

Una Propuesta de Implementación MDD y TDD en el Dominio de Sistemas de Procesamiento Transaccional

Hernán Zbucki¹, Claudia Pons²

¹ Facultad de Informática UNLP

² Facultad de Informática, UNLP. Universidad Abierta Interamericana, UAI. CIC (Comisión de Investigaciones Científicas de la Provincia de Bs. As.

{hernan.zbucki}@gmail.com

{cpons}@lifa.info.unlp.edu.ar

Abstracto. 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 construir estas soluciones, se introduce una propuesta de implementación dual MDD y TDD. Esta propuesta define una base de conceptos comunes, obtenidos del análisis de soluciones preexistentes, y experiencias de los desarrolladores, con el objeto de definir elementos del dominio. En paralelo, se conjugarán estrategias de pruebas unitarias, como simulación de las transacciones entrantes en los sistemas bajo desarrollo. El conjunto de ambas metodologías intentará definir un *framework* orientado al dominio que fomente la reutilización de código, reduciendo los esfuerzos de mantenimiento. Así también, dada la naturaleza gráfica de los modelos, se intentará mejorar la visibilidad del sistema construido, tanto para los colegas de equipo, como para los *stakeholders*.

Palabras clave: Payment Gateways, MDD, TDD, refactoring, patrones.

1 Introducción

Los sistemas de procesamiento de transacciones son unos de los más usados y de los más sofisticados dentro de los sistemas de información de gran escala. Particularmente un *CAS*¹ 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]. 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

¹ CAS: Acrónimo inglés para el término *Card Authorization System*, o Sistema de Autorización de Tarjetas.

financiera. [2] Ambos requerimientos están ligados: Los procesos de verificación, autenticación y criptografía, que hacen a 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.

Otra problemática común en estas soluciones es la *frecuencia de actualización del sistema* (referido a 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 ligado a la competencia entre compañías y bancos, de modo que son presionadas para ofrecer nuevos servicios o modificar los servicios existentes con frecuencia. [1] Estas situaciones causan constantes revisiones de los sistemas de autorización, aumentando la complejidad de mantenimiento de los mismos.

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 incurrir negativamente en la calidad de la solución integral.

Por otro lado, la ingeniería de software establece que la construcción de programas debe ser encarado de la misma forma 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*. [3] Sin embargo, los enfoques actuales de construcción de software no siempre son suficientes para tratar los inconvenientes relacionados a los efectos de cambios de tecnología y a los efectos que surgen de la necesidad de cambios en los requerimientos de forma recurrente. [4]

En este artículo se introduce un trabajo que es parte de un proyecto académico de I/D/I, pero que estuvo en un principio enmarcado en el ámbito de una empresa multinacional (*PointPay Inc.*, www.pointpay.net). Inicialmente, esta compañía definió una serie de problemáticas encontradas en su proceso de construcción de sistemas de procesamiento transaccional. Los autores de este trabajo evaluaron entonces los tres requerimientos principales recolectados: la *reusabilidad* de la mayor cantidad de componentes posibles, la *reducción de costos de mantenimiento*, y la *convergencia de criterios* de diseño entre desarrolladores.

Como resultado, dentro del ámbito académico del LIFIA-UNLP, se desarrolló esta propuesta de implementación MDD y TDD orientada específicamente a este dominio. Esta propuesta de implementación dual comprende en primer lugar la definición de un *framework* de elementos de dominio, la posterior formalización del mismo a través de un lenguaje específico de dominio (DSL), la construcción de una herramienta de transformación de las instancias de los metamodelos a código funcional, y la integración de pruebas de unidad (metodología TDD) por tipos de transacción

soportados. De esta forma este proyecto pretende cumplir los objetivos planteados de forma tanto corporativa (requerimientos iniciales) como académica (metodología formal para la solución de los requerimientos planteados).

La organización de este artículo es la siguiente: en la sección 2 se introducirá el estudio realizado sobre el dominio transaccional para refactorizar el código y definir una base de conceptos, que a su vez fueran los pilares fundacionales para implementar un DSL (metodología MDD-MDA) basada en elementos de dominio. En la sección 3 se describe una propuesta de implementación TDD orientada al dominio transaccional, e integrada al modelo MDD que se introdujo en la sección 2. En la sección 4 se describe la evaluación de esta implementación, con los resultados y mediciones obtenidos dentro del ámbito de la compañía. En la sección 5 se presenta un conjunto de artículos de otros autores relacionados al tema, que ayudaron a orientar este trabajo. Por último, en la sección 6 se suman las conclusiones y el eventual aporte de este trabajo sobre el estudio del dominio del procesamiento de transacciones.

2 Propuesta de Refactorización e Implementación MDD

El trabajo concreto realizado hasta la actualidad consta de tres productos: la construcción de una primera versión totalmente funcional de un *framework* (clases, interfaces y métodos para soportar conceptos comunes del dominio), una versión de un DSL que formaliza el *framework* bajo una metodología MDA, y una versión funcional de la herramienta de transformación automática de instancias del metamodelo del DSL a código fuente. Todos los productos fueron desarrollados bajo el IDE *Visual Studio 2012*, con la extensión *MDD Dsl Tools*, para el lenguaje C#, y bajo soporte de *.NET Framework 2.0*.

La construcción del *framework* fue el primer paso de todo el trabajo. Comenzó con el análisis de soluciones transaccionales preexistentes, de la cual la compañía era dueña. Estos programas (alrededor de 10 servicios transaccionales concurrentes, o STC) estaban corriendo en ambientes productivos, contabilizando mensualmente 8 millones de transacciones en promedio. Los STC fueron desarrollados por grupos de programadores empleados por la compañía, disgregados geográficamente en todo el continente americano, sin un concepto de equipo de IT centralizado, sin control de los criterios usados para su construcción, ni del lenguaje de programación que debían usar, y ni de los mecanismos de persistencia que se debían implementar. Los equipos de desarrollo de cada país se autogobernaron por un lapso de cinco años consecutivos, haciendo que las diferencias en los códigos y en los criterios fueran cada vez más pronunciadas. Las tareas previas realizadas para converger a un único criterio, a menos lenguajes y a menos tecnologías nunca pudieron llevarse a cabo con éxito. La causa principal fue que se intentó forzar la migración de la filosofía de todos los equipos de desarrollo a la de un país, es decir eligiendo como modelo a una sola filosofía de uno de los países, sin analizar sus ventajas y desventajas, y sin considerar las ventajas de los demás. Lógicamente esto generó discusiones entre los forzados a migrar sobre los fundamentos para elegir el *criterio modelo* de un país con respecto al suyo.

Durante la fase de análisis de los códigos fuentes preexistentes encontramos una divergencia profunda de criterios de programación, falta de reusabilidad de componentes, múltiples tecnologías implementadas, metodologías informales de testing y varios bugs conocidos que no tenían solución (*memory leaks* por ejemplo; principalmente en los STC que fueron programados con C++; anecdóticamente estos eran obligados a reiniciarse cada madrugada para evitar la acumulación excesiva de memoria). Previo a iniciar nuestro trabajo, se definió que era necesario formar una capa de trabajo o *framework*, y que eventualmente ella sería la candidata a modelo de criterio único para lograr la convergencia y fomentar la reusabilidad.

La primera tarea fue la recolección iterativa de factores comunes y artefactos de software que se repetían entre los distintos STC analizados. Si bien las soluciones variaban dados sus requerimientos específicos, muchos comportamientos eran comunes. Por ejemplo, una de las primeras lecciones aprendidas de este análisis fue conocer el mecanismo de crecimiento (a nivel constructivo) de un sistema transaccional: un procesador de transacciones crece a través *del agregado de nuevas transacciones, o la modificación/sofisticación de transacciones existentes*. Fue clave entender entonces que el concepto de *Transacción* era estelar en todo el modelaje de los STC. Hasta ese entonces, muchos de los STC preexistentes estaban *orientados al flujo de las operaciones* en vez de estar *orientados al tipo de transacción*, sin distinguir de forma coherente los flujos operativos de cada una de las transacciones aceptadas por éstos. Esto causaba que al querer modificar el código de una transacción, se ponía en riesgo la lógica de las otras transacciones (bajo nivel de desacople del código). También se encontraron *God Objects* (generalmente con uno o dos métodos principales) que articulaban el orden de ejecución de todos los pasos que forman una transacción. Si una transacción se diferenciaba de otra en su comportamiento, se usaban condicionales *if-else* que a la larga complicaban las tareas de mantenimiento y/o evolución del código de forma limpia y segura. El *framework* y los STC generados de éste, deberían ser entonces *orientados a la transacción*, en vez de *al flujo de la operación*. De este modo, con la implementación de un patrón del tipo *Strategy*, se podrían definir las transacciones como unidades lógicas de construcción. Es entonces que el primer concepto que se definió en el *framework* fue la clase abstracta *transacción* (*AbstractTransactionHandler*). La clase esta compuesta por métodos que definen los pasos que deberían ejecutarse en una transacción típica. Cada vez que llega una trama de datos a los nuevos STC, se analiza el tipo de transacción, y se instancia la clase homónima correcta que represente al tipo entrante, heredando las características y los pasos comunes definidos en la clase abstracta anterior.

De forma consecutiva, fueron necesarios capitalizar en clases abstractas otros aspectos comunes. En segundo lugar se definieron los *motores de entrada*, que son los encargados de la recepción de requerimientos y del envío de respuestas (datos) hacia los POS (*Point of Sale*) que interactúan con el STC (a través de un puerto TCP). Si la trama de entrada tiene coherencia (cumple con el protocolo) y el tipo de transacción es soportado por el STC, se genera un *thread* para atender el procesamiento de la transacción, y se instancia la clase *transacción* correcta (patrón *Strategy*) según el tipo entrante. Cabe acotar que los *motores de entrada* no son una única clase abstracta, sino una jerarquía de clases abstractas ya que existen varios tipos. Consecutivamente para que los *motores de entrada* reconozcan al tipo de transacción entrante, fue

necesario definir los conceptos de dominio *parser*, *parser stream* y *parser structure*. El primer concepto almacena todas las funcionalidades para entender un protocolo transaccional (por ejemplo ISO8583), el segundo concepto es un envoltorio que almacena la trama de datos entrante o saliente de forma serializada, y el tercer concepto es una estructura de campos con la información deserializada de la trama de datos (*stream*). Los *parsers* pueden serializar y deserializar la información, es decir pueden transformar de *parser stream* (lo que llega o lo que sale por un puerto TCP) a un *parser structure* (información de contexto que nutre a la transacción durante todo su proceso), o viceversa. Por último se definieron los conceptos *contexto transaccional* (*AbstractTransactionContext*), *acceso de datos* (*AbstractTransactionDataSource*) y los *motores de salida* (que son parte de una jerarquía de clases abstractas, como en el caso de los *motores de entrada*). El *contexto transaccional* es un objeto *thread-safe* que acompaña al *thread* de la transacción durante toda su vida y sirve para almacenar datos de cualquier proveniencia que le sean de utilidad a las tareas de procesamiento. También almacena los estados por los que va incurriendo una transacción. Los *accesos a datos* permiten el acceso seguro a una base de datos, a un archivo, o a cualquier mecanismo de persistencia, con el objeto de evitar *memory leaks* e independizar al STC de la lógica de acceso hacia una tecnología de persistencia particular. Y los *motores de salida* son muy similares a los de *entrada* permitiendo comunicar al STC con otros STC, ya sea por una conexión TCP o por un protocolo SOAP (*Web Service*), WCF, XML, etc.

Luego de definir los conceptos comunes de los STC en clases abstractas, se definió en qué orden se ejecutarían los pasos que hacen a la transacción. Se propuso analizar y definir una *secuencia de trabajo genérica de procesamiento*: cada transacción debería poderse resolver en pasos, bajo una secuencia pre-ordenada, y la transacción implementa solo aquellos que sean de utilidad para realizar el correspondiente procesamiento. Es decir, que la *secuencia de trabajo genérica* tiene como objetivo que cualquier transacción pueda realizarse con pasos definidos siempre iguales, pero con posibilidad de sobrecarga para los que desee implementar. Los pasos son los siguientes: *recepción*, *pre-proceso*, *reenvío*, *proceso*, *devolución*, *post-proceso* y *mantenimiento*. La definición de esta secuencia requirió mucho trabajo de revisión de STC preexistentes, y de trabajo iterativo de corrección, de modo de conseguir este conjunto de pasos y el orden que puedan eventualmente satisfacer a múltiples STC desconocidos de antemano. Esta secuencia está implementada en la clase abstracta *transacción*, y una clase hija solo debe sobrecargar aquellos pasos que necesite personalizar según el requerimiento dado para la transacción real. Más aún, dada la jerarquía interna de clases dentro del *framework*, varios de estos pasos pueden tener comportamientos por defecto que faciliten la construcción del STC. Este concepto de sobrecarga de pasos dentro de una jerarquía de clases, bajo una secuencia de ejecución pre-ordenada se formalizó implementando diferentes técnicas de *Inversión de Control (IoC)* y usando el patrón de diseño *Template Method*. A nivel de código fuente, cada paso se definió como un delegado (puntero a función), permitiendo conectar solo aquellos métodos concretos dentro de las subclases.

Una vez definida la *secuencia genérica de trabajo* y la mayoría de las clases abstractas del *framework*, se procedió a realizar un análisis de refactorización a patrones. De este modo se intentó sacar el mayor provecho de los patrones para que los elementos de dominio del futuro DSL puedan ser lo más sólidos posibles a nivel

arquitectura (escalabilidad), y que puedan brindar las funcionalidades planeadas de una forma eficiente y entendible para otros desarrolladores. Por ejemplo, la única forma de acceder al STC es a través de una clase *façade* a través de sus métodos *Start()* y *Stop()*. Por su parte, tanto esta clase *façade* como los *motores de entrada y salida* se definieron como clases *Singleton* ya que deben ser únicos en todo el contexto de ejecución. Esto se definió así para que todos los *threads*, es decir todas las transacciones, puedan accederlos unívocamente para comunicar requerimientos y respuestas.

Por último se propuso formalizar el conocimiento acumulado en un lenguaje de dominio específico. Utilizando las herramientas *DSL Tools* provistas por el IDE *Visual Studio*, se ha generado un metamodelo con la representación gráfica de todos estos elementos de dominio, con propiedades de dominio que permiten su personalización desde el mismo modelo. Se definieron pre-validaciones y post-validaciones por cada elemento, y las relaciones entre estos, para poder interconectarlos. Así también fue necesario desarrollar una herramienta de transformación que interpretara las instancias del metamodelo del DSL, y las transforma a código fuente. Para codificar esta herramienta se utilizó el lenguaje T4 que provee *Dsl Tools* para comprender las instancias del metamodelo. A futuro, estamos considerando poder transformar los modelos generados por este DSL en otros modelos, como por ejemplo diagramas de clase o de secuencia. La lección aprendida de esta implementación MDA-MDD fue que no es posible automatizar por completo la creación de código funcional: siempre habrá características particulares para una solución que deberán personalizarse en cada caso. Es por esto que la herramienta de transformación, genera una estructura de archivos y carpetas ordenadas por capas, de modo que una clase hija que hereda de algún elemento del *framework*, y que haya sido diseñada por el DSL, quede definida en *clases parciales* dentro de dos archivos, uno con la *componente automática* y otro con la *componente manual*. La *componente automática* estará almacenada en un archivo de código, y cada vez que se modifique la instancia del metamodelo del DSL, este archivo se modificará por la herramienta de transformación para quedar sincronizado con el modelo. En cambio, en la estructura de archivos y carpetas existirá una carpeta *CustomCode*, donde se almacenarán los archivos de código que implementan la *componente manual*. Si la *componente manual* no existe y el modelo de DSL se transforma a código, el archivo con esta componente se creará. Si el archivo ya existe no se sobrescribirá, permitiendo almacenar los cambios que hagan los programadores con las personalizaciones que no se puedan representar desde el modelo.

3 Propuesta de Implementación TDD

La metodología *Test-Driven Development* (TDD), es un enfoque evolucionario para la construcción de software, en la que se fomenta la escritura del código de los casos de prueba previo a la construcción de código productivo que cumpla con esa prueba. Esta metodología se basa en la repetición de ciclos de desarrollo de software cortos que contengan las siguientes prácticas: primero el desarrollador escribe casos de prueba automatizados (que fallan en un principio por ausencia de código funcional) y

que definen la funcionalidad o comportamiento deseado, luego se produce la mínima cantidad de código para cumplir con el caso de prueba, y luego se refactoriza el código generado para cumplir con los estándares de aceptación [7]. Como resultado de implementar TDD, los desarrolladores están más concentrados en definir la interface de pruebas que cumple con los requerimientos, que con la implementación propiamente dicha. Es decir, el desarrollo se ajusta más a un *diseño por contrato* que resulta en código más conciso, más flexible y modularizado. A su vez se reduce el acoplamiento entre clases e interfaces, haciendo al código de las mismas más limpio y puntual en pos de cumplir con los casos de unidad definidos.

Durante la fase de análisis de los STC preexistentes, se detectó que no existían políticas de pruebas de validación bien definidas. Tampoco existían casos de unidad desarrollados en el código que validen los requerimientos de estos sistemas. Favorablemente, la mayoría de las pruebas de validación que se realizaban informalmente consistían en enviar transacciones al STC, analizar los *logs* (las líneas escritas en un archivo de texto en cada paso del procesamiento) y validar las respuestas recibidas del STC. En otras palabras, es una ventaja de los STC que el único punto (o el punto principal) de interacción hacia el exterior sea a través del intercambio de transacciones. Es por ello que se propuso implementar una metodología TDD, orientada al dominio del procesamiento transaccional, basado en el factor común del intercambio de requerimientos y respuestas contra un STC.

Las instancias del metamodelo permiten definir en los *motores de entrada* la propiedad de dominio "*incluir pruebas unitarias*". Si esta propiedad es *true* la herramienta de transformación a código fuente generará dos casos de prueba unitarios por cada *transacción* atada al *motor de entrada* en la instancia del metamodelo: un *test* para simular la aprobación de la transacción, y otro para simular el rechazo de la misma. Además, el código que implementa los casos de prueba se distribuye también entre una *componente manual* y otra *automática*. En la *componente manual*, la herramienta de transformación genera automáticamente una serie de métodos, particularmente cuatro métodos por cada transacción vinculada al *motor de entrada*. Dos de esos métodos permiten al desarrollador armar una instancia de *parser structure* de modo de simular una trama aprobada, y otra trama rechazada. El archivo que almacena esta *componente manual* se mantendrá persistente más allá de sucesivas automatizaciones de la herramienta de transformación, guardando los cambios hechos por los desarrolladores. En cambio, en la *componente automática*, se generan los métodos de prueba unitarios propiamente dichos, que pueden ser llamados desde herramientas de ejecución de *unit test*, como *NUnit*. Este archivo sí se regenerará por cada proceso de automatización de la herramienta. Las pruebas unitarias hacen llamadas a los métodos de la *componente manual* para poder serializar y enviar requerimientos al STC, y eventualmente recibir respuestas y *deserializarlas* en pos de realizar las validaciones pertinentes. Por último, las validaciones son realizadas solo si se reciben respuestas del STC. Los otros dos métodos de la *componente manual* son los encargados de realizar estas validaciones, siendo uno de ellos el agente validador para el caso de aprobación y el otro para el caso de rechazo. Cada respuesta deserializada es pasada por parámetro a uno de estos métodos (según sea el caso) como una instancia del tipo *parser structure*. Dentro de estos métodos se implementará el código que definirá si la prueba unitaria fue exitosa o fallida.

4 Evaluación de la Implementación MDD y TDD

El proyecto de I/D/I comenzó en Febrero de 2012, y durante todo ese año se realizó el análisis de STC preexistentes, la definición de los elementos comunes, la refactorización de código, y la implementación de las clases abstractas para la consecutiva construcción del marco de trabajo. Durante el primer semestre se realizó la implementación de la metodología MDD y TDD, entregando la primera versión de los productos (el marco de trabajo, el DSL y la herramienta de transformación) en Julio de 2013. Los productos se evaluaron en el ambiente corporativo, de modo de conocer si cumplía con los objetivos iniciales definidos.

Contabilizando desde el último semestre de 2013 hasta el primer semestre de 2014, se realizaron bajo esta implementación desde *scratch* 9 autorizadores STC: 6 de venta de tiempo-aire de telefonía celular, 1 para venta de vales, 1 seguros de robo y 1 de lotería. Además se analizaron 8 STC (escritos en C++ y C#) previos a la implementación dual. Más allá que los STC no son exactamente iguales, si comparamos aquellos con requerimientos similares (cantidad de transacciones parecidas, complejidad similar, etc.), se pueden obtener resultados comparativos de la evaluación entre los servidores transaccionales anteriores y posteriores a dicha implementación. Las evaluaciones son las siguientes:

- Los tiempos de desarrollo disminuyeron entre 45% y 70% en 8 de los 9 casos post-implementación (siendo la excepción el autorizador de vales). Para este último caso, el *motor de salida* debía cumplir con una serie de requisitos imprevistos en la primera versión del *framework*, por lo que se definió un nuevo tipo de *motor de salida*, se implementó, se integró a este *framework* y se representó en el DSL. Ese *costo* puede considerarse *inicial*, dado que para próximas soluciones ya estará listo para ser reutilizado.
- Se logró homologar los conceptos de los desarrolladores en los equipos distribuidos de trabajo. De pasar a trabajar de forma independiente, se logró centralizar el equipo y ahora todos colaboran desarrollando STC para países diferentes al de locación de cada individuo.
- La tasa de errores durante los primeros tres meses (luego de la puesta en marcha) de cada STC, también se redujo. Las incidencias registradas para aquellos STC desarrollados previos al 2013 denotan que fueron necesarios cambios *en caliente* en la mayoría de los casos, al corto plazo de haberse puesto cada STC en producción. En los STC desarrollados post-implementación, hubo veces que fue necesario hacer cambios sobre la marcha al corto plazo pero principalmente por errores de comprensión de un requerimiento en particular. Los errores de código dentro de un requerimiento comprendido se han disminuido casi en su totalidad, dado que los elementos de dominio han sido probados en otras soluciones con anterioridad.
- Se logró aumentar la visibilidad del código: A partir de la implementación realizada, se han hecho reuniones retrospectivas de revisión de arquitectura trimestrales. Tanto los desarrolladores, como los líderes de

equipo de cada país están al tanto de la arquitectura de los STC en producción.

También se encontraron puntos débiles en los STC contruidos bajo esta propuesta MDD/TDD. En primer lugar, la implementación del DSL y de la herramienta de transformación está atada en esta primera versión a la tecnología *.NET Framework* de *Microsoft*. Del mismo modo, las clases para accesos a base de datos fueron diseñadas para trabajar contra un motor de DB *SQLServer*, por lo que en la versión actual no hay compatibilidad con otros motores tales como *Oracle*. Desde el lado TDD, la implementación tuvo éxito aunque se aprovechó solo la capacidad de validación de la metodología (con las pruebas unitarias solo se validó el código de las *componentes automáticas y manuales*). Aún no se conjugó la filosofía TFD (*Test-First Development*), que podría ser de utilidad, por ejemplo, para la construcción de la *componente manual*.

5 Trabajos Relacionados

Del conjunto de artículos recolectados durante la primera fase del proyecto, se destacan estos tres trabajos:

El primer trabajo (*Multi-threading technique for authorization of credit card system using .NET and JAVA*) del autor W. Y. Ming [1] propone un STC 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 STC pueden paralelizarse dentro de cada hilo de ejecución de cada transacción, por ejemplo, las operaciones criptográficas. Muchas de estas operaciones se hacen con dispositivos de E/S (un HSM conectado al servidor), y la interacción contra estos puede llegar a ser costosa en tiempos de respuesta. Este STC es realizado tanto en plataformas *Java* como en *.NET Framework*. Sin embargo no trabaja sobre la formalización de esta técnica *multithreading* en elementos de dominio reutilizables.

El segundo trabajo (*Lagniappe: Multi-* programming made simple*) [6] 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 hizo en nuestro artículo, de modo de representar conceptos comunes), en cambio solo define unos elementos de dominio *reducidos*, que pueden conectarse entre sí para formar elementos de dominio más complejos.

El tercer trabajo (*Domain-Specific Languages for Enterprise Systems*) [8] analiza una arquitectura llamada POETS, para la construcción orientada a eventos en sistemas del tipo ERP. En este artículo se hace una propuesta de modelización MDD sobre POETS, proponiendo una serie de elementos de dominio. El trabajo describe un caso de uso del DSL implementado a modo de evaluación. La forma de exposición de este trabajo (dominio, DSL y evaluación) sirvió como modelo para nuestro artículo.

6 Conclusiones y Aportes

Este artículo presentó un análisis de elementos comunes para el dominio de sistemas transaccionales concurrentes (STC). A partir de este análisis se realizó la organización de todos los conceptos recolectados en una jerarquía de clases abstractas dentro de un *framework*. A su vez este *framework* fue la base para la construcción de un DSL que permita la implementación de una metodología MDD. Por último se desarrolló una herramienta de transformación de instancias del meta-modelo a código funcional, y se integró parcialmente una técnica de casos de unidad (TDD) orientada al dominio bajo análisis.

El aporte de este trabajo es la definición del conjunto de elementos de dominio funcionales basados en factores comunes analizados de diversos STC. El DSL pretende ser útil para la construcción de otros STC, cualquiera fuera el tipo. Bajo la premisa que los elementos de dominio aquí analizados deberían encontrarse en cualquier STC, nuestra visión a futuro es que estos faciliten el diseño de otros servidores transaccionales, y que otros trabajos sobre el tema puedan extender el DSL con nuevos elementos de dominio aun no definidos. Así también se podrían rediseñar aquellos aquí presentados para que representen la realidad de forma más precisa, o funcionen de mejor forma. Por último la implementación TDD aumenta la confiabilidad del sistema a través de la automatización de pruebas unitarias de validación. En este trabajo se aporta un enfoque para implementar las pruebas unitarias como la combinación de transacciones y sus resultados esperados, operando al STC como un objeto cerrado (*black box*) que recibe *entradas* y devuelve *salidas*.

Referencias

1. Ming, W. Y.: Multi-threading technique for authorization of credit card system using .NET and JAVA, Kuala Lumpur, 2007.
2. Kang, K., Lee, J., Kim, B., Kim, M., Seo, C., Yu, S.: Re-engineering a credit card authorization system for maintainability and reusability of components—a case study. In: Reuse of Off-the-Shelf Components, pp. 156--169. Springer Berlin Heidelberg (2006)
3. Pons, C., Giandini, R., Perez, G.: Desarrollo de software dirigido por modelos. Conceptos teóricos y su aplicación práctica. EDULP, La Plata, Argentina (2010)
4. Singh, Y., Sood, M.: Models and Transformations in MDA. In: First International Conference on Computational Intelligence, Communication Systems and Networks. (2009)
5. Bishop, J.: CSharp 3.0 Design Patterns. O' Reilly (2008)
6. Riché, T.L., Lavender, G., Vin, H.M.: Lagniappe: Multi-* programming made simple. Proceedings - International Conference on Computer Communications and Networks, ICCCN, art. no. 4317815, pp. 173-178. (2007)
7. Kent, B.: Test-Driven Development by example. Addison Wesley (2003)
8. Andersen, J.B. , Bahr, P.A. , Henglein, F.A. , Hvitved, T.A.: Domain-specific languages for enterprise systems - Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8802, pp. 73-95. (2014)