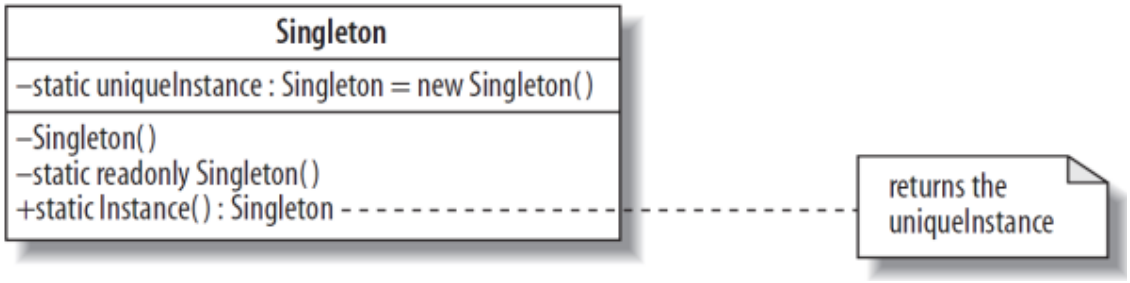

Figura 42 - Diagrama de clases del patrón Singleton [10]

A su vez, el rol del patrón *Façade* es proveer distintas vistas de alto-nivel de los sistemas internos al objeto *fachada*, ocultando los detalles a los clientes. En general, las operaciones que son intencionalmente expuestas para que los clientes hagan uso de ellas, pueden estar armadas por varias llamadas a métodos internos de los subsistemas que componen la solución. Este patrón a su vez, asume casi implícitamente que solo se instanciará un objeto de la clase *fachada* para un conjunto de subsistemas. Por esta razón, este patrón generalmente se combina con el patrón *Singleton,* para complementar su rol de única fachada.

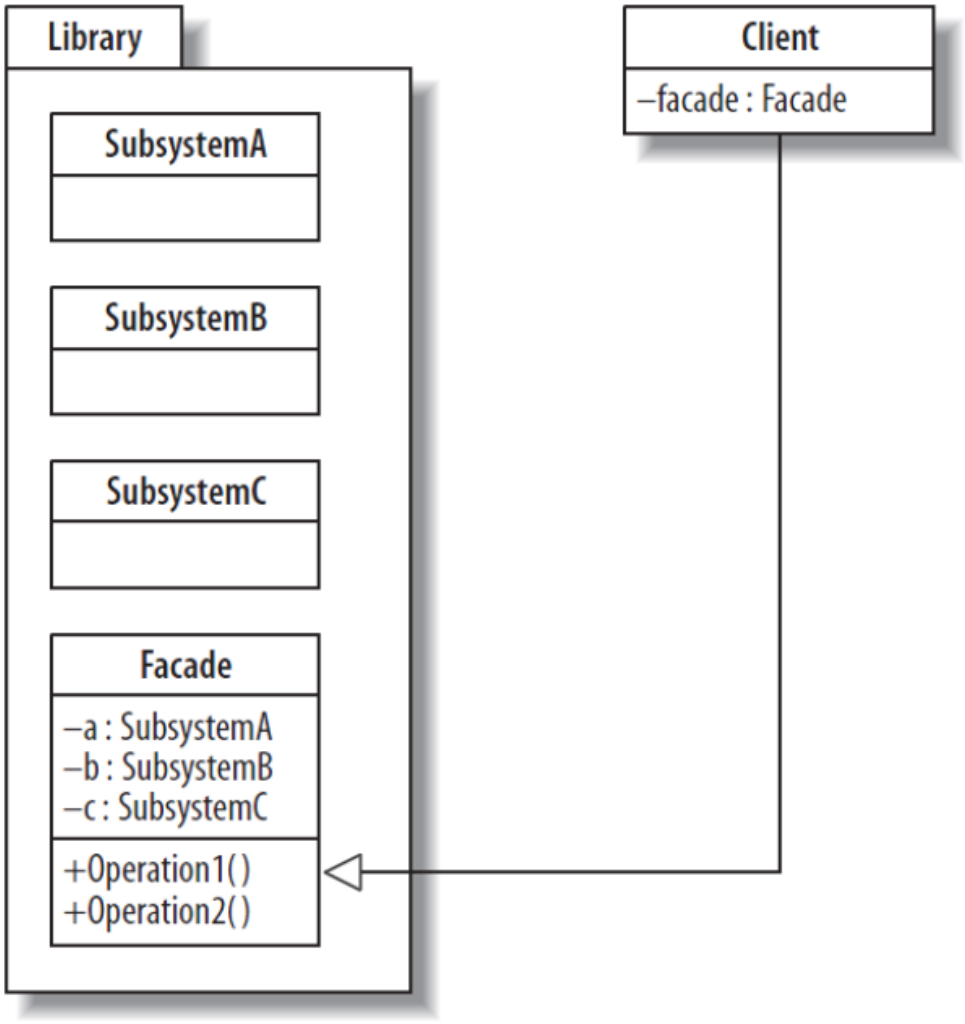

Figura 43 - Diagrama de clases del patrón Façade [10]

En esta propuesta de *framework* este patrón fue utilizado para manejar todas aquellas entidades que no estuvieran relacionadas a las múltiples posibles instanciaciones de clases asociadas a una transacción. Por ejemplo, las clases que heredan de *AbstractTransactionHandler* serán transacciones, y si en un momento dado hay mil clientes conectados a un *motor de entrada*, el motor generará mil objetos. Así también las clases de contexto (*AbstractTransactionContext*) se instanciarán mil veces, y los analizadores de protocolo también (*AbstractTransactionParser*). Sin embargo, otras entidades convienen ser mantenidas de forma individual, impidiendo su reproducción, como los *motores* y la *fachada*.

Como se explicó en la sección correspondiente, se decidió armar una clase *façade* para cada solución, y así concentrar todos los métodos a exponer al manejador de un servicio, principalmente los de encendido y apagado de la solución en si misma (que deberían ser llamados desde los eventos *OnStart()* y *OnStop()* del servicio

respectivamente). A su vez, cada solución puede tener uno o muchos motores, que deben poder encenderse y apagarse de forma individual, y necesitan una secuencia de inicialización de datos. Resultó eficiente entonces, seguir la filosofía de este patrón compuesto, de modo que haya una única fachada, con una sola instancia asociada, y una única instancia por cada motor dentro de la solución. En esa única instancia por motor, se pueden concentrar todas las operaciones de inicialización en el constructor (básicamente parámetros obtenidos de lecturas en una base de datos), y exponer algunos métodos que serán llamados por la fachada para encenderlo y apagarlo conveniente y ordenadamente (*Start()* y *Stop()*).

Cabe acotar que por herencia no se logró forzar el patrón *Singleton*: cuando se intentó generar la propiedad estática de la clase *AbstractTransacionEngine* dentro de sí misma, se sufrió el efecto colateral de una única instancia de motor posible por *AppDomain*. Es decir, un único motor, más allá del tipo en todo el sistema. Como este no era el efecto deseado, la única posibilidad que surgió es que cada implementación concreta de *AbstractTransactionEngine* definiera su propio campo del tipo correspondiente, y como contraparte este patrón debía implementarse por afuera del *framework* y de forma manual. El enfoque de esta capa es reducir las variaciones y desvíos que cada programador pudiera agregar por ingreso manual de código, por lo que este patrón se puede completar utilizando transformaciones de código, según MDD. Este punto se verá más adelante en el capítulo de formalización del *framework* a través de esta metodología

4.6.4 Chain of Responsibility

El patrón *Chain of Responsibility* trabaja con una lista de objetos manejadores (*handlers*) para un requerimiento dado, tal que si un objeto no puede resolver el pedido, se lo pasa al siguiente objeto de la cadena de responsabilidad. Al final de la cadena, y de no haberse encontrado algún objeto que se haga responsable del requerimiento, se puede encontrar tanto un comportamiento por defecto, o uno por excepción. [10]

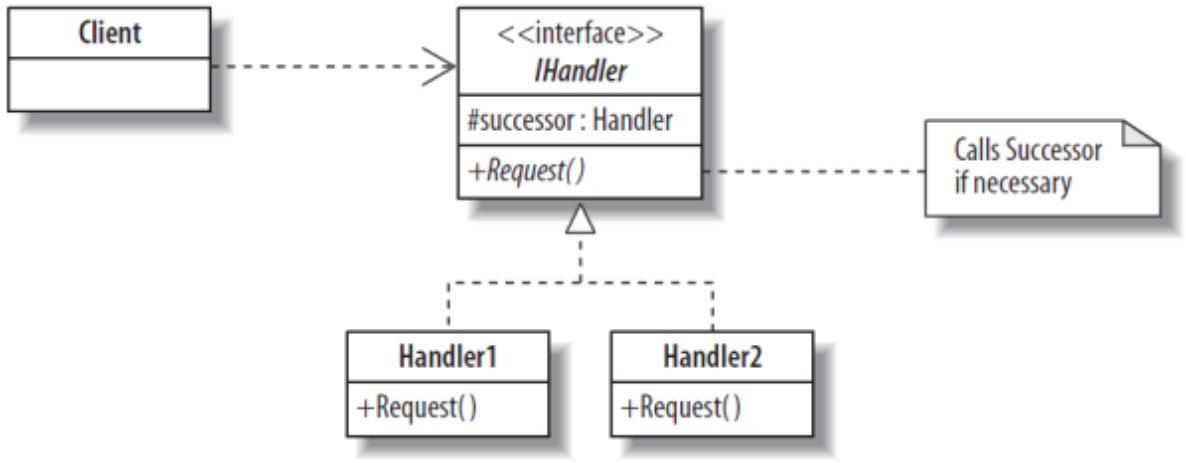

Figura 44 - Diagrama de clases del patrón Chain Of Responsibility [10]

Chain of Responsibility sirvió como inspiración para la resolución de un problema que se observaba en los concentradores originales, que era la habilidad de poder resolver transacciones en un modelo por capas, que podía incluir distintos nodos de procesamiento, si el nodo actual no era capaz de resolver la transacción. Este patrón sirvió para manejar una cadena de responsabilidad dentro de este modelo propuesto, donde si una clase no podía resolver el problema, le pasaba el procesamiento a la siguiente clase, o a un nodo lejano (otro servicio, u otra instancia corriendo y escuchando peticiones de alguna forma). Gracias a esta idea de traspaso de responsabilidad, se logró resolver los problemas de diseño referidos a las capacidades de ruteo de transacciones, en las soluciones de recarga de tiempo-aire. El diseño por capas es producto de las observaciones de varios servicios y soluciones a lo largo del tiempo, donde no siempre el desarrollo contemplaba la realización de un autorizador (en donde se resuelve una transacción por sus propios medios, y de forma local), sino que a veces era necesario retransmitir la información entrante de un cliente. Por lo que uno de los conceptos claves del *framework* es poder capitalizar capas de *forwarding*, cada una conteniendo sus motores definidos (ya sean de entrada o salida), y las transacciones ligadas a esos motores.

En el diseño generado, las transacciones son las responsables de hacer efectivo ese *forwarding*, de una manera u otra. De hecho se diseñó para que lo pueden hacer de dos maneras: en la actualidad los dos momentos en donde se puede hacer el traspaso de responsabilidad son:

- Durante *DoSecondStage():* De hecho la segunda etapa del procesamiento está dedicada a que si la transacción no puede resolverse en la clase asignada, se traspase la responsabilidad a la siguiente, que puede estar dentro de la solución o en un nodo remoto. Por ese motivo, los delegados que son llamados en *DoSecondStage()* son todos los relacionados con las habilidades definidas en la interface *ITransactionForwardable.* Para que una clase pueda comunicarse en esta etapa, se dispuso de la propiedad *_ForwardHandler* (ver diagrama de clases siguiente).

- Durante *DoThirdStage()*, luego de haber contestado al cliente: De esta forma, se puede *enganchar* una transacción anexa de mantenimiento, luego de haber respondido una respuesta al cliente que inició el requerimiento. La transacción original solo debe llamar al delegado *DoMaintenance()* que debería estar oportunamente conectado con *DoTransaction()* de la clase siguiente. Si a su vez, esta última requiere otro mantenimiento, se engancha una tercera transacción/manejador/clase en *DoMaintenance()* de la segunda, y así continuamente. No se debería enganchar nunca una transacción con sí misma, de modo de evitar *loops* de procesamiento. Para que una clase pueda comunicarse en esta etapa, se dispuso de la propiedad *_MaintenanceHandler* (ver diagrama de clases siguiente).

Sin embargo, la redefinición de las secuencias y la implementación de los métodos son elección propia del desarrollador. Se puede aprovechar del patrón dado que todas las transacciones heredan de la misma clase, por lo que tienen conocimientos de los delegados internos de sí mismas, y además porque pueden heredar

habilidades convenientemente para poder distribuir las responsabilidades de procesamiento de la forma que se desee.

En la Figura 45 se muestra las dos propiedades (*_ForwardHandler* y *_MaintenanceHandler*) que son del tipo *AbstractTransactionHandler*, de modo de poder generar dos cadenas de responsabilidad, una para el procesamiento de la transacción primaria y otra para el procesamiento de manutención.

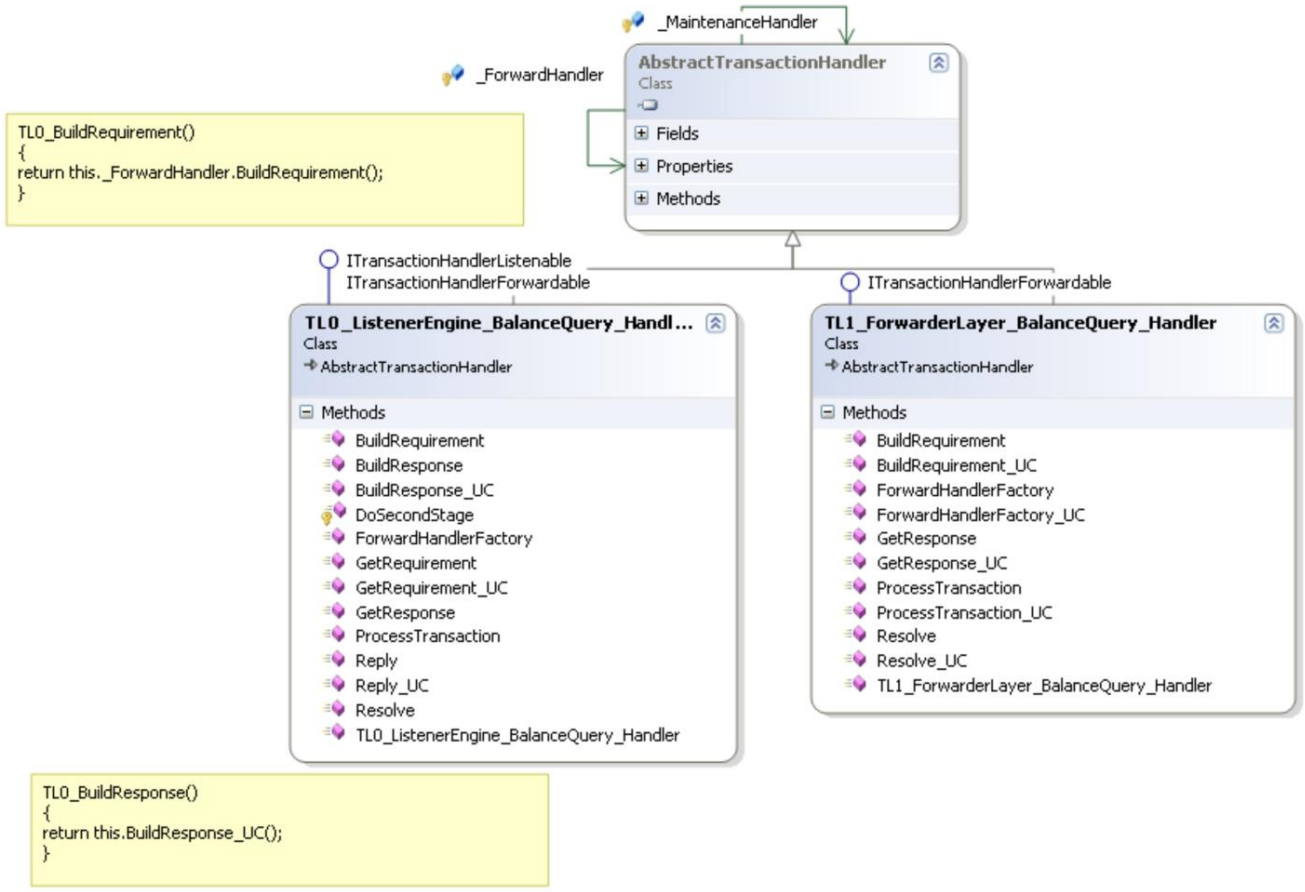

Figura 45 - Diagrama de clases de una transacción base, mostrando el patrón Chain of Responsibility

4.6.5 Factory Method

El patrón *Factory Method* es una de las formas de crear objetos dejando a las subclases decidir exactamente qué clase debe instanciarse. Las clases que se instanciarán deben implementar una interface o heredar de una clase padre, y es el método *fábrica* quien las instancia de forma apropiada, basándose en información contextual o extraída del estado de un objeto. [10]

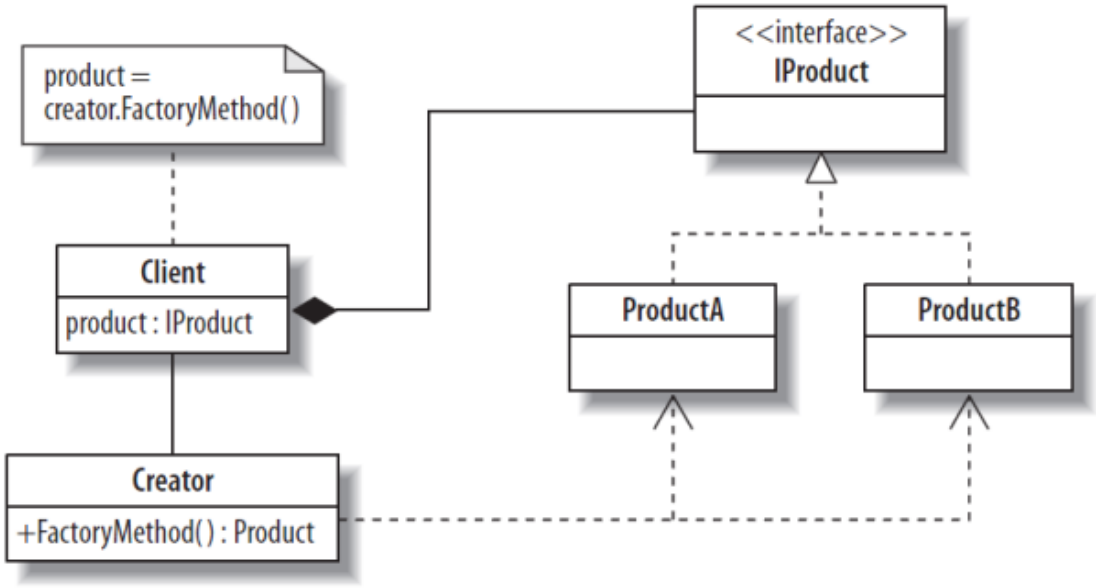

Figura 46 - Diagrama de clases del patrón Factory Method [10]

Este patrón fue de utilidad para poder resolver la instanciación dinámica de objetos dependiente de datos de contexto, principalmente cuando se trabaja con secuencias de trabajo predefinidas. Los métodos *fábrica* se aprovecharon dentro de esta propuesta principalmente para los *motores de entrada* y en las transacciones, pero también en algunos *motores de salida*. A continuación se enumeran estos casos:

- Dentro de los *motores de entrada*, la clase *AbstractTransactionInputEngine* define dos métodos *fábrica: ParserFactory*() y *TransactionHandlerFactory*(). Debido a que la secuencia por defecto de escucha de un *motor de entrada* es fija (ver secuencias en sección *Motores Transaccionales*), es necesario que los objetos dentro del motor puedan ser instanciados correctamente. Es decir que el *Parser* y el *TransactionHandler* sean del tipo deseados para la transacción que el motor está recibiendo. Cada *subclase abstracta* de *AbstractTransactionInputEngine* definirá su propia secuencia de escucha, pero nunca podrá personalizarse para que genere un *parser* específico o una transacción específica. Solo podrán hacer eso las clases concretas de cada motor, que conocerán qué protocolo se usará, y qué transacciones atenderá una solución en especial. Dado entonces un sistema específico, *ParserFactory*() devolverá un objeto del tipo *AbstractTransactionParser,* que dentro del método se ha inicializado con el protocolo a escuchar. Así mismo, en *TransactionHandlerFactory*() se pasa por parámetro el identificador de la transacción entrante, obtenido por el *parser* de la lectura del socket del cliente. *TransactionHandlerFactory*() devuelve un objeto *AbstractTransactionHandler,* y que se ha inicializado según la clase que atenderá la transacción.

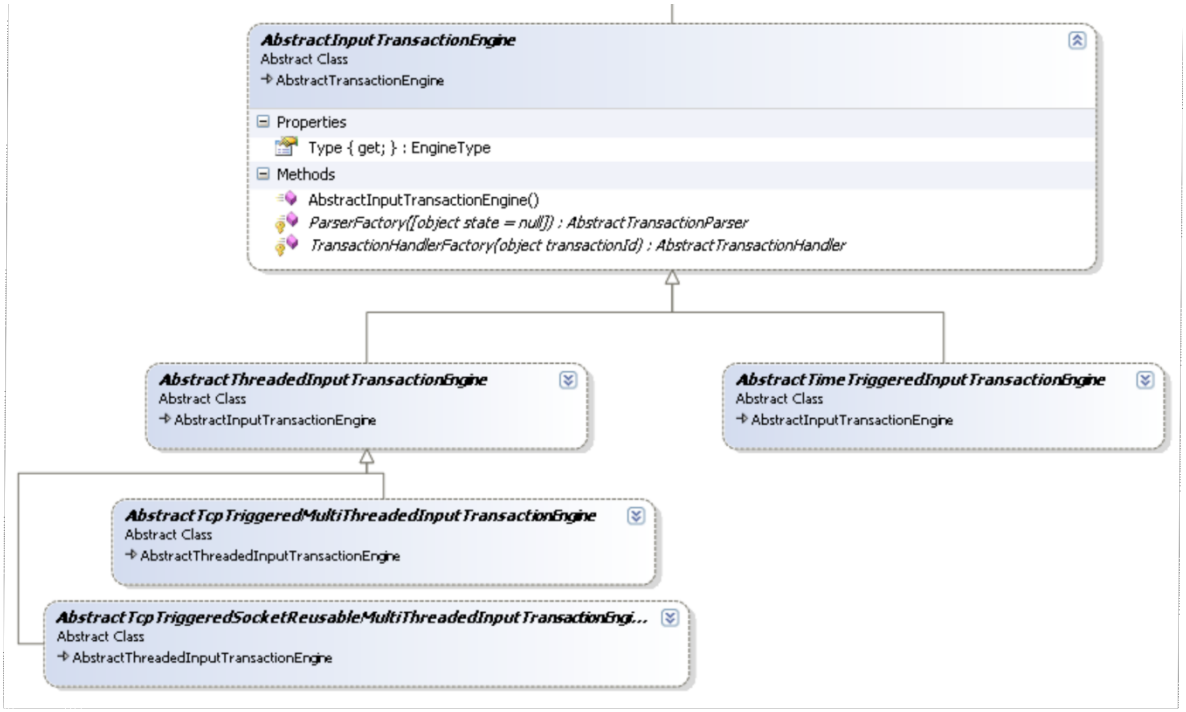

Figura 47 - Diagrama de clases de los motores de entrada, resaltando los métodos *fábrica* que se heredan.

- En las transacciones la secuencia por defecto también es fija. Como se explicó en *Chain of Responsibility*, las transacciones pueden generar dos cadenas conceptuales de traspaso de responsabilidad: una para el reenvío (*forwarding*) y otra para la manutención. En ambos casos, las clases que atenderán el traspaso deberán ser instanciadas en momentos específicos de la secuencia de procesamiento. Dicha secuencia está plasmada en los métodos virtuales *DoFirstStage(), DoSecondStage()* y *DoThirdStage()*, y que por sus naturalezas abstractas desconocen en tiempo de compilación cuál clase será eventualmente la siguiente en la cadena de *forwarding*, y cuál sería la siguiente en la de manutención. Para resolver esto, la secuencia consta con dos métodos *fábrica* que decidirán en tiempo de ejecución si hay clases que puedan cargarse en *_ForwardHandler*, o en *_MaintenanceHandler*; estos métodos son *ForwardTransactionHandlerFactory()* y *MaintenanceHandlerFactory()*. En función de valores contextuales, los métodos decidirán si hay alguna clase que deba ser instanciada para cualquiera de los dos fines, y si la encuentra la instanciará, devolviéndola como un objeto *AbstractTransacionHandler*.

- En el *motor de salida directo*, es necesario crear un objeto de conexión con el nodo externo por cada transacción que ejecuta su secuencia de trabajo. Esto es así ya que el objetivo de este tipo de motor es que cada hilo de ejecución genere su propio objeto de conexión, que permanecerá con *vida* mientras que la etapa de *forwarding* de la transacción *viva*. Dentro de la secuencia predefinida para este tipo de

motor, uno de los pasos es la llamada a *OutputHandlerFactory()* (*ver sección Motor de Salida Directo*). Este método devuelve un objeto que debe heredar la interfaz *IDisposable,* nativa de .NET. De este modo, el método puede instanciar clases que hereden del tipo *Stream*, una clase de .NET que es base para lectoescritura y transmisión de datos por varios medios, y que cumple con el contrato dictado por *IDisposable*. A partir de este tipo, desde *OutputHandlerFactory()* se pueden devolver objetos de tipos como *TcpClient*, *NamedPipeServerStream,* y otros que permitan comunicación entre objetos o interprocesos. La importancia de devolver *IDisposable*, en vez de *Stream*, es asegurar al motor que al finalizar la ejecución de su secuencia, el mismo pueda llamar indiferentemente al método *Dispose()*, eliminando de memoria el espacio asignado para el *handler*.

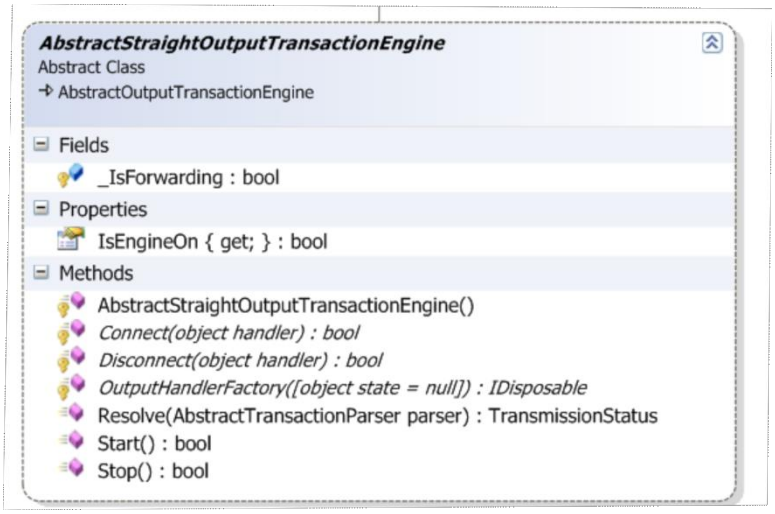

Figura 48 - Clase base de un motor directo de entrada

4.6.6 Observer

El patrón *Observer* define una relación entre objetos observadores y observados, tal que cuando un objeto observado cambia su estado, los observadores son notificados de dicho cambio. Generalmente, el patrón aplica a un conjunto de observadores que desean ser notificados de los cambios de un solo objeto observado, identificable por publicar cualquier cambio en su estado. [10]

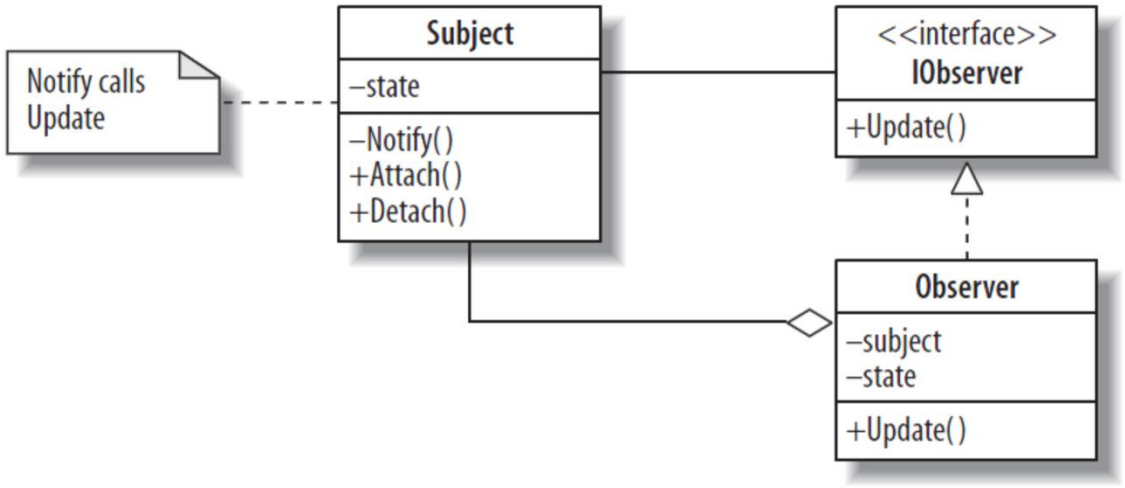

Figura 49 Diagrama de clases involucradas en el patrón Observer

Este patrón fue de utilidad para el *motor de salida* de tipo *mono-punto* o *embudo*, representado por la clase *AbstractFunneledOutputTransactionEngine*. La principal peculiaridad de este motor con respecto al de *Salida Directo*, es que una única conexión debe ser mantenida, y el motor debe encolar, recuperar y emparejar los requerimientos y las respuestas solicitadas desde múltiples transacciones que llegan simultáneamente al mismo (para mayor detalles ver sección *Motor de Salida Mono-Punto*).

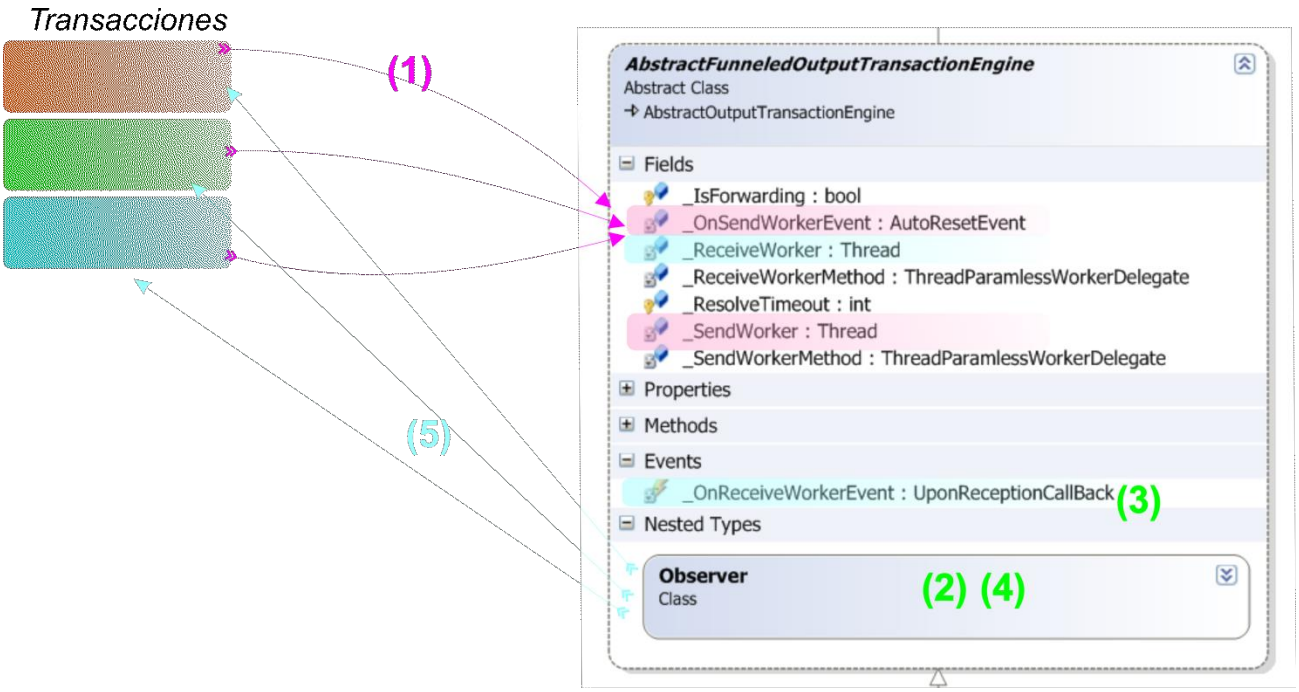

Figura 50 - Mecanismo de implementación del patrón Observer

El motor contempla la posibilidad de recibir requerimientos de múltiples hilos de ejecución, y cada uno llamará a *Resolve()* (paso 1 de la Figura 50). De esta forma, y como se comentó en la sección que describe al motor, *Resolve()* es el único método proveído por el motor que se ejecuta tantas veces de forma simultánea, como hilos lo hayan llamado. Se recuerda al lector que este define dos secuencias de pasos para procesar las transacciones, una será ejecutada por el hilo de la transacción propiamente dicho (esos pasos se definen en *Resolve*()), y la otra secuencia será ejecutada en los *threads de envío y recepción*.

El desafío encontrado reside justo en esa característica: el motor debía encolar mensajes provenientes de múltiples hilos que llaman a *Resolve()*, pero a su vez debía redistribuir las eventuales respuestas entre los hilos. Para solucionar esto se utilizó el patrón *Observer*: usando el principio de subscripción a eventos, los hilos que estuvieran interesados en *recibir noticias* de posibles respuestas debían generar un objeto *observador* que, dentro de la secuencia definida en *Resolve()*, se subscribirían (paso 2 de la Figura 50) al evento de recepción de mensajes. Por su parte, el motor avisa mediante este evento cada vez que llega un mensaje (paso 3 de la Figura 50). Los *observadores* son señalados (paso 4 de la Figura 50) y cada objeto *observador* despierta al hilo correspondiente. Es responsabilidad de cada hilo recolectar el mensaje de respuesta disponible desde la fuente de persistencia, y verificar si le pertenece a él (paso 5 de la Figura 50).

4.7 Pendientes y Desafíos

Aún queda mucho por hacer. Se recuerda que esto es una propuesta de marco de trabajo transaccional con tres años de antigüedad, cuya visión es la concentrar y capitalizar las experiencias y conocimientos colectivos de los desarrolladores y de las soluciones preexistentes analizadas. Sin embargo, a medida que las soluciones tecnológicas avanzan, nuevas piezas de código podrán ir reemplazando a las clases que se describieron en este capítulo. Por ejemplo, este *framework* fue diseñado para la versión .NET 2.0; en versiones posteriores se puede hacer uso de LINQ o de objetos dinámicos para reducir colecciones y diccionarios estáticos.

A continuación se describen los puntos que seguramente serán extendidos y mejorados a futuro, a medida que se obtenga *feedback* del uso de este *framework*, y las tecnologías subyacentes evolucionen:

- **Debe extenderse el marco de trabajo a otras plataformas y/o lenguajes:** Por ejemplo, el próximo paso de migración debería ser el soporte en Java.
- **Mejorar el grado de usabilidad del potencial del lenguaje y/o de la plataforma:** Por ejemplo para .NET, se deberían reemplazar los algoritmos de filtrado y búsqueda con sentencias LINQ genéricas. Así mismo se pueden aprovechar los objetos dinámicos como el *ExpandoObject,* para generar los contextos transaccionales (en *AbstractTransactionContext*).

- No se logró enmarcar el concepto de conexiones *WebService* en un *motor de salida* concreto. Se debe analizar con detalle la jerarquía de los motores para poder incluirlos, y facilitar la integración con estos para los futuros usuarios del *framework.*
- Se debe continuar expandiendo la jerarquía de *motores de entrada y salida*, para incrementar la riqueza del *framework* en componentes reutilizables.
- El *framework* debería extenderse no solo a servidores transaccionales concurrentes, sino también a terminales de punto de venta (POS).

4.8 Resumen del Capítulo

En este capítulo se trataron los siguientes temas:

- Se describió el trabajo de observación y análisis sobre soluciones preexistentes, aclarando los motivos para realizar esta propuesta, y mostrando las ventajas y desventajas de la soluciones *pre-framework*.
- Se introdujo el concepto de sintetizar una transacción mediante una secuencia única y personalizable: Para tal fin se mostraron los pasos que deberían realizarse según la propuesta para procesar cualquier transacción, dado un servidor transaccional.
- Se introdujeron los conceptos transaccionales encontrados como factores comunes durante el análisis y la observación de soluciones preexistentes, como así también de la síntesis de la experiencia del equipo de desarrollo.
- Se describieron los motores más usados en detalle.
- Se describieron los patrones usados para soportar algunos de los desafíos que se presentaron durante la implementación de los conceptos transaccionales.
- Se marcaron algunos de los caminos evolutivos para seguir en el desarrollo incremental de este marco de trabajo.

5 Implementación de una Propuesta MDD Basada en el Framework TransactionKernel

5.1 Introducción

En este capítulo se describe la propuesta de formalización del *framework* que se presentó con anterioridad. Este *framework* se denomina *TransactionKernel,* y la formalización se realizará a través de una metodología MDD, en particular usando DSM *(Domain Specific Modeling)*. Como se adelantó en el primer capítulo, para este fin se desarrollará un lenguaje específico de dominio que integre algunos de los conceptos analizados durante la presentación del *framework.* En el contexto de *Ingeniería de Software*, el enfoque de *desarrollo dirigido por modelos* (MDD) está surgiendo como un cambio de paradigma, favoreciendo las prácticas de desarrollo orientado a modelos, respecto de las prácticas de desarrollo que se centralizan en el código (*Code-Centric Software Development*). Este enfoque promueve la sistematización y automatización de la construcción de artefactos de código. De esta forma, los modelos pasan a tener un papel preponderante, siendo considerados estos modelos como entidades de igual o mayor jerarquía que el código que describen. Es en los modelos donde el conocimiento y la experiencia de los desarrolladores son capitalizados, en forma de *transformaciones de modelos.* La característica esencial de MDD es que los productos generados sean modelos. [11]

5.2 Propuesta

La propuesta de formalización del *framework TransactionKernel* constará de dos partes: La primera es el *lenguaje de dominio específico* propiamente dicho, llamado *PointPay.Framework.Language*. Allí se definen un conjunto de conceptos que serán *elementos de dominio* y que tendrán correspondencias con las clases definidas por *TransactionKernel*. Además se generaron propiedades de dominio (*domain properties*) por cada elemento del modelo para poder caracterizar el comportamiento de cada uno. Así también se expresarán las relaciones posibles entre los elementos, y el *formato* (*shape*) con los que serán representados gráficamente cada vez que se instancien dentro de un modelo (forma, color, ícono, etc.).

La segunda parte de la propuesta es una *herramienta de transformación automática de código a partir de los modelos dibujados*, llamada *TransactionKernelTransformationTool.* Esta herramienta será la encargada de entender cómo está definido un elemento en una instancia particular del modelo (es decir, cómo están cargadas las propiedades de dominio y cómo está relacionado ese elemento con los demás para un modelo dado). A partir de ese análisis deberán generarse uno o varios archivos de código funcional, que reflejen el funcionamiento diagramado en el modelo. Además del código funcional, la herramienta podría eventualmente generar otros modelos de diferentes niveles de abstracción, como diagramas de clases UML, diagramas de secuencia, etc. Al

momento de escribir este trabajo, la herramienta de transformación solamente contempla la conversión de una instancia de modelo del DSL a archivos de código funcional.

Por último se aclara al lector que en este capítulo no se describirá en detalle la herramienta de transformación a código, dada la dificultad de plantear las reglas de transformación particulares de este dominio en un documento de texto. Sin embargo se detallarán los elementos que componen el lenguaje de dominio y que representan de alguna forma los conceptos revisados en el *framework TransactionKernel.*

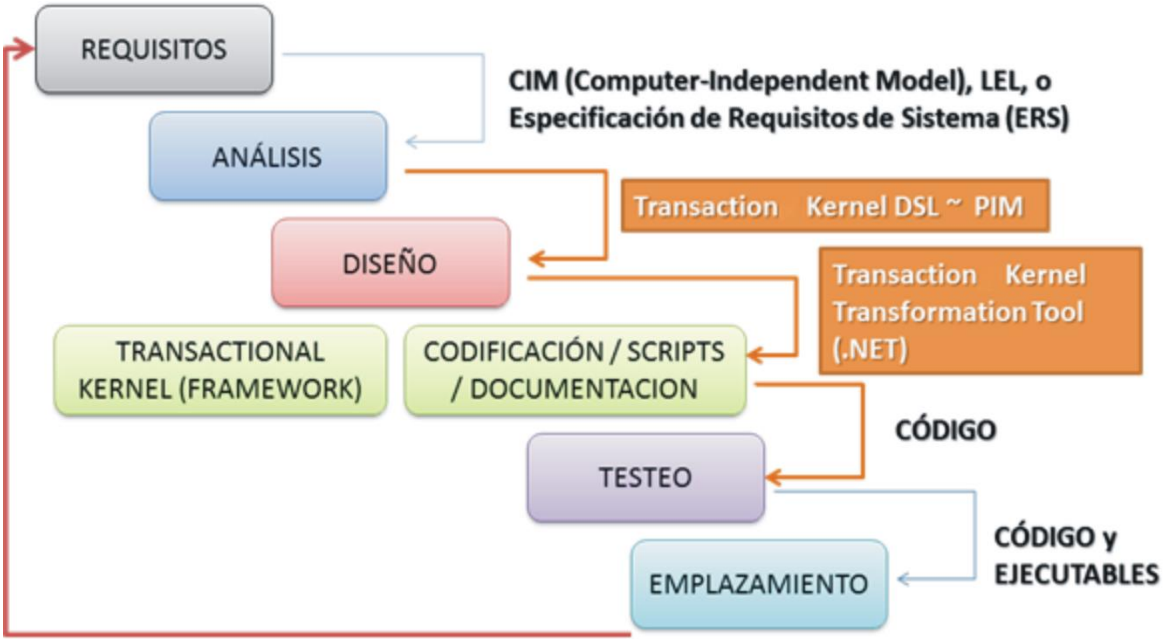

Figura 51 - Ciclo de vida de la propuesta DSM para *TransactionKernel*

Estas dos partes de la propuesta se conjugan con las etapas del ciclo de vida de desarrollo de *software,* analizadas en capítulos anteriores. Para la etapa de *Requisitos* por ejemplo, a través de un documento llamado *especificación de requisitos del sistema* (ERS), o un LEL[17], o un CIM (*Computer-Independent Model*) se pueden cuantificar un conjunto de requisitos funcionales, no funcionales y de expectativas de los *stakeholders*, o del *Product Owner* de un sistema. De esta forma se logra hacer un análisis de lo que debería hacer un sistema, y también de lo que no debería hacer, o de lo que no va a contemplarse para que pueda ser informado a las partes interesadas del proyecto con la mayor antelación posible. Para llegar a la etapa de diseño, se propone usar *el lenguaje de dominio específico TransactionKernel*, de modo de traducir todos los requisitos elicitados y

[17] Un LEL (Léxico Extendido del Lenguaje) es una técnica que procura describir los símbolos de un lenguaje. La idea central del LEL es la existencia de lenguajes de aplicación.

analizados, a una instancia de modelo que pueda luego transformarse en otros modelos de menor abstracción, o directamente en código funcional. Por último, para pasar de la etapa de *Diseño* a la de *Codificación,* se propone el uso de la herramienta de transformación automática de código *TransactionKernelTransformationTool.* El código generado al momento de escribir este trabajo es solamente en lenguaje C#, y en *scripts* SQL para la capa de acceso a datos. Las clases generadas como producto de la transformación serán soportadas por la herencia de las clases del *framework TransactionKernel*. Se recuerda al lector que esa capa contiene los conceptos comunes analizados y obtenidos de la experiencia de los desarrolladores y de las soluciones preexistentes, tal cual se lo presentó durante el capítulo anterior. Así mismo se generarán clases de *test unitario*, con las transacciones definidas en el modelo. De este modo se puede utilizar una estrategia TDD para el desarrollo transacción por transacción, o como estímulos válidos para el sistema transaccional (así se puede probar al sistema como una *caja negra*). Por último, la etapa de *Testing* se puede llevar a cabo a través de *NUnit* o un programa similar que permita comprender y ejecutar los casos de prueba definidos en la transformación. Así mismo la etapa de *Emplazamiento* se puede llevar a cabo con un IDE, en este caso *Visual Studio 2012.*

5.3 Lenguaje de Dominio Específico Propuesto

5.3.1 Consideraciones Iniciales

El lenguaje específico de dominio que se describe durante este capítulo tendrá en cuenta las siguientes consideraciones.

- **Esquematización de los modelos por capas de procesamiento:** La propuesta de modelado se realizó en base a capas transaccionales, es decir, que se puede leer cualquiera de los modelos producidos por este DSL de izquierda a derecha manteniendo una analogía casi temporal de cómo va procesándose la transacción entrante hasta que se redirige al próximo nodo de la cadena de procesamiento. Entonces, cualquier elemento del modelo debe estar emplazado dentro de una capa transaccional. El diagrama permite poner múltiples capas transaccionales por modelo, identificándolas de dos modos: a través de un nombre descriptivo, y a través de un nivel de capa numérico. Por ejemplo, para un sistema transaccional que tiene que recibir transacciones y redirigirlas a otro nodo, se podría modelizar el mismo a través de dos capas. Una capa (nivel de capa '0', nombre "*Capa de Entrada*") que contenga al *motor de entrada* y a las transacciones con las que trabajará, y otra segunda capa (nivel de capa '1', nombre "*Capa de Salida*") que contendrá al *motor de salida* y a las transacciones que se pueden eventualmente redirigir. Este concepto se verá con mayor detalle durante este capítulo, y durante el capítulo de *Evaluación del* DSL.
- **Dualidad en la codificación automática y manual:** Si bien uno de los focos de MDD es poder mitigar las dificultades de construir un programa a través del desarrollo por modelos y de la transformación de éstos a código funcional, no es viable la construcción de programas de forma totalmente

automatizada. Es decir, en algún momento hay que especificar los requisitos que son necesarios para cumplir con la totalidad de un alcance dado, y para la automatización total debería generarse un lenguaje que cumpla con cualquier alcance posible. Además la jerarquía consta solo de dos niveles de abstracción: una instancia de modelo del DSL, y el código funcional. Otra razón es que no se puede prever de antemano el nivel de personalización que el usuario final necesitará configurar para un sistema transaccional que surja como producto de este DSL. Entonces, es mandatorio dejarle la libertad al usuario de poder personalizar algunos aspectos del código que se generó como producto de la transformación automática.

Para poder manejar entonces esta dualidad, se implementaron transformaciones de modelos a *clases parciales*. En pocas palabras, las *clases parciales* hacen posible que la definición de una clase pueda dividirse en múltiples archivos físicos. A nivel de lógica, las clases parciales no tienen ninguna diferencia para el compilador, comparadas con aquellas cuya definición esté en un único archivo. Durante la fase de compilación, simplemente se agrupan todas las definiciones parciales y se compila el código como si hubiese estado en un único archivo físico. La ventaja de usar esta parcialidad es que en uno de los archivos que contiene a la definición de la clase se autogenerará cada vez que se guarde el modelo del DSL (a través de la herramienta de transformación a código funcional). Ese archivo será entonces borrado y regenerado 'n' veces, por lo que cualquier agregado generado de forma manual por el usuario será eliminado cuando se vuelva a guardar el modelo. Entonces se utiliza el segundo archivo de definición de clase parcial, que solo se genera de forma automática si el mismo no existe, es decir la primera vez. Luego, allí se pueden definir de forma segura todos los agregados que el usuario desee implementar en la clase.

- **Los modelos son archivos**: Cada modelo que sea generado por el DSL se almacenará en un archivo, con extensión *.ppaydsl*

- **Las entradas y las salidas del sistema son transacciones**: El sistema de procesamiento que se intentará desarrollar por medio de este DSL se considerará como una *caja negra*, es decir, sus entradas y salidas son las transacciones que podrá recibir y responder desde uno de los lados, y las que podrá redirigir y recibir desde el otro lado del sistema. En otras palabras, suponiendo un *switch* que conecta a 100 EFT-POS con un autorizador de un producto dado, las entradas y salidas por un lado son el envío de tramas por parte de los POS y sus respectivas respuestas (generadas desde el *switch*). Del otro lado, las entradas y salidas son las tramas que se generan desde el *switch* hacia el autorizador final (nodo externo), y las tramas recibidas como respuestas desde el autorizador en cuestión.

- **Todos los sistemas de procesamiento generados se identifican por un número de instancia**: Es decir, se asignará por el diseñador un valor entero positivo al diseño que se realiza. Todos los valores de configuración propios estarán referenciados por este valor, evitando conflictos de datos o colisiones entre sistemas.

- **Se utilizará log4net para fines de logging en los códigos generados de forma automática**: *log4net* es una herramienta que brinda soporte para realizar tareas de bitácora, con varios tipos de salida (*debug, error, advertencia, información, y error fatal*). En caso de algún inconveniente con el sistema de procesamiento, se utilizan las líneas de log generadas para ver dónde puede estar el problema. Se seleccionó *log4net* como herramienta de *logging* dado el grado de adopción de la librería por el equipo de desarrollo que llevó adelante la implementación. De hecho, *log4net* es actualmente una de las herramientas de facto para *logging* a nivel global.

- **Métodos *GetValue* y *SequenceFactory***: Estos son dos métodos que se generarán de forma automática durante las transformaciones a código de los sistemas diseñados por el DSL propuesto. A través de estas, se proponen dos funcionalidades que son de utilidad general en cualquier sistema de procesamiento:
    - ***GetValue()*** permite obtener un valor de una tabla de configuración, dentro de una base de datos, un archivo de configuración, un XML, o cualquier otro medio de persistencia. El método permite el pasaje por parámetro de una *sección* (del tipo *string*), una *clave* opcional dentro de la sección (*string*), y el método devolverá todos los valores encontrados para esa *clave* y esa *sección*. Cabe acotar que es imprescindible pasar como parámetro obligatorio el *número de instancia* asignado al sistema, para poder trabajar solo con el conjunto de datos asignado al sistema en curso.
    - ***SequenceFactory()*** permite la generación de un número secuencial único por instancia, con un techo máximo, que al ser alcanzado reiniciará la cuenta a 1. Este número secuencial es comúnmente usado en los sistemas de procesamiento para enumerar cada transacción saliente del mismo hacia otro nodo externo. De hecho muchos protocolos transaccionales como el *ISO8583* reservan un campo (campo 11 *STAN*, del inglés *System Trace Audit Number*) para tal fin. Por esta razón, se generará de forma automática un método que trabajará contra algún mecanismo de persistencia, con el fin de generar número secuenciales unívocos. La unicidad del número debe darse en un lapso de tiempo que generalmente es de un día como máximo, es por eso que el número se considera unívoco a pesar que cuando llega al techo máximo para ese contador, el mismo reinicie la cuenta a 1.

- **Parametrización de los sistemas:** La parametrización de los sistemas generados a través del DSL se configura en dos etapas: una configuración **mínima** (obligatoria) y una configuración **extendida** (obligatoria y variable según cada sistema).
    - **Configuración mínima**: Se implementa como un archivo *app.config*, que contendrá los valores mínimos para el funcionamiento del sistema, ordenados como elementos XML. Dentro de estos valores mínimos se destacan el *ID de Instancia* y el *Connection String***,** que servirán para poder configurar las llamadas durante la etapa de configuración extendida (explicada en el párrafo siguiente). Allí mismo, en el *app.config* se guardan también los nombres de los *stored procedures* usados en el sistema y la configuración de los campos de los *parsers* usados. Estos últimos son vitales para poder usar los *analizadores de protocolo* (*parsers*) ya que las estructuras de requerimiento y respuesta se configuran dinámicamente con los valores definidos dentro del *app.config*.
    - **Configuración extendida:** Se implementa de forma indexada a través de un *número de instancia*, una *sección* y una *clave*. Las configuraciones de los sistemas se deberán guardar en un archivo, en una base de datos o en algún otro medio persistente. En esta propuesta, está implementado el mecanismo de persistencia vía *SQL Server:* es decir que existirá una tabla *Configuracion* que tenga datos almacenados de forma indexada por un *identificador de instancia, secciones y claves* dentro de cada *sección*. Oportunamente el método *GetValue()*, deberá estar implementado para recuperar uno o más valores dentro de esta tabla, según se especifique solo la *sección*, o una *clave* dentro de una *sección*. Será obligatorio indicar a qué instancia se refiere el sistema, de modo de recuperar únicamente los datos que le pertenecen.

- **Elementos del metamodelo en *DSL Tools*:** Para el desarrollo del lenguaje específico de dominio, se utilizó la herramienta provista por *Microsoft* dentro de su IDE *Visual Studio*. Los elementos usados durante la definición del metamodelo (el lenguaje) son:

    - **Domain Model:** Un *Domain Model* es un tipo de *lienzo base* donde se desarrollará el metamodelo que representará a un *lenguaje especifico de dominio*. Existirá solo una única instancia del *Domain Model*, donde se contendrán todos los elementos del DSL bajo desarrollo. Dentro de cada *Domain Model*, existirán dos divisiones: Una contendrá la esquematización de las relaciones entre *clases de dominio (Domain Class)*, y la segunda contendrá la definición de la representación gráfica de cada *Domain Class* dentro de una instancia del metamodelo.
    - **Domain Class:** Las *Domain Classes* (o *Clases de Dominio*) son los elementos principales de un *Modelo de Dominio* (*Domain Model*). Estas clases pueden ser usadas para representar cualquier tipo de entidad que pertenezca al lenguaje de dominio específico en cuestión. Por

ejemplo, se pueden representar estados, conceptos del negocio, casos de uso, actividades, y/o cualquier otra noción que sea de importancia para el lenguaje bajo desarrollo.

- **Domain Relationship:** Se utilizan las relaciones de dominio (*Domain Relationships*) para establecer las relaciones entre las *Domain Classes,* dentro del ámbito de un lenguaje específico de dominio. Hay dos tipos de relaciones, las *Empotradas* (*Embedded Relationships*) y las de *Referencia* (*Reference Relationships*) [12].
  - **Embedded Domain Relationship**: En una *relación empotrada*, los elementos de la clase destino son incluidos en los elementos de la clase de dominio origen. Es decir que la clase de dominio origen contiene de alguna forma a las clases de dominio destino que estén relacionadas por una relación empotrada. Cada elemento del metamodelo (es decir, una instancia del lenguaje específico de dominio), debe estar relacionado con otro por exactamente una *Embedded Relationship,* de lo contrario no podrá ser parte del diagrama cuando se quiera guardar (*save*) el mismo. La única excepción es válida para el elemento raíz (*root*) del modelo, que en nuestro caso es el *Domain Model*. Por defecto, cuando se elimina un elemento *padre*, el elemento empotrado también se elimina. Las relaciones empotradas se representan como líneas sólidas, dentro del diagrama del DSL en cuestión. Las conexiones empotradas terminan formando un árbol de *Domain Classes,* en una instancia de modelo del DSL. [12]
  - **Reference Domain Relationship:** En una *relación por referencia*, los elementos de una clase de dominio origen hacen referencia a elementos en una clase de dominio destino. Este tipo de relación es vital para conectar las distintas entidades del *Domain Model* entre ellas. Al usarlas, se crean propiedades por cada uno de los roles de la relación (en cada relación existen los roles *origen* y *destino*), que facilitan la navegación programática entre clases de dominio. Estas propiedades permiten una eventual transformación de código automática a través de una herramienta complementaria para tal fin. Las relaciones por referencia se representan como líneas de guiones, dentro del diagrama del DSL en cuestión.
    [12]
- **Herencia:** Las relaciones por herencia son similares a las habituales en lenguajes orientados a objetos. Dentro de un modelo de dominio, se pueden relacionar dos o más *Domain Classes***,** como una clase de dominio base y al menos otra clase de dominio heredada. Las clases de dominio heredadas obtienen de la clase base todas sus propiedades, al igual que en un paradigma orientado a objetos. Las relaciones por herencia, a diferencia de las *relaciones empotradas* y las *relaciones por referencia* no tienen nombres que las identifiquen, no tienen multiplicidades, y no tienen roles que generen propiedades de navegación.

Por otro lado, las representaciones usadas para los elementos del lenguaje son:

- **Swimlane:** Es un tipo de representación que genera un *carril* para depositar otros elementos de dominio dentro. Pueden tener disposición horizontal o disposición vertical.

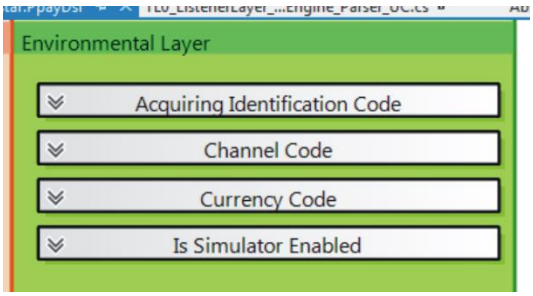

Figura 52 - Ejemplo de *Swimlane*

- **Compartment Shape:** Es un tipo de representación en forma de cajón que permite mostrar de forma dinámica los nombres de otros elementos del dominio que estén referenciados al elemento que el *Compartment Shape* representa.

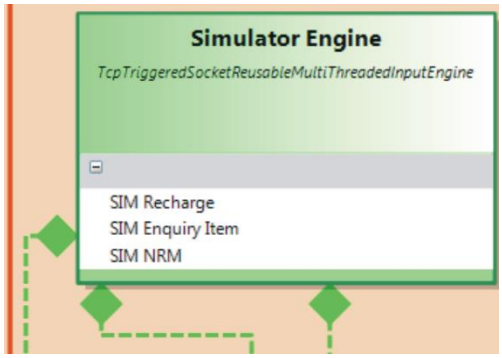

Figura 53 - Ejemplo de un *Compartment Shape*

- **Connector:** Los conectores pueden ser usados para representar las distintas relaciones entre clases de dominio (ya sean empotradas, o por referencia).

Figura 54 - Ejemplo de *Connector*

- **Geometry Shape:** Son representaciones geométricas simples que pueden decorarse con descripciones de la instancia del elemento de dominio que están representando en un momento dado.

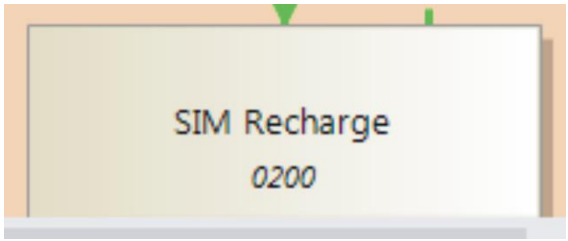

Figura 55 - Ejemplo de *Geometry Shape*

- **Image Shape:** Es posible utilizar imágenes para especificar la representación de determinadas clases de dominio.

Figura 56 - Ejemplo de *Image Shape*

5.3.2 Modelo Base

El modelo base del lenguaje se implementa como una *clase de dominio*. Esta clase de dominio definirá las propiedades del modelo que serán de interés para las transformaciones a código que se proponen más adelante

en este trabajo. La *clase de dominio* se llama *TransactionModel* y es el elemento raíz del modelo, por lo que no es destino de ninguna relación empotrada.

5.3.2.1 Propiedades y Relaciones

Figura 57 – Diagrama de relaciones empotradas entre el elemento raíz TransactionModel, y los elementos Transaction Layer y Transaction Environment Layer

La *clase de dominio TransactionModel* (o TM) se relaciona de forma empotrada con otras dos *clases de dominio*, llamadas *TransactionLayer* (TL), y *TransactionEnviromentLayer* (TEL). Ningún otro objeto del metamodelo podrá instanciarse dentro de *TransactionModel*, si primero no está empotrado en alguna de las dos capas.

La herramienta de transformación de código obtendrá del modelo una serie de propiedades para construir una clase *Facade* funcional. Esa clase contendrá todos los métodos definidos y explicados oportunamente cuando se desarrolló la clase *AbstractTransactionFacade* en los capítulos anteriores. Así mismo, la clase *Facade* generada será diseñada para cumplir con una única instancia del sistema (patrón *Facade Singleton*), por lo que la transformación automática de modelo a código implementará las propiedades y métodos estáticos para cumplir con esta premisa, entre otras personalizaciones basadas en el alcance particular del proyecto. De esa forma se genera una clase que es *Singleton*, que hereda de *AbstractTransactionFacade*, y que contiene información específica de la solución bajo modelización.

Las propiedades principales de *TransactionModel* permiten modelar algunos de los parámetros para traducir de forma automática los datos del sistema bajo construcción. Estas propiedades son:

Propiedades de *TransactionModel*	**Objetivo**
Name: *string*	La propiedad *Name* es primordial en el DSL ya que permitirá generar los nombres de los archivos de código autogenerados

	durante el proceso de transformación, y también se usará para los nombres de las *clases, delegados, interfaces, propiedades, campos, etc*. Es un campo de carácter obligatorio, es decir, cuando se guarda el modelo en disco, se pre-validará que este campo no esté ni vacío ni *null.* De otro modo, no se podrían identificar ninguno de los objetos que se pretenden generar.
InstanceId: ***Int16***	Es otra propiedad principal que identifica la instancia del modelo bajo desarrollo. Es un valor numérico que debe asignarse unívocamente al sistema en curso. Este valor será utilizado para darle nombre al servicio que se instalará en el sistema operativo cuando esté productivo. Además se usa como parámetro principal del método *GetValue()* para recuperar el conjunto de parámetros que le corresponde al sistema. Es un campo de carácter obligatorio, es decir, cuando se guarda el modelo en disco, se pre-validará que este campo esté cargado con un número entero positivo.
Namespace: ***string***	Esta propiedad se utiliza, como su nombre indica, para definir el *namespace* de todas las clases generadas de forma automática. Es un parámetro obligatorio, que se pre-validará antes de guardar el archivo del modelo.
GetValueImplementationType: ***PersistableSourceType (enum)***	Esta propiedad sirve para definir cuál es el tipo de implementación que se utilizará para el método *GetValue()*. En función del valor de esta propiedad, durante la transformación automática de código se implementará un método para cumplimentar con la definición dada. Los valores aceptados al momento de este trabajo son: ➢ *NotTyped:* El método *GetValue()* se deja listo para ser implementado complemente por el usuario, es decir solo se transformará el prototipo de la función vacío, con una excepción del tipo *NotImplementedException* a modo de recordatorio funcional de la falta de implementación.

	➢ *SQLServerDatabase*: El método *GetValue()* se implementará para poder hacer el llamado a un *stored procedure* que realice la búsqueda en una tabla, dentro de una base de datos con motor *SQL Server*.
SequenceFactoryImplementationType: *PersistableSourceType (enum)*	Similar al caso anterior, esta propiedad define cuál es el tipo de implementación del método *SequenceFactory()*. Al momento de esta propuesta, los tipos de implementación son los mismos que para el método *GetValue()* (*NotTyped, SQLServerDatabase*).
SatelliteInstances: *string*	Si bien el DSL estuvo inicialmente pensado para que cada modelo que se generase terminara siendo identificado por un único ID de instancia, en la práctica se dieron varios casos donde la librería de clases generada debía ser utilizada en diferentes servicios, con diferentes configuraciones y parámetros. Es por eso que se generó este campo, para poder indicar los valores de *ID de instancia* que serán satélites del *ID de instancia principal*. Bajo este nuevo concepto, se pueden tener diferentes servicios, cada uno con la misma librería de clases pero con diferentes ID de instancia, y por ende con configuraciones separadas. Durante las transformaciones de código, este valor impactará en el archivo *app.config*

Tabla 13 – Propiedades principales de la clase de dominio *TransactionModel*

Por último, *TransactionModel* cuenta con otras cuatro propiedades para generar un *ConnectionString* por defecto, que permita el funcionamiento de la implementación de *GetValue*, y *SequenceFactory*, y cualquier otro mecanismo auto-transformado a código del modelo. Estas propiedades son:

- ***DatabaseServerInstance: string*.** Define la dirección y el nombre de la instancia del motor de base de datos a conectarse.
- ***DatabaseInitialCatalog: string.*** Define el nombre de la base de datos a usar.
- ***DatabaseUser: string.*** y ***DatabasePasword: string.*** Definen las credenciales de acceso a la DB.

5.3.3 Capa Transaccional

Tal cual se introdujo en las *Condiciones Iniciales* de este capítulo, la *capa transaccional* (*Transaction Layer*) o (TL) tiene como objetivo funcionar como un contenedor de elementos del DSL. Todos los elementos dentro de la TL pertenecen a una etapa dentro del sistema de procesamiento. La mayoría de los elementos del DSL deben pertenecer obligatoriamente a una TL, salvo por la excepción de algunos elementos que tienen que ser contenidos únicamente por la *capa transaccional de entorno* (TEL), como se verá más adelante en este capítulo. No hay límite máximo de cantidad de TL's, aunque al menos debe haber una (1) capa obligatoria. A su vez, para facilitar la organización de las capas, las mismas tienen un nivel, representado por un número entero positivo o igual a cero. Puede haber niveles numéricamente repetidos, ya que estas capas se transformarán en carpetas que sirven para organizar el código generado según las etapas de procesamiento. Los elementos de dominio TL y TEL son los únicos dos que pueden ser insertados sobre el *Transaction Model*. No hay ningún elemento del lenguaje que no pueda estar por fuera de alguna de las dos capas.

Para facilitar la comprensión de cómo usar las capas, se propone un ejemplo al lector:

a) En el caso inicial de armar un repetidor de tramas (solo devuelve un eco de lo que se lee), el DSL podría estar diagramado con una única capa transaccional llamada *"Capa de Escucha"* con nivel *"0"*. Como se explicará en los párrafos siguientes, se insertará un objeto "*Motor de Entrada*" y de ese motor se conectará una "*Transacción Eco*". Ambos elementos (*motor* y *transacción*) deberán ser contenidos por la capa transaccional. Una vez guardado el diagrama, se transformará el diagrama a código, generando las clases necesarias para que el usuario pueda personalizar el comportamiento de la transacción.
b) En el caso de expandir el repetidor a un reenviador de tramas (a otro nodo), el modelo anterior podría modificarse agregando una segunda capa, de nivel "1" por ejemplo, llamada "*Capa de Reenvío*". En esa capa se inserta un elemento "*Motor de Salida*" y una transacción "*Transacción Reenvío*" conectada de ese motor. Luego se conectan ambas transacciones (es decir, la de la capa '0' con la de la capa '1') con un *conector de reenvío* que se verá más adelante. Cabe acotar que si bien la transacción es desde la lógica una sola (es decir que habrá un *thread por cada transacción*), se representa en el modelo como muchas transacciones/fases distintas (una por cada TL) según el estado de procesamiento. Durante la transformación del modelo a código, se generarán dos carpetas (una por cada TL), con las clases necesarias y sus propiedades vinculadas según las conexiones definidas en dicho modelo. Luego el usuario es quien personalizará las clases de las transacciones / motores para poder cumplir con los requisitos dados para este "reenviador de tramas" ficticio.

En resumen, los elementos incluidos en las TL se transformarán en archivos de código, que a su vez se almacenarán en carpetas definidas para cada TL. Las carpetas tendrán la nomenclatura **TLx** donde **x** es el valor numérico de nivel asignado por el diseñador. A su vez los nombres de los archivos dentro de estas carpetas

estarán prefijados de la misma forma (en el ejemplo anterior los elementos de la capa “0” se prefijarán como **TL0_[Nombre de Archivo]** y los de la capa “1” como **TL1_[Nombre de Archivo]**).

5.3.3.1 Propiedades y Relaciones

Figura 58 Diagrama de relaciones empotradas del elemento *TransactionLayer*.

En la Figura 58 se pueden ver todas las relaciones empotradas de los elementos del DSL involucrados con el elemento *Transaction Layer* (TL). Las relaciones de dominio que se explican a continuación tienen definidas propiedades de navegación para cada clase de dominio. No se entrará en detalle en estas propiedades, sin embargo se explica a continuación el concepto de las relaciones definidas:

- Dentro de una TL puede haber CERO o MUCHAS *transacciones*, y esa *transacción* puede pertenecer a una única TL.
- De la misma forma, en una TL puede haber CERO o MUCHOS *motores de entrada*, y CERO o MUCHOS *motores de salida*. A su vez los motores pertenecen a una única TL.
- En una TL puede haber CERO o MUCHOS *OutputTransactionWebServices.*
- En una TL puede haber CERO o MUCHOS *TimeTriggers*
- En una TL puede haber CERO o MUCHOS *orígenes de datos* (*TransactionSQLServerDataSource*) y puede haber CERO O MUCHOS *soporte de orígenes de datos* (*TransactionDataSourceSupport*)

Propiedades de *TransactionLayer*	**Objetivo**
Name: *string*	Es el nombre de la TL, que es asignado a libertad del diseñador del sistema. Es aconsejable que el nombre sea descriptivo de la etapa que se está modelando.
Level: *Int16*	Es un número de la capa. Representa de forma numérica la etapa del proceso de la transacción, dentro de un autorizador, concentrador, etc. Por ejemplo, las capas iniciales (que reciben las transacciones, o las generan) deberían tener '0', luego las finales (las que reenvían o procesan internamente el pedido, el número debería ser '1'). En el caso de manejar varios procesos intermedios, podrían usarse otros números: por ejemplo '0' para la de recepción, '1', '2' y'3' para las intermedias, y '4' para la final.

Tabla 14 – Propiedades principales de la clase de dominio *TransactionLayer*

5.3.3.2 Formato

Figura 59 – Formato asignado a la clase de dominio *TransactionLayer*

Las capas transaccionales se decoran con una forma del tipo *SwimLane*, o *andarivel,* que permite el agregado de otros elementos del modelo dentro de la capa.

5.3.4 Motor Transaccional

Los motores transaccionales son los elementos del diagrama que permiten implementar de forma gráfica los artefactos de código para poder recibir y/o enviar transacciones hacia otros nodos o motores. Estos elementos de dominio se verán reflejados en transformaciones de código que permitirán implementar un *motor transaccional* como los analizados durante la revisión del *framework*. Es decir, la herramienta de transformación, usará este motor gráfico para construir un motor funcional basado en las clases de motores del *framework,* y a su vez personalizará algunos aspectos del motor, como el nombre, y algunas características más. Por ejemplo, qué tipos de transacciones entenderá, cuáles rechazará apenas sean recibidas, etc.

5.3.4.1 Propiedades y Relaciones

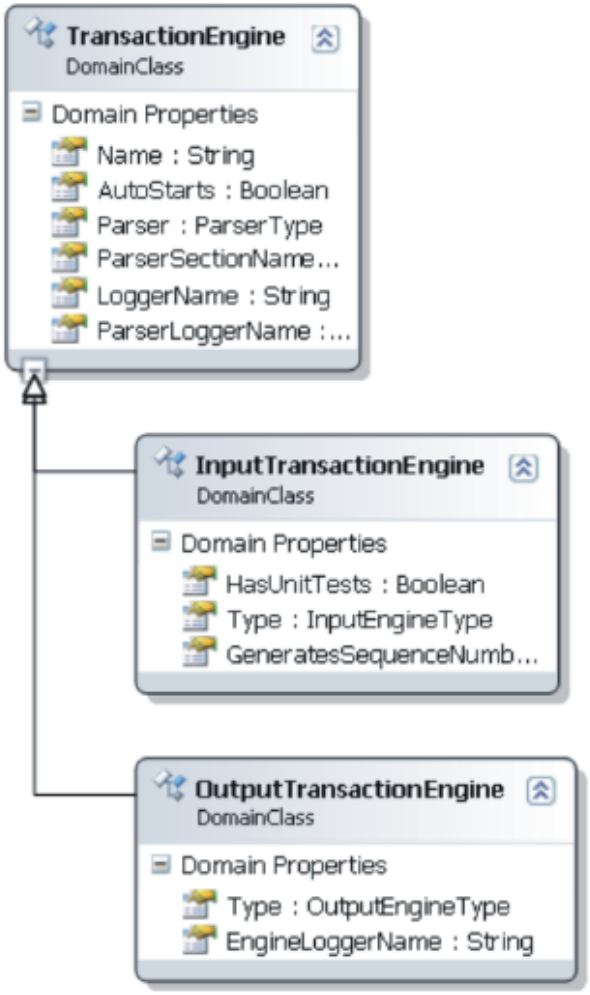

Figura 60 - Diagrama de relaciones jerárquicas del elemento *TransactionEngine*, con sus clases de dominio heredadas, *InputTransactionEngine* y *OutputTransactionEngine*.

El metamodelo propone una jerarquía de clases para los motores del DSL: la clase con las propiedades comunes a todos los motores se implementa en *TransactionEngine,* mientras que *InputTransactionEngine* y *OutputTransactionEngine* son clases derivadas que permitirán modelar los *motores de entrada y de salida* respectivamente.

Propiedades de *TransactionEngine*	**Objetivo**
Name: *string*	Es el nombre simbólico que se le da al motor para distinguirlo dentro del modelo. Se utilizará durante la transformación al código funcional para darle nombre a la clase y al archivo que contiene la implementación.

AutoStarts: *bool*	Es una propiedad que indica si el motor debe ejecutarse como consecuencia de la previa ejecución del método *Start()* de la clase *Facade* del sistema bajo diseño.
Parser: *ParserType*	Es una propiedad que define el tipo de *analizador* (*parser*) que tendrá definido el motor para entender el protocolo de las transacciones entrantes. Ese *ParserType* es una enumeración con valores válidos de clases que heredan de *AbstractTransactionParser*
ParserSectionName: *string*	Es una propiedad que define el nombre del *tag* XML, que servirá dentro del *app.config* de la solución para definir el uso de los campos del protocolo dado. Es decir, si bien los campos de un protocolo (por ejemplo ISO8583) están definidos para todas las soluciones, puede haber distintos sistemas transaccionales que interpreten los contenidos de diferentes formas. Es dentro de esos *tags* donde debe definirse la interpretación del protocolo para el alcance bajo construcción.
LoggerName: *string*	Es el nombre del *logger* dentro del contexto de *log4net*. *Log4net* permite implementar varios *loggers* dentro de un *AppDomain*, y a través de esta propiedad el modelador define en qué *logger* escribirá el motor bajo desarrollo.
ParserLoggerName: *string*	Similar a la propiedad anterior, pero en este caso se permite configurar otro *logger* adicional, de modo que cuando el motor transfiere la ejecución a los métodos del *parser* configurado, este último pueda generar mensajes de *log* en un archivo diferente, o bajo otras condiciones distintas a las del motor.

Tabla 15 - Propiedades principales de la clase de dominio *TransactionEngine*

Las clases *TransactionEngine* no pueden modelizarse directamente, ya que para el metamodelo dado, es una clase sin formato gráfico asociado. Sin embargo, se pueden modelar sus dos clases derivadas que se describen a continuación: *InputTransactionEngine* y *OutputTransactionEngine*.

5.3.5 Motor Transaccional de Entrada

Los *motores transaccionales de entrada* (ME) representan a los componentes de código que inyectan una transacción en el sistema, principalmente porque suelen configurarse para escuchar por un medio (por un puerto o canal dado), o porque se los configura para emitir transacciones bajo determinados eventos (un *timer* por

ejemplo). La mayoría de las veces son los encargados de leer un paquete de datos codificado bajo un protocolo prestablecido, y en función del tipo de la transacción detectada se crea un hilo de ejecución. Ese *thread* estará destinado a procesar el método *DoTransaction()* de una clase heredada de *AbstractTransactionHandler* para el procesamiento de la transacción entrante.

Estos elementos de dominio permiten (luego de la correspondiente transformación a código funcional) crear una clase *motor de entrada* personalizada con propiedades específicas, y cuyo funcionamiento se basa en los distintos tipos de *Input Engines* analizados en el *framework TransactionKernel*. Es decir, la clase generada por la transformación heredará los comportamientos de alguno de los motores analizados dentro de *TransactionKernel*.

5.3.5.1 Propiedades y Relaciones

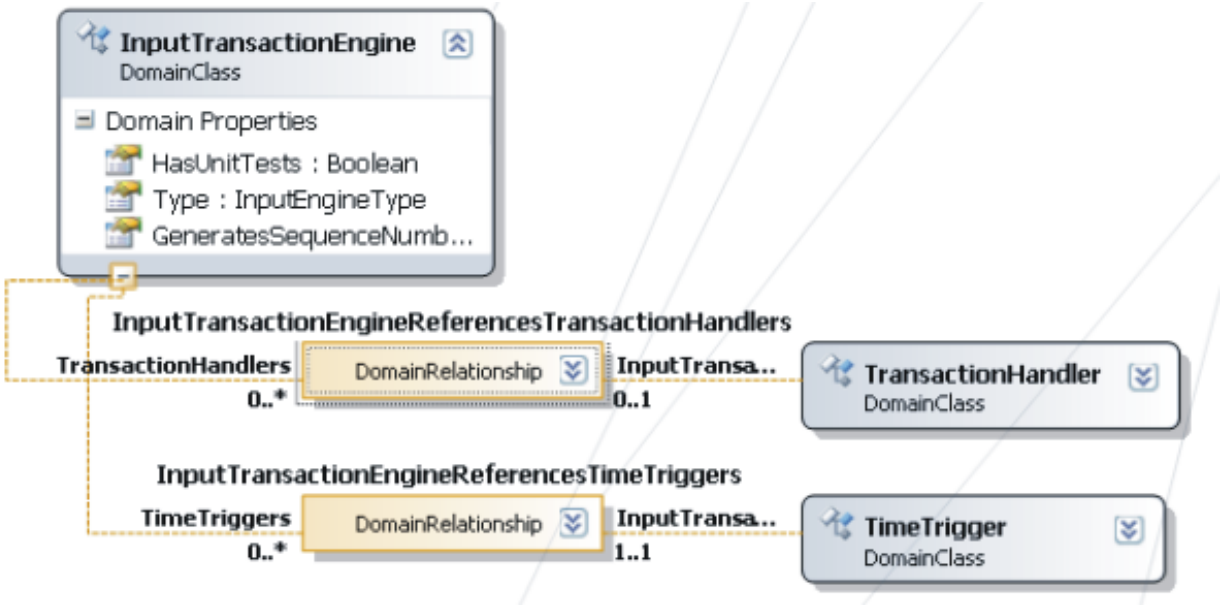

Figura 61 - Diagrama de relaciones referenciales del elemento *InputTransactionEngine*.

De la Figura 61, se pueden observar las *relaciones de dominio* de la clase *InputTransactionEngine*, de modo que:

- Un ME puede tener asociados ninguno o muchos *TransactionHandlers* (o *Manejadores Transaccionales*), es decir que puede tener configurada ninguna o varias transacciones. A través del método *GetOperationId()* del *parser* asociado al ME, se instanciará el *TransactionHandler* correcto para la atención de la transacción. Las transacciones que puede instanciar el ME son solo las que estén conectadas con el motor.

- A su vez cada transacción puede estar a lo sumo conectada a un motor (es decir, 0 a 1 ME).
- Así también, puede darse el caso que un *TransactionHandler* no esté conectado a un motor. Es decir que en ese caso, las transacciones pueden estar armadas con muchos *TransactionHandlers* conectados entre sí, como una forma de subdividir el proceso total en varias etapas. Esto es lo que se hace comúnmente en los *Transaction Layers*: dada una solución con dos capas por ejemplo, en la capa de nivel 0 se podría modelizar un ME con un *manejador transaccional* (MT), y en la capa de nivel 1 un *motor de salida* con otro MT. Los dos MT estarán probablemente vinculados para representar dos etapas del mismo proceso transaccional. Cabe acotar que esto puede extenderse a 'n' etapas, siendo la configuración decisión del modelador para su conveniencia en el entendimiento del proceso total.
- Un ME puede tener asociados ninguno o muchos *Time Triggers*, o *disparadores de tiempo* (DT). Estos *disparadores* son sólo válidos si el tipo del ME elegido es *Time Triggered Input Engine*. Una vez realizada la transformación a código, estos DT se traducirán a eventos de *timer* programables que dispararán transacciones de forma periódica en el sistema a través del ME, y sin necesidad de que estas transacciones provengan de otro nodo externo. Estos ME se han usado para transacciones de *Echo* en sistemas financieros, por ejemplo. Particularmente para la implementación de los DT en la tecnología .NET, se ha utilizado un objeto del tipo *AutoResetEvent*.

Propiedades de *InputTransactionEngine*	**Objetivo**
HasUnitTests: *bool*	Es una propiedad que define si las transacciones asociadas al *motor de entrada* tendrán casos de pruebas unitarios pre-generados al momento de la transformación a código.
Type: *InputEngineType*(enum)	Esta propiedad define el tipo de *motor de entrada*, que al momento de la transformación a código definirá la herencia que tendrá el motor bajo desarrollo. Esa clase base deberá ser del tipo *AbstractInputTransactionEngine*, teniendo la posibilidad de ser alguna de las clases de ME ya implementadas dentro del *framework TransactionKernel*. Se recuerda al lector que las clases de ME definidas en *TransactionKernel* son: • *AbstractThreadedInputTransactionEngine* • *AbstractTcpTriggeredThreadedInputTransactionEngine* • *AbstractTcpTriggeredSocketResuableThreadedInputTransactionEngine* • *AbstractTimeTriggeredInputTransactionEngine*

GeneratesSequenceNumber: ***bool***	Define si el motor genera un número secuencial por cada transacción entrante o inyectada al sistema. Puede ser de gran utilidad para la posterior auditoría de transacciones, porque es un método de identificación de la misma. Funcionalmente, esta propiedad (si es *true*) hace obligatorio que cada *manejador transaccional* atado al ME ejecute el método *SequenceFactory()*. Por otro lado, se generarán automáticamente durante la transformación a código, los *scripts* necesarios para la base de datos, de modo de albergar el último valor generado, y el *Stored Procedure* para recuperar e incrementar en 1 la cuenta total.

Tabla 16 - Propiedades principales de la clase de dominio *InputTransactionEngine*

5.3.5.2 Formato

El formato gráfico definido en el metamodelo es un *CompartmentShape* simple, que acumula en la parte inferior los nombres de los *manejadores transaccionales* (MT) que están conectados al ME. Está decorado por el nombre del motor y el tipo de motor elegido. Los MT quedan gráficamente conectados al ME a través de una flecha de guiones, definida en el conector que se muestra en la Figura 62.

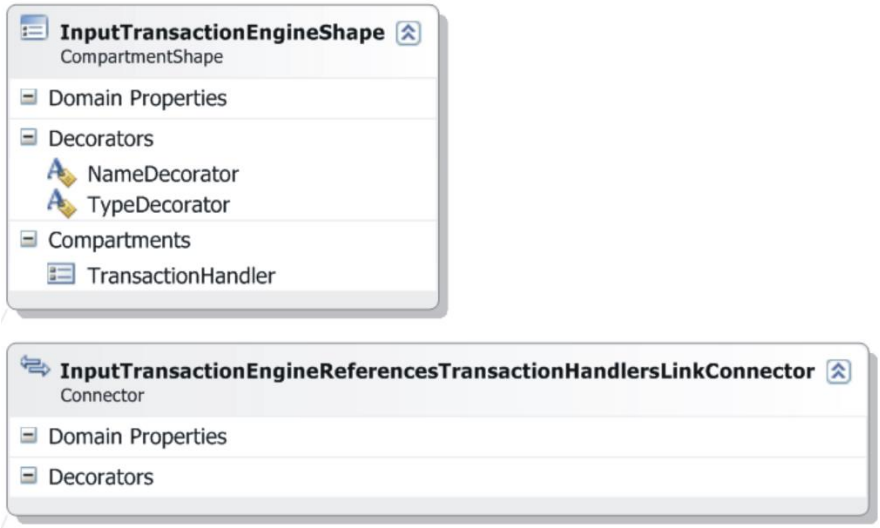

Figura 62 - Formato asignado a la clase de dominio *InputTransactionEngine,* y al conector de este tipo de objeto con objetos del tipo *TransactionHandler*

5.3.6 Manejador Transaccional

Los *manejadores transaccionales* (MT) son los elementos definidos en el metamodelo de este lenguaje para representar a las transacciones en diferentes etapas de su transición. Estos *elementos de dominio*, se transformarán en clases que hereden de *AbstractTransactionHandler,* durante la fase de transformación a código funcional.

El lenguaje habilita a los MT a poder representar una transacción lógica en etapas, siendo cada etapa representada por único MT. Por heredar de *AbstractTransactionHandler,* las clases que se generen automáticamente de los modelos basados en el DSL obtendrán el conocimiento acumulado y capitalizado en el *framework*. También podrán conectarse directamente con *motores de entrada*, *motores de salida* y otros elementos del dominio con conectores gráficos.

5.3.6.1 Propiedades y Relaciones

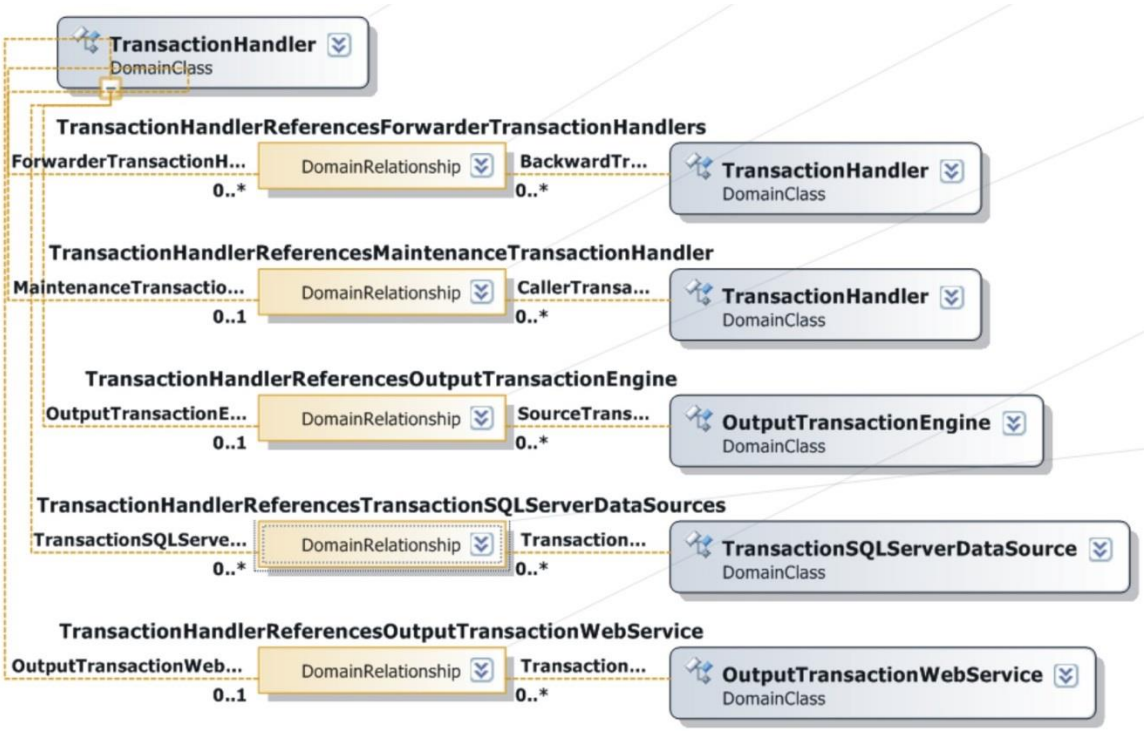

Figura 63 - Diagrama de relaciones referenciales del elemento *TransactionHandler*.

Los *manejadores de transacciones* pueden conectarse con otras instancias del modelo, de modo que:

- Un MT puede conectarse con ninguno o muchos *manejadores transaccionales*. Por ejemplo, cuando el *manejador transaccional* se conecta solo con un *motor de entrada*, y la transacción real es representada por una sola etapa, hay un único MT (que no se conecta con ningún otro manejador) para

representar la transacción real. En cambio, un MT podría conectarse con otros MT cuando la representación de la transacción real está compuesta por más de una etapa durante su transición.

- Existen dos tipos de relaciones posibles entre *manejadores transaccionales*: del tipo *ForwarderTransactionHandlerLink*, y del *MaintenanceTransactionHandlerLink*. En el primer caso, dos o más MT pueden relacionarse cuando una transacción real está dividida en varias etapas, cada una de estas etapas representada por cada MT. Este tipo de vinculación de *manejadores transaccionales* permite la instanciación en cascada de MT's durante la transición del procesamiento. Por ejemplo desde que se instancia por el *motor de entrada*, hasta que es autorizada de forma local o reenviada por un *motor de salida*. Específicamente, la instanciación en cascada de MT's se realizará dentro del método *ForwardHandlerFactory(),* como ya se explicó en capítulos anteriores. El método para instanciar el siguiente MT se generará como producto de la transformación del modelo a código. Por último, en el caso de *MaintenanceTransactionHandlerLink,* la relación permite que cada MT tenga como máximo un solo manejador transaccional vinculado con fines de mantenimiento, es decir que se instanciará durante el método *MaintenanceHandlerFactory()* de la secuencia genérica. El método para instanciar al MT correcto también será fruto de la transformación del modelo a código.
- Un MT puede conectarse opcionalmente a un *motor de salida*. Es decir que la transformación a código vinculará la llamada al método *Resolve()* de *AbstractTransacionHandler* con la llamada de *Resolve()* del *motor de salida* (MS) (*AbstractOutputTransactionEngine*). De esta forma la transacción podrá hacer uso del MS para reenviar información transaccional hacia nodos externos. El *motor de salida* en cambio puede tener conectados cero o muchos MT.
- Un MT puede conectarse a ninguno o a muchos *orígenes de datos transaccionales* (*TransactionSQLServerDataSource*). Este DSL permite elegir tres momentos para la conexión con el origen de datos: la etapa de *preproceso,* la etapa de *postproceso* o la etapa de *postproceso final*. La transformación a código vinculará los métodos *PreProcessTransaction()*, *PostProcessTransaction() y FinalProcessTransaction()* de la secuencia genérica con llamadas secuenciales a otros métodos (uno por cada origen de datos vinculado), para operar contra un mecanismo de persistencia (en esta versión del DSL, una base de datos *SQLServer*).
- Por último un MT puede conectarse opcionalmente a un *web service transaccional de salida* (WSTS), de forma análoga como lo hace con los *motores de salida*. En este caso, la transformación a código generará una plantilla en el método *Resolve()* de la clase del MT para poder consumir una operación del WSTS. De esta forma la transacción podrá también reenviar información transaccional hacia nodos externos. El WSTS en cambio puede tener conectados cero o muchos MT.

Propiedades de *TransactionHandler*	**Objetivo**
Name: *string*	Es el nombre de la transacción. Sirve para identificarla en el modelo, y para darle nombre a la clase y a los archivos de código durante la transformación del modelo a código.
TransactionId: *string*	Es la propiedad que identifica a la transacción a nivel lógico. Servirá para que los *parsers* asociados a los *motores de entrada* puedan vincular en el código la instanciación de una clase de un MT dado. La instanciación correcta se decidirá en función del campo *TransacionId* de la trama entrante, que deberá coincidir oportunamente con la definida en este campo homónimo del MT. Es decir, se transformará a código una sobrecarga del método *TransactionHandlerFactory()* (de la clase *AbstractInputTransactionEngine*) en la clase del ME. Por último se vinculará el resultado del método *GetOperationId()* (del *parser* asociado a ese ME) con una instanciación del MT en función del *TransactionId* (se instanciará la clase correcta para la atención de la transacción). Si la trama entrante tiene un *OperationId* que no pueda vincularse con ningún *TransactionId*, la transacción se descarta en el ME.
LoggerName: *string*	Es el nombre del *logger* usado por *log4net* para configurar de forma modular la escritura del MT.

Tabla 17 - Propiedades principales de la clase de dominio *TransactionHandler*

5.3.6.2 Formato

El formato gráfico definido en el metamodelo es un *GeometrytShape*, que contiene dos decoradores: Uno está asociado al nombre de la transacción (parte superior del elemento gráfico), el segundo está asociado al *TransactionId*. Luego se describen las definiciones del metamodelo de cada uno de los conectores, que representan de forma gráfica a las relaciones descritas en los párrafos anteriores.

Figura 64 - Formato asignado a la clase de dominio *TransactionHandler*.

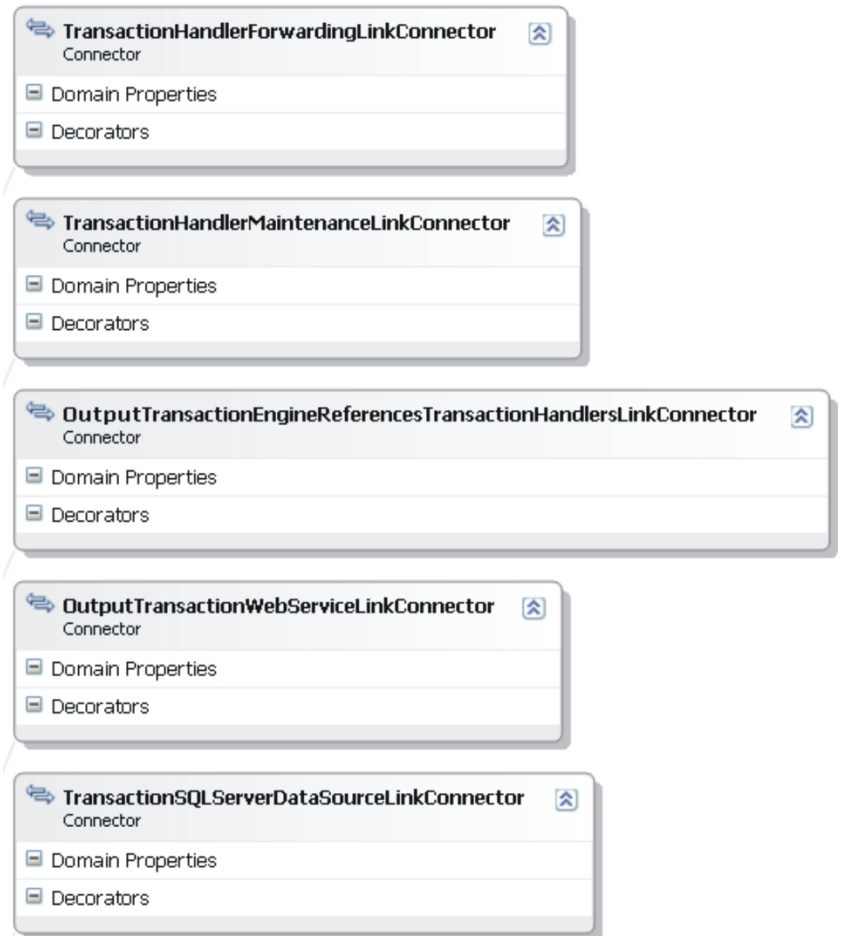

Figura 65 - Formato asignado a los conectores que involucran a la clase de dominio *TransactionHandler*

5.3.7 Motor Transaccional de Salida

Los *motores de salida (*MS) son los responsables de resolver la comunicación con un nodo externo, durante la fase de reenvío de un requerimiento hacia el exterior. Los MS se implementan (a nivel de código fuente) como una clase específica de la solución, pero heredada de *AbstractOutputTransactionEngine* (correspondiente al *framework* presentado en capítulos anteriores). El usuario que diseña la solución puede interconectar *manejadores de transacción* (MT) según un alcance dado a un MS, de modo que todo ese conjunto de transacciones puedan emitir requerimientos hacia el exterior del sistema, y eventualmente recibir respuestas.

5.3.7.1 Propiedades y Relaciones

Si bien la vinculación primaria de los *motores de salida* es con los *manejadores de transacciones* (como se analizó durante la sección correspondiente), los MS pueden estar conectados opcionalmente con un *soporte a base de datos* (*TransactionDataSourceSupport*), en una relación 0 a 1. Este concepto permite incrementar el nivel de automatización de la transformación a código de la herramienta para algunos tipos de MS.

Por ejemplo, para el *motor de salida mono–punto*, correspondiente a la clase del *framework AbstractTcpFunneledOutputTransactionEngine*, muchos de los métodos abstractos en la definición requieren algún sistema de persistencia para guardar los mensajes entrantes y salientes que se encolan, para que los *threads* de envío y recepción puedan obtenerlos, y/o guardarlos dentro de la cola. Como no hay una única solución de persistencia, y puede quedar a decisión del usuario cuál implementar a conveniencia, los métodos definidos en la jerarquía de clases del motor no dan indicios (en su transformación a código funcional) de preferencia por una u otra forma. Por eso, desde el lenguaje, se propone este tipo de vinculación opcional entre un MS con un *TransactionDataSourceSupport*: si el usuario decide usar este soporte, en función del tipo del motor y del tipo de persistencia elegida la herramienta de transformación decide implementar (en código funcional) una capa de acceso a datos para que el MS funcione automáticamente y sin interacción del usuario. En otras palabras, la herramienta generará tablas*, stored procedures* y métodos de acceso a datos en código fuente, pertenecientes a una DB de motor *SQLServer*. Una de las líneas de trabajo futuro para este DSL es implementar más mecanismos de soporte a persistencia, que permitan expandir las posibilidades de almacenar datos de importancia para cada uno de los tipos de MS definidos.

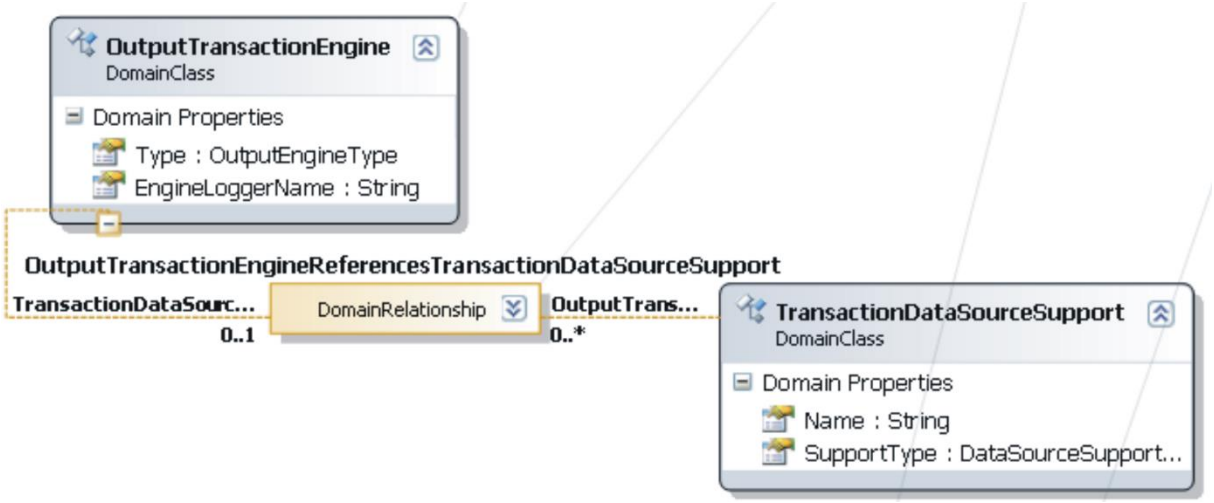

Figura 66 - Diagrama de relaciones referenciales del elemento *OutputTransactionEngine*.

Propiedades de *OutputTransactionEngine*	**Objetivo**
Type: *OutputEngineType*	Es una enumeración de tipos de *motor de salida,* que al momento de la transformación a código, generará una clase de usuario heredada de alguna de las clases abstractas dentro de la jerarquía del *framework*. Los tipos de MS en la enumeración actual son: • *Output Engine* • *Straight Output Engine* • *Funneled Output Engine* • *Tcp Straight Output Engine* • *Tcp Funneled Output Engine*
EngineLoggerName: *string*	Es el nombre del *logger* usado por *Log4Net* para configurar de forma modular la escritura de mensajes de log del MS.

Tabla 18 - Propiedades principales de la clase de dominio *OutputTransactionEngine*

5.3.7.2 Formato

Similar a los *motores de entrada,* los MS usan un formato gráfico del tipo *CompartmentShape* simple, que acumula en la parte inferior los nombres de los *handler* de transacción que están conectados al MS. Está decorado por el nombre del motor y el tipo de motor elegido.

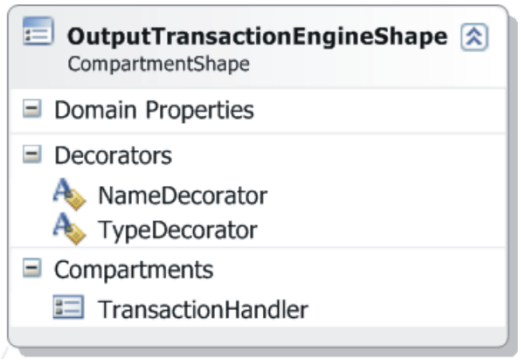

Figura 67 - Formato asignado a la clase de dominio *OutputTransactionEngine*

5.3.8 Origen de Datos Transaccional

Los *orígenes de datos transaccionales* (ODT) son los elementos que representan puntos de acceso a datos en el modelo del DSL. Se vinculan con *manejadores transaccionales*, es decir que explícitamente indican que un MT debe acceder a un medio de almacenamiento de datos para correr una operación (o un conjunto de operaciones) CRUD. Al momento de este trabajo, el único mecanismo de persistencia compatible con los ODT son los conectores a base de datos *SQLServer*. Por lo que, cada vez que un MT se conecte con un ODT, el manejador

deberá ejecutar un *Stored Procedure*, que a su vez realizará una o varias operaciones en las bases de datos involucradas.

Como ya se describió, dada la propuesta de secuencia de trabajo genérica dentro del *framework TransactionKernel,* la vinculación de los ODT con lo MT puede generar código automático de acceso a datos en distintas etapas de ejecución de una transacción: *PreProcessTransaction()*, *PostProcessTransaction()* y *FinalProcessTransaction().* Esto se puede configurar desde el modelo, por propiedades de dominio del ODT.

5.3.8.1 Propiedades y Relaciones

Existe una relación entre *TransactionDataSource* y *TransactionSQLServerDataSource,* y está definida en el DSL en carácter hereditario. De todos modos, la única clase instanciable en los modelos es *TransactionSQLServerDataSource*, y a futuro deberían implementarse una gama mayor de ODT a diferentes tipos de almacenamiento. Dentro de un modelo dado, la única relación disponible de los ODT se da con los *manejadores de transacción*.

Figura 68 – Diagrama de relaciones jerárquicas del elemento *TransactionDataSource*.

Propiedades de *TransactionDataSource*	**Objetivo**
Name: *string*	Es el nombre que recibe el elemento, y que dará también nombre a los *scripts, stored procedures*, tablas, y código fuente para el acceso a datos.
StepOrder: *Int16*	Dado que un *manejador transaccional* puede conectarse con más de un ODT, esta propiedad permite dar un orden de ejecución entre todos los ODT asociados a ese manejador, dentro de una misma etapa (*PreProcessStage, PostProcessStage o FinalProcessStage*). Por ejemplo, si un MT está conectado con dos ODT en la etapa de *PreProcessStage*, uno de los ODT deberá tener el *StepOrder* igual a 1, y el otro el *StepOrder* igual a 2. De ese modo, se configura el orden de ejecución entre ambos ODT.

Tabla 19 – Propiedades principales de la clase de dominio *TransactionDataSource*

Propiedades de *TransactionSQLServerDataSource*	Objetivo
InstanceName: *string*	Nombre de la instancia del motor de DB a la cual se conectará el sistema para la ejecución de las operaciones incluidas dentro del ODT.
DatabaseName: *string*	Nombre de la base de datos a la cual se conectará el sistema para la ejecución de las operaciones incluidas dentro del ODT.
ConnectionString: *string*	*String* de conexión contra la base de datos. Se utilizará para automatizar la configuración durante la conexión, dentro del código fuente de acceso a datos.
ExecutedOnStage: PersistableStage (enum)	Con esta propiedad de dominio, se puede configurar en qué etapa se ejecutará la operación contra la base de datos. • *PreProcessStage* • *PostProcessStage* • *FinalProcessStage* Según la configuración de etapa, el acceso a datos de la DB se hará en el método *PreProcessTransaction()* (para el caso de *PreProcessStage*), en el método *PostProcessTransaction()* (para el caso de *PostProcessStage*), o en el *FinalProcessTransaction()* (para el caso de *FinalProcessStage*).

Tabla 20 - Propiedades principales de la clase de dominio *TransactionSQLServerDataSource*

5.3.8.2 Formato

El formato configurado para estos elementos es del tipo *GeometryShape*, con un formato circular violeta, y con decoradores de las propiedades *Name* y *StepOrder*.

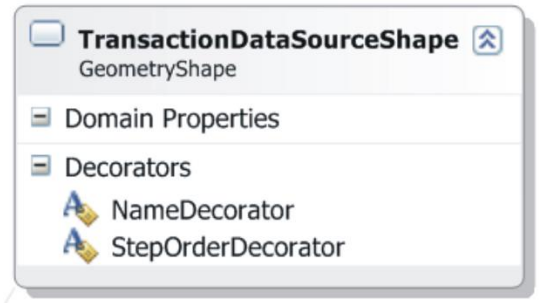

Figura 69 - Formato asignado a la clase de dominio *TransactionDataSourceShape*

5.3.9 Disparador de Tiempo

Los *disparadores de tiempo* (DT) fueron diseñados para que sean elementos del dominio con posibilidad de combinación con otros elementos, de modo de poder generar bases de tiempo auto-reiniciables. De esta forma, los DT pueden generar, por ejemplo, transacciones disparadas por eventos periódicos cuando están vinculados con un *motor de entrada*. Son de gran utilidad en aquellos casos que se requiera configurar transacciones para mantener una sesión lógica de red, o porque un autorizador/*switch* requiere mantener un *keep-alive* para mantener la conexión. También son útiles cuando se requiere enviar una transacción de *echo* periódico hacia algún nodo externo, o para generar tareas de mantenimiento internas.

5.3.9.1 Propiedades y Relaciones

Al momento de escribir este trabajo, los DT se pueden vincular con los *motores de entrada* solamente. Para el futuro existe la propuesta de incrementar las fuentes generadoras de disparo de transacciones (es decir, no solo debido a una base temporal auto-reiniciable), e incrementar las combinaciones de los DT con otros elementos del dominio.

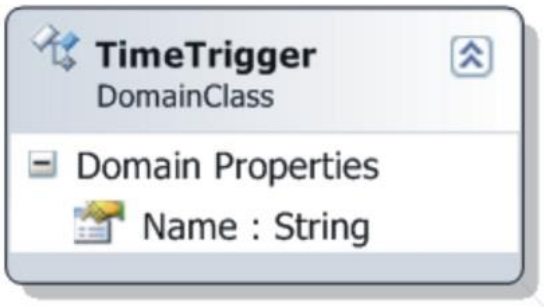

Figura 70 - Diagrama de la clase de dominio *TimeTrigger*

Propiedades de *TimeTrigger*	**Objetivo**
Name: *string*	Es el nombre del elemento que dará también nombre a la clase cuando se realice la transformación a código funcional.

Tabla 21 - Propiedades principales de la clase de dominio *TimeTrigger*

5.3.9.2 Formato

El formato seleccionado para el elemento en el modelo es del tipo *Image Shape*. Se seleccionó un dibujo (.jpg) de un reloj despertador, y tiene como decorador el nombre del elemento.

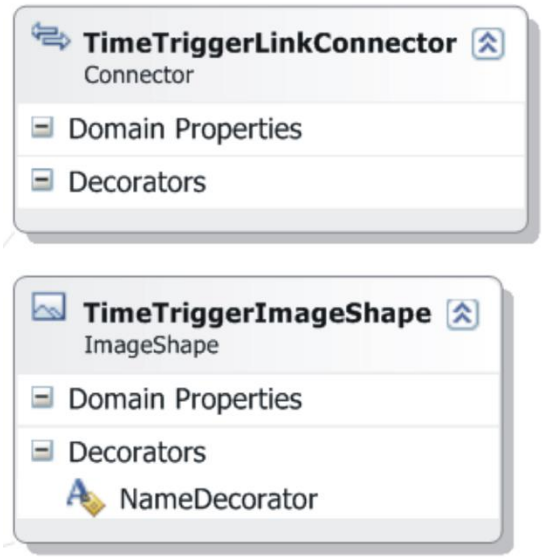

Figura 71 - Formato asignado a la clase de dominio *TimeTrigger* y a su conector

5.3.10 Web Service Transaccional de Salida

Los *Web Services Transaccionales de Salida* (WSTS) son una variación de los *motores de salida*. Los WSTS permiten conectar uno o más *manejadores transaccionales* para que a través del método *Resolve()*, puedan reenviar requerimientos a través de *métodos web* definidos en un *web service* remoto. Al no tener de antemano una definición concreta de un *web service*, los WSTS no pueden conceptualizarse dentro de la jerarquía de los *motores de salida* (en los MS si se conoce de antemano el mecanismo de reenvío, y lo que se modifica en cada caso es el protocolo usado y la información que viaja en el mismo).

5.3.10.1 Propiedades y Relaciones

Con respecto a otros elementos del dominio, los WSTS pueden relacionarse con muchos *manejadores de transacciones* (0 a *). Cuando se detecta este vínculo durante la transformación a código, el método *Resolve()* es construido automáticamente en cada MT para hacer una llamada genérica a un *web method*. El *web method* se puede representar en el modelo a través de la propiedad de dominio *TransactionId* del MT, siempre que ese MT esté conectado a un WSTS. Es decir, que el nombre configurado en *TransactionId* será el nombre del *método web* a llamar dentro de *Resolve()*.

Con respecto a la definición del metamodelo del lenguaje, los WSTS son en realidad instancias de la clase *OutputTransactionWebService* (ver Figura 72). Se desarrolló esta jerarquía para que en una propuesta futura se puedan configurar también *web service transaccionales de entrada* (WSTE), que funcionen como receptores de transacciones de forma análoga a un *motor de entrada*.

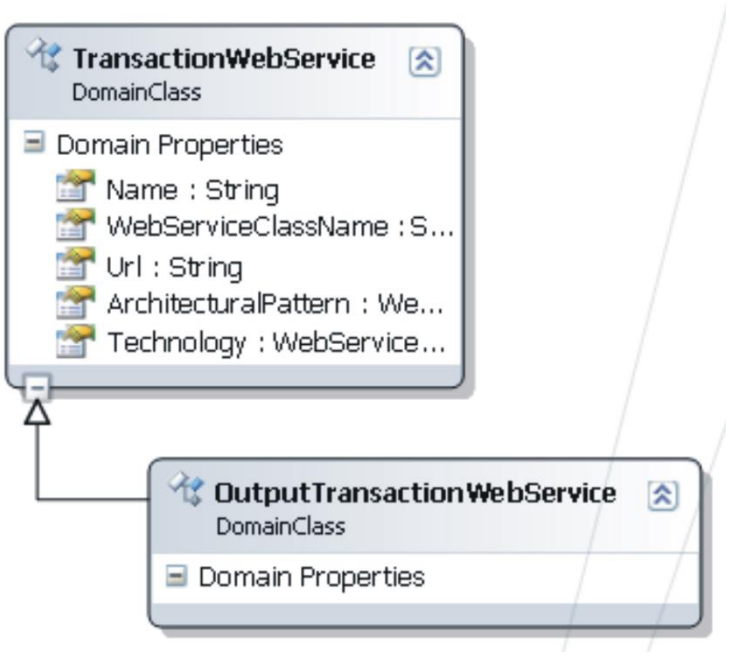

Figura 72 - Diagrama de relaciones jerárquicas del elemento *TransactionWebService* y *OutputTransactionWebService*.

Propiedades de *TransactionWebService*	**Objetivo**
Name: *string*	Es el nombre el *web service* en el modelo que permitirá dar nombre a las referencias generadas automáticamente al WSTS.
WebServiceClassName: *string*	Es el nombre de la clase *proxy* del *web service*, generada por el IDE, por *wsdl.exe* o por cualquier otra herramienta de transformación de definición de *web service* a código.
Url: *string*	Es la URL a donde está alojado el w*eb service.*
ArchitecturalPattern: *WebServiceArchitecturalPattern (enum)*	Es una enumeración que permite definir cómo se manejan las instancias de la clase *proxy* del *web service* en el sistema. Al momento de este trabajo las opciones son las siguientes dos: • *Singleton*: Se genera automáticamente código para manejar una única instancia de la clase *proxy* del *web service*. Estas propiedades que configuran un patrón *Singleton,* se generan de forma automática por el transformador de código en la clase *Façade* del sistema.

	• *MultipleInstances*: Configurando esta opción, en cada método *Resolve()* de los MT conectados al WSTS se generará un bloque *using,* donde se instanciará la clase *proxy* del w*eb service*. Al finalizar la transacción contra el WSTS, la instancia se desecha.
Technology: ***WebServiceTechnology (enum)***	Es una enumeración que permite definir la tecnología del *web service.* A través de esta propiedad de dominio, la herramienta de transformación puede sacar provecho de las propiedades de las clases base del *proxy* del *web service*. Al momento de este trabajo, solo se han definido dos tecnologías: • *SoapHttpClientProtocol: Web service* con tecnología SOAP. • *WCF: Windows Communication Foundation.*

Tabla 22 - Propiedades principales de la clase de dominio *TransactionWebService*

5.3.10.2 Formato

El formato elegido para los WSTS es un *Geometry Shape*, de color azul rectangular. A diferencia de los *motores de salida*, no tiene *Compartment,* es decir que no acumula los *manejadores de transacciones* conectados al WSTS. Tiene dos decoradores: uno para el nombre del *web service* en el modelo, y el otro para el nombre de la clase *proxy*.

Figura 73 - Formato asignado a la clase de dominio *OutputTransactionWebServiceShape*

5.3.11 Capa Transaccional de Entorno y Variables Transaccionales de Entorno

Las *capas transaccionales de entorno* (TEL), son gráficamente similares a las *capas transaccionales* (*Transaction Layers*) ya analizadas, pero permiten albergar *variables transaccionales de entorno* (VTE) que se

almacenarán en algún medio de datos. Estas VTE solamente pueden incluirse en las TEL, y definen un mecanismo para usar variables de entorno personalizadas para el servicio bajo construcción. En tiempo de ejecución, las VTE pueden ser eventualmente llamadas cuando sean necesarias, y están enlazadas por una capa de acceso de datos al mecanismo de persistencia que las guarda. En el futuro se propone que en esta capa se puedan incluir otros elementos que faciliten la personalización del código de forma automática por medio de la herramienta de transformación.

5.3.11.1 Propiedades y Relaciones

La relación entre estos dos elementos de dominio (TEL y VTE) es exclusiva entre ellos, si bien en el futuro se propone ampliar el uso de las TEL para otros elementos de dominio que permitan mejorar la personalización de sistema transaccional. En concreto, las TEL pueden contener cero o muchas VTE, y las VTE pueden pertenecer a una única *capa transaccional de entorno*.

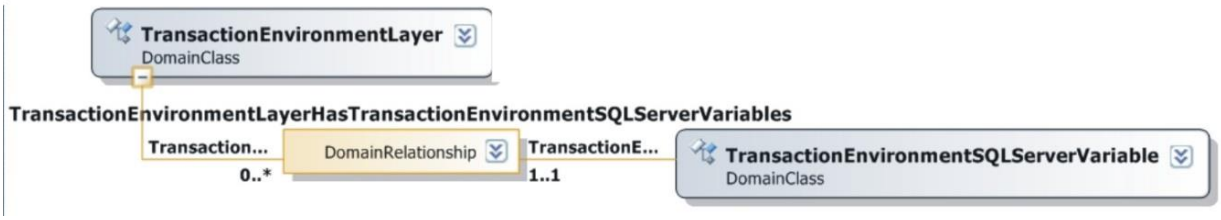

Figura 74 - Diagrama de relaciones referenciales del elemento *TransactionEnvironmenLayer*.

Figura 75 - Diagrama de la clase de dominio *TransactionEnvironmenSQLServerVariable*.

Propiedades de *TransactionEnvironmentLayer*	**Objetivo**
Name: *string*	Es el nombre de la TEL, con fines figurativos. No se utiliza para nombrar ningún archivo o clase.

Tabla 23 - Propiedades principales de la clase de dominio *TransactionEnvironmentLayer*

Propiedades de *TransactionEnvironmentSQLServerVariable*	**Objetivo**
Name: *string*	Es el nombre de la variable de entorno. En este caso el nombre sirve para las referencias a la variable de entorno en el código funcional, y para definir la variable en el mecanismo de persistencia.
DefaultValue: *string*	Es el valor por defecto que se generará para la VTE en el mecanismo de persistencia.

Tabla 24 - Propiedades principales de la clase de dominio *TransactionEnvironmentSQLServerVariable*

5.3.11.2 Formato

El formato elegido para la TEL es similar a la de la *capa transaccional,* un *swimlane* de color verde, con un decorador en la parte superior derecha con el nombre de la *capa transaccional de entorno.* En cambio, las VTE son del tipo *GeometryShape*, de color gris, y con el decorador del nombre de la variable. Además las VTE pueden minimizarse y maximizarse.

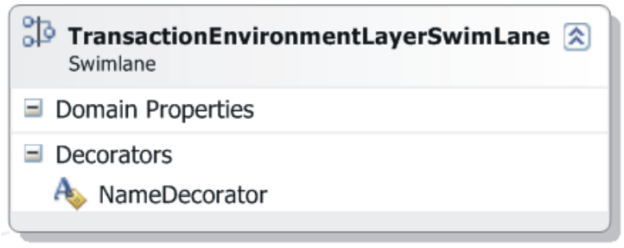

Figura 76 - Formato asignado a la clase de dominio *TransactionEnvironmentLayer*

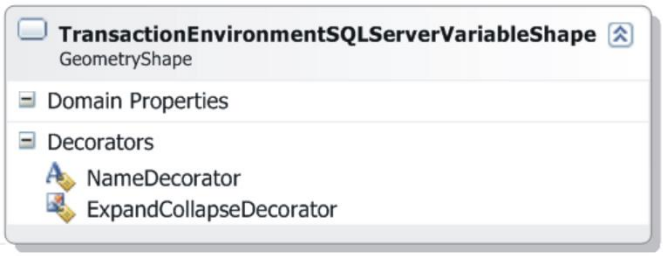

Figura 77 - Formato asignado a la clase de dominio *TransactionEnvironmentSQLServerVariable*

5.4 Resumen del Capítulo

En este capítulo se trataron los siguientes temas:

- Se introdujo la propuesta de formalización del *framework TransactionKernel* a través de metodologías MDD. La propuesta consta de dos partes, la primera es el lenguaje de dominio específico, y la segunda es la herramienta de transformación de modelos a código funcional. No se han incluido de forma explícita las reglas de transformación a código ya que no suman a la descripción de la propuesta de formalización.
- Se introdujeron las ventajas del uso de metodologías MDD frente a la problemática de construcción de *software* conocida como *el atajo del programador*. Se vincularon las dos partes de la propuesta de formalización de *framework* dentro del modelo de construcción iterativo: el DSL será la herramienta que permite la transición desde la definición de requerimientos (*análisis*) a la etapa de *diseño*, mientras que la herramienta de transformación hace a la transición desde la etapa de *diseño* a la de *codificación*.
- Se presentaron las consideraciones iniciales bajo las cuales se desarrolló el lenguaje. Así se describió el principio de funcionamiento de la herramienta de transformación a código y algunas definiciones base para la modelización de una transacción y para la construcción del lenguaje específico de dominio.
- Luego se describieron los elementos del dominio que hacen al lenguaje. Principalmente se remarcó el objetivo de cada elemento, cómo se relaciona con otros elementos dentro de un modelo, cuáles son sus propiedades de dominio y para qué sirven, y como se relaciona con clases del *framework TransactionKernel.*

6 Evaluación del DSL

6.1 Introducción

En este capítulo se introducen los requerimientos para el armado de un sistema transaccional simple. Se propone ilustrar al lector cómo se utiliza el DSL y cómo se relacionan los elementos de dominio del *framework* con los requerimientos en un caso concreto.

Los requerimientos que describen al sistema que se pretende desarrollar fueron obtenidos de un ambiente real de trabajo, y que actualmente está funcionando como un *bridge* transaccional (adaptador) en Colombia. Las instantáneas de pantallas que se muestran durante este capítulo son muestras reales del modelo usado para la construcción de esta solución.

También se expondrá una comparación de esfuerzo para dos sistemas realizados en la compañía con características similares (a nivel de especificaciones, transacciones soportadas, protocolo de comunicación, etc.). Uno de los sistemas se realizó previo a la implementación de la metodología MDD (DSL + el *framework* transaccional), y el otro con la metodología descrita en este trabajo. Entonces, se intentará concluir que la metodología efectivamente logra reducir los tiempos de desarrollo, como así también agudiza el entendimiento de los sistemas por los miembros del equipo de desarrollo y mejora la visibilidad del proyecto para los *stakeholders*.

6.2 Sistema a Resolver

El proyecto para crear este sistema se armó a mediados de 2014, siendo el objetivo del mismo el armado de un adaptador (*bridge*) que pudiera adaptar las transacciones entrantes provenientes de un concentrador (*switch*) de venta de tiempo-aire. Particularmente este *bridge* tiene que adaptar las transacciones de este *switch* para ser reenviadas a un autorizador externo de una conocida empresa de telefonía celular. Dado que la empresa de telefonía celular tiene su propio protocolo de comunicación (no solo a nivel de trama de datos, sino a nivel sesión, seguridad, etc.), el *adaptador* será el responsable de enmascarar (desde la perspectiva del *switch*) los detalles comunicativos que presenta el nodo provisto por la empresa de telefonía.

Los requerimientos relevados durante la primera semana del proyecto fueron los siguientes:

- El adaptador tendrá que resolver tres transacciones provenientes del *switch:* **Venta**, **Cancelación / Reverso**, y **Consulta de Venta.**
- Se debe mantener una única conexión activa contra el nodo externo, más allá de la cantidad de clientes conectados en un momento dado. Es decir que no se permite el armado de múltiples conexiones de forma simultánea contra el nodo externo.

- Dada esta única conexión activa, se deberá mantener una sesión lógica contra el nodo provisto por la compañía de telefonía celular. Para ello se deberá enviar una transacción de ***Echo*** cada 60 segundos**,** y una transacción de ***Log In*** cuando el sistema arranca, o cuando se detecta un estado de desconexión.
- Los estados de desconexión surgen eventualmente cada vez que no se recibe respuesta de un ***Echo*** luego de 60 segundos de enviada la transacción correspondiente.
- El protocolo de conexión contra el nodo externo es ISO8583 1987 – BASE 24 (con 128 campos).
- Debe poder recibir conexiones múltiples de forma simultánea desde el *switch*, escuchando en un puerto TCP.
- El protocolo de conexión contra el *switch* es propietario, y es el utilizado de forma común entre el *concentrador* y otros *adaptadores*. De este modo se separan responsabilidades entre el *concentrador* y los demás *adaptadores*.
- Se debe vender un único producto, sin control de código de área telefónica, y sin control de montos máximos ni mínimos, ni de control por montos distribuidos (montos predefinidos, por ejemplo $20, $50, etc.)
- Se debe codificar el proyecto usando el lenguaje C#, versión .NET 2.0

6.3 Diseño Usando el DSL

La solución se desarrollará bajo un *framework* .NET 2.0 compatible, utilizando el *framework TransactionKernel* y el DSL que se presentó en este trabajo. El diseño de la solución constará de dos proyectos, uno del tipo *Biblioteca de Clases / Librería* (.dll) con toda la lógica propia del *adaptador*, y otro segundo proyecto del tipo *Ejecutable* (.exe) que podrá funcionar tanto como servicio de *Windows*, como aplicación de consola.

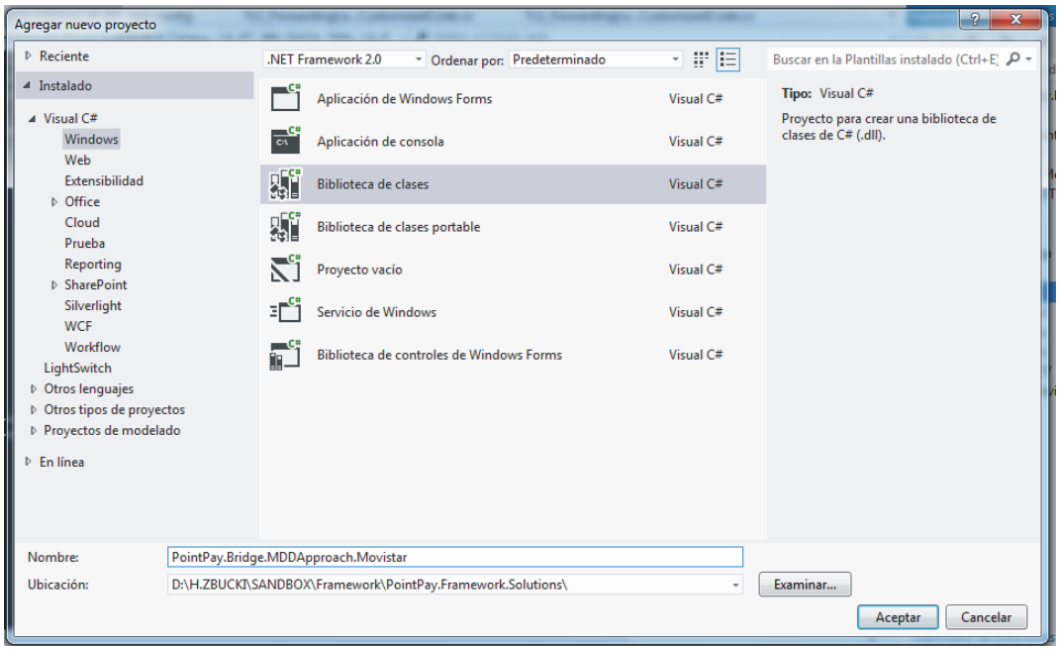

Figura 78 – Selección del tipo de proyecto en el entorno *Visual Studio 2012 Ultimate*.

Se usará el entorno *Visual Studio 2012 Ultimate* con la extensión *NuGet* y el SDK de *Visualización y Modelización* para este IDE. Por último, se instaló la extensión *PointPay.Framework.Language*, que contiene la definición del DSL y la herramienta de transformación de modelo a código. De este modo, el IDE puede agregar a cualquier proyecto los archivos de modelo (con extensión *.ppaydsl*) que entienden los elementos y las reglas de dominio definidas y descritas en el capítulo anterior. Si bien el DSL fue construido con *Visual Studio 2010 Ultimate*, la extensión generada fue instalada en otra computadora, con la versión 2012. Durante este capítulo se describirá el armado del proyecto *Librería*, omitiendo a propósito el armado del proyecto *Ejecutable*.

El primer paso es el armado de un proyecto de *Biblioteca de Clases / Librería*. Para lo cual haciendo *click* en *Archivo -> Nuevo -> Proyecto.* Se elige la carpeta y el nombre de la solución. Por último se debe elegir el tipo de proyecto, del tipo *Visual C# -> Windows -> Biblioteca de Clases*, como se muestra en la Figura 78. Eventualmente hay que configurar la versión de .NET que se decida utilizar; para esta solución se utiliza versión 2.0.

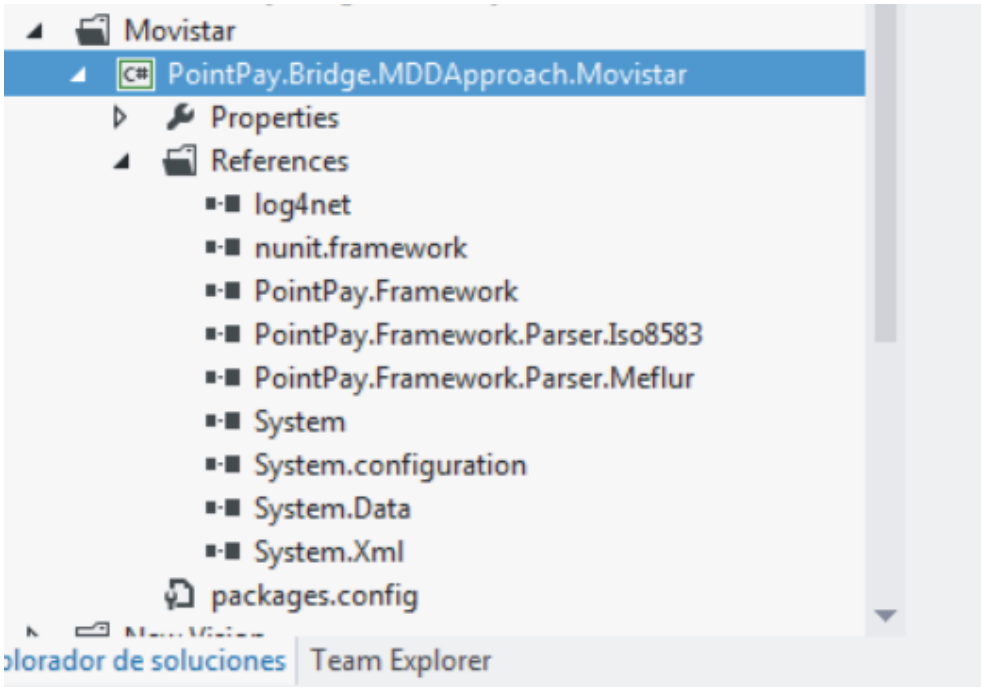

Figura 79 – Referencias a otras librerías dentro del esquema del proyecto.

Una vez creado el proyecto, en el *Explorador de Soluciones* (donde figuran los archivos de una solución o de los proyectos dentro de la solución), se mostrarán los distintos archivos agregados al proyecto de forma automática, como también las referencias a otras librerías básicas para poder asegurar la primera compilación del proyecto. Particularmente en este caso, y como se desea utilizar el *framework TransactionKernel* de modo de soportar los elementos de dominio comunes transaccionales recolectados, se deben agregar las siguientes referencias adicionales:

- *log4net*: Como ya se describió con anterioridad, es una librería de licencia gratuita que facilita las tareas de *logging*, y que contiene la implementación a la cual el *framework TransactionKernel* hace referencia cada vez que intente *loguear* un mensaje hacia el exterior. Esta librería se puede obtener del administrador de paquetes *NuGet* dentro del IDE, o desde su página de *internet* (*http://logging.apache.org/log4net/*).
- *nunit.framework:* Es una librería gratuita que implementa una serie de atributos decoradores, habilitando la posibilidad de escribir código de pruebas unitarias para implementar metodologías TDD. *PointPay.Framework.Language* necesitará de esta librería como dependencia, dado que entre las transformaciones programadas para un modelo dado, por defecto cada transacción conectada a *motores de entrada* tendrá asociados dos casos de prueba, uno para el caso de un resultado de aprobación de la misma, y el otro para simular el caso de rechazo. Esta librería también se puede obtener del administrador de paquetes *NuGet* dentro del IDE.

- *PointPay.Framework:* Es el nombre particular de la librería donde está implementado el *framework TransactionKernel,* dado que el proyecto surgió como una necesidad específica para la empresa *PointPay*. Esta librería contiene el código de todas las clases jerarquizadas que componen al marco de trabajo, y donde el DSL se reflejará cuando codifique las clases de usuario de esta solución. Por lo tanto es dependencia para poder utilizar *PointPay.Framework.Language*.
- *PointPay.Framework.Parser.Iso8583:* Esta librería contiene una implementación específica (basada en la clase *AbstractTransactionParser* de *TransactionKernel*) para el protocolo transaccional *Iso8583,* que se usará particularmente en esta solución para comunicar el *bridge* que se está por construir con un nodo externo.
- *PointPay.Framework.Parser.Meflur:* Esta librería contiene otra implementación específica para un protocolo propietario llamado *Meflur*, que comunica en este caso a un *switch* (concentrador) con varios *bridges,* incluyendo a éste que se está por construir. También está basado en la clase *AbstractTransactionParser* de *TransactionKernel.*

Una vez agregadas estas referencias, el proyecto contiene todas las dependencias mínimas cumplidas para empezar a modelizar el *bridge* a través del IDE de forma gráfica. Solo resta ajustar los archivos iniciales agregados en el proyecto y agregar un archivo del tipo apropiado para modelar el sistema. Primero, hay que seleccionar cualquier archivo de código preexistente sobre el raíz del proyecto (probablemente cuando uno genera la *Biblioteca de Clases* por primera vez, se agregue automáticamente el archivo *Class1.cs* o *Class1.vb*). Si este archivo existe, eliminarlo. Luego hacer *click* derecho sobre el proyecto, *Agregar -> Nuevo elemento*. Se abrirá una pantalla de diálogo para seleccionar el tipo de elemento que deseamos agregar, eligiendo convenientemente el tipo de archivo *PointPay.Framework.Language,* y escribimos el nombre del archivo del modelo, como muestra la Figura 80.

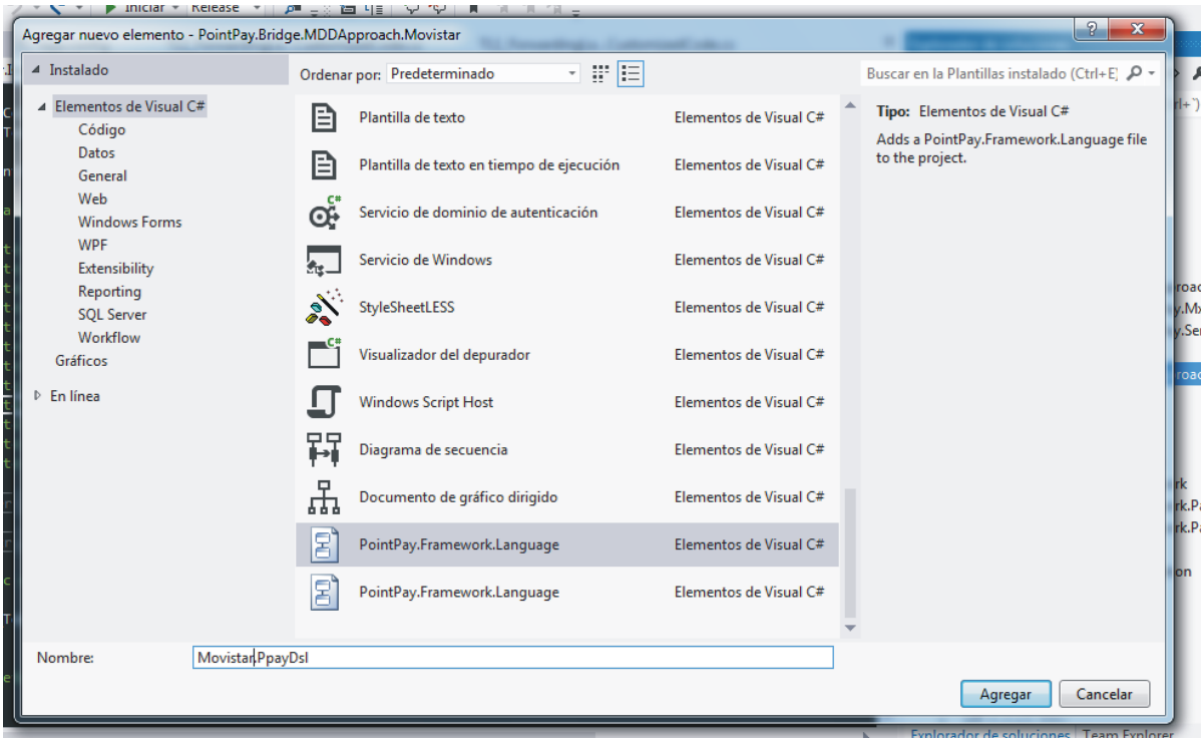

Figura 80 – Agregado de un modelo del tipo DSL – PointPay.Framework.Language

Una vez agregado el modelo, hacemos doble *click* en el archivo para abrirlo. Inicialmente el modelo viene con dos capas, una de *transacción* (*TransactionLayer1*) de nivel 0 (cero), y otra de *entorno* (*TransactionEnvironmentLayer1*). Esta última no maneja niveles. Sobre estas dos capas (en principio) se deberán agregar todos los elementos de dominio transaccionales definidos para describir a la solución que estamos construyendo.

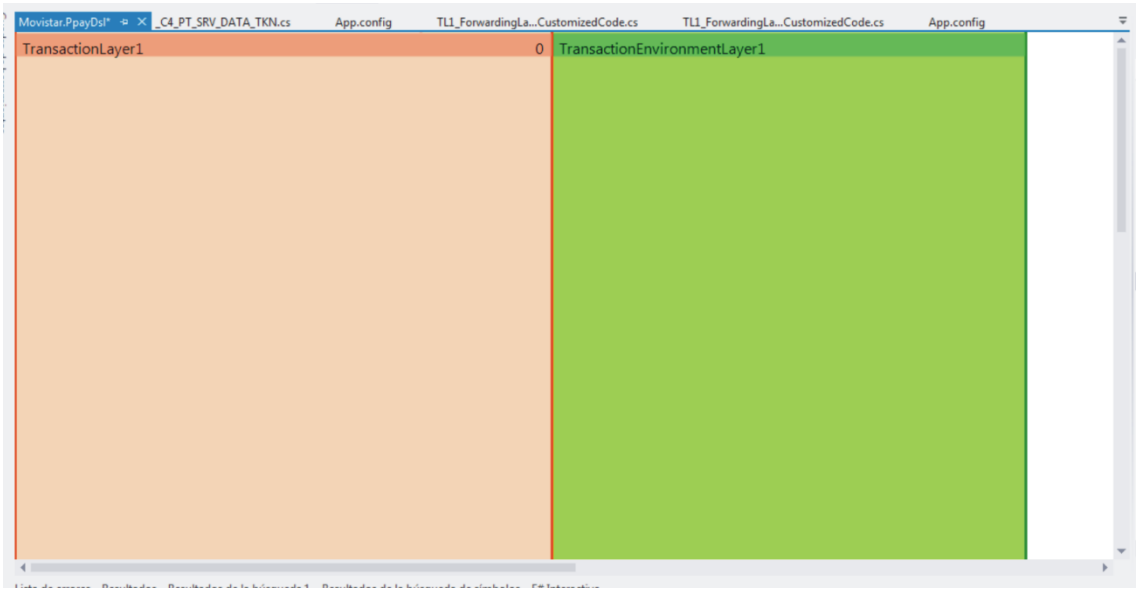

Figura 81 – Modelo del DSL inicial, recién agregado al proyecto.

La primera modificación es la de renombrar convenientemente los nombres de las capas. La capa transaccional, la renombramos a *Listener Layer (capa de escucha),* mientras que la de entorno la llamamos *Environmental Layer (capa de ambiente).*

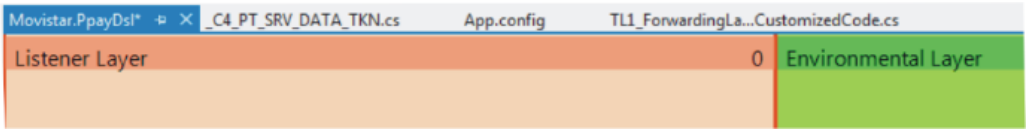

Figura 82 – Capas de transacción renombradas convenientemente.

El siguiente paso es el de especificar los valores principales del modelo, para lo cual hay que acceder a la pestaña de *Propiedades*. Para abrir esta pestaña, hay que hacer *click* derecho sobre algún punto del modelo (donde no haya capas transaccionales o de entorno). Luego hacer *click* sobre *Propiedades*. Los valores dentro de esta sección de *Propiedades* fueron descritos oportunamente en el capítulo anterior como propiedades del *Modelo Base.* A continuación se explican aquellos valores que se modificaron.

- *Get Value / Sequence Implementation Type*: Se seleccionó el valor de enumeración *SQLServerDatabase*, para persistir los valores de configuración del sistema y del secuenciador. Cada mensaje que salga del adaptador hacia el nodo externo debe generar un valor numérico de secuenciador. Además todos los valores que configuran a la instancia (como el puerto TCP de escucha, la dirección IP donde se alojará, el directorio de *log*, etc.) se guardarán en una tabla CONFIGURACION dentro de una base de datos manejada por un motor *SQLServer.*

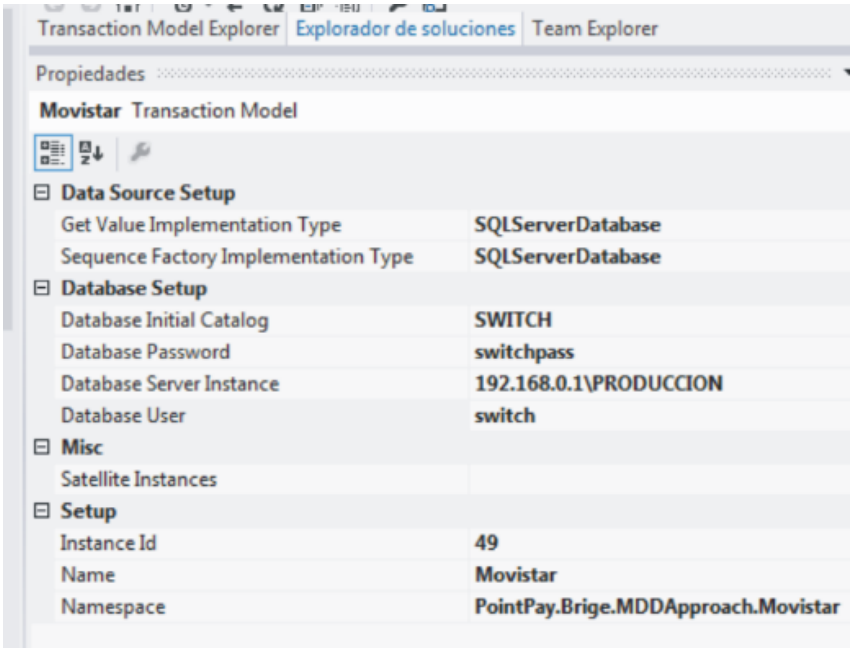

Figura 83 –Personalización de datos esenciales del modelo

- *Database Setup*: Se configuran los valores mínimamente necesarios para construir una *connection string* contra la base de datos.
- *Setup*: Aquí se asignó el *número de instancia* 49 para identificar a la solución entre otras más. Todos los valores de configuración en la tabla para tal fin de la DB estarán indexados por este valor, y así los valores configurados podrán ser obtenidos solo por este *bridge*. El *nombre* que se le dio fue *Movistar* dado que el *bridge* adaptará pedidos para un nodo externo que provee este producto. Por último el *namespace* que se definió será el que utilizará la herramienta automática de transformación a código para que cada uno de los archivos generados pertenezcan lógicamente a ese espacio de definiciones y nombres. Los valores de la sección *Setup* son todos obligatorios, por lo que en el DSL se tuvo que definir una pre-validación que verifique que estos tres valores se encuentren definidos antes de comenzar con las tareas de transformación a código funcional.

Ahora se comenzará a agregar elementos de dominio al modelo. Por lo que expandiendo la pestaña del *Cuadro de Herramientas* de *Visual Studio*, en la sección *Engines*, se debe elegir la herramienta *Input Transaction Engine Tool*. Esta herramienta permite agregar al modelo cualquier tipo de *motor de entrada,* por lo que desplazando la herramienta en la *capa transaccional de escucha (Listener Layer)* se consigue agregar un *motor de entrada* básico, listo para configurar, como se ve en la Figura84.

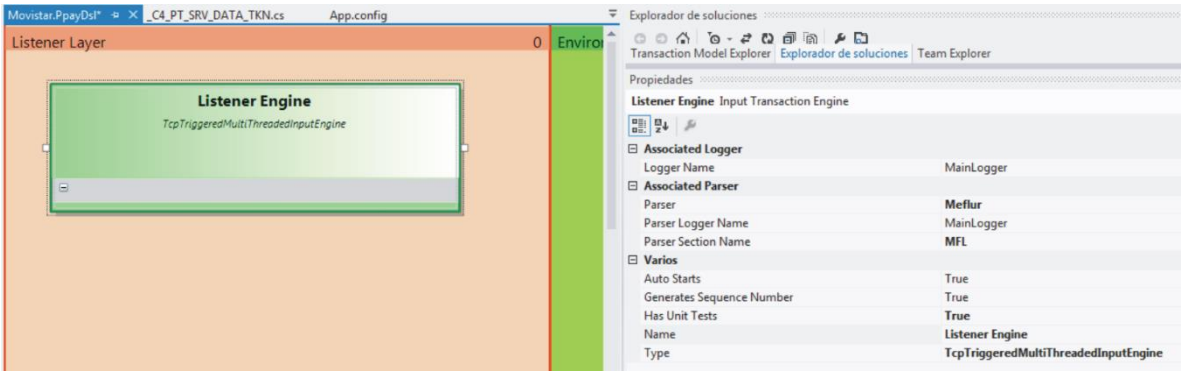

Figura 84 – Agregado y personalización de un *motor de entrada*

El siguiente paso es personalizar el *motor de entrada* a conveniencia de la solución. Renombramos al *motor de entrada* como *Listener Engine*, ya que servirá como *entrada* para el servidor transaccional. Es decir, escuchará por una dirección IP configurable, por un puerto TCP también configurable, y se seleccionarán tramas de datos que sean compatibles con el protocolo configurado. Los valores modificados son los siguientes:

- *Associated Parser:* Se modificó el tipo de *parser* asociado al *motor de entrada*, indicando el valor *Meflur,* dado que ese es el protocolo que queremos que interprete el mismo. El valor *Parser Section Name* se modificó al valor *MFL*; a través de ese valor en el XML de configuración inicial se puede definir un *elemento tag MFL* que contenga qué significa cada campo del protocolo *Meflur* para esta instancia en particular. Recordar que *Meflur* es un protocolo propietario que tiene una cantidad de campos expandibles (cantidad no fija) y que el contenido lógico que viaje por esos campos queda a definición de cada solución en particular. Entonces a través de ese XML se puede definir qué contenido viaja en cada posición de la trama.
- *Various:* De esta sección se modificó:
    - *Has Unit Tests*, ya que se precisaba que cada una de las transacciones que se vinculen con este *motor de entrada* tuvieran dos casos de prueba unitarios (un caso para simular una aprobación de la transacción y otro para simular un rechazo).
    - *Name,* el nombre del motor paso a ser *Listener Engine.*
    - *Type,* ya que precisamos que el motor genere un *thread* por cada conexión TCP entrante. De este modo, la clase del motor autogenerada heredará de la clase del *framework AbstractTcpTriggeredMultiThreadedInputTransactionEngine*.

Una vez configurado el motor, se decidió agregar una segunda capa transaccional que permita configurar otros elementos del dominio en pos de redireccionar las peticiones entrantes hacia el nodo externo. Entonces se hizo *click* derecho sobre la capa *Environmental Layer, Agregar antes -> Transaction Layer*.

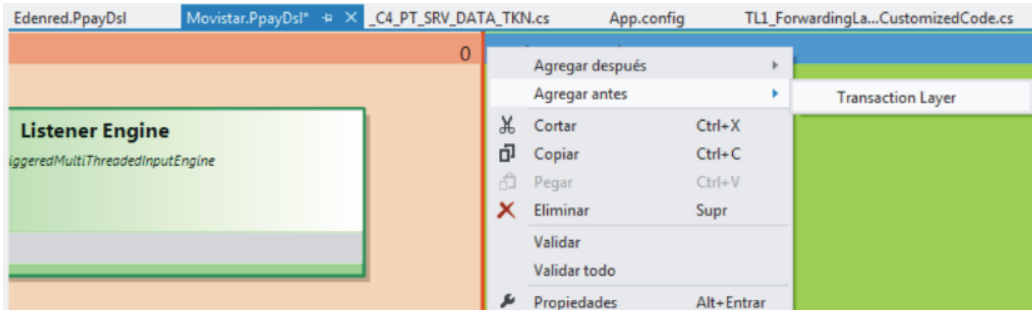

Figura 85 – Agregado de una segunda capa transaccional

Esta segunda capa transaccional tendrá todos los elementos de dominio que tienen por responsabilidad (dentro del *bridge*) el reenvío de las peticiones, mientras que aquellos elementos en la capa transaccional de escucha (*Listener Layer*) tienen como responsabilidad el proceso de las peticiones entrantes desde el *switch* y el armado de la respuesta hacia el *switch*. La segunda capa transaccional se renombró a *Forwarder Engine* (aunque hubiera sido más representativo haberla llamado *Forwarder Layer*), y se eligió el *nivel de capa* en 1 (uno).

Luego se expandió el *Cuadro de Herramientas*, y dentro de la sección *Engines* se eligió la herramienta *Output Transaction Engine Tool*. Se desplaza la herramienta hacia la *capa de reenvío* y se genera un *motor de salida* genérico. Del mismo modo como se hizo con el *motor de entrada,* se abren las *Propiedades* del *motor de salida* y se renombra a *Funneled Forwarder Engine*.

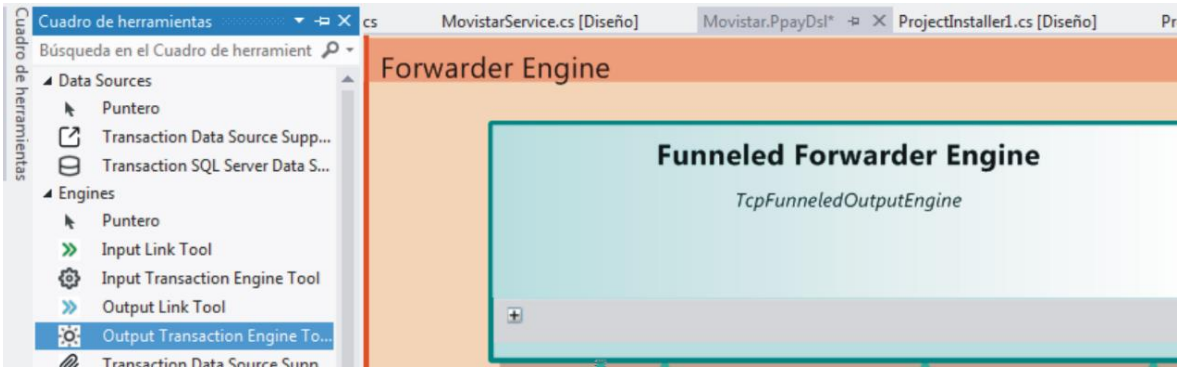

Figura 86 – Agregado de un motor de salida dentro de la segunda capa transaccional

Así también se elige el tipo *TcpFunneledOutputEngine*, ya que como se pedía en los requerimientos de la solución, todas las transacciones salientes deben salir por un único socket TCP que se mantiene conectado (en el mejor de los casos) todo el tiempo con el nodo externo. Es decir que todos los *threads* creados por el *motor*

de entrada, de algún modo tienen que sincronizarse para encolar las peticiones hacia un único punto, que es este *motor de salida*. De ahí sale el término *embudo* o *funnel* en inglés.

Este *motor de salida* es complejo, y para poder encolar los mensajes este tipo de motor dispone de un conjunto de métodos virtuales y métodos abstractos que deben ser implementados para asegurar el correcto ordenamiento de los mensajes entrantes y salientes. Para facilitar la codificación a través de la mayor cantidad de automatización posible, se desarrolló un elemento que usa mecanismos de persistencia conocidos para dar soporte a otros elementos de dominio, en este caso a los *motores de salida* de este tipo. Desde el *Cuadro de Herramientas* se arrastra el elemento *Transacion Data Source Support,* deslizándolo hasta la capa transaccional de reenvío. Se renombra a *Funneled Engine Support*, y en las *propiedades*, se modifica el campo *Support Type*, indicando que queremos que el mecanismo de persistencia sea del tipo *SQLServerSupport*.

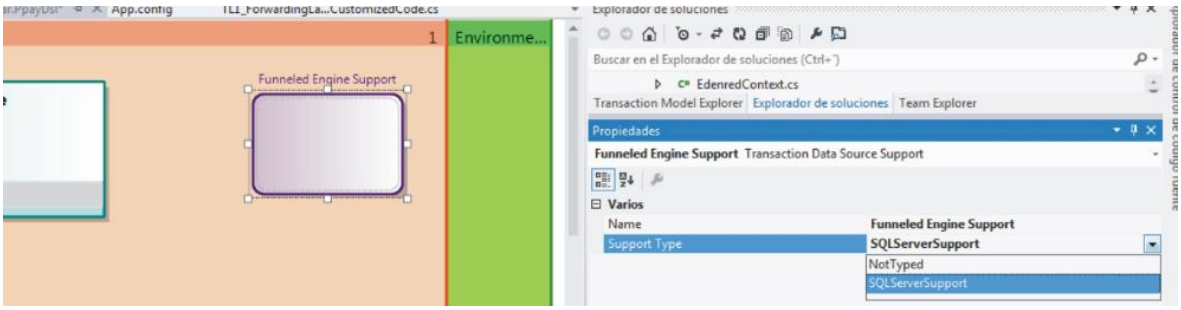

Figura 87 – Agregado de un soporte de acceso a datos para personalizar algunas características del motor de salida.

También desde el *Cuadro de Herramientas* se elige la herramienta *Transaction Data Source Support For Output Engine Link*. Esta herramienta es el conector válido para vincular los *motores de salida* con los *soportes de datos*. Con esta conexión se vinculan entonces el motor *Funneled Forwarder Engine* y el soporte *Funneled Engine Support*.

Figura 88 – Vinculación del motor de salida con el soporte de acceso a datos.

Dado que por requerimientos se pedía que por la conexión entre el *bridge* y el nodo externo se debían enviar *Echos* y *Logins* de modo de mantener *viva* la conexión a nivel lógico, será necesario programar algún elemento que dispare de forma temporal las transacciones que cumplan con estas tareas. Es por eso que en la *capa*

transaccional de escucha se agrega un segundo *motor de entrada,* que renombramos a *Network Related Messages Engine.* Este segundo *motor de entrada* será del tipo *TimeTriggeredInputEngine*, por lo que no abrirá ningún socket TCP de escucha: solo escuchará a que algunos eventos temporales indiquen que es el momento para instanciar alguno de los manejadores de transacción, que se vincularán convenientemente al motor en una etapa posterior.

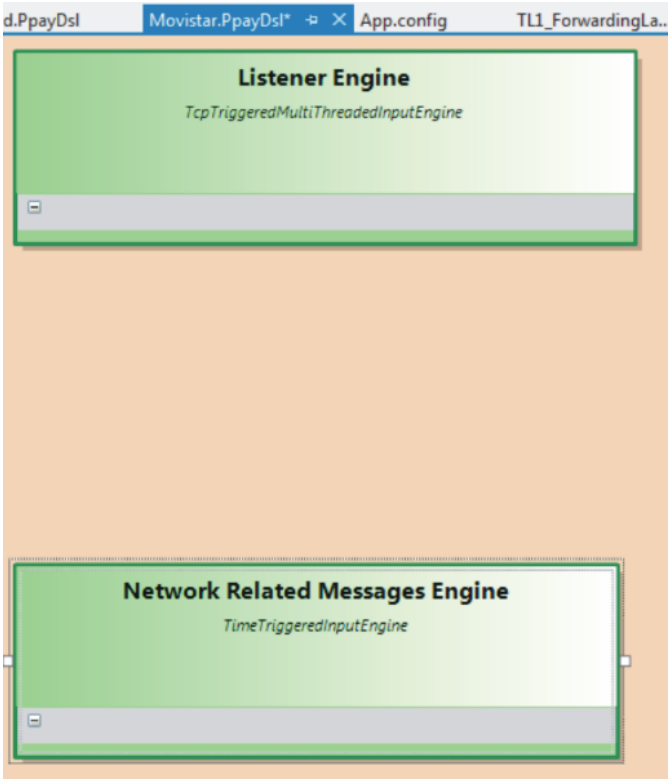

Figura 89 – Agregado de un segundo motor de entrada en la primera capa transaccional

Entonces, dado que necesitamos contar con dos *timers* independientes que le indiquen a este motor cuándo instanciar los manejadores de *Echo* y de *Login,* en el *Cuadro de Herramientas* se debe seleccionar la herramienta *Time Trigger Tool* y se deben desplazar dos *timers* dentro de la *capa transaccional de escucha*. Estos dos *timers* los renombramos a *EchoTrigger* y *LoginTrigger*. Luego volviendo al *Cuadro de Herramientas* se elige la herramienta *Time Trigger Link Tool*, que permite vincular a *motores de entrada* con los *Time Triggers*. Existe una post-validación sobre el modelo que valida que los *motores de entrada* vinculados con los *Time Trigger* efectivamente sean del tipo correcto (*TimeTriggeredInputEngine*). Para este caso, se vinculan los disparadores *EchoTrigger* y *LoginTrigger* con el motor *Network Related Message Engine*, como se muestra en la Figura 91.

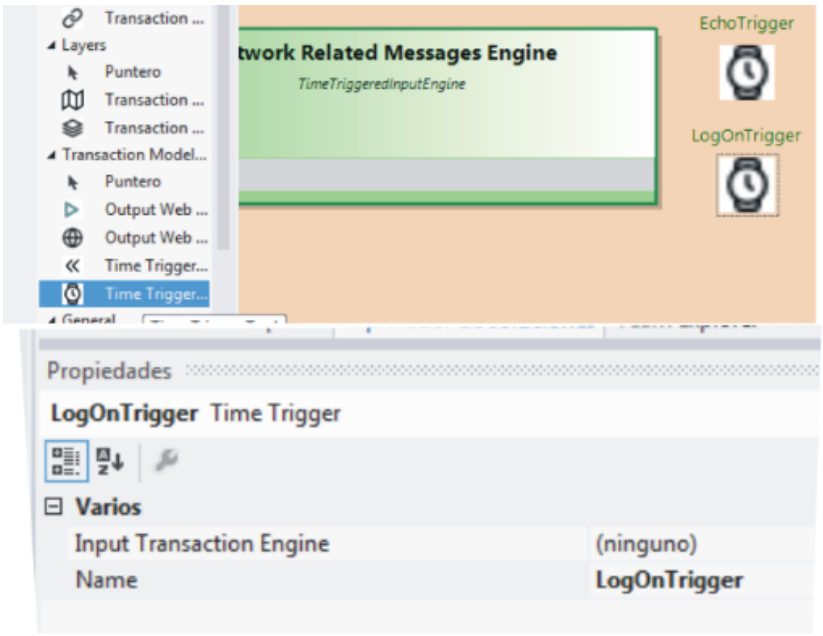

Figura 90 – Agregado de dos timers que irán conectados con el motor de entrada disparado por eventos temporales.

Ahora es el momento de agregar los *manejadores de transacción* al modelo. Se aclara al lector que estos manejadores representan a una transacción en un estado, o momento dado. Por ejemplo, los *manejadores de transacción* que se agreguen a los *motores de entrada* representarán a la transacción real cuando la misma está siendo recibida por el *bridge*, o cuando se esté armando la respuesta y respondiendo consecutivamente al *switch*. En cambio, aquellos *manejadores de transacción* vinculados a la *capa de transacción de reenvío* representan a la transacción real en su estadía por la etapa de reenvío.

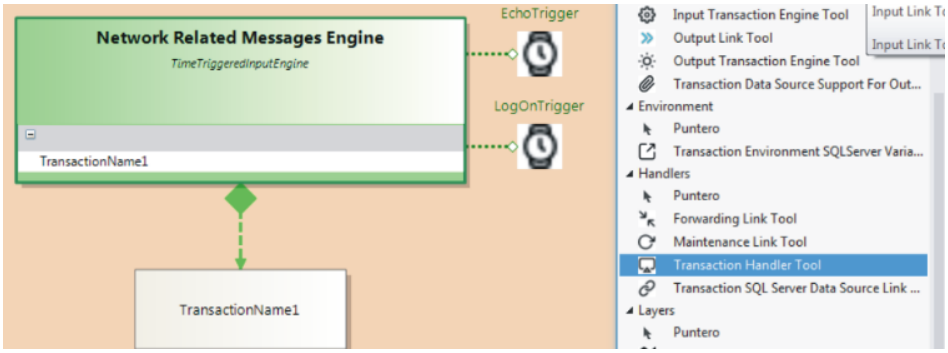

Figura 91 – Vinculación del motor de entrada con los dos timers configurados.

Del *Cuadro de Herramientas* se debe elegir la herramienta *Transaction Handler Tool* y desplazarla a la *capa de transacción* deseada, en este caso la *capa transaccional de escucha*. Con la herramienta *Input Link Tool* se pueden vincular *manejadores de transacción* con *motores de entrada*. Entonces se vincula el manejador

llamado por defecto *TransactionName1* con el motor *Network Related Message Engine*. En la figura 92 se muestran las propiedades del manejador, que fue renombrado a *NRM.* Los dos campos más importantes del manejador son los que están definidos en la sección *Setup;* uno es el *nombre* (*Name*) y el otro es el *Transaction Id* que se configuró en 0800. Esto quiere decir que el motor entenderá que tiene que recibir solamente (o en este caso, solo generar) transacciones cuyo identificador es el 0800. Como ya se explicó en capítulos anteriores, para poder reconocer el *Transaction Id* dentro de una trama de entrada TCP, el archivo de configuración XML de la solución contiene un atributo que permite indicar cuáles campos definidos para una trama son *identificadores de la transacción*. Entonces, una vez desarmado el mensaje entrante, el motor busca en esos campos y compara con todos los manejadores vinculados al mismo, instanciando eventualmente a la clase correcta.

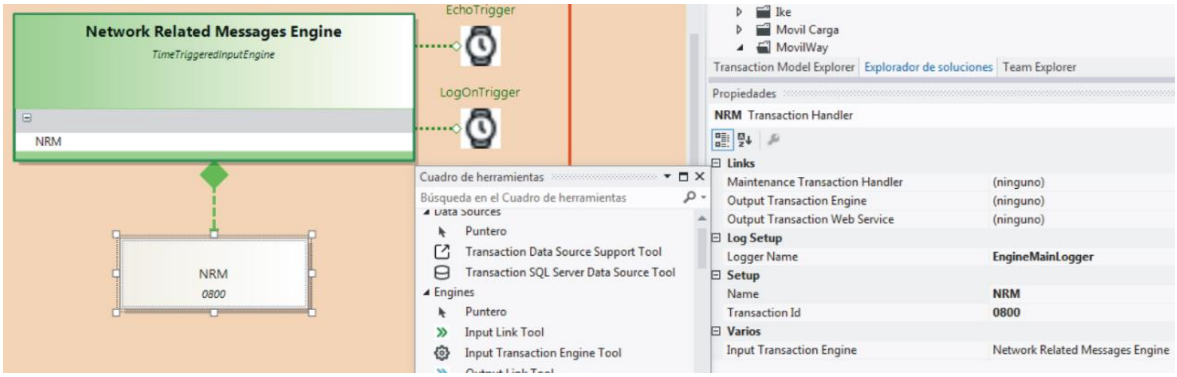

Figura 92 – Agregado de un manejador de transacción vinculado al motor de entrada disparado por eventos temporales.

Peculiarmente para esta solución, al manejador *NRM* se le configuró un *LoggerName* distinto al que viene por defecto. Esto quiere decir que todos los mensajes de *log* que la transacción genere serán enviados a un archivo de *log* distinto al configurado por defecto. Esto se eligió así para poder separar el *log* de *NRM* del *log* generado por los *manejadores de transacciones* de peticiones reales entrantes (los manejadores vinculados al *motor de entrada Listener Engine*).

Este manejador, instalado en la *capa transaccional de escucha* representará a la transacción real *NRM* (0800) durante una etapa particular, o en un estado particular. En este caso representará a la transacción durante las etapas de llegada y de salida al sistema. Si bien físicamente no llegó una transacción por un socket (dado que el motor está siendo disparado por eventos temporales), en este manejador se deben capturar los datos de una trama ficticia, que será entregada por el motor *como si hubiese provenido de un switch*. Entonces es en el manejador donde se deberán capturar estos datos valiosos. Ahora bien, como se comentó cuando se describió el propósito de los *manejadores transaccionales*, una transacción real puede estar representada por varios manejadores. En la Figura 93 se agrega otro *manejador transaccional* para la misma transacción *NRM* (0800),

pero que se ubicará en la *capa transaccional de reenvío*. Este manejador tendrá como propósito implementar el reenvío de la transacción al nodo externo, preparando el requerimiento según el protocolo convenido con ese nodo, y recibir correctamente una eventual respuesta desde allí. Para lograr esto, se vincula el *manejador transaccional* recién agregado con el *motor de salida* ya instanciado en el modelo en pasos previos.

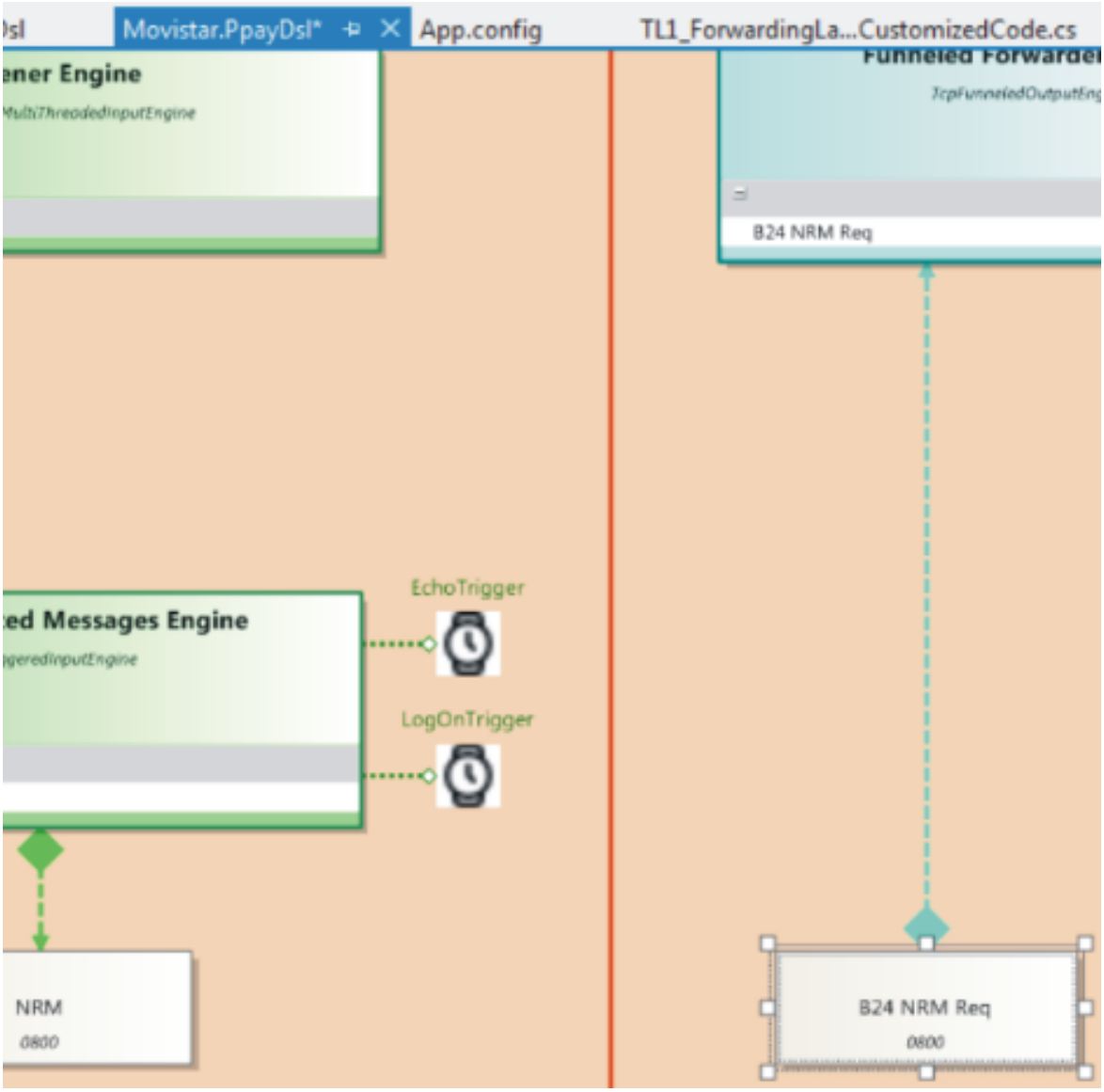

Figura 93 – Agregado de un *manejador de transacción* vinculado al *motor de salida*

Continuando con el modelado, los manejadores transaccionales se pueden vincular entre sí de diferentes formas, en función de la conveniencia del usuario que modela el sistema. Para tal fin existen varios conectores diferentes analizados en el capítulo anterior, que terminan teniendo acciones distintas sobre el código funcional (luego de

la correspondiente transformación). En este caso, se utiliza el *conector de reenvío* para vincular a ambos manejadores.

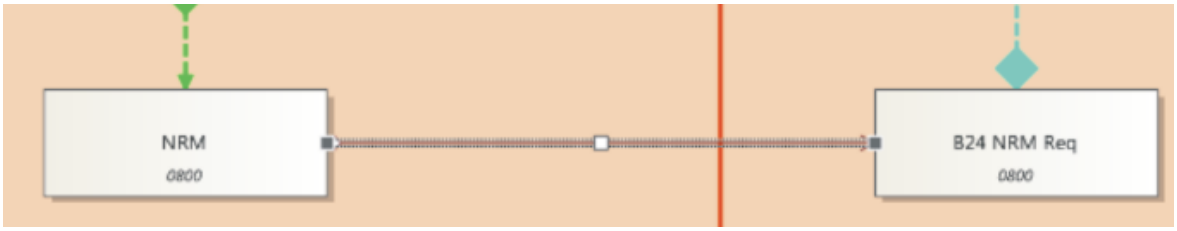

Figura 94 – Vinculación entre ambos manejadores de transacciones entre capas transaccionales.

El modelo entonces se interpreta de la siguiente manera: el método *ForwardHandlerFactory()* del manejador en la *capa de escucha* instanciará a la clase correspondiente del manejador en la *capa de reenvío*. Entonces, la segunda etapa (de la secuencia genérica de trabajo) se realizará usando las implementaciones armadas dentro del *handler* de la *capa de reenvío*, en vez de usar las implementaciones del primer manejador (el de la *capa de escucha*). En cuanto se termina la etapa de envío y recepción de datos entre este sistema y el nodo externo (propósito de la segunda etapa), la tercera etapa de la secuencia genérica de trabajo se ejecutará, retornando a la implementación del primer manejador (el de la *capa de escucha*). Correspondientemente se procesará y armará la respuesta en ese manejador, habilitado al *motor de entrada* a que devuelva los datos al punto de origen (si lo hubiera; en este caso no lo hay ya que el motor es del tipo *TimeTriggered*).

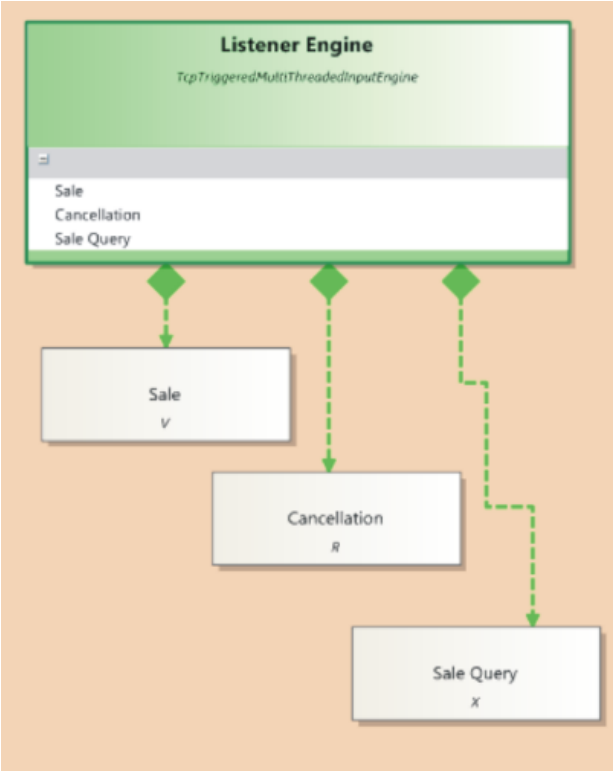

Figura 95 – Agregado de los tres manejadores de transacción requeridos, vinculados al motor de entrada principal.

En la Figura 95 se insertan otros *manejadores transaccionales* dentro de la *capa de escucha*, ahora si conectados al *motor de entrada* disparado por conexiones entrantes TCP. Este motor instanciará un manejador para que represente una transacción proveniente del *switch*, en función del *Transaction Id*. Se agregan los manejadores de *Venta (Sale), Cancelación (Cancellation)* y *Consulta de Transacción (Sale Query)*, según los requerimientos iniciales.

Por otro lado, se ingresan los *manejadores transaccionales* correspondientes a las tres transacciones (*Venta*, *Cancelación* y *Consulta de Transacción*), pero dentro de la *capa de reenvío.* Cabe acotar que los *Transaction Id* de los manejadores en una capa y la otra pueden variar, justamente porque tienen que representar a las transacciones en distintas etapas. Particularmente, en las interfaces *switch / bridge* y *bridge / nodo externo* los identificadores pueden ser distintos ya que los protocolos de comunicación pueden ser distintos.

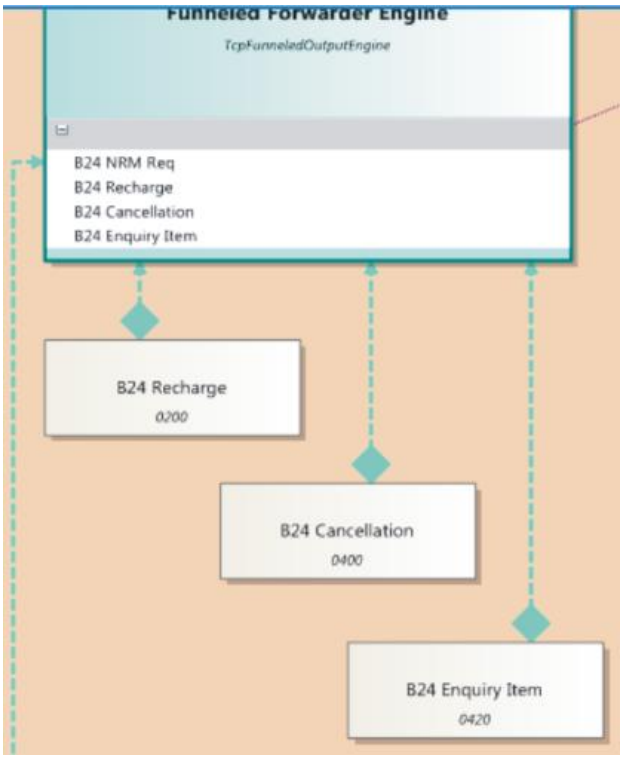

Figura 96 – Agregado de tres manejadores transaccionales vinculados al motor de salida.

Como sucedió en el caso de *NRM*, cada manejador en la *capa de escucha* se vinculará con el correspondiente manejador en la *capa de reenvío*. Los métodos *ForwardHandlerFactory()* dentro de los manejadores asociados al *motor de entrada* instanciarán las clases correspondientes a los manejadores en la última capa transaccional. La mecánica de la secuencia general de trabajo es la misma: la *segunda etapa* se ejecutará con la implementación dentro de los *handlers* en la *capa de reenvío*, mientras que la primera y tercera etapa se ejecutarán en la implementación dentro de los *handlers* de la *capa de escucha*, como se muestra en la Figura 97.

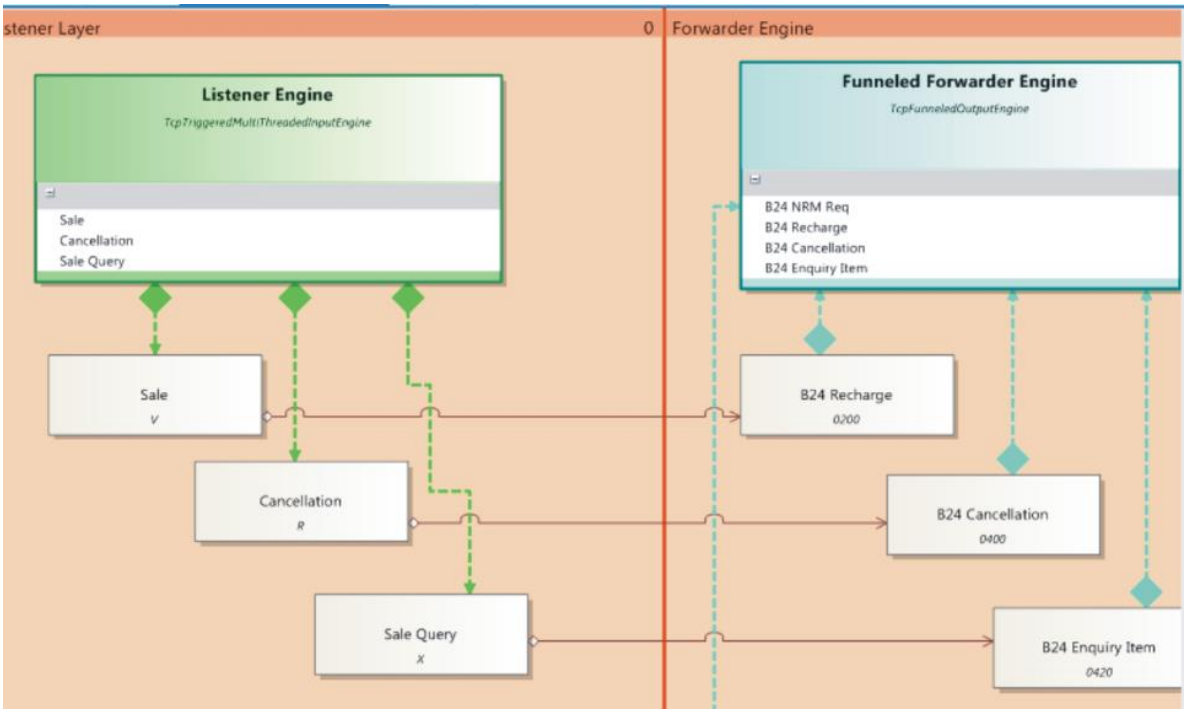

Figura 97 – Vinculación de los *manejadores transaccionales* de la *capa transaccional de escucha*, con los correspondientes en la *capa transaccional de reenvío*.

Con esta diagramación de elementos de dominio, estaría terminada la modelización del sistema, y solo quedaría *guardar el archivo* del modelo para que se ejecute la herramienta de transformación a código fuente. Sin embargo, se decidió agregar unas características extras al sistema previo a transformar el modelo a código. Dado que en la práctica no se logró conseguir un ambiente de pruebas durante las primeras fases del desarrollo, se optó por generar un simulador de respuestas del nodo externo. De este modo se podría solventar la falta de entorno de pruebas, programando algunas respuestas conocidas (o esperables). Más aún, en función de algún parámetro dentro de las transacciones que se envían a ese simulador (como por ejemplo el código de área del teléfono que se reenvía, o el monto de la transacción) se lo puede programar para que responda transacciones aprobadas, rechazadas y pendientes. De este modo se reutilizan técnicas propias de TDD para probar el sistema como una unidad singular con entradas y salidas.

Para tal fin, se agregó una tercera capa transaccional, que se renombró a *Simulator Layer*, o *capa de simulación*. Se colocó la capa con el nivel 2, como se muestra en la Figura 98.

Para poder recibir las transacciones entrantes desde la *capa de simulación*, se insertó un *motor de entrada disparado por conexiones TCP*, similar al usado en la *capa de escucha*, pero con una leve diferencia: el tipo del motor indica la frase *socket reusable*, es decir que el *thread* instanciado para atender a la transacción original, no cierra el socket en cuanto termina de procesarse la secuencia genérica de trabajo (es decir, cuando el método

DoTransaction() del manejador retorna para ceder la ejecución al *motor de entrada*). En cambio, el hilo de ejecución continua vivo, esperando la llegada de un nuevo requerimiento por el socket TCP abierto. Eventualmente el *thread* finaliza cuando el otro par conectado al socket cierra la conexión o cuando el método *Receive()* del *motor de entrada* sale por *Time-Out*.

Figura 98 – Agregado de una tercera capa transaccional para armar un simulador de respuestas provenientes del nodo externo.

Se eligió este tipo de motor para esta capa, dado que la *capa de reenvío* usa un *motor de salida* del tipo embudo (*TcpFunneled*….). De esta forma se puede simular el comportamiento de mantener una conexión viva. Convenientemente, se configuró el tiempo de disparo de la *transacción NRM* en un valor menor que el tiempo de *Time-Out* del motor del simulador, de modo que se generen *Echos* antes que este motor de por muerta la conexión. En la Figura 99 se muestra la configuración del motor en la *ventana de Propiedades*: cabe acotar que a diferencia de los otros dos motores ingresados, este motor tiene la propiedad *Auto Starts* en *False*. Esto se configuró así, ya que el simulador debe iniciarse de forma opcional, mientras que los otros dos motores son parte de la solución y deben iniciarse siempre que se inicie el servicio. Así también se le configuró el mismo

protocolo que el *motor de salida* de la *capa de reenvío*, y se le configuró un *logger* diferente, de modo que las líneas de *log* generadas por el simulador no se mezclen con aquellas generadas por el sistema en sí mismo.

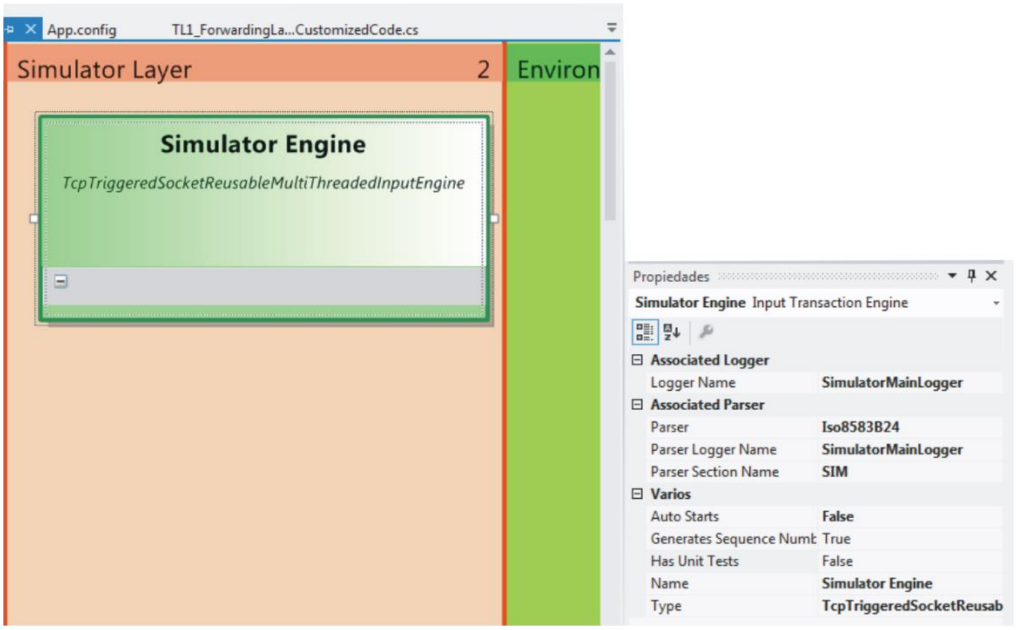

Figura 99 – Agregado de un motor de entrada en la capa transaccional del simulador.

En la Figura 100 se denota el agregado de cuatro *manejadores transaccionales* que se vinculan con el *motor de entrada* de la *capa de simulación*, representando a las transacciones *Venta*, *Cancelación*, *Consulta de Transacción* y *NRM* (*Echos* y *Logins*). Notar que no hay una cuarta *capa de reenvío de simulación*, porque en este caso, un solo *manejador de transacción* alcanzó para representar a la transacción real en su totalidad de estados / etapas: es decir, como llegan los datos al simulador, se arma la respuesta simulada y se envía la respuesta.

El *motor de salida* de la *capa de reenvío* es configurable desde la base de datos, dentro de la tabla *Configuracion*, como se explicó en párrafos anteriores. Es ahí donde se puede definir la dirección IP y el puerto TCP adonde deberá conectarse, y por ende, será ahí donde se decida si el motor efectivamente se conecta al nodo externo productivo, o al simulador que se acaba de modelizar. A su vez en esa misma tabla, la dirección IP y puerto TCP del *motor de entrada* de la *capa de simulación* deberá oportunamente coincidir con lo configurado por el del *motor de salida*, si se desean vincular a ambos por una red real. La lógica del simulador se configuró de modo que todas las ventas que tengan montos terminados en 0, den aprobadas, mientras que los montos terminados en 5, sean transacciones rechazadas. Además se configuraron dos teléfonos inexistentes en la realidad para simular casos de *Time-Out* del motor, e incumbencias en la red que resultaban de interés.

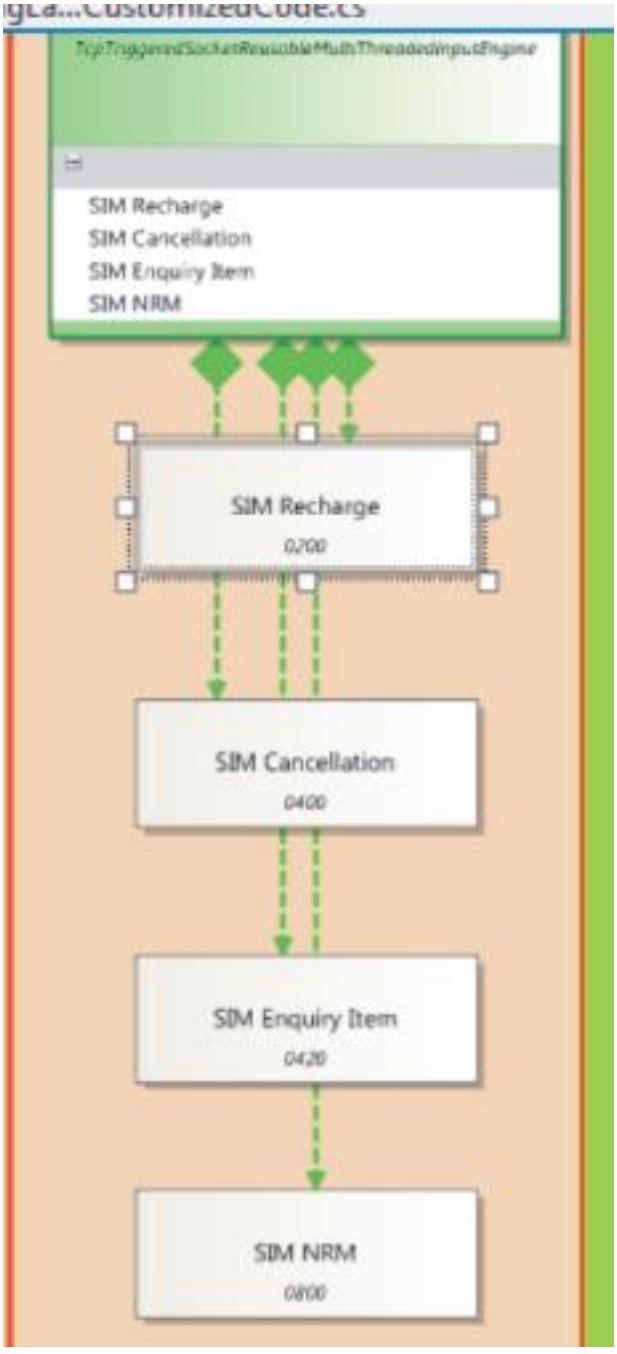

Figura 100 – Agregado de *manejadores transaccionales* que quedan vinculados al *motor de entrada* correspondiente al simulador.

En la Figura 101 se muestra un panorama general de los elementos de dominio agregados al modelo. Lo último que resta por diagramar son las variables de entorno de la solución, que serán almacenadas en la base de datos.

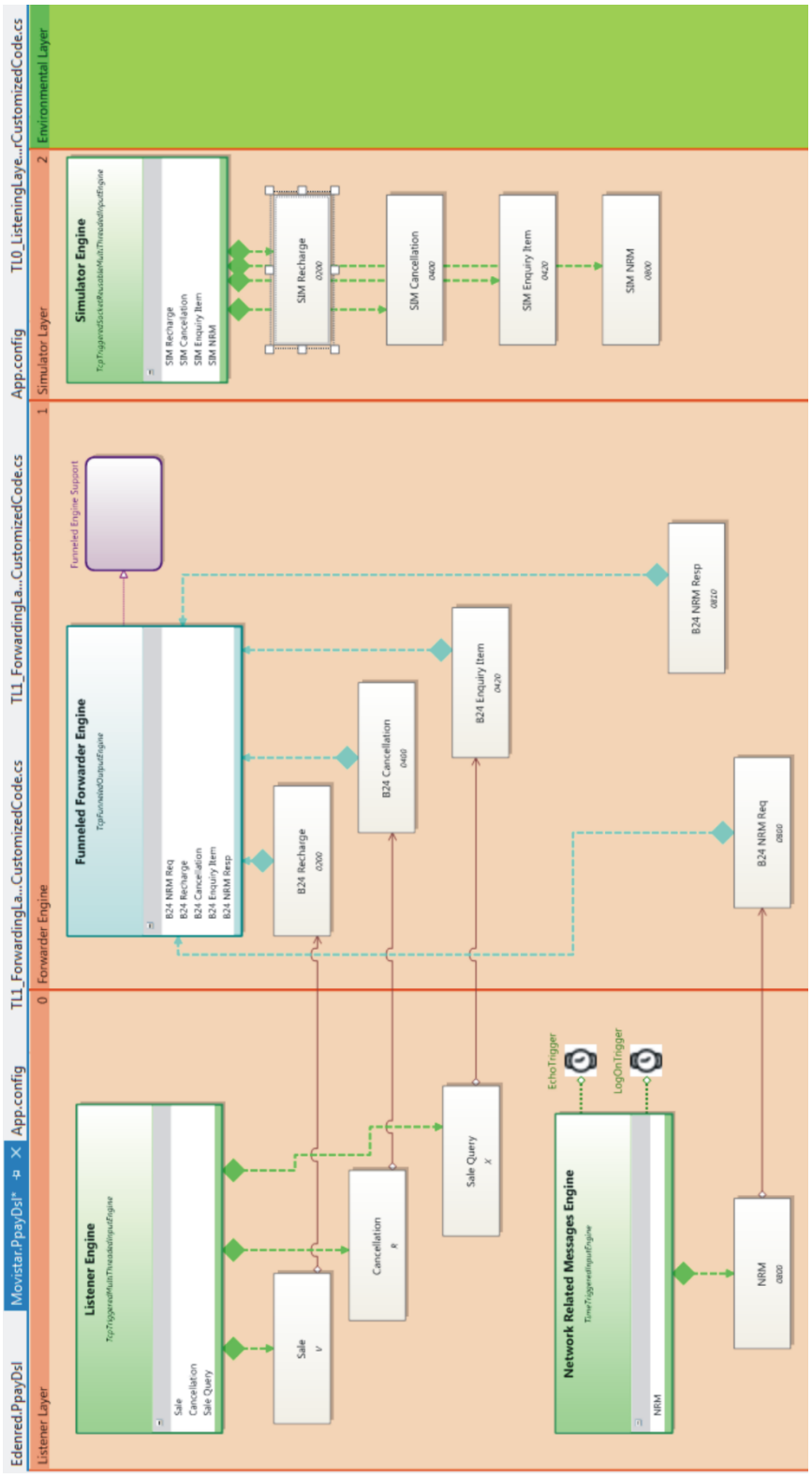

Figura 101 – Panorama general del modelo armado.

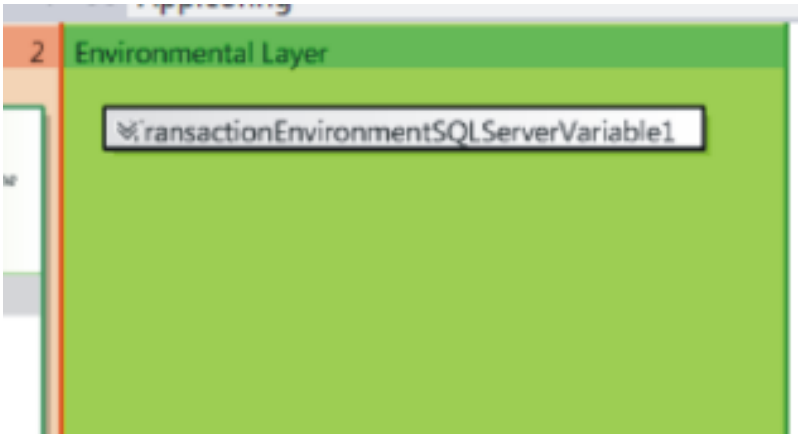

Figura 102 – Agregado de una variable de entorno transaccional

Para agregar una variable de entorno se debe abrir el *Cuadro de Herramientas*, y utilizar la herramienta *Transaction Environment SQLServer Variable*. De allí se debe arrastrar una variable de entorno hacia la *capa de entorno* de la solución (se recuerda al lector que solo puede haber una sola por solución, a diferencia de las *capas transaccionales*). Luego se debe renombrar a conveniencia la variable, y se debe configurar el valor inicial de la variable. Cada una de las variables de entorno ingresadas se transformará en *scripts* (para insertar el valor en la base de datos) y código fuente para acceder a esa variable.

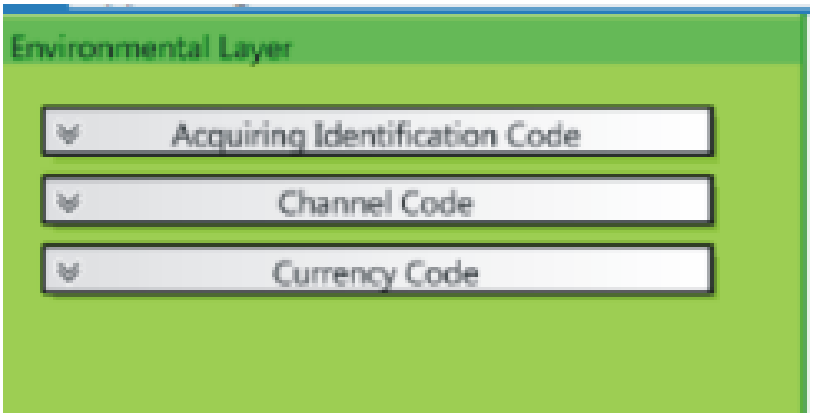

Figura 103 – Valores de entorno transaccional usados en esta solución.

Para esta solución se utilizaron tres variables de entorno (las que se muestran en la Figura 103), que permiten configurar unos campos en la trama de entrada / salida del protocolo ISO8583B24, pertenecientes a la interfaz *bridge / nodo externo* (o bien *bridge / simulador*).

6.4 Transformación a Código

6.4.1 Estructura de Archivos

Una vez ejecutada la herramienta de transformación automática a código fuente, se generaron los siguientes archivos, según la Figura 104.

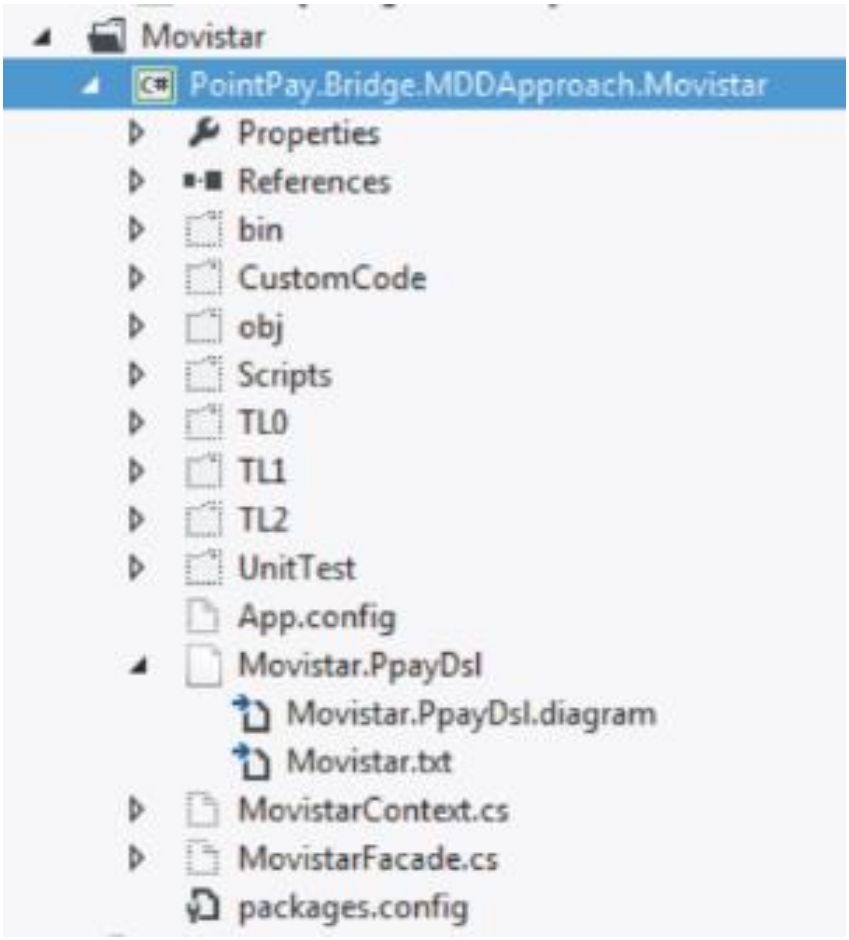

Figura 104 – Estructura de archivos generado en la solución.

- **TL0, TL1, TL2**: Carpetas por cada nivel de capa transaccional definido en el modelo *(capa de escucha*, *capa de reenvío* y *capa de simulación*)
- **MovistarFacade.cs**: Clase generada automáticamente, y sincronizada con el modelo en todo momento. Cualquier cambio en el modelo regenera este archivo. Contiene una clase *Facade* con definiciones del sistema y varios métodos para manipular el sistema (hereda de la clase *AbstractTransactionFacade* del *framework TransactionKernel*).

- **MovistarContext.cs**: Clase generada automáticamente, y sincronizada con el modelo en todo momento. Cualquier cambio en el modelo regenera este archivo. Contiene la clase que representa al contexto de datos definido para la solución, que hereda de la clase *AbstractTransactionContext*.
- **App.config**: Archivo XML de configuración de parámetros esenciales de la solución (por ejemplo, número de instancia, *connection strings*, configuración de campos de los protocolos, *stored procedures*, etc.). Este archivo se genera automáticamente y esta sincronizado con el modelo solo si no existe (es decir, los cambios de usuario agregados de forma posterior a la transformación no se pierden si cambia el modelo).
- **UnitTest:** Carpeta que contiene los archivos con los casos de pruebas unitarias.
- **Scripts:** Carpeta con los scripts SQL, que deben ejecutarse en el servidor donde se ejecutará el sistema transaccional. Estos scripts contienen los valores para la tabla CONFIGURACION, donde se configura el comportamiento de las instancias involucradas por el servidor transaccional bajo desarrollo.
- **CustomCode:** Contiene los archivos de código **MovistarFacade_UserCustomizedCode.cs** y **MovistarContext_UserCustomizedCode.cs,** pero con la diferencia que en esta carpeta es en donde los usuarios pueden modificar y/o agregar código sin peligro que la transformación automática (desde el modelo al código) elimine los cambios generados por ellos. Se recuerda al lector que se implementaron las clases *MovistarFacade* y *MovistarContext* como clases parciales, donde la definición se realiza en los archivos del directorio raíz y de este directorio (*CustomCode*).

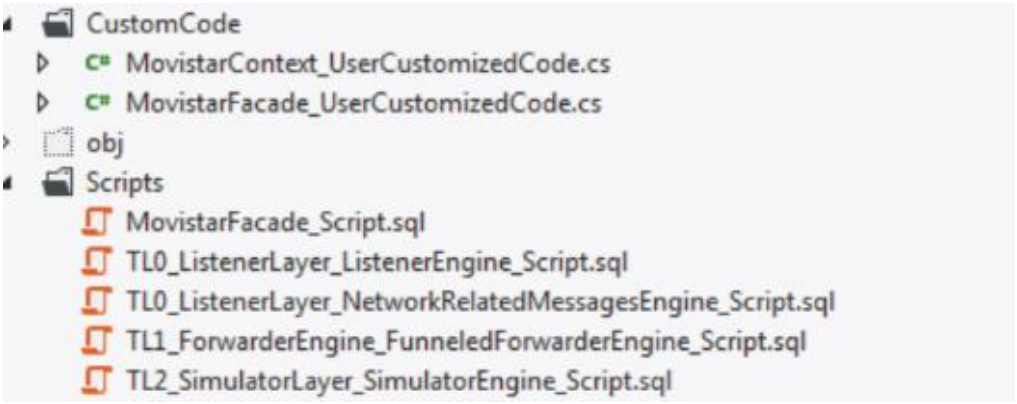

Figura 105 – Archivos dentro de la carpeta Script y CustomCode

Dentro de la carpeta TL0 se puede encontrar una carpeta (*CustomCode*) y los siguientes archivos:

- Los archivos de código de los manejadores creados por la herramienta de transformación para la *capa transaccional de escucha* (*Cancellation, NRM, Sale y SaleQuery*): *TL0_ListenerLayer_Cancellation_Handler*, *TL0_ListenerLayer_NRM_Handler, TL0_ListenerLayer_Sale_Handler, TL0_ListenerLayer_SaleQuery_Handler.*

- Los archivos de código de los motores definidos dentro de esta capa (*NetworkRelatedMessages* y *ListenerEngine*): *TL0_ListenerLayer_ListenerEngine_Engine.cs, TL0_ListenerLayer_NetworkRelatedMessagesEngine_Engine.cs*
- Los archivos de código de los motores y de los manejadores dentro de la carpeta *CustomCode*, donde los usuarios pueden agregar y/o modificar código sin perder cambios durante una próxima transformación del modelo.

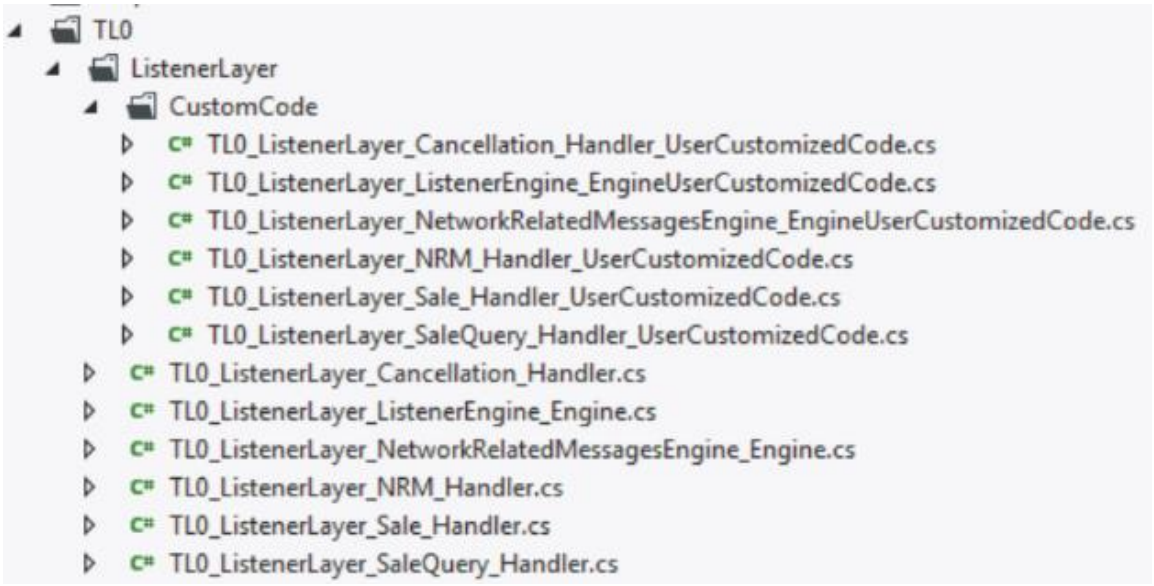

Figura 106 – Estructura de la carpeta TL0 (Transaction Layer 0)

Dentro de la carpeta TL1 se puede encontrar una carpeta (*CustomCode*) y los siguientes archivos:

- Los archivos de código de los manejadores creados por la herramienta de transformación para la *capa transaccional de reenvío* (*Cancellation, EnquiryItem, Recharge, NRM*): *TL1_ForwarderEngine_B24Cancellation_Handler.cs, TL1_ForwarderEngine_B24EnquiryItem_Handler.cs, TL1_ForwarderEngine_B24Recharge_Handler.cs, TL1_ForwarderEngine_B24NRMReq_Handler.cs, TL1_ForwarderEngine_B24NRMResp_Handler.cs*
- El archivo de código del *motor de salida* creado por la herramienta de transformación para la *capa transaccional de reenvío* (*FunneledForwarderEngine*): *TL1_ForwarderEngine_FunneledForwarderEngine_Engine.cs*

- Los archivos de código de los motores y de los manejadores dentro de la carpeta *CustomCode*, donde los usuarios pueden agregar y/o modificar código sin perder cambios durante una próxima transformación del modelo.

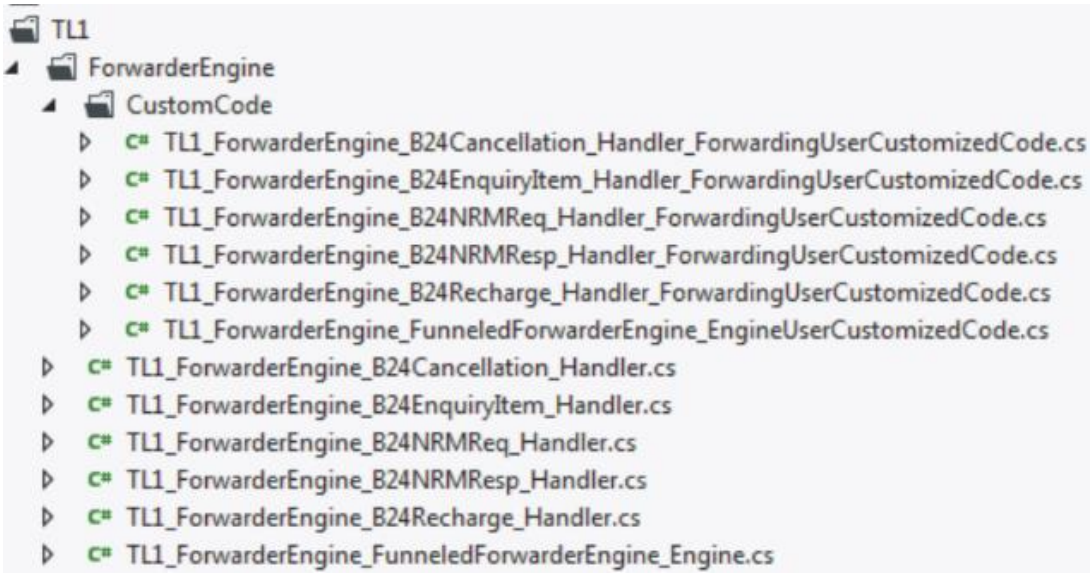

Figura 107 – Estructura de archivos dentro de la carpeta TL1 (Transaction Layer 1)

Dentro de la carpeta TL2 se puede encontrar una carpeta (*CustomCode*) y los siguientes archivos:

- Los archivos de código de los manejadores creados por la herramienta de transformación para la *capa transaccional de simulación* (*Cancellation, EnquiryItem, Recharge, NRM*): *TL2_SimulatorLayer_SIMCancellation_Handler.cs, TL2_SimulatorLayer_SIMEnquiryItem_Handler.cs, TL2_SimulatorLayer_SIMRecharge_Handler.cs, TL2_SimulatorLayer_SIMNRM_Handler.cs*
- El archivo de código del *motor de entrada* creado por la herramienta de transformación para la *capa transaccional de simulación* (*SimulatorEngine*): *TL2_SimulatorLayer_SimulatorEngine_Engine.cs*
- Los archivos de código de los motores y de los manejadores dentro de la carpeta *CustomCode*, donde los usuarios pueden agregar y/o modificar código sin perder cambios durante una próxima transformación del modelo.

Figura 108 – Árbol de archivos de la capa de simulación (TL2)

6.5 Evaluación de los Objetivos Planteados

A continuación se evaluarán dos sistemas, uno desarrollado previo a la implementación MDD (pre-metodología), y otro con el DSL que se describió en este trabajo. El propósito de esta tarea es conseguir alguna medida de cumplimiento de objetivos de este modelo. Es decir, conocer si la metodología MDD implementada fue realmente de ayuda para la comunicación entre desarrolladores, la congregación de conceptos comunes en elementos de dominio que realmente representen elementos repetibles, y si efectivamente aumentó la visibilidad de la arquitectura del sistema para los s*takeholders*. También interesa encontrar si hay puntos débiles que surjan de la comparación, de modo de definir los próximos objetivos en las líneas de trabajo futuras.

Las métricas que se propusieron para hacer esta comparación son:

- Cantidad de requerimientos de cada solución
- Desarrolladores involucrados
- Estimación inicial del esfuerzo para realizar cada solución
- Esfuerzo real consumido para realizar cada solución
- Tiempos de entrega de cada solución vs tiempo de entrega estimado inicialmente.
- Metodologías de construcción y de reutilización
- Metodologías de testing.
- Comunicación entre desarrolladores dentro del proyecto.
- Orientación de los sistemas transaccionales
- Generación automática de código

Se realizará una comparación entre estas métricas para la solución A (pre-metodología) y la solución B (post-metodología).

6.5.1 Sistema A (Pre-metodología)

Este sistema "A" fue desarrollado en el Q1 de 2011 por un desarrollador que ya no pertenece a la compañía. Este sistema es una evolución de un autorizador preexistente (esa primera fase del autorizador ya estaba construida para Q1 2011), y en esta segunda fase el objetivo era aumentar la gama de funcionalidades del mismo.

A través de la recolección de documentos y mails enviados durante la etapa de construcción del sistema, hoy día podemos averiguar los tiempos de desarrollo y las dificultades encontradas durante ese período. Así mismo se recuperaron los documentos de especificación original, de modo de tener las bases comparativas entre el sistema "A" y el sistema "B" que mostraremos en la próxima sección.

Los datos recolectados del sistema "A" son los siguientes:

- ✓ Desarrolladores involucrados: 1
- ✓ Inicio de la etapa de desarrollo: Lunes 24/01/2011
- ✓ Fin de la etapa de desarrollo (fecha original estimada): Jueves 03/02/2011
- ✓ Inicio de la etapa de pruebas de integración y puesta productiva: Viernes 04/02/2011
- ✓ Fin de la etapa de pruebas de integración y puesta productiva (fecha original estimada): Martes 08/02/2011
- ✓ Horario Laboral Diario: 8 horas / hombre por día.
- ✓ Esfuerzo asignado a la etapa de desarrollo: 64 horas / hombre
- ✓ Esfuerzo asignado a la etapa de pruebas de integración y puesta productiva: 24 horas / hombre
- ✓ **Esfuerzo total del proyecto (estimación original): 88 horas / hombre**

La especificación del sistema "A" es la siguiente:

- ✓ El sistema transaccional debe ser un *autorizador* (*authorizer*) que deberá validar cupones y premios provenientes de un POS, y que a su vez los pedidos de los POS son concentrados por un *switch* transaccional. Ese *switch* transaccional es el que se conecta con este autorizador para validar dichos cupones y premios.
- ✓ El protocolo SWITCH-AUTORIZADOR es del tipo *Meflur* (protocolo propietario basado en campos ASCII separados por *pipes* "|", sin CRC, con caracteres de comienzo (0x02) y fin de trama (0x03).
- ✓ El sistema debe manejar dos transacciones: una de *redención de cupones por premios*, y otra de *reverso de la redención*. Para la *redención de cupones*, el POS debe leer con un *scanner* (pistola) el código de barras de un cupón, enviar este código al *switch,* y el autorizador validar si el cupón es válido. Si las validaciones dan OK, el usuario debe redimir el premio descrito en dicho cupón: desde el lado del autorizador, se debe descontar el producto redimido por el cupón validado.
- ✓ El sistema debe armar un archivo de conciliación diario con todas las transacciones de redención y de reverso generadas el día anterior.
- ✓ El sistema debe contar con tres reportes que se deben obtener de la estructura de base de datos configurada para este sistema. Esos reportes son enviados por mail diariamente.

Sucesos importantes detectados:

- ✓ El desarrollador escaló en un mail del miércoles 26/01/2011 que una funcionalidad preexistente en la fase 1 (de conciliación de datos contra el proveedor) tomaba la fecha de una forma dada, y para esta fase 2 la fecha sería configurada en un formato diferente.

- ✓ Dentro del código del sistema se detectaron algunas piezas de código comentadas, y de nombres de campos que parecen provenir de otro sistema (dados los nombre de dichos campos y propiedades). Esto denota que la técnica usada por el desarrollador fue *copiar y pegar* código de una solución anterior similar. Si bien esta tarea es totalmente válida, el desarrollador no llegó a tiempo o no consideró importante emprolijar el código, refactorizando los nombres de los campos, o eliminando el código copiado que evidentemente no le resultó útil.
- ✓ En un mail del lunes 07/02/2011, el supervisor y el desarrollador pidieron una prórroga de la puesta productiva para el miércoles 09/02/2011 al cliente (8 horas / hombre más en la etapa de puesta productiva).
- ✓ El sistema quedó productivo en un 90% el viernes 11/02/2011 (se demoró la entrega en 16 horas / hombre más, ocupadas en tareas de *redoing*). Quedaron pendientes los tres reportes y una tarea de mantenimiento a nivel de base de datos, que se pusieron productivos luego de dos semanas, dado que el desarrollador empezaba sus vacaciones el lunes 14/02/2011.
- ✓ Según un mail del cliente del día 25/02/2011, el sistema tuvo su presentación oficial (demostración con el cliente) para ese mismo día. El cliente describió un error de validación con las terminales que estaban en la demo, y pidió soporte *urgente* a la compañía.
- ✓ En un mail del 26/02/2011 el cliente dio por finalizada la demostración indicando que la etapa terminó exitosa. Así también avisó que el 02/03/2011 comenzaría la distribución de la versión a todos los POS en campo. Se asume que el 10% restante (los tres reportes) quedaron realizados en los días del mes de febrero de 2011, ya que no hubo reclamos.

6.5.2 Sistema B (Post-metodología)

El sistema "B" que se introduce involucra el desarrollo de un *bridge* (adaptador) para soportar la venta de tres productos de tiempo-aire (para *switch* telefonía celular prepaga). Este desarrollo debe funcionar en conjunto con un (concentrador) en la plataforma de la oficina de México de la compañía. El proyecto que involucra todas las tareas para realizar este sistema se organizó durante la primera semana de Febrero de 2015. Se documentó la organización del proyecto a través de un *diagrama de Gannt*, en el cual podemos destacar los siguientes puntos:

Datos recolectados del sistema "B" (de la documentación del proyecto):

- ✓ Desarrolladores involucrados: 1 recurso total, desglosado en 1 desarrollador en la oficina de Argentina, más 1 de *backup* en la oficina de Colombia (en el caso eventual de tener que movilizar el recurso de Argentina para otro proyecto de urgencia).
- ✓ Inicio de la etapa de desarrollo (fecha original estimada): Lunes 09/02/2015
- ✓ Fin de la etapa de desarrollo (fecha original estimada): Miércoles 11/02/2015

- ✓ Inicio de la etapa de pruebas de integración y puesta productiva: Jueves 12/02/2015
- ✓ Fin de la etapa de pruebas de integración: Viernes 13/02/2015
- ✓ Puesta productiva (fecha original estimada): Lunes 16/02/2015
- ✓ Horario Laboral Diario: 8 horas / hombre por día.
- ✓ Esfuerzo asignado a la etapa de desarrollo: 24 horas / hombre
- ✓ Esfuerzo asignado a la etapa de pruebas de integración y puesta productiva: 8 horas / hombre
- ✓ **Esfuerzo total del proyecto (estimación original): 32 horas / hombre**

La especificación del sistema "B" es la siguiente:

- ✓ El sistema transaccional debe ser un *autorizador* (*authorizer*) que facilitará la compra de tiempo-aire de telefonía celular prepaga para los productos mexicanos TELCEL, IUSACELL y MOVISTAR. Como se describió en el sistema "A", los pedidos de los POS se concentran en un *switch* transaccional, y que a su vez re-direcciona la venta de esos productos al adaptador "B", si se dan algunas condiciones comerciales preferenciales.
- ✓ El protocolo SWITCH-AUTORIZADOR es del tipo *Meflur* (protocolo propietario basado en campos ASCII separados por *pipes* "|", sin CRC, con caracteres de comienzo (0x02) y fin de trama (0x03)
- ✓ El sistema debe manejar tres transacciones: la *transacción de venta* propiamente dicha, una transacción de *consulta de saldo* (que informe el saldo que tiene la compañía para poder vender a sus clientes los productos de tiempo-aire), y la transacción de *consulta de recarga*.
- ✓ El sistema debe armar un archivo de conciliación diario con todas las transacciones de venta aprobadas del día anterior.
- ✓ A diferencia del sistema "A", el sistema "B" no tiene reportes visibles.

Sucesos importantes detectados:

- ✓ A pesar de haber organizado el proyecto para arrancar el 09/02/2015, el proyecto no arrancó en la fecha estipulada, dado que fue aprobado por la gerencia comercial el 11/03/2015. El *kick-off* se demoró para el lunes 16/03/2015.
- ✓ Este desfasaje de 6 semanas en el *kick-off* afectó en el proyecto sobre la asignación original de recursos. El desarrollador primario no podía estar dedicado 8 horas a la construcción del adaptador "B" dado que tenía que cumplir con otros proyectos. Es decir, que debido a un solapamiento de proyectos, se decidió diversificar los esfuerzos en los dos desarrolladores (situados en Argentina y Colombia), para que cada uno aporte 4 horas/hombre.
- ✓ Para la organización estructural del *bridge* (adaptador) se utilizó el DSL. Los desarrolladores declararon en una entrevista para este trabajo que durante el primer día de desarrollo ya habían terminado el modelo estructural del sistema, es decir una instancia de modelo del DSL. Para esta tarea

se armó una teleconferencia de aproximadamente 90 minutos de *Skype* con pantalla compartida. Gracias a la naturaleza gráfica de los modelos, el DSL permitió facilitar la comunicación, la discusión de ideas y la definición de elementos involucrados, según declaró el desarrollador colombiano.

- ✓ Dado que la comunicación de la transacción se realiza de POS a *switch*, de *switch* a *bridge*, y por último de *bridge* a proveedor, se decidió paralelizar las tareas restantes entre ambos desarrolladores: el recurso argentino se encargó de las modificaciones necesarias a nivel *switch* para la comunicación *switch*-*bridge*, y el recurso colombiano de la personalización de código generado automáticamente por el DSL. Estas dos tareas ocurrieron el martes 17/03/2015.
- ✓ Según un mail del miércoles 18/03/2015, el recurso colombiano informó que ya estaban las personalizaciones realizadas, como así también los casos de test unitario. Según declararon los involucrados, durante ese día se hicieron tareas de *redoing* dado que algunos *unit tests* fallaban para el caso de transacciones de *venta rechazadas*.
- ✓ Otro dato importante es que durante esta etapa no se utilizó el circuito completo (POS >> *switch* >> *bridge* >> proveedor) para hacer pruebas. Es decir, no fue necesario pedir un POS productivo al sector de operaciones de la empresa, para que lo configuren especialmente apuntando a un ambiente de desarrollo, que permita generar transacciones de prueba. En cambio se usaron casos unitarios tanto en *switch* como en el *bridge*.
- ✓ El sistema se terminó el jueves 19/03/2015 y se subió al ambiente pre-productivo para hacer una prueba de homologación con personal del proveedor, personal de la oficina de México de la compañía, y los dos desarrolladores involucrados. Las pruebas duraron aproximadamente una hora, y se realizaron transacciones a través de todo el circuito POS >> *switch* >> *bridge* >> proveedor. Para tal fin, se configuró un POS en México, que enviaba las transacciones de prueba al ambiente pre-productivo en Argentina (*switch*-*bridge*), y este por último se conectaba con el proveedor en México.
- ✓ Las pruebas de integración se finalizaron de forma correcta el mismo jueves 19/03/2015, dando como resultado el OK del proveedor para comenzar las operaciones (desde el aspecto técnico). Ese mismo día se subió al ambiente productivo.
- ✓ El lunes 23/03/2015 se ingresaron las reglas de ruteo, las políticas de prioridad para comprar los productos por el canal "B", las credenciales para acceder al servicio productivo y se realizaron las compras de saldo para poder comenzar a vender. Se monitoreó el *bridge* durante una hora, observando que las transacciones entraran sin problemas. Al día siguiente (24/03/2015) se dio por finalizada la tarea de puesta en producción, dado que los archivos de conciliación entre las plataformas de ambas compañías no dieron diferencias de transacciones.

6.5.3 Comparativa Sistemas A y B

En esta sección se redactará una comparación de ambos proyectos de desarrollo, habiéndose analizado las especificaciones de ambos sistemas, las vivencias de los desarrolladores y los sucesos generados durante las fases de construcción. El objetivo de esta comparación es resumir algunos puntos que denoten fortalezas y debilidades de cada uno de los métodos, a fin de enriquecer y construir una evaluación cualitativa del *framework TransactionKernel* y del DSL presentado durante este trabajo.

- ✓ Con respecto a las *metodologías de construcción y de reutilización*, en el sistema "A" se utilizó código que se copió y pegó de otras soluciones. El desarrollador, lógicamente, aportó sus líneas de modo de cumplir con los requerimientos. Esta metodología es totalmente válida. De todos modos, en esta propuesta debemos destacar que este enfoque no deja de ser informal, si se pretende aumentar el grado de reusabilidad. Es decir, el desarrollador es libre de reutilizar porciones de código a discreción y voluntad de él mismo. Otro desarrollador podría copiar y pegar código de cualquier otra fuente de información que haga la misma tarea de igual forma o similar. El punto es que al largo plazo, si bien funcionalmente la aplicación puede cumplir con las expectativas, las porciones de código utilizadas no son siempre iguales, y la construcción de artefactos concretos de *software* se diluye. Esto produce que aumente el acople de código si no se mantienen tareas de revisión de arquitectura recurrentes. Además se agregan *desvíos y vicios* propios de cada desarrollador durante el proceso de codificación manual, en conjunto a los *desvíos* y *vicios* agregados por las piezas copiadas de fuentes externas. Por último la diversidad de fuentes de información puede terminar impactando negativamente cuando se conjugan todas las piezas de código en un mismo sistema. Comparativamente, en el sistema "B" las libertades para los desarrolladores están restringidas ya que se desalienta el uso de *copy paste* indiscriminado. Pero como contraparte se fomenta una formalización de artefactos reutilizables, que estén bien delimitados y que permitan facilitar la construcción de código con piezas bien definidas.
- ✓ Con respecto a las *metodologías de testing*, durante el desarrollo del sistema "A" el programador no utilizó una estrategia de *unit test,* ni implementó una tecnología TDD. Probablemente las sesiones de pruebas se realizaron con un POS real, que efectuaba las operaciones contra el *switch*, recorriendo todo el circuito transaccional de punta a punta. En el sistema "B" se utilizaron casos de prueba unitarios para verificar y validar los comportamientos de las transacciones. Se recuerda al lector que el DSL fomenta el uso de *unit test*, generando dos casos de prueba unitarios por cada transacción conectada a los *motores de entrada* definidos en el modelo. Esto le da la posibilidad al desarrollador de usar estos *tests unitarios* ya sea para hacer simples validaciones, o para orientar por completo el desarrollo del sistema a través de una metodología TDD. La orientación del sistema "B" no fue TDD, simplemente se usaron los casos de prueba para hacer las primeras validaciones y verificaciones del *adaptador bridge*, por lo que no se aprovechó por completo el poder real de los casos de prueba unitarios. De una o de otra forma, el sistema "B" llegó a la etapa de *pruebas integrales* con una medida cuantitativa de calidad, dado que cumplía con los casos unitarios que se habían generado durante la transformación

del modelo a código funcional. Ambos sistemas (A y B) completaron la fase de *pruebas de integración* del mismo modo, es decir, recibiendo tramas desde un POS, recorriendo todo el circuito transaccional completo.

- ✓ Con respecto a la *comunicación entre desarrolladores dentro del proyecto*, el sistema "B" brindó mayor visibilidad que el sistema "A", dada la naturaleza gráfica de los modelos, y la experiencia que un equipo gana al reutilizar los mismos elementos de forma conveniente. A modo anecdótico, el autor de este trabajo tomó la responsabilidad en 2011 de mantener y continuar los proyectos realizados por el desarrollador del sistema "A", que había renunciado al corto tiempo de su llegada. El trabajo de análisis realizado para comprender cómo funcionaban los sistemas preexistentes (el sistema "A" entre otros) fue muy extenso, y la comprensión del código generado se había tornado muy difícil dado que no estaba el desarrollador disponible para dar soporte. Luego de 3 meses en la empresa, se logró comprender en un 75% aproximadamente sobre la funcionalidad del código fuente heredado de esta persona. No es una crítica al desarrollador, sino a la naturaleza del código fuente *per se*, que al ser texto no es tan visible como un diagrama, gráfico o modelo. Todos pensamos distinto, y eso no tiene que ser un problema al tratar de entender el código de otra persona. Por eso, para el entendimiento entre colegas desarrolladores es de gran ayuda el uso extensivo de elementos reutilizables en un *marco de trabajo*. Desde la experiencia recolectada durante la implementación de la metodología MDD, se evidenció que los integrantes del equipo lograban comprender las intenciones de los otros colegas con solo ver qué clases del *framework* se instanciaron, cuáles fueron los métodos utilizados y qué elementos de dominio habían sido planteados.
- ✓ Sobre la *orientación de los sistemas transaccionales*, los primeros analizados (pre-implementación MDD, como el sistema "A") estaban orientados a un bloque monolítico de código (un archivo ejecutable *.exe*) que contenía la lógica de ejecución como servicio, el manejo de la capa de comunicaciones y de *threads,* la lógica de acceso a datos y la lógica de negocios. El código de todas formas estaba correctamente dividido en varios archivos, e incluso se utilizaban algunos patrones de diseño (principalmente el *Strategy* y *Singleton* para algunas situaciones particulares), pero no se detectó una política de refactorización o de reingeniería de la solución para detectar elementos comunes una vez finalizado el sistema. Por ejemplo, se notó en uno de los *switch* pre-implementación que las lógicas de negocio de cada tipo de transacción estaban conglomeradas en un solo método. Esto causaba un incremento en condicionales (*if-else*) que terminaba aumentando el nivel de acople del código. Ha sucedido que al tratar de corregir un error en la lógica de una transacción se descomponga otra que funcionaba correctamente en un principio. Por esta razón, el desacople mediante alguna receta conocida (algún patrón conocido) era vital para poder generar soluciones de mayor confiabilidad. Fue así que estas tareas de refactorización y reingeniería sucedieron a la hora de diseñar el *framework*. Por ejemplo uno de los pilares del marco de trabajo es el desacople de las lógicas de las transacciones en clases separadas, siguiendo el patrón *Strategy*. Es por esta característica que internamente definimos a

los sistemas realizados con este *framework* como *orientados al tipo de transacción*, como contrapartida de los antiguos *orientados al flujo transaccional*.

- ✓ La *generación automática de código* es un punto fuerte de la implementación de MDD en el sistema "B" comparado con el sistema "A". Por ejemplo el sistema "A" requería de tablas y *stored procedures* dentro de la base de datos para funcionar correctamente. Los *scripts* que creaban estas tablas y SP no fueron nunca encontrados en el código fuente heredado del desarrollador. Nuevamente no es una crítica al desarrollador, ya que analizando el código se pueden deducir los nombres de los elementos de DB involucrados, y de esta forma obtenerlos del ambiente productivo, por ejemplo. Sin embargo, el sistema "B" es generado inicialmente desde una herramienta de transformación de código que no solo escribe código funcional, sino también los *scripts* necesarios para crear las tablas y *stored procedures* para hacerlo funcionar. En otras palabras, el sistema "B" ofrece mayor consistencia a nivel de archivos fuentes para facilitar la replicación del sistema en otro entorno productivo o de test.
- ✓ Entre los *puntos débiles* del sistema B es que la implementación del *framework TransactionKernel* y su correspondiente MDD implica actualmente estar atado a una tecnología en particular, en este caso .NET y el IDE Visual Studio. Parte de los sistemas creados previos a la implementación del *framework* y del DSL tenían la ventaja y la desventaja de la libertad de elección de una tecnología de implementación en particular. Como la compañía no tenía una política global para definir qué idiomas y IDE's usar, los desarrolladores usaron C++, PHP, .NET, Java y C. El hecho de migrar todo a una sola tecnología trajo rispideces, ya que todas tienen ventajas y desventajas. Esto es un punto débil de este trabajo, y que debe ser atendido en líneas de trabajo futuras, de modo que se encare la migración del *framework* y del DSL a otras plataformas, como Java, PHP y C++.
- ✓ Con respecto a la *tasa de errores,* los sistemas "A" y "B" no sufrieron errores de alta criticidad una vez puesto en campo, pero se han dado casos donde las interfaces con los proveedores cambiaron (luego de un tiempo considerable en campo sin errores), y por esa razón se generaron *bugs* en nuestros sistemas. Los tiempos de corrección de algunos errores en el sistema "A" fueron notablemente mayores que los tiempos insumidos para corregir algún problema similar en el "B". Otros errores en cambio consumieron casi el mismo tiempo en ambos. La conclusión nuevamente recae en que las demoras se dieron por tener que investigar cómo funcionaban porciones del sistema "A" generado por otra persona, mientras que en el "B" los elementos de dominio son conocidos más allá del responsable que los instanció. En el caso cuando el tiempo insumido fue similar tanto para el "A" como para el "B", la conclusión recae en alguna personalización de código en el sistema "B", que no fue fácil comprender en ausencia del responsable y que demoró la corrección (recordar que el sistema "B" generado por el DSL tiene una componente de código generado automáticamente y otra componente que es libre de personalización por los desarrolladores).
- ✓ Con respecto al *cumplimiento de tiempos*, si bien para la demostración se tomó un solo par de soluciones *ejemplo* (sistemas "A" y "B"), la percepción de los desarrolladores y de los *stakeholders* es

que se está cumpliendo con los tiempos de mejor forma que en el pasado. Así también el grado de confiabilidad es mayor ahora que antes, ante una nueva salida productiva. Probablemente esto sucede por reutilizar código preexistente (que con el tiempo pasa a ser *bien conocido* por todos los involucrados) y por poder dar visibilidad hacia los colegas de equipo de forma consistente a través del modelo de una instancia.

6.6 Resumen del Capítulo

Durante este capítulo se desarrollaron los siguientes puntos:

- Se introdujo un caso real de desarrollo para un sistema transaccional del tipo *adaptador* (*bridge*), a través de un set de especificaciones. Esas especificaciones contenían las transacciones soportadas, la forma de conexión, el protocolo con el cual deben ser enviados los datos, y el mecanismo para mantener una sesión lógica contra el nodo externo.
- Se hizo una descripción paso a paso del desarrollo del *bridge* a través del DSL.
- Se mostró el resultado de la transformación a código desde la perspectiva de la estructura de archivos generada.
- Se realizó una comparación entre dos sistemas (A y B) realizados con y sin la implementación MDD. Se realizó una descripción de cada caso y se redactaron párrafos comparando cada sistema con respecto diversas métricas propuestas.

En el próximo capítulo se definirán las conclusiones finales de la tesis y las líneas de trabajo futuro.

7 Conclusiones y Línea de Trabajo Futuro

Este capítulo es el último de este trabajo de tesis. En él se describirán las líneas de trabajo futuras, que continuarán seguramente con el desarrollo del DSL y del *framework* descritos durante este trabajo. El objetivo será hacer de estas dos piezas un artefacto de *software* más completo, con la posibilidad de abarcar nuevas arquitecturas de sistemas transaccionales aun no contempladas en la versión actual mostrada durante esta tesis.

También se expondrá un resumen de la opinión general de los desarrolladores y de los *stakeholders,* que estuvieron involucrados de algún modo con los proyectos. El objetivo de estas líneas será remarcar y describir algunos puntos fuertes y débiles de este trabajo.

Con todos estos datos recolectados se elaborará la conclusión final del trabajo.

7.1 Feedback de los Stakeholders

Los *stakeholders* involucrados en este trabajo son los colegas desarrolladores del equipo IT (5 desarrolladores), los líderes del equipo de IT (3), el CTO de la compañía (que congrega las tareas de conducción de todo el área de IT) y los *focal points* (4) (analistas funcionales externos al equipo de IT que hacen de conexión con los clientes) y los gerentes comerciales y generales de cada subsidiaria (4). Los datos obtenidos a continuación fueron obtenidos de entrevistas informales, charlas restrospectivas del equipo de desarrollo, e intercambio de mails del autor con los demás colegas:

- Se realizaron seis entrevistas (CTO de la compañía, Gerente General de Argentina, Focal Point de México, Focal Point de Colombia, Gerente de Desarrollo, Gerente Comercial de Argentina). Dentro de la entrevista informal, se les pidió valorar la contribución del *framework* en la compañía con un sistema de puntajes del 1 al 5, donde 1 significa “no agregó valor significativo”, y 5 significa “generó valor fundamental para la cultura de la compañía”.
- Se congregan las opiniones, críticas y sugerencias de los desarrolladores dentro del equipo de IT durante las reuniones retrospectivas trimestrales durante todo el año 2014 e intercambio de mails entre ellos.

La opinión de los involucrados varía en función del grado de intervención de cada uno en los proyectos. Sin embargo la opinión general es positiva. De las seis entrevistas realizadas, el *CTO* y el *Gerente de Desarrollo* puntuaron 5 / 5 al *framework*. Los *Focal Point de México y Colombia* opinaron que la contribución del *framework* tiene un puntaje de 4 / 5 y ambos gerentes se limitaron a puntuar 3 / 5.

Se considera que esta tendencia de puntuacuón es positiva ya que se logró uno de los objetivos corporativos: aumentar la comunicación entre los equipos de los países (Colombia, Chile, Argentina y México). Esta

comunicación se vio fomentada dado que los proyectos de un país pueden ser atendidos por cualquiera de los desarrolladores de la compañía, más allá de su locación física (antiguamente se solía distribuir los proyectos exclusivamente entre los desarrolladores del país dueño de dicho proyecto). Otros de los proyectos que actualmente se está encarando y que está soportado por la filosofía presentada en este trabajo es el de centralizar todos los servicios de todos los países en un solo lugar físico. Para concretar esta meta, es preciso migrar todos los sistemas preexistentes (en varios lenguajes, tecnologías y arquitecturas) a una base de diseño conocida por todos, que sea extensible por los desarrolladores de la compañía y que congregue los conceptos y elementos de dominio comunes. Es por eso que este *framework* y su correspondiente formalización encajan para cumplir con estas tareas.

Por esta razón a nivel gerencial, tanto el *framework* como la implementación de un lenguaje específico de dominio (DSL) fueron vistos como un hecho positivo dentro del mapa evolutivo (*road map*) del sector de IT. El único inconveniente encontrado fue el estar atado a una tecnología en particular, en este caso .NET. Esto conlleva el costo de obtener las licencias de los ambientes de desarrollo y de los servidores productivos para cumplimentar con todos los aspectos legales pertinentes. Así mismo, la imposición de una tecnología en particular y la recolección de factores comunes no fueron tareas sencillas dadas las distintas opiniones técnicas de cada uno de los integrantes del equipo de IT. Nunca estuvo la intención de imponer una tecnología en particular, pero dado un ambiente divergente en alternativas tecnológicas (ya existían soluciones transaccionales en C++, PHP, Java, C y .NET) se tenía que definir el que mejor encajara en los objetivos de largo plazo de la empresa. Las discusiones para elegir una tecnología entre las cinco implementadas trajo rispideces e intercambios de ideas importantes, más allá que la filosofía MDD intenta desacoplar la implementación tecnológica de un modelo de dominio (PIM y PSM por ejemplo). La recolección de elementos repetibles para capitalizar elementos de dominio no fue tarea sencilla: los desarrolladores veían conceptos parecidos pero no todos veían la conceptualización de la misma forma, con las mismas clases, métodos y propiedades.

Sin embargo, los integrantes del equipo de desarrollo y los líderes de equipo de IT de cada subsidiaria están satisfechos con el cumplimiento de los tiempos de los proyectos, que se vio fomentado por la reutilización de elementos de dominio a través de *TransactionKernel* y la implementación MDD. Esto permite mayor previsibilidad a la hora de estimar fechas de entrega, esfuerzo de los recursos involucrados y distribución de recursos para proyectos en paralelo.

Hoy por hoy quedan muchas cosas por hacer: por ejemplo, el soporte para acceso de datos está limitado a bases de datos del tipo *Microsoft SQL Server*. No hay una capa de acceso a datos que sea independiente de esta implementación. Así también, como se comentó con anterioridad, tanto el *framework* como el DSL son comprensibles desde el IDE *Visual Studio*, y para lenguaje C# .NET. Para hacerlos extensibles a mayor cantidad de usuarios, es necesario conseguir la independencia de plataforma tecnológica.

Desde el punto de vista de la filosofía de MDD, muchos dentro del equipo de IT coincidimos en que se puede sacar provecho del conjunto *framework+DSL* generando modelos intermedios entre el DSL y el código fuente. Por ejemplo, que del DSL se produzcan diagramas de clases del sistema generado, casos de uso, diagramas de secuencia y documentación de los protocolos de entrada y salida. Es decir, apuntar a obtener todos estos *nuevos outputs* sumados a los *outputs actuales*: la componente de código automático, la componente de código personalizable por el desarrollador, los scripts de acceso a datos y los casos de prueba unitarios.

Desde el punto de vista de la filosofía de TDD, los integrantes del equipo quedaron todos de acuerdo en implementarlo, ya que los sistemas transaccionales encajan perfectamente en un modelo de casos unitarios de prueba. Cada caso unitario puede ser considerado una transacción en particular, con datos en la trama que el sistema debería aprobar o rechazar. De este modo el equipo logró reducir tiempos de verificación de los sistemas ya que no necesitaba contar con un POS real. Tampoco se necesitaba de un simulador de POS, que muchas veces era desarrollado como un programa colateral al proyecto y que no siempre funcionaba correctamente, agregando errores la propia herramienta de simulación. El mayor punto débil encontrado en implementar TDD es que los sistemas transaccionales dependen de mecanismos de persistencia para consolidar datos o validarlos. Este *framework* aún no dispone de una capa de datos *mock* bien definida para poder desacoplar la dependencia de datos en las bases, y correspondientemente para que los *test unitarios* resulten siempre de la misma forma.

Más allá de todos puntos débiles, los *stakeholders* en mayor o menor medida de inferencia, hicieron el esfuerzo de adoptar esta nueva metodología de trabajo obteniendo resultados positivos hasta el día de hoy para los objetivos y el alcance planteado. Hay que agradecer a la calidad de equipo humano y profesional que forma la compañía, que hizo posible la implementación de MDD desde lo técnico. Desde el aspecto gerencial se brindó la confianza y soporte para modificar las prácticas antiguas de desarrollo, creyendo en la propuesta y creyendo en que era posible mejorar la productividad a través de la Ingeniería de Software. Queda mucho por mejorar, y se espera ir resolviendo cada punto débil en las próximas versiones del *framework* y del *DSL*. Así mismo se espera seguir cumpliendo con los objetivos y expectativas que se presenten en el futuro.

7.2 Conclusión Final

A continuación se resumen las conclusiones obtenidas de todos los puntos analizados durante este trabajo:

- ✓ El objetivo primario de esta tesis, que consistió en formalizar el *framework TransactionKernel* a través de metodologías MDD – DSM se cumplió correctamente para un alcance acotado, es decir, para en una plataforma *Microsoft Windows,* con *framework* .NET 2.0 o superior instalado y un IDE *Visual Studio 2010* o superior instalado para desarrollar.

- ✓ Se corroboró que la siguiente hipótesis es parcialmente correcta*: A partir de un conjunto de conceptos repetibles y pertenecientes al dominio del procesamiento electrónico de transacciones bancarias, de lealtad o similares, es posible generar sistemas de proceso multiconcurrentes de forma semi-automática, que a través de la transformación de modelos cooperativos (de mayor a menor nivel de abstracción), del armado de artefactos, y del soporte de metodologías* MDD*, mejoren tanto el grado de reusabilidad de componentes, como la disminución de los costes de mantenimiento, tasas de errores por línea de código, y los tiempos de entrega del producto*. La parcialidad de la corroboración se da por los siguiente motivos:
  - Se pudo comprobar fehacientemente la reducción en los tiempos de entrega de los productos y la mejora en el grado de reusabilidad de componentes. Se corroboraron mejoras en reusabilidad y en tiempos de entrega, que resultaron en una reducción de los esfuerzos para dar mantenimiento o crear un sistema. No se pudo comprobar directamente que se haya bajado las tasas de errores, pero si se corroboró que una vez que un elemento de dominio está funcionando correctamente, sigue funcionando bien en otras soluciones (o por lo menos no presentó errores en implementaciones subsiguientes bajo alcances similares).
  - Queda aclarar que no se consiguió en esta versión trabajar con modelos cooperativos de mayor a menor nivel de abstracción. Simplemente desde una instancia de modelo del DSL se transforma a código funcional, *scripts* de DB y código de *unit testing*.

- ✓ Se cumplió con el objetivo secundario de este trabajo, que consistía en demostrar el uso del *framework* y del DSL en un ejemplo real con requisitos definidos, aunque no se expuso el código final, dado que no sumaba a la demostración de uso de los elementos de dominio.

- ✓ Se detectaron puntos fuertes y débiles del *framework* y del DSL. Los últimos serán puntos de trabajo para el mediano-corto plazo, en las próximas versiones del lenguaje específico de dominio.

- ✓ La opinión de los *stakeholders* fue positiva en general, aunque hubo opiniones divergentes durante la definición de los elementos de dominio a nivel técnico. Durante la implementación de la metodología los desarrolladores fueron los que más tuvieron que trabajar en adaptarse por estar acostumbrados a otras formas de trabajo. La definición del lenguaje y tecnología de desarrollo fue la que más discusión trajo en el equipo, siendo un punto débil del DSL y del *framework TransactionKernel* el estar atado a la tecnología .NET.

- ✓ La comunicación entre desarrolladores fue uno de los puntos positivos más importante de todo este trabajo. Luego de trabajar en la adaptación y la puesta en marcha de los elementos de dominio que serían incluidos en la plataforma, los proyectos entre desarrolladores de diversos países se

incrementaron, e incluso los equipos podían intercambiarse ya que comprendían por igual como estaba formado un sistema.

- ✓ Si bien la filosofía MDD fomenta la transformación de modelos, desde los de mayor nivel de abstracción hasta los de menor nivel, (llegando hasta el código funcional), en este trabajo concluimos que no es posible dejar automatizada por completo dicha transformación, al menos en el dominio de procesamiento transaccional. Es clave contar con un espacio para que el desarrollador pueda agregar código personalizado para alterar algunos comportamientos propios de cada solución, que no pueden ser transferidos desde modelos de mayor nivel de abstracción. La implementación de clases parciales fue vital para cumplir de forma elegante con esta premisa.

- ✓ Muchos de los inconvenientes técnicos que se presentaban durante el desarrollo de servidores transaccionales previos a la implementación del *framework* y el DSL fueron solucionados, entre ellos el manejo de recursos computacionales cuando hay picos transaccionales, el uso de un sistema estándar de *logging*, el desaprovechamiento del uso del lenguaje orientado a objetos, el desacople de código monolítico y la refactorización a patrones conocidos.

- ✓ Los elementos de dominio definidos en el *framework* y en el DSL representan los conceptos repetibles y comunes encontrados entre la cantidad finita de soluciones analizadas al alcance. Sin embargo no hay garantías que para nuevos requerimientos en el futuro, los elementos definidos no sean suficientes y haya que hacer re-ingeniería de forma recursiva para remodelar los elementos de dominio preexistentes, o definir nuevos.

- ✓ Así mismo, los protocolos transaccionales soportados en este *framework* son muy acotados, principalmente ISO8583 y protocolos propietarios. Dados nuevos proyectos en el futuro, se deberían incluir *analizadores de protocolo* (*parsers*) capaces de ensamblar y desensamblar HPDH, SPDH, HL7 entre otros.

- ✓ El uso de TDD (*Test-Driven Development*) ha sido de gran ayuda para comprobar el impacto de los cambios en el código. Se logró adaptar de una forma elegante esta metodología considerando a las transacciones como inputs independientes del sistema. El sistema transaccional es visto como una *caja negra*, y se generan los *unit test* como envíos de tramas de *transacciones aprobadas y rechazadas*, validando que el sistema consiga aprobarlas o rechazarlas según los datos cargados en estas tramas.

7.3 Líneas de Trabajo Futuro

A continuación se resumen los puntos en los que se trabajará en pos de aumentar las capacidades funcionales del conjunto *TransactionKernel+DSL:*

- ✓ Independencia de plataforma tecnológica. Migración a Java, C++ y PHP.
- ✓ Corrección de errores encontrados en el DSL
- ✓ Transformación no solo a código fuente, sino a diagramas de clases, casos de uso y diagramas de secuencia.
- ✓ Transformación del modelo a documentación de protocolo y del sistema modelizado.
- ✓ Compatibilidad on otros mecanismos de persistencia: *Oracle*, *SQLite*, archivos XML, bases de datos orientadas a objetos, etc.
- ✓ Extrapolar las tareas de reingeniería realizadas para los sistemas transaccionales, en pos de encontrar un conjunto de elementos de dominio pero para el desarrollo de aplicaciones POS.

8 Anexo A: Secuencia Genérica y Redefinible de AbstractTransactionHandler.

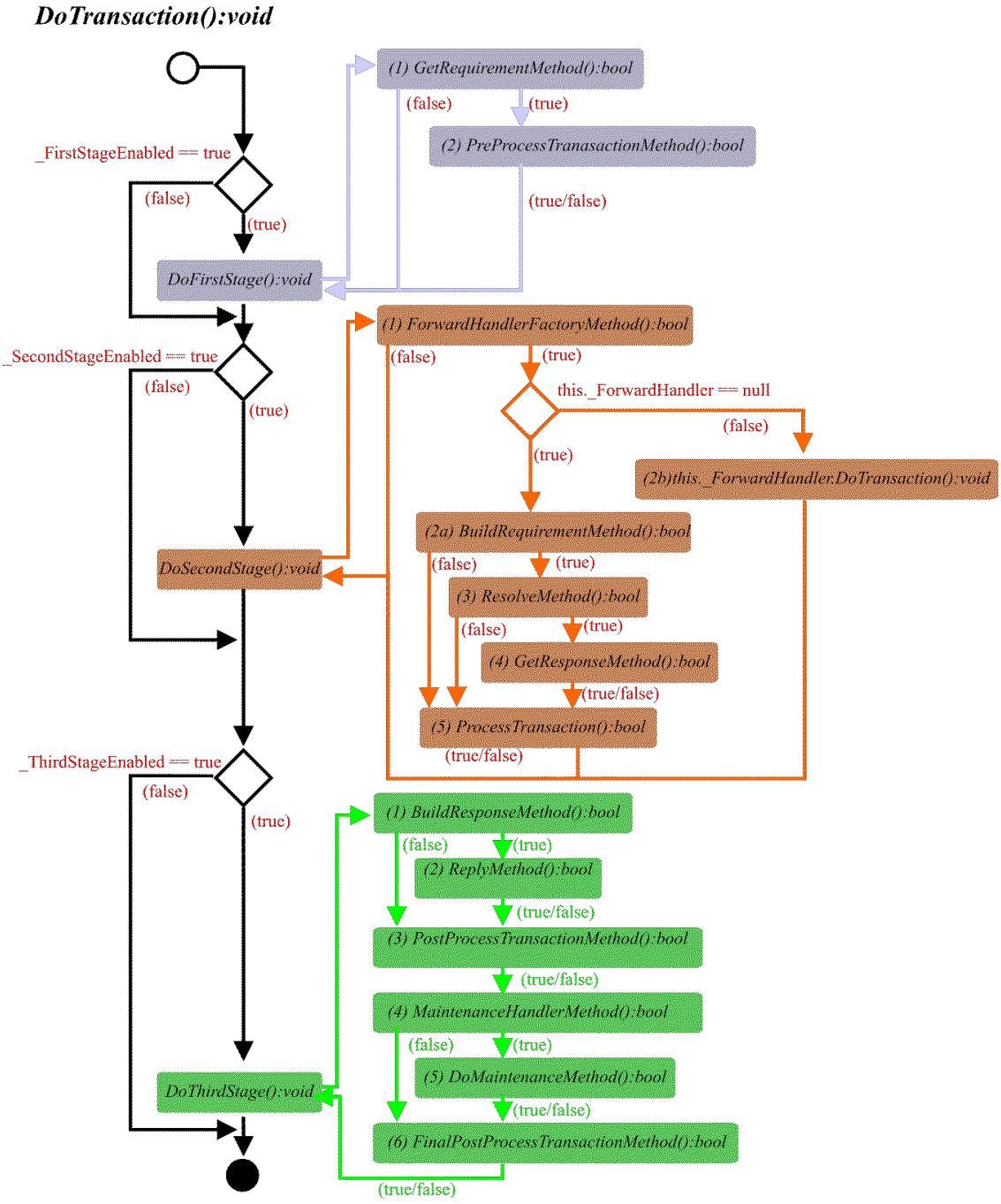

Figura 109 - Secuencia de trabajo por defecto propuesta en *TransactionKernel*, de tres etapas para el proceso genérico de transacciones

9 Bibliografía

[1] W. Y. Ming, *Multi-threading technique for authorization of credit card system using .NET and JAVA,* Kuala Lumpur, 2007.

[2] J. J. L. B. K. M. K. C.-w. S. S.-l. Y. Kyo Chul Kang, «Re-engineering a credit card authorization system for maintainability and reusability of components–a case study,» de *Reuse of Off-the-Shelf Components*, Springer Berlin Heidelberg, 2006, pp. 156-169.

[3] M. H. N. M. N. W. Y. M. H. H. Siti Hafizah Ab Hamid, «Multi-Threading and Shared-Memory Pool Techniques for Authorization of Credit Card Systems Using Java,» *INFOCOMP Journal of Computer Science,* vol. VII, nº 3, pp. 60-69, 2008.

[4] Nasir, M.H.N., Ab Hamid, S.H. y Hassan, H., «Thread-Level Parallelism & Shared-Memory Pool Techniques for Authorization of Credit Card System,» de *Communications and Information Technologies, 2008. ISCIT 2008. International Symposium on*, Lao, 2008.

[5] T. L. G. V. H. Riché, «Lagniappe: Multi-* programming made simple,» de *International Conference on Computer Communications and Networks, ICCCN*, 2007.

[6] J. ,. B. P. ,. H. F. ,. H. T. Andersen, «Domain-specific languages for enterprise systems,» de *Lecture Notes in Computer Science*, Springer, 2014, pp. 73-95.

[7] M. S. Yashwant Singh, «Models and Transformations in MDA,» de *First International Conference on Computational Intelligence, Communication Systems and Networks*, 2009.

[8] R. G. G. P. Claudia Pons, Desarrollo de software dirigido por modelos. Conceptos teóricos y su aplicación práctica, La Plata, Pcia. de Buenos Aires, Argentina: EDULP, 2010.

[9] «Agile Data,» [En línea]. Available: http://www.agiledata.org/essays/tdd.html.

[10] J. Bishop, CSharp 3.0 Design Patterns, O' Reilly, 2008.

[11] L. M. Bibbo, D. Garcia y C. Pons, «A Domain Specific Language for the Development of Collaborative Systems,» de *International Conference of the Chilean Computer Science Society*, 2008.

[12] M. Inc., «Overview of Domain Relationships,» Microsoft Inc., [En línea]. Available: http://msdn.microsoft.com/en-us/library/bb126471(v=vs.80).aspx. [Último acceso: 08 Setiembre 2014].

[13] M. V. K. C. Thomas Stahl, Model-Driven Software Development: Technology, Engineering, Management (Wiley Software Patterns Series), 2006.

[14] J. C. M. W. Marco Brambilla, Model-Driven Software Engineering in Practice., Morgan Publisher, 2012.

[15] Engelen, «gSOAP,» [En línea]. Available: http://www.cs.fsu.edu/~engelen/soap.html.