AGOSTO DE 2018

SISTEMA DE CONMUTACIÓN DE TARJETAS DE REGALO (GIFT CARDS) BASADO EN JPOS

EBER C. BEZZONE

DNI: 36.222.501

MODALIDAD: PRÁCTICA EN EMPRESA

PROYECTO FINAL PARA LA OBTENCIÓN DE TÍTULO DE INGENIERO EN SISTEMAS PLAN 2004

EMPRESA: TECRO INGENIERÍA S.A

Agradecimientos

Llegar hoy a esta instancia universitaria es producto de haber tenido siempre e incondicionalmente a mi lado a personas que fueron de incalculable apoyo. En primer lugar, mi familia, que fueron, son y serán un pilar incondicional en cada una de las actividades en las que deseo incurrir, incluso en aquellas que involucran estar a muchos kilómetros de distancia durante periodos prolongados de tiempo. A mis amigos, los de siempre, por seguir estando presentes en cada momento. A los amigos que hice durante este periodo, con quienes tuve la suerte de compartir charlas, mates y trasnoches de estudio, y que a pesar de los momentos más estresantes siempre estuvieron para brindar su ayuda, su palabra y calma.

A la UNLPam por brindarme la posibilidad de formarme como profesional, en especial a la Facultad de Ingeniería y el personal que la compone, que siempre brindó lo mejor para lograr que más alumnos puedan cumplir sus metas. A la empresa Tecro Ingeniería S.A por dejarme realizar este trabajo dentro de la organización, por ayudarme a crecer como profesional y persona. A cada uno de mis compañeros de trabajo y jefes, por la ayuda brindada y el aguante, tarea que sé que a veces puede no ser fácil.

Por último, a cada integrante del grupo de investigación LISI, que, durante mi periodo de investigador, me ayudaron de manera desinteresada a crecer como profesional. En especial a mi directora de investigación y tutora universitaria de este proyecto, Dra. Gabriela Minetti, quién con su paciencia y calma me guio a lo largo de estos años.

Índice

Ilustraciones

Algoritmos

Resumen

En la actualidad la empresa Tecro Ingeniería S.A trabaja tanto en proyectos propios de la organización, como también así, en convenios con otras empresas situadas en otros países. Una de las empresas con la cual Tecro Ingeniería S.A trabaja continuamente, dedicada principalmente a la conmutación de tarjetas de regalo, presenta la necesidad dentro de uno de sus proyectos de conectarse con un nuevo *endpoint*. Esto le permitirá poder comenzar a operar con tarjetas y transacciones que son soportadas por este *endpoint* en cuestión.

Esto conllevó a la necesidad de anexar a dicho proyecto, denominado *switch (software de conmutación)*, la capacidad de poder conectarse con un nuevo *endpoint*, como así también de desarrollar las características necesarias para poder procesar los nuevos tipos de transacciones. Esto produjo como resultado que todas las organizaciones que actualmente se encuentran utilizando el servicio del *switch*, al igual que aquellas empresas que comenzaran a operar sobre el sistema, puedan brindar en sus comercios productos que son producidos por el *endpoint*.

Actualmente, el sistema pasó exitosamente por todas las etapas de pruebas a la que son sometidas las nuevas características de un proyecto de esta índole. Seguido a esto, el proyecto fue puesto en producción para que los comerciantes que trabajan con este *switch* puedan utilizar las nuevas características desarrolladas y que dan origen a este trabajo.

Capítulo 1 – Introducción

Este capítulo aborda el problema que motiva la realización del presente trabajo, los objetivos propuestos, la metodología que será utilizada para alcanzar los mismos, los resultados que se esperan obtener y, por último, cómo se organiza el presente documento.

1.1 Problema

Tecro Ingeniería S.A es una empresa ubicada en la localidad de General Pico, la cual se dedica (en mayor medida) al desarrollo, mantenimiento y actualización de software de conmutación de tarjetas, los cuales son llamados "*switch*". Tecro Ingeniería S.A trabaja con la empresa Emp-X[1] desde el año 2011. En este esquema de negocios, es que la Emp-X maneja su propia cartera de clientes, a los cuales se les desarrolla diferentes aplicaciones según las necesidades de cada uno de ellos. La empresa Emp-X terciariza a través de Tecro Ingeniería S.A gran parte del desarrollo, mantenimiento y actualización de estas aplicaciones.

En la actualidad, un nuevo cliente comenzará a operar con la empresa Emp-X, esto quiere decir, que esta nueva empresa comenzará a procesar *gift cards* de diferentes *endpoints*. Dichos *endpoints* son las compañías que, para comenzar, emiten las tarjetas de regalos (*gift cards*) y las distribuyen en los diferentes puntos de venta, para que puedan ser compradas por los consumidores finales. Estas personas adquieren los productos, y, en línea de caja, realizan diferentes operaciones sobre dichas tarjetas. Estas operaciones dependen del tipo de tarjeta, el proveedor, entre otros.

Actualmente la empresa Emp-X tiene muchos *switchs* de diferentes características en funcionamiento. El nuevo cliente que comenzará a operar requiere de un producto de características similares a las ya desarrolladas, por tal motivo se considera la posibilidad de integrarlo a uno de los *switchs* actuales. Dicho *switch* se encuentra actualmente en producción y en correcto funcionamiento para una cierta cantidad de clientes de la empresa Emp-X. Pero, la llegada de otro cliente implica requerimientos nuevos, y, por consiguiente, nuevas funcionalidades y características deben ser desarrolladas e implementadas.

En este caso, el *switch* que se verá afectado cuenta actualmente con la conexión a dos *endpoints*; que van a ser utilizados por el nuevo cliente, quien, a su vez, necesita la conexión a dos nuevos *endpoints*. Por lo tanto, para ello, se deben realizar las conexiones pertinentes, como así también, las modificaciones necesarias para poder soportar el procesamiento de las operaciones que estas tarjetas requieran.

1.2 Objetivos de trabajo

Los objetivos que propone la empresa al grupo de trabajo son amplios y complejos, lo que conlleva a la división de tareas y actividades entre los diferentes miembros del grupo.

Basado en las actividades asignadas, es que el siguiente trabajo se propone explicar las tecnologías, metodologías y conocimientos que son necesarios para llevar a cabo una nueva

[1] Por razones de confidencialidad no se dará el nombre real de la empresa.

comunicación entre el *switch* que actualmente está en funcionamiento y un nuevo *endpoint*. Esto implicará poder establecer dos canales de comunicación entre el *switch* y el nuevo *endpoint*, como así también el desarrollo de los mensajes que participantes tiene que enviar y procesar para saber si dichos canales de comunicación se encuentran activos. Es de vital importancia en estos sistemas críticos detectar la falla de cualquier canal de comunicación lo antes posible, lo que permite tomar decisiones que minimicen las pérdidas.

El hecho de que el sistema soporte un nuevo operador implica que, además de la comunicación, se deben desarrollar las características necesarias para que el *switch* pueda procesar todas las operaciones que las poseen las tarjetas vendidas por el nuevo cliente. Esto quiere decir que, este *switch* tiene que estar preparado para procesar activaciones, desactivaciones, compras, recargas de saldo, entre otras.

En función a lo especificado en los párrafos anteriores y para llevar a buen término este trabajo se establecen los siguientes objetivos a cumplir:

- Relevar y analizar la situación actual del *switch* para poder contar con una perspectiva de las funcionalidades que actualmente están desarrolladas. También, permitirá familiarizarse con el proyecto, como así también con el *framework* que se utiliza.
- Analizar las distintas soluciones posibles a los fines de ver cuales de estas se adaptan mejor a la situación problemática anteriormente planteada.
- Diseñar como se llevará a cabo la implementación de la/s solución/es anteriormente seleccionada/s.
- Desarrollar las características necesarias para cumplir con los requerimientos planteados por la Emp-X.
- Desarrollar las pruebas de test necesarias con el fin de corroborar el correcto funcionamiento de lo desarrollado.
- Verificar que la documentación suministrada por la Emp-X esté actualizada, caso contrario, sugerir los cambios que sean pertinentes.

1.3 Metodología de trabajo

Dada que la modalidad seleccionada para este trabajo es "práctica en empresa", la primera actividad consistirá en la aclimatación con el ambiente de trabajo de la empresa Tecro Ingeniería S.A. Una vez adaptado con el lugar de trabajo, se llevará a cabo la familiarización con el proyecto que se está desarrollando. Esto implica tomar conocimiento de las tecnologías que están siendo utilizadas, como los lenguajes de programación, *frameworks* y las diferentes herramientas con las que se está trabajando. A su vez, se requerirá la instrucción sobre el estado actual del proyecto, la problemática a resolver, como así también, las diferentes actividades a desarrollar.

Paralelamente a la tarea anterior, una vez familiarizado con las actividades a desarrollar, es que se avanzará en el análisis de las posibles soluciones a ser implementadas. Esto implica la toma de decisiones sobre cuáles serán las mejores alternativas para implementar la conexión

entre el *switch* y el nuevo *endpoint*. También comprende el análisis de cómo estas afectan al proyecto final y al grupo de desarrollo.

Luego, una vez que las decisiones sean tomadas, se comenzará con la implementación de esta nueva conexión. Dicha actividad estará dirigida y avalada por el tutor dentro de la empresa. Esto producirá que el *switch* pueda establecer una comunicación constante con el nuevo *endpoint*. A sí mismo todas las actividades de desarrollo, que se lleven a cabo dentro de la empresa, serán acompañadas de los pertinentes test que sean necesarios para asegurar el correcto funcionamiento de lo desarrollado.

Completada la fase de desarrollo para poder establecer la conexión entre el *switch* y el *endpoint*, se continuará con la familiarización de las operaciones que soportará el nuevo *endpoint*. Esto implica, no solo comprender qué producen como resultado final cada una de las diferentes transacciones, sino también entender qué tipos de mensajes generan estas nuevas operaciones. De esta manera, se podrá llevar a cabo el correspondiente desarrollo de cada uno de los mensajes que van a ser procesados por el sistema.

Dichas actividades, a su vez, también estarán acompañadas de los test correspondientes, al igual que la supervisión del tutor dentro de la empresa.

Dado el modelo de negocios con el que cuenta, tanto la Emp-X como así también Tecro Ingeniería S.A, es que eligen trabajar con metodologías ágiles. En particular, la metodología de trabajo utilizada para este proyecto es SCRUM, que permite la creación de equipos auto-organizados. Los cuales impulsan la continua comunicación verbal entre todos los miembros del equipo, basándose en reuniones diarias y comunicación constante, lo que produce un trabajo iterativo e incremental. Dicha metodología de trabajo será abordada con más detenimiento en el capítulo 4.

1.4 Resultados esperados

Al finalizar dicha experiencia dentro de la empresa se espera haber concluido exitosamente todas las actividades planteadas, en los tiempos que fueron estipulados. Esto incluye, no solamente, el correcto desarrollo de las actividades que darán nuevas funcionalidades al *switch*, sino que también, un desenvolvimiento apropiado dentro de un ámbito laboral ya establecido, con una adecuada y fluida comunicación, principalmente, con aquellos compañeros con lo que se deberá trabajar dentro del mismo proyecto.

Por otro lado, en cuanto a lo académico, se espera poder aprender sobre nuevas tecnologías y herramientas de trabajo, que son utilizadas por las diferentes empresas y organizaciones día tras día. Muchas de las herramientas y *frameworks* a utilizar fueron al menos nombrados dentro de las diferentes materias a lo largo de la carrera universitaria. Pero, por razones de tiempo, no todos son vistos en profundidad, por lo que resulta beneficioso poder aprender con más detenimiento sobre estas tecnologías.

1.5 Organización del documento

El presente documento se encuentra divido en una totalidad de 5 capítulos. En el primer capítulo llamado "Introducción" se encuentra la información referente al problema que será

abordado, los objetivos de este trabajo, la metodología que será utilizada para llevar a cabo dicho trabajo, como así también los resultados que se esperan obtener al concluir.

En el capítulo número 2 "Lenguajes de programación y *frameworks*" se describa la tecnología java como también los *frameworks* que serán utilizados, describiendo de esta manera *hibernate* y jPOS.

En el capítulo 3 "Metodología de trabajo", se describirá cual es la metodología utilizada por la empresa para llevar a cabo sus diferentes proyectos, de esta manera se abordará la metodología Scrum y el trabajo con integración continua, también se abordará la realidad de la empresa en su día a día.

En el capítulo número 4 "Arquitectura del sistema" se describirá la arquitectura básica de un *switch*, y a su vez, será descripto con mayor precisión el funcionamiento del *switch* particular sobre el cual se estará trabajando.

Por último, en el capítulo 5 "Desarrollo de las actividades" se especificará en detalle las diferentes actividades que se deben realizar a lo largo del proyecto.

Por último, se especifica cuáles son las conclusiones obtenidas del presente trabajo.

Capítulo 2 – Lenguaje de programación y frameworks

En el presente capítulo se aborda una introducción al lenguaje de programación java, el cual será utilizado en el proyecto de desarrollo. A su vez, también se abordarán los diferentes *frameworks* utilizados, siendo estos mismos *Hibernate* y jPOS. Cabe destacar que java no es el único lenguaje de programación involucrado en el proyecto. En la actualidad, este cuenta con diferentes grupos de trabajo, afectados a diferentes tareas donde se desarrolla el software con múltiples lenguajes de programación, como así también con diferentes *frameworks*. Estos no serán abordados aquí, dado que no intervienen en las tareas propuesta en el presente trabajo.

2.1 JAVA

La tecnología java [1] consta, esencialmente, de dos partes: una plataforma y un lenguaje de programación. La plataforma java, a su vez, se encuentra conformada también por dos partes: JRE *(Java Runtime Environment)* y JDK (*Java Development Kit*).

El JRE (*Java Runtime Environment*), traducido al español como "entorno de ejecución java" es el que contiene las bibliotecas que ofrecen los diferentes servicios, como asi también, la JMV *(Java Virtual Machine),* traducida al español como máquina virtual de java, la cual se encarga de ejecutar los *bytecode* java. El JRE no posee compiladores ni herramientas para desarrollar las aplicaciones Java, solo posee las herramientas para ejecutarlas.

El JDK (*Java Development Kit*) o, en español, herramientas de desarrollo java. Son un conjunto de herramientas que permiten el desarrollo de programas en el lenguaje java. Dentro de estas herramientas podemos encontrar el compilador (javac), el cual se encarga de convertir el código fuente java a *bytecode,* un formato de código objeto independiente del sistema operativo y el hardware, el cual será interpretado por la JVM. Dentro de este paquete encontramos otras herramientas como el desensamblador de binarios (javap), *debugger*, entre otras.

Dentro de las plataformas de java se reconocen inicialmente cuatro: Java SE, javaFX, java ME y java EE. A lo largo de la historia, java ha ido actualizando sus plataformas, uniéndolas y creando nuevas basadas en las necesidades que la actualidad requiere. Aquí se detallan las más utilizadas a lo largo del tiempo.

Java SE: [2] *Java Standard Edition* (SE) es la edición más difundida de las plataformas java. Esta plataforma ofrece las funcionalidades básicas, sin dejar esta de ser una plataforma completa en cuanto a funcionalidades. Esta plataforma define desde los tipos básicos de datos, hasta clases de alto nivel que facilitan el trabajo con el acceso a datos, redes, seguridad, análisis de XML y desarrollo de interfaces gráficas de usuario, también llamadas de GUI (*graphical user interface*).

Como se muestra en la Ilustración 1 - componentes de java SE-, esta plataforma se compone de JRE como todas las plataformas javas, un JDK y JSR (*Java Specification Requests*, solicitudes de especificación de java en español) de funcionalidades limitadas para la plataforma java SE.

Ilustración 1 - Componentes de Java SE
Fuente: https://upload.wikimedia.org/wikipedia/commons/e/e1/Java_Platforms.PNG

JavaFX: "La plataforma de cliente enriquecido". Esta plataforma proporcionaba una interfaz de usuario ligera y acelerada por *hardware* para aplicaciones empresariales. Esta plataforma amplía su potencia permitiendo utilizar librerías de cualquier plataforma java en aplicaciones javaFX. A partir de Java SE 7 *Update* 6, Oracle JavaFX forma parte de Java SE de Oracle [3]. Es decir, que con una sola instalación (java SE) se puede hacer uso de las herramientas y librerías de javaFX, ya que estas vienen insertadas dentro del JDK (JSE).

Java ME: *Java Micro edition,* [4] es una plataforma de java que ofrece un entorno robusto, pero a su vez flexible, el cual es ejecutado en dispositivos integrados y móviles. La API de java ME está compuesta por un subconjunto de la API de java SE. En muchas de las ocasiones que se desarrollan aplicaciones con java ME, éstas son clientes de los servicios de plataformas java EE. Las aplicaciones en la plataforma java ME son portátiles en muchos dispositivos, pero, a su vez, aprovechan las capacidades nativas de cada dispositivo.

Java EE: *Java Enterprise Edition* [5], traducida al español como java empresarial, es una plataforma impulsada principalmente por la misma comunidad que la utiliza, esto implica la contribución de expertos de la industria, organizaciones comerciales, los propios usuarios de la plataforma, entre otros. Java EE está a la vanguardia de las necesidades de la industria actual, con el fin de brindarle soluciones para las problemáticas actuales, mejorar la portabilidad de las aplicaciones, y a su vez, aumentar la productividad del desarrollador.

Como se puede observar en la Ilustración 2, java EE está compuesto por los mismos componentes de java SE, pero a su vez se le suman un JSR y un JDK extendido. Java EE cuenta dentro de sus librerías con herramientas para diferentes funciones; algunas de ellas son para conexión a base de datos: *Java DataBase Connectivity* (JDBC), invocación a métodos remotos *Remote Method Invocation* (RMI), e-mails como *Java Message Service* (JMS), servicios web como *Extensible Markup Language* (XML), entre otros.

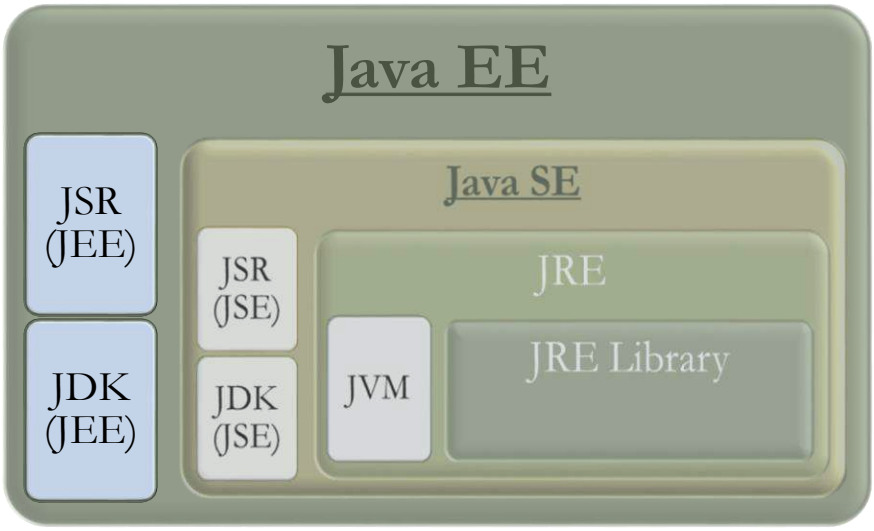

Ilustración 2 - Componentes de Java EE
Fuente: https://upload.wikimedia.org/wikipedia/commons/e/e1/Java_Platforms.PNG

2.2 *Frameworks*

Una de las primeras definiciones de *framework* fue brindada por Johnson y Foote en la revista *Journal of Object-Oriented Programming*: "Un *framework* es un conjunto de clases que incorpora un diseño abstracto para soluciones a una familia de problemas relacionados, y admite reutilizaciones en una granularidad mayor que las clases" [6]. Partiendo de esta definición, nos encontraremos con una gran similitud entre lo que se define como un *framework* y lo que conocemos como una librería de clases. Si bien en una primera impresión pueden ser objeto de confusión, un *framework* es un entorno de desarrollo completo que suele ofrecer herramientas tales como el compilador, el *debugger* e incluso un editor de código. Pero la principal diferencia, es que, un *framework* cuenta con una amplia biblioteca de librerías, las cuales hacen el corazón del *framework* y por lo general pueden ser extendidas, pero no modificadas.

En las siguientes secciones se describen los dos *frameworks* que son utilizados en este proyecto: *Hibernate* y jPOS.

2.2.1 *Hibernate*

Antes de explicar qué es y cómo funciona *Hibernate*, es necesario introducir ORM (*Object-Relational mapping),* o en español, mapeo de objetos relacionales. ORM es un modelo o técnica de programación que consiste en la conversión de datos entre el sistema de tipos y una base de datos relacional como motor de persistencia. Esto se lleva a cabo a través de una programación orientada a objetos, es decir, una serie de entidades simplifican las tareas básicas de acceso a los datos, lo cual permite programar de manera independiente al motor de base de datos que se esté utilizando, permitiendo al programador concentración absoluta sobre el desarrollo de la aplicación sin la necesidad de conocimientos específicos sobre el motor de base de datos particular. Un claro ejemplo de esto es una simple consulta SQL "*select*", pero limitando la cantidad de filas que se desea sean devueltas por el motor de base de datos. Como es apreciado en el Algoritmo 1 –"Consulta SQL *select* usando distintos

motores de búsqueda" se puede apreciar como una misma consulta es escrita en lo que el autor considera los 3 motores de base de datos más utilizados.

```
1.  SELECT TOP 10 * FROM usuarios    //SqlServer
2.  SELECT * FROM usuarios LIMIT 10    //MySQL
3.  SELECT * FROM usuarios WHERE rownum<=10; //Oracle
```

Algoritmo 1 - Consulta SQL *select* usando distintos motores de búsqueda

La misma consulta se escribe de maneras diferente, dependiendo del motor de base de datos utilizados, gracias a los ORM esto pasa a ser transparente para el programador. El ORM al tener una capa intermedia, abstrae al programador de la base de datos y lo centra en el desarrollo de la aplicación como se muestra en la Ilustración 3.

Hibernate es un *framework* ORM desarrollado en java, con una versión para .Net conocida como *NHibernate*. Dicho *framework* es de código abierto bajo la licencia GNU LGPL [7]. Utiliza el principio de reflexión de java que permite a un objeto que se encuentra en ejecución, examinarse y manipularse a sí mismo.

Hibernate [8] tiene la capacidad de adaptarse a una base de datos en funcionamiento, como asi también, de crear una nueva a partir de la información que contiene y se le fue especificada. Para que esto se pueda llevar a cabo, el desarrollador debe detallar el modelo de datos, qué relación existe entre estos y de qué tipo es cada uno. Con esta información, *hibernate* también tiene la capacidad de manipular los datos de la base de datos, operando simplemente sobre los objetos. Todo esto acompañado de las ventajas que contiene la programación orientada a objetos. *Hibernate* cuenta con la capacidad de poder convertir los datos utilizados por java a los datos que son aceptadas en los diferentes motores de base de datos. El *framework* traduce las peticiones del desarrollador en sentencias SQL exclusivas para cada motor de base de datos, ofreciéndolo al programador un lenguaje de consultas llamado HQL (*Hibernate Query Language*), o en español lenguaje de consultas de *Hibernate*, junto con una API para construir las consultas de manera programática, las cuales son llamadas de *criterias*.

Una perspectiva de alto nivel de cómo es la arquitectura de *Hibernate* es mostrada en la Ilustración 4. Aquí se puede observar que el *framework* crea una capa intermedia entre la base de datos y la aplicación java donde se detallan los archivos de configuración como la conexión a la base de datos, las entidades, las relaciones entre estas, etc.

Ilustración 3 - Comportamiento de un ORM

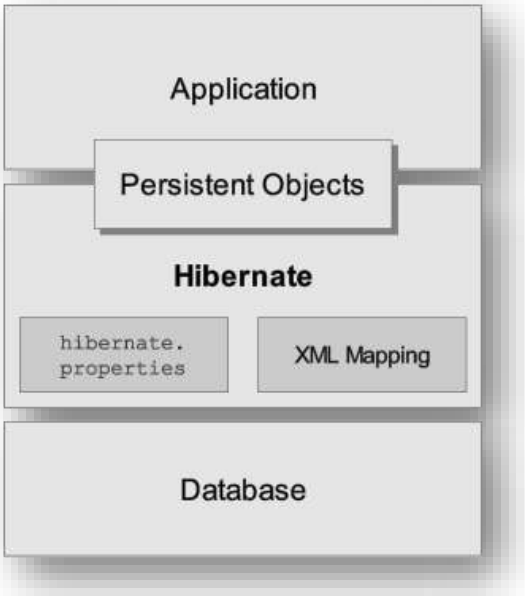

Ilustración 4 - Arquitectura de *Hibernate* (Perspectiva de alto nivel)
Fuente: https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/architecture.html

Hibernate permite trabajar en una arquitectura "mínima" como se muestra en la Ilustración 5. De este modo la aplicación es la encargada de proporcionar sus propias conexiones JDBC, además, es la misma aplicación la encargada de administrar las transacciones. Por lo tanto, se requiere la utilización de un subconjunto mínimo de las APIs de *Hibernate*.

En el caso de la arquitectura "completa" como se ejemplifica en la Ilustración 6, la aplicación se abstrae de las APIs de JDBC/JTA y permite que *hibernate* sea el encargado de administrar las conexiones. Una vez creada la conexión, para que los datos persistan en la base de datos, el *framework* crea una clase de instancia entidad, la cual es una clase java que esta mapeada a una tabla de la base de datos. Este, es un objeto transitorio, ya que todavía no ha almacenado información alguna ni está asociado a ninguna sesión. Para guardar el objeto de base de datos, se crea la instancia de la interfaz *SessionFactory*, la cual es una instancia *singleton*. Cada conexión de

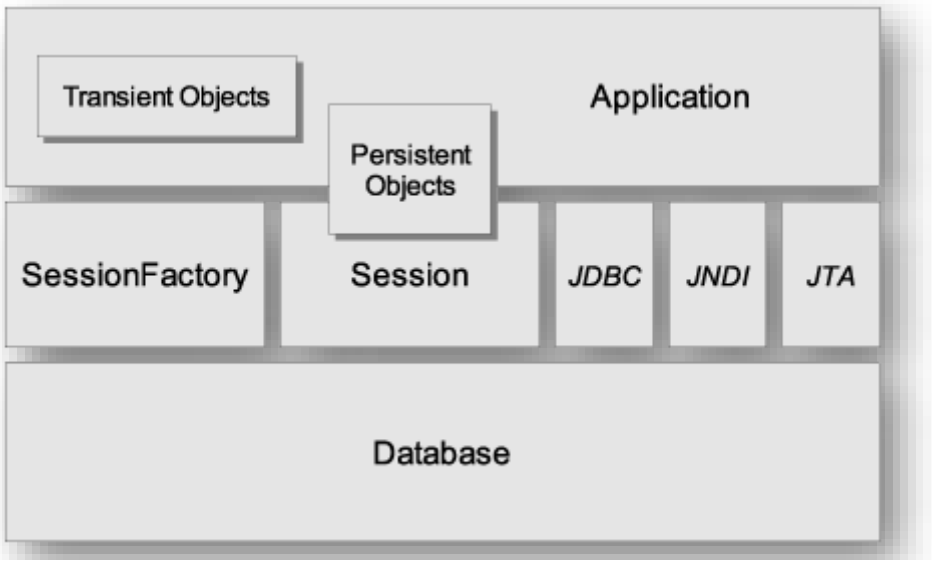

Ilustración 5 - Arquitectura mínima de uso
Fuente: https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/architecture.html

base de datos se produce mediante la creación de una instancia de la interfaz *Session*. Ésta representa una única conexión con la base de datos. A continuación, se detallan los componentes más importantes que se pueden apreciar en la Ilustración 6.

SessionFactory (org.hibernate.SessionFactory): Es una memoria caché que contiene los mapeos compilados para una sola base de datos. Puede contener de manera opcional una memoria caché de segundo nivel, donde aloja datos que son reusables entre las transacciones a nivel de proceso.

Session (org.hibernate.Session): Es un objeto *singleton* de corta duración, que representa la conversión entre la aplicación y el almacenamiento persistente. *Session* mantiene una caché de primer nivel de objetos persistentes que se utiliza cuando se busca objetos por el identificador.

Objetos y colecciones persistentes: Son objeto/s de corta duración, *single-threaded* (de hilo único) que contienen un estado persistente como asi también, una funcionalidad de negocio. Estos se encuentran asociados a solo una sesión (*session*), una vez que esta sea cerrada, los objetos son liberados para usar en cualquier capa de aplicación.

Objetos y colecciones transitorios y separados: Son instancias de clases persistentes que no se encuentran actualmente asociadas con una *Session*. Pueden haber sido instanciadas por la aplicación y aún no son persistentes, o pueden haber sido instanciadas por una *Session* cerrada.

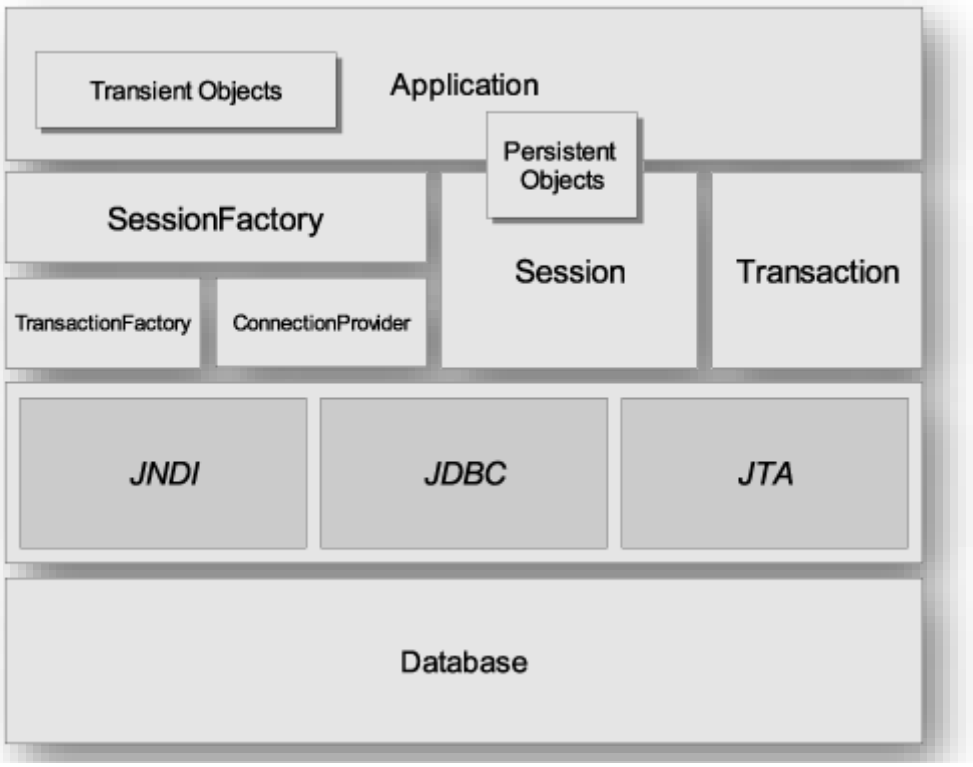

Ilustración 6 - Arquitectura completa de uso
Fuente: https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/architecture.html

Transaction (org.hibernate.Transaction): (Opcional) Es un objeto de corta vida, *single-threaded* (de hilo único) que la aplicación utiliza para especificar unidades atómicas de trabajo. En algunos casos, una *Session* puede extenderse sobre varias transacciones.

ConnectionProvider (org.hibernate.connection.ConnectionProvider): (Opcional) Es una fábrica o conjunto de conexiones JDBC. Abstrae a la aplicación del *Datasource* o *DriverManager* subyacente. No se expone a la aplicación, pero puede ser extendido/implementado por el desarrollador.

TransactionFactory (org.hibernate.TransactionFactory): (Opcional) Es una fábrica de instancias de transacciones. No está expuesta a la aplicación, pero puede ser extendida o incluso implementada por el desarrollador.

Extension Interfaces: Hibernate ofrece un extenso conjunto de interfaces que pueden ser implementadas para personalizar el comportamiento de la capa de persistencia.

2.2.2 JPOS

JPOS es un software empresarial de código abierto y misión crítica, basado en la estandarización internacional ISO-8583. Dicho estándar especifica una interfaz común mediante la cual los mensajes originados por tarjetas de transacciones financieras (tales como tarjetas de regalo) pueden intercambiarse entre compradores y emisores de tarjetas. Especifica la estructura, el formato y el contenido del mensaje, los elementos de datos y los valores de los elementos de datos [9]. En cuanto al *framework*, este se encuentra basado en la plataforma java y proporciona un puente entre los mensajes basados en ISO-8583 y los sistemas y redes de procesamiento interno que el desarrollador desea implementar. Es por esto que JPOS sirve como base de mensajería para sistemas que intercambian transacciones electrónicas hechas con tarjetas, sean estas de crédito, débito o tarjetas de regalo (*gift cards*). JPOS tiene la capacidad de poder ejecutarse en ambientes de alta usabilidad, con incluso millones de transacciones diarias permitiendo no solo procesar dichas transacciones, sino también documentar cada una de estas. Dicha documentación es realizada a través de archivos de *logs* en formato XML o de cualquier formato descripto previamente en las plantillas [10].

El *framework* JPOS permite ser usado como una librería, que un desarrollador puede simplemente importar al proyecto que se encuentra desarrollando. Uno de los puntos más destacados de este *framework* es la posibilidad de poder aplicar, modificar o eliminar clases en "caliente". A esta opción se la conoce como *"hot deploy"* y es uno de los puntos fundamentales de dicho software, ya que esto permite a agregar nuevas funcionalidades, corregir errores o eliminar una determinada característica sin la necesidad de tener que dejar de ofrecer el servicio. Este es uno de los puntos más destacables a la hora de ofrecer un servicio de alta disponibilidad o servicios de misión crítica.

Otro de los puntos más destacables de este *framework* es la posibilidad de escalabilidad que posee y la configuración de esta. Este *framework* posee la capacidad de procesar hasta miles de transacciones por segundo. Esto a través de la cantidad de *threads* (hilos), que podemos asignarles.

Como se puede aprecian en la Ilustración 7 –"Estructura del proyecto" el *framework* está basado en la división de módulos integrables, donde cada uno de estos desarrollan una tarea en particular y concreta. A su vez, dichos módulos pueden ser extendidos, implementados y/o utilizados por otros módulos. Los módulos cuentan esencialmente de 4 partes: "**lib**", "**cfg**", "**deploy**", "**src**". La carpeta de **lib** contiene todas las librerías '.jar' que serán necesarias para el desarrollo del módulo sobre el que se esté trabajando. La carpeta **cfg** contiene todos los archivos de configuración que serán necesarios para el desarrollo de dicho módulo, esto incluye: especificaciones DTD, archivos de constantes, especificaciones de otros *frameworks* (como podría ser *Hibernate*), o el formato de un determinado mensaje. En la carpeta "**deploy**" se encuentra los archivos xml. Estos son los encargados de describir los servicios QBeans, los cuales son la parte "ejecutable" de los diferentes módulos de JPOS. Estos archivos XML especifican la clase java que los ejecuta. Por último, la carpeta "**src**" contiene todas las clases javas que se verán involucradas en los diferentes servicios QBeans anteriormente mencionados.

Para la compilación del proyecto, se utiliza generalmente un proceso *ant*[2] [11] produciendo como salida una carpeta *build*. Esta carpeta está compuesta, por las mismas sub-carpetas anteriormente descriptas, solo que es en esta oportunidad no se encuentran solo los archivos relacionados a un determinado módulo, sino los archivos de todos los módulos en conjunto. Una vez que *ant* termina con esta sub-tarea, procede con el proceso de compilación java, obteniendo así los *bytecode*. Por último, la tarea *ant* de compilación genera el archivo .JAR que contiene toda la información en el mismo orden en que fue presentada en la carpeta *build*.

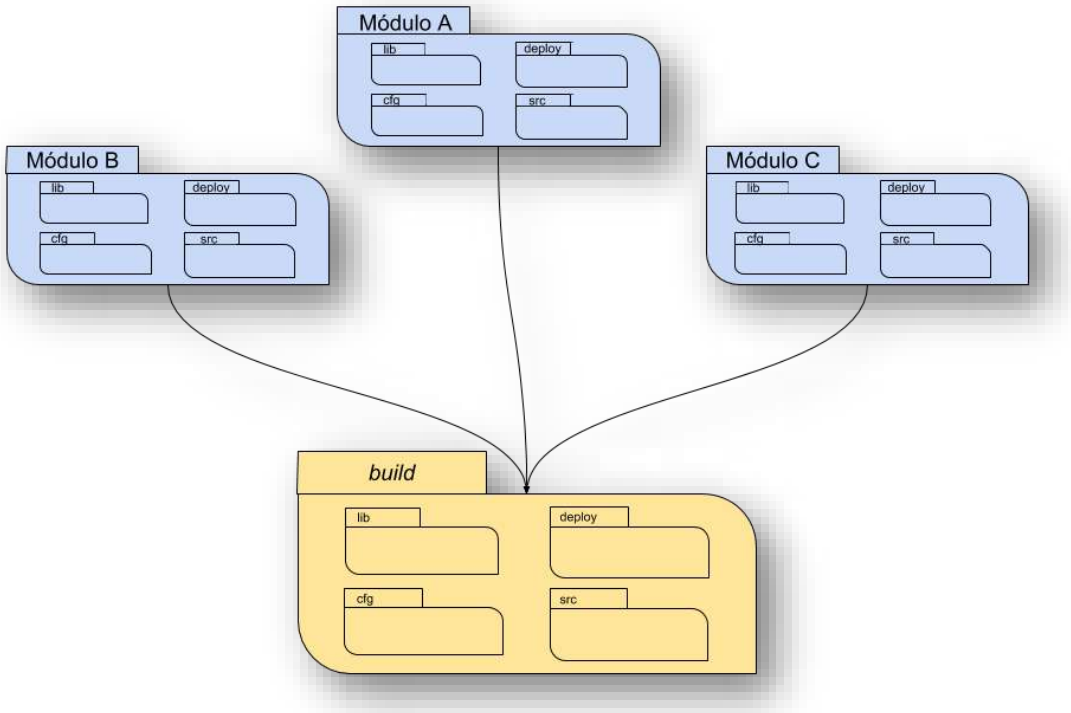

Ilustración 7 - Estructura del proyecto

[2] Apache Ant es una biblioteca de Java y una herramienta de línea de comandos cuya misión es guiar los procesos descritos en los archivos de compilación como objetivos y dependencias.

Capítulo 3 – Metodología de trabajo

Para llevar a cabo las metas propuestas por la empresa y adaptándose al ambiente de trabajo con el que cuenta Tecro Ingeniería S.A se utilizarán metodologías ágiles para el desarrollo del trabajo, más precisamente la metodología Scrum. El trabajo con Scrum permite a la empresa no solo tener entregas (segmentos de software) totalmente funcionales, sino que a su vez permite el análisis de la situación actual y brinda la posibilidad de realizar cambios durante el proceso. Esto facilita el agregado de características al producto final sin desechar el trabajo anteriormente logrado.

3.1 Scrum

Una definición dada por Ken Schwaber y Jeff Sutherland [12] es que "Scrum es un *framework* con el cual las personas pueden abordar complejos problemas adaptativos, mientras que ofrecen productiva y creativamente productos del más alto valor posible". Dichos autores expresan a *Scrum* como un framework y no como una metodología o proceso de trabajo. Si bien, no hay una convención respecto a esto, otros autores reconocen a *Scrum* como metodología, ubicándola como una de las metodologías ágiles más utilizadas. Es por esto, que, a lo largo del documento, se reconoce y describe a Scrum como metodología y no como *framework*.

Scrum fue utilizado desde principios de la década del 90 tanto para el desarrollo de software, hardware, redes, entre otros. Tiene un buen manejo del trabajo en productos complejos, ya que deja a la vista la eficacia relativa que existe entre la gestión de productos y las técnicas de trabajo para poder mejorar continuamente el producto final, el equipo y el entorno de trabajo.

La principal filosofía de Scrum es trabajar en pequeños grupos, lo que produce que estos sean más flexibles y adaptables. En el caso de trabajos complejos, se crean redes de grupos de trabajos que interaccionan y se relacionan, cada uno de estos con metas propias, orientadas a una meta en común.

Scrum se basa en una teoría de control de procesos empírica. El empirismo afirma que el conocimiento proviene de la experiencia y toma decisiones basadas en lo que se conoce. Por lo tanto, Scrum emplea un enfoque iterativo e incremental para optimizar la predictibilidad y controlar el riesgo basado en los conocimientos adquiridos previamente.

Tres pilares sostienen el control del proceso empírico: transparencia, inspección y adaptación. La ***transparencia*** hace referencia a que los aspectos significantos del proceso tienen que ser visible para todas aquellas personas que sean responsable de producir una salida. Es importante que estos aspectos sean definidos por un estándar común, y así todas las personas que observan comparten un entendimiento común de lo que se está observando.

En cuanto a la ***inspección***, es indispensable que se hagan inspecciones periódicas sobre los artefactos que detallan cuáles son las tareas y actividades que se deben entregar en cada *sprint*[3]. Esto permite una detección temprana de todas aquellas tareas o actividades que se están

[3] *Sprint:* Periodo de tiempo prefijado durante el cual se crea un incremento de producto "Hecho o Terminado" utilizable, potencialmente entregable

desviando de lo planificado, y así, poder tomar decisiones para la corrección de éstas. Es recomendable que dichas inspecciones estén a cargo de personal idóneo y especializado sobre el tema, como así también es importante que sean periódicas y frecuentes, siempre y cuando estas no entorpezcan con el trabajo que las otras personas del grupo realizan. En cuanto a la adaptación, si la/s persona/s encargada/s de la inspección determina que uno o más de los aspectos inspeccionados anteriormente se desvían más allá de los límites aceptables y/o esperados, se debe ***adaptar*** el proceso o producto que se está desarrollando. De esta forma, se pretende evitar desvíos más allá de los planificados y, de ser posible, la corrección de las actividades que dieron origen a esta situación imprevistas, así como también la atenuación de sus consecuencias negativas.

Como se puede apreciar en la Ilustración 8, el proceso es iterativo y colaborativo. Es iterativo ya que para lograr la meta final se requiere de 'N' *sprints*. En tanto que se lo considera colaborativo, dado que todos sus participantes si bien tienen tareas específicas, muchas de ellas deben ser realizadas con otros participantes, o incluso agentes ajenos a la empresa, como por ejemplo clientes o consultores.

3.1.1 El equipo de trabajo

El equipo Scrum tiene dos características principales: son auto-organizados y multifuncionales. Se los considera auto-organizados porque cada uno de sus miembros elige

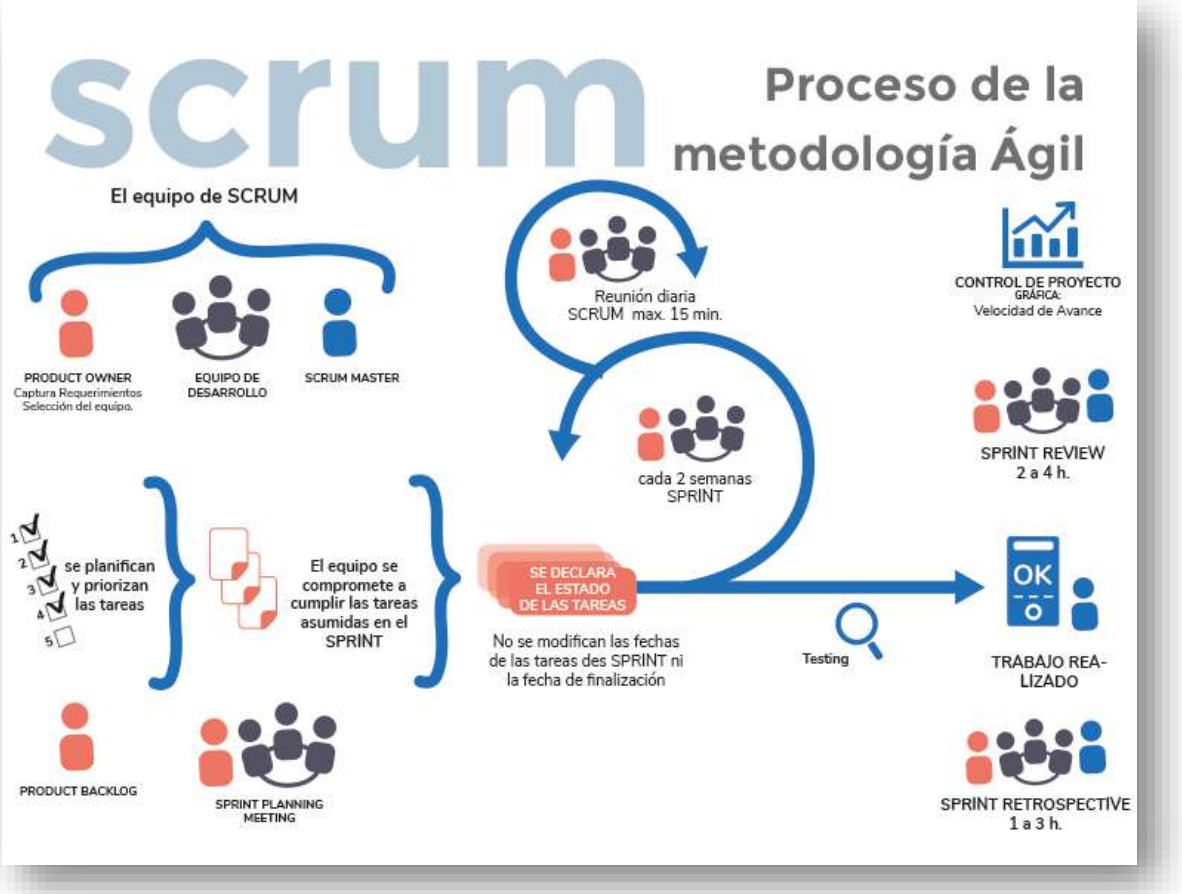

Ilustración 8 - Procesos y participantes Scrum
Fuente: https://development.grupogaratu.com/metodologia-scrum-desarrollo-software

la mejor manera para llevar a cabo sus actividades, en vez de ser dirigido por algún otro miembro del equipo a cargo. También se los considera multifuncionales, esto se debe a que como equipo se poseen todas las habilidades necesarias para resolver las diferentes tareas sin necesidad de depender de terceras personas ajenas al equipo de trabajo.

Este modelo de equipo está diseñado para optimizar la flexibilidad, la creatividad y la productividad de cada uno de los diferentes miembros del equipo. Dicho equipo está compuesto el dueño del producto, el equipo de desarrollo y un *Scrum master*.

3.1.1.1 Dueño del producto

El dueño del producto es el encargado de que el sistema final desarrollado por el equipo de trabajo cuente con la mejor calidad y valor agregado posible. La metodología utilizada por esta persona para lograr los objetivos difiere mucho en cada equipo de trabajo, organización y de la persona en sí.

El dueño del producto es la única persona responsable de administrar el p*roduct backlog*, en español conocido como requerimientos del producto. La administración de estos documentos incluye expresar correctamente las tareas a desarrollar, organizar las actividades en base a las priorizaciones, asegurarse que cada uno de los miembros del equipo de desarrollo tenga acceso a lo necesario para poder llevar a cabo su trabajo, como así también, asegurarse que hayan comprendido las tareas a desarrollar. Cabe destacar que este rol debe ser llevado a cabo por una sola persona, sin importar si esta persona debe relacionarse con superiores o comités, será el dueño del producto quien expresará y explicará los deseos de sus superiores, pero solo éste será quien decida en que prioridad se desarrollan las diferentes actividades.

Es importante que para que esta persona pueda tener éxito en su trabajo, todos respeten las decisiones que éste tome, esto no significa que cada una de las personas, superiores o no, no pueden expresar opiniones, consultas y/o sugerencias, pero llegado el momento de tomar la última decisión, esta recae totalmente sobre el dueño del producto.

3.1.1.2 Equipo de desarrollo

El equipo de desarrollo está conformado por las personas que se encargarán de desarrollar el código y las entregas cuando hayan sido finalizadas las diferentes tareas para que sean incluidas en el *sprint* acordado. Los equipos de desarrollo están estructurados para organizar y gestionar su propio trabajo. La sinergia resultante optimiza la eficiencia y eficacia general del equipo de desarrollo. Una de las características de estos grupos es que no se reconocen títulos entre los miembros del equipo, independientemente del trabajo que realice la persona. A su vez, en esta modalidad de trabajo, no se reconocen sub-equipos en el equipo de desarrollo, independientemente de los dominios que deben abordarse como pruebas, arquitectura, operaciones o análisis de negocios. Estos equipos se basan en que, si bien una persona puede tener habilidades específicas, áreas en las que sobresalen, dichas habilidades pertenecen al equipo como un todo, perdurando así la ideología de grupo sobre el individualismo, lo que produce cooperativismo entre compañeros.

Un punto importante a la hora de confirmar estos grupos de trabajo, es elegir correctamente el tamaño del grupo, la cantidad de personas que conformarán el equipo de desarrolladores.

El tamaño óptimo del mismo tiene que ser lo suficientemente pequeño como para seguir siendo ágil y lo suficientemente grande como para completar un trabajo significativo dentro de un *sprint*.

3.1.1.3 El Scrum master

Esta persona es la encargada de asegurarse que todos estén en sintonía, esto incluye cerciorarse de que todos entienden como trabajar con Scrum, de que cada uno de los miembros del equipo entiende su rol, las metas grupales, etc. Esta persona es considerada el guía/líder del grupo, ayuda a aquellas personas ajenas al grupo a entender cómo se lleva a cabo el trabajo, como así también cuales interacciones del exterior con el equipo son productivas y cuáles no.

Esta persona es la encargada de interactuar con el equipo de trabajo, como así también con la organización y el dueño del producto. En cuanto a la relación con este último, se espera que el *Scrum master* se asegure que el resto del equipo comprenda las metas y el dominio del problema especificadas por el dueño del producto. Se espera que *Scrum master* colabore con el dueño del producto en la reorganización de las tareas, recomendando y guiándolo, como así también, proveyéndole un estimativo de los resultados que éste espera. Sin embargo, el *Scrum master* solo guía al dueño del producto y le provee la información, pero es este último quien toma las decisiones finales.

En cuanto a la relación entre el Scrum master y el equipo de desarrollo, se espera que esta persona guíe al equipo de desarrollo a través de la auto organización de las tareas, ayudándolo al equipo a crear productos de calidad, eliminando obstáculos que puedan estar haciendo que el equipo no progrese, como así también facilitando las herramientas necesarias.

3.1.2 Eventos Scrum

Los eventos de Scrum están diseñados para crear regularidad en las reuniones, formalizar el proceso, tiempo y continuidad de estas, evitando así el desperdicio de tiempo, la creación de reuniones de urgencias o no contempladas en los tiempos iniciales. Como se puede apreciar en la Ilustración 9, los eventos son: *sprint*, planificación del *sprint*, las reuniones diarias, revisión del *sprint*, retrospectiva de *sprint*, todos estos eventos están acotados en el tiempo, es decir, cuentan tanto con un tiempo de inicio, como así también, con un horario de finalización o tiempo máximo para llevarlos a cabo. Además del *sprint* que es el evento principal, y el que a su vez contiene todos los otros eventos dentro de sí mismo, cada evento que surge es una oportunidad formal para revisar lo ya desarrollado, evacuar dudas futuras y corregir desviaciones dentro de lo planeado. La lógica de estos eventos es dar transparencia a lo que se está haciendo, la evasión de alguno de estos eventos tiene como resultado la pérdida de transparencia, desperdiciando así, la esencia de la metodología, la cual es, producir un proceso iterativo e incremental.

3.1.2.1 sprint

El evento principal de esta metodología es el *sprint*. Este evento engloba todos los otros eventos de la metodología: la planificación del *sprint*, reuniones diarias, la revisión del *sprint* y la retrospectiva del *sprint*. En la finalización de cada *sprint* se busca incrementar la cantidad de

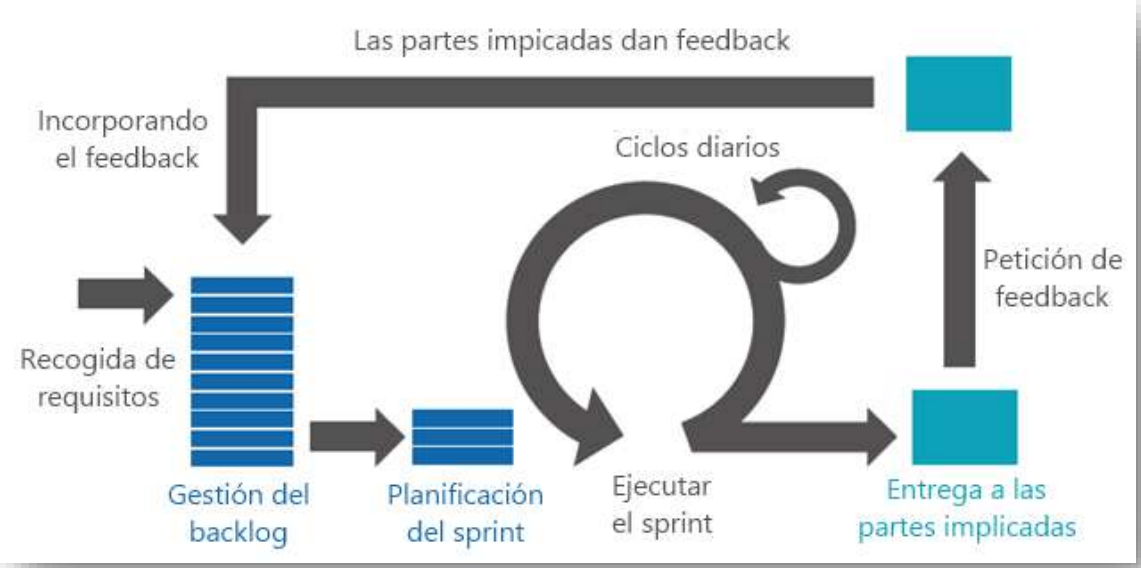

Ilustración 9 - Eventos Scrum
Fuente: http://fernandoescolar.github.io/public/uploads/2013/01/scrum-proceso.png

producto terminado, esto quiere decir poder terminar una parte del proyecto, la cual haya sido correctamente probada y revisada y esté lista para entrar en producción de ser necesario.

Uno de los puntos principales de este evento es que tiene duraciones de tiempo constante a través del tiempo, y si bien la unidad de tiempo óptima no está definida, la mayoría de los autores coinciden, en que este evento no debería durar más de un mes. En esencia, un periodo de tiempo relativamente extenso para que el equipo de trabajo pueda desarrollar y probar las tareas seleccionadas para dicho *sprint*, pero tampoco un periodo de tiempo tan extenso como para que lo planeado no puede llegar a ser realizado, ya que la definición de lo que se está desarrollando puede cambiar, aumentar la complejidad y, por ende, el riesgo. La idea principal de esta medida de tiempo es que sea lo suficientemente grande para progresar, pero no lo suficientemente largo como para que se convierta en un ambiente no controlado. Esto permite la previsibilidad al garantizar la inspección y la adaptación del progreso hacia un objetivo de *sprint* al menos cada mes calendario, lo que produce limitar los riesgos a solo un mes.

Cada *sprint* tiene una meta de lo que se construirá, un diseño y un plan flexible que guiará su construcción, el trabajo y el incremento resultante del producto. La finalización de un *sprint* produce automáticamente el inicio de uno nuevo. Dentro de este evento no se realizan cambios que puedan llegar a poner en riesgo el objetivo final del *sprint*, los objetivos de calidad jamás deben de disminuir, y el alcance se puede aclarar y renegociar entre el propietario del producto y el equipo de desarrollo a medida que durante el proceso se aprende.

Durante el periodo en el que se lleva a cabo este evento, puede haber circunstancias extraordinarias que hagan que no tenga sentido continuar con un *sprint*. Un ejemplo de ello es cuando la empresa decide cambiar su producto final o si cambian las condiciones del mercado o de la tecnología. Esto producirá que todos los elementos incompletos del *backlog* sean reagrupados en los *sprint* que seguirán a continuación. Esta situación debido a la duración del evento suele ser muy poco frecuente, ya que los contratiempos anteriormente

nombrados, no suelen tener cambios tan radicales en periodos de tiempo como éste. La decisión final de la cancelación la tiene el dueño del producto. Dicha decisión puede ser influenciada por el Scrum master o los mismos desarrolladores. De producirse dicho infortunio, lo desarrollado hasta el momento pasa a una revisión especial para tomar la decisión de que parte del trabajo desarrollado es viable para producción, que debe ser descartado, y que debe ser transferido a un nuevo *sprint* para su finalización.

3.1.2.2 Planificación del sprint

En este evento se espera que durante un tiempo preestablecido (usualmente una jornada laboral) se pueda lograr la planificación del *sprint*. Esto implica establecer cuáles serán las actividades a desarrollar y quienes las llevarán a cabo. Es importante para lograr dicha planificación que todos los miembros del grupo se encuentren presentes, esto incluye a los desarrolladores y el dueño del producto. Es el Scrum master es quien se encarga de que todo se lleve a cabo dentro del tiempo prestablecido, como así también, que todos comprendan el alcance que se propone para dicho *sprint*.

Uno de los dos puntos principales de este evento es definir qué puede ser logrado en el *sprint* que se desea planificar. Los desarrolladores colaboran con el conocimiento necesario de que es lo que puede ser desarrollado en ese periodo de tiempo. Es trabajo del dueño del producto analizar cuál será el objetivo de dicho *sprint*, como así también, cuáles serán los requerimientos del producto que se verán implicados en dicho periodo de tiempo. Es por ello que resulta importante contar para esta reunión con los requerimientos del producto, el último producto terminado, la capacidad de producción del equipo de desarrollo y la última performance del equipo de desarrollo. Aquí todo el equipo colabora en la definición de la meta para el *sprint*.

La meta del *sprint* es un conjunto de objetivos que son propuestos a realizar en este periodo de tiempo. La definición de una meta ayuda a que todos los participantes involucrados tengan una definición clara y común de que es lo que se quiere llegar a lograr cuando el tiempo del evento haya trascurrido. También ayuda a que los desarrolladores tengas las mismas metas presentes, inclinándolos así a un trabajo conjunto y no individualista.

La cantidad de ítems elegidos de la lista de requerimientos que serán desarrolladas en dicho *sprint* está definida por el equipo de desarrollo. Esto se debe a que solo este equipo puede tener una verdadera noción de que es lo que puede lograr en el periodo de tiempo establecido.

Usualmente el equipo de desarrollo comienza diseñando el sistema y el trabajo que necesita realizar las diferentes tareas. Estos trabajos pueden ser de diferentes tamaños y complejidades, pero nunca deben ocupar más de una jornada laboral poder realizarlos. De esta manera quedan claros los puntos a desarrollar y es más fácil poder estimar el tiempo que llevará poder desarrollarlos.

Es en esta etapa donde los desarrolladores negocian con el dueño del producto que tanto trabajo pueden llegar a desarrollar durante el *sprint*. Aquí es donde se consideran cuáles ítems del *backlog* tienen más prioridad y cuáles de estos serán desarrollados. Es en estas reuniones donde los desarrolladores pueden invitar, además del dueño del producto y el Scrum master, como así también el personal que consideren idóneos para la tarea o bien, que tengan la

capacidad/habilidad de esclarecer puntos donde nadie es experto. Al finalizar la reunión, el equipo de desarrollo debería contar con la habilidad de explicarle al dueño del producto como es que se llevarán a cabo las diferentes tareas que el equipo va a desarrollar.

3.1.2.3 Reuniones diarias

Estas reuniones se llevan a cabo durante no más de 15 minutos. Aquí se planea que se llevará a cabo en las próximas 24hs, horario en el que tendrá lugar la siguiente reunión. En estos encuentros es donde se comparte los progresos y se inspecciona lo desarrollado hasta el momento, de esta manera se puede tener una noción general de cuán cerca o alejados están de la meta propuesta para el actual *sprint*. Es en estas reuniones donde el equipo puede observar cuales ítems de la lista de tareas han sido ya desarrollados y cuáles de estos aún faltan realizar. Dichas reuniones siempre deben tomar lugar en el mismo horario para evitar retrasos o complicaciones que hagan perder el tiempo en la organización de nuevas reuniones.

Las reuniones diarias no tienen una estructura o formato predefinido, esto se debe a que cada equipo de trabajo es diferente, por lo tanto, en la medida que el propio grupo de trabajo comienza a conocer su dinámica es que se va definiendo el formato que toma cada una de las reuniones. Si bien, teóricamente, no hay una estructura que guíe la reunión, si es importante que tenga lugar la discusión sobre 3 puntos fundamentales:

¿Qué fue lo que se hizo ayer que colaboró a la meta del *sprint?*

¿Qué es lo que se hará en las próximas 24hs que ayudará a la meta del *sprint*?

¿Hay algún impedimento que obstaculice al equipo o a alguno de sus individuos a seguir trabajando?

Es importante que las reuniones no se extiendan más de lo establecido para que cada uno de los miembros pueda comenzar a trabajar. Si bien es el *Scrum master* quien se asegura que todos participen de la reunión, son los miembros del equipo de desarrollo quienes guían hacia donde se va a dirigir, el *Scrum master* solo guía a los desarrolladores a que las reuniones no se extiendan más de lo asignado. Aquellas personas del grupo que tengan algún impedimento para continuar con su trabajo o necesitan una readaptación de alguno de los ítems de la lista deben reunirse en una reunión especial con las personas afectadas o que pueden brindarles una solución. De esta manera no se desperdicia el tiempo de quienes no se ven afectados dentro de la reunión diaria.

El grupo de desarrolladores y el *Scrum master* son quienes se ven obligados a participar de cada una de las reuniones. Cualquier persona ajena a estos, es bienvenida a las reuniones, pero es el *Scrum master* quien se encargará de que dichas personas no disturben. Estas reuniones ayudan a la comunicación, evitar reuniones improvisadas, detectar problemas o desviaciones tempranas y a la toma rápida de decisiones.

3.1.2.4 Revisión del sprint

Este evento es llevado a cabo al finalizar el *sprint* para corroborar el incremento del producto final y actualizar ítems del *backlog*. Esta es una reunión informal donde el objetivo principal es poder ver lo que fue desarrollado y como el equipo de trabajo fue creciendo y mejorando.

A esta reunión asisten todas las partes interesadas, esto incluye al equipo de desarrollo, el *Scrum master*, el dueño del producto y cualquier persona ajena al grupo que de alguna manera se ve involucrada en el producto final. Esta suele ser una reunión de no más de media jornada laboral donde todos participan, pero es el *Scrum master* quien se encarga de que no se extienda más tiempo del previsto.

Durante la reunión el dueño del producto explica que ítems fueron desarrollados, mientras que el equipo de desarrollo contesta las preguntas que justifican que fue lo que se realizó. A su vez, este equipo es el que introduce los problemas que surgieron en el *sprint* y cómo los mismo fueron resueltos. En esta reunión se analiza cómo se trabajó durante este periodo para poder contar con nueva información en la próxima reunión de planificación de *sprint*. Aquí se analiza si la situación actual del proyecto se desvió de lo planificado, sea por cambios en el mercado, en la tecnología, etc. Si hubo cambios, se hacen los arreglos necesarios, como así también se discute si hubo desviaciones sobre lo especulado y se realizan las correcciones necesarias. Como resultado de esta reunión es que se obtiene la lista de requerimientos del producto revisada y los ítems candidatos para el próximo *sprint*.

3.1.2.5 Retrospectiva del sprint

Esta reunión es desarrollada para poder hacer un análisis sobre todo lo sucedido en el último *sprint*. Esto permite, a su vez, hacer un análisis de cómo mejorar para los siguientes *sprint*.

Este análisis se lleva a cabo en una reunión que, a priori, no debería durar más de media jornada laboral, y se debería llevar a cabo después de la revisión del *sprint* y anterior a la planificación del siguiente.

Los objetivos de estas reuniones no son solo analizar cómo fue el último *sprint* respecto de las personas, relaciones, herramientas y procesos, sino que también, identificar cuáles fueron las situaciones que salieron bien y mal para poder mejorarlas en los *sprint* futuros. Si bien las mejoras puedan implementarse en cualquier momento de la metodología, la retrospectiva del *sprint* brinda una oportunidad formal de identificarlas e implementarlas.

3.1.3 Documentos Scrum

Los documentos generados por esta metodología permiten dar transparencia a lo realizado, como así también, brindar claridad a todos los involucrados dentro del proyecto. Esto permite tener escritos que brindan transparencia y colaboran a eliminar ambigüedades.

Como se aprecia en la Ilustración 10, los documentos principales dentro de esta metodología son los requerimientos del producto, y los *sprint backlog*, traducidos al español como requerimientos del *sprint*. Estos dos documentos colaboran en la obtención de partes del producto final que son correctamente finalizadas, testeadas, aprobadas por el cliente y puestas en producción en etapas tempranas del desarrollo.

En el caso del cliente, este trabaja de manera intensa con el dueño del producto para poder obtener los correctos requerimientos del sistema. Sin embargo, a lo largo del proceso desarrollo Scrum, el mismo no se ve involucrado hasta las revisiones finales a menos que se requiera de su presencia en alguna de las reuniones previas a los fines de poder evacuar dudas que impiden continuar con el desarrollo.

Ilustración 10 - Documentos Scrum
Fuente: http://www.diegocalvo.es/metodologia-scrum-metodologia-agil/metodologia-scrum/

3.1.3.1 Requerimientos del producto – product backlog

Este documento contiene una lista detallada de todo lo que es necesario que se encuentre en el producto final. En su comienzo no contiene toda la información necesaria para llevar a cabo el proyecto de manera completa. Este documento, en sus inicios, contiene una lista acotada para poder comenzar a trabajar en el primer *sprint*, a medida que el proyecto evoluciona, también lo hace la lista. Este escrito enumera todos los requerimientos, características, funciones, mejoras y soluciones que deben ser implementadas. Es en este documento que se encontrará para cada uno de sus ítems, una estimación de costo, una descripción y un orden de prioridad.

A medida que el producto comienza a ser utilizado y nuevos comentarios surgen, la lista va creciendo, al igual que la información que ésta contiene. Lo mismo sucede con cambios en las tecnologías que puedan surgir durante el proceso, esto puede permitir ingresar nuevos funcionalidades o características, lo que produce cambios en las especificaciones del producto. Los requerimientos en esta clase de productos nunca dejan de cambiar, por lo que sería imposible que nuestro documento sea estático. Es trabajo del dueño del producto mantenerla actualizada, como así también, disponible y accesible para el resto de los integrantes del equipo.

Un proceso importante por el que atraviesa este documento es el "refinamiento". Este es el proceso por el cual el equipo completo revisa cada uno de los ítems de la lista y hacen los cambios que puedan llegar a ser pertinentes. Uno de los puntos importantes en este proceso es el del ordenamiento de estos ítems, una lista ordenada hace que sea más sencillo saber por cual ítem se debe continuar o como se debe proceder. Cuando estas listas de ítems no están

ordenadas hay una valiosa pérdida de tiempo en una ordenación parcial cada vez que alguien debe de consultarla.

La manera de ordenarla debe ser elegida por el equipo en conjunto, dependiendo del proyecto. Una de las maneras más usadas es hacerlo por el tiempo que va a consumir cada tarea, eligiendo aquellas que van a consumir más tiempo de manera prioritaria. Si bien todos pueden hacer una estimación de cuánto es el tiempo que puede consumir una tarea, es el programador final quien decide este número, debido a que es esta persona quien la tiene que desarrollar y tiene plena conciencia de sus habilidades y limitaciones. En estos casos el dueño del producto se limita simplemente en asegurarse de que el resto de los integrantes del equipo *scrum* comprende cada uno de los ítems que aparecen en el documento.

3.1.3.2 Requerimientos de sprint - sprint Backlog

Este documento guarda ciertas similitudes con el descrito anteriormente en la sección 3.1.3.1. En este caso, su principal diferencia radica que aquí se detalla la lista de ítems que se espera hacer en el *sprint* actual. Otra diferencia que presenta, es que este documento es una proyección de lo que se espera terminar completamente para el siguiente *sprint*, y la precisión de esta información solo puede ser dada por el equipo de desarrollo. Por lo que la selección de ítems del *product backlog* que pasarán a formar parte del *sprint backlog* está a cargo de los desarrolladores.

Este documento está acompañado de un plan de progreso, que los desarrolladores crean a los fines de tener conocimiento de cómo proceder. Este plan, como así también, el documento en general, no son estáticos. Una versión inicial es creada para poder comenzar a trabajar, pero a medida que las reuniones diarias toman lugar, los problemas y las nuevas ideas son expuestas, el documento toma cambios propuestos por el equipo de desarrollo.

Este documento tiene la particularidad de siempre estar visible para toda persona del equipo *scrum*. Es una foto actual de que fue lo que se desarrolló, que es lo que está en proceso, y que tareas se esperan completar a la finalización del *sprint*.

3.2 Integración y entrega continua a través de herramientas de automatización

La integración continua es una práctica de desarrollo de software donde los miembros de un equipo integran su trabajo con frecuencia, por lo general cada persona integra al repositorio su progreso al menos diariamente, lo que produce integraciones múltiples por día. Cada integración se verifica mediante una compilación automatizada (incluyendo test) para detectar errores de integración lo más rápido posible. De esta manera los problemas que se producen causados por la integración de código son significativamente menores, lo que conlleva en un gasto de tiempo mucho menor para poder solucionar dichos problemas. De esta manera cada miembro del equipo de trabajo logra colaborar más dinámicamente con el proyecto [13].

Otras prácticas utilizadas que presentan grandes similitudes con la integración continua, son las prácticas de trabajo de **entrega continua** como asi también el **despliegue continuo**. La

principal diferencia entre estas prácticas es cuando la intervención humana es necesaria para llevar los nuevos cambios a producción.

En la actualidad, prácticamente sin importar el tamaño del proyecto de desarrollo, ningún equipo de desarrollo lleva a cabo un proyecto sin usar al menos alguna plataforma de control de versiones que permita concentrar en un servidor una versión única y actualizada con todos los cambios que el equipo de desarrollo ha llevado a cabo hasta el momento. Esto mismo, junto a otras herramientas de automatización permiten no solo ahorrar tiempo en el momento de la integración de trabajos, sino que también reduce las posibilidades de errores a la hora de hacer grandes integraciones donde todos aportan código.

3.2.1 Integración continua

La integración continua es la práctica mediante la cual el proceso de integrar código, compilarlo y testearlo se hace de manera automática y de manera periódica, generalmente una vez al día, o bien al completar una tarea, produciendo de esta manera una nueva versión del producto en cada *commit* [4].

Como se muestra en la Ilustración 11 la integración continua cuenta esencialmente de 6 pasos: El desarrollador es el encargado de desencadenar el primer paso, el cual se produce cuando este termina una tarea que tenía en desarrollo y procede a efectuar un *commit*. Esto produce como resultado el incremento en el versionado del código completo. Seguido a esto, el servidor de control de versiones procede a aceptar los cambios o rechazarlos en el caso de que haya conflictos en los archivos (un ejemplo de esto es cuando dos desarrolladores editan la misma línea de código). En caso afirmativo el propio servidor de versiones es quien ejecuta el paso 2, enviando los cambios al servidor de integración continua. Este último es quien se encarga de ejecutar los pasos 3, 4 y 5. En primer lugar el servidor se encarga de compilar de manera automática el proyecto completo. Seguido a esto, los test que hayan sido especificados será ejecutados para corroborar que el proyecto en su totalidad sigue funcionando correctamente. Por último, el servidor de integración continua produce los resultados y procede a analizarlos, obteniendo de esta manera una respuesta exitosa, si los pasos anteriores fueron ejecutados correctamente y ninguno de los test ejecutados han fallado, o bien, una respuesta de error si cualquier paso anteriormente nombrados ha fallado. Cualquiera sea el resultado final obtenido, este desencadenará el paso final, el cual es la notificación a cada uno de los miembros del equipo de desarrollo.

De esta manera cada uno de los desarrolladores están informados si otra tarea fue ya realizada y correctamente integrada, o si bien, se produjo algún error, lo que hace que el desarrollador que efectuó el *commit* (también notificado), a través de los archivos de log y resultados, pueda identificar el problema de la integración y proceda a hacer un nuevo *commit* con los cambios pertinentes para la correcta integración, o en el peor de los casos revierta su último cambio a la última versión estable.

[4] *Commit:* En este *Context*o hace referencia a la acción de subir código fuente a algún repositorio de versiones, como puede ser GIT o SVN.

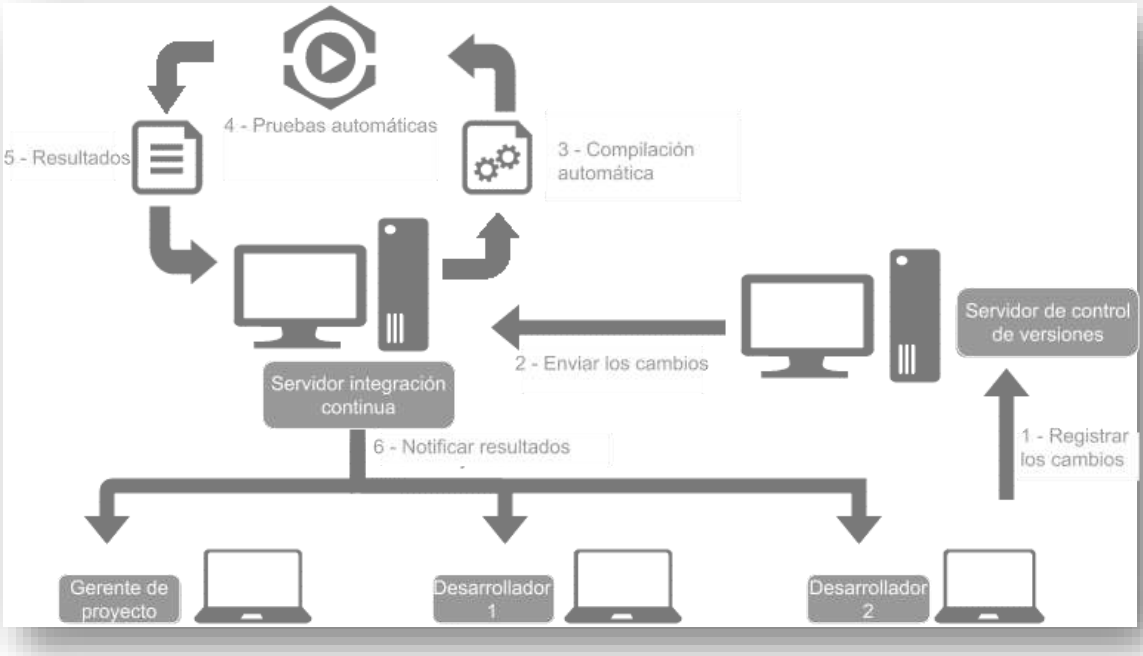

Ilustración 11 - Ciclo de integración continua
Fuente: http://xurxodev.com/tag/integracioncontinua/

3.2.2 Entrega y despliegue continuo

La **entrega continua** y el **despliegue continuo** no son prácticas de trabajo que disten mucho de la integración continua, sino más bien son prácticas que añaden ciertos pasos extras a los anteriormente detallados. La principal característica con la que cuentan estas prácticas es que se producen de manera totalmente automática en el caso del despliegue continuo, y algunos de ellos de manera automática en la entrega continua. Tanto la entrega como el despliegue continuo, trabajan en su esencia con integración continua, es decir que los 6 pasos anteriormente detallados son los primeros pasos con los que cuenta estas prácticas. Como se ve reflejado en la Ilustración 12, la entrega continua procede a automatizar la mayor parte del trabajo, dejando solo el último paso (puesta en producción) como una tarea manual. Es decir, que esta práctica todavía se requiere que un agente humano proceda a realizarla. En este caso, lo habitual es tener fechas de puesta en producción las cuales son acordadas entre la empresa que desarrolla el sistema, el cliente y las partes involucradas. De esta manera las entregas en producción pueden llevase a cabo de manera más regular, ya que esta práctica automatiza todos los pasos anteriores produciendo en cada *commit* todo lo necesario para proceder a producción.

En el caso del despliegue continuo, como se observa en la Ilustración 12 todos los pasos se llevan a cabo de manera automática, incluso la puesta en producción, por lo que en cada *commit* que un desarrollador haga, tenemos una nueva versión instalada en producción completamente funcional. Esto hace que no se requiere de interacciones para llevar el producto a producción, lo que produce que cada característica que sea agregada al sistema, o cualquier problema que sea reparado se encuentre inmediatamente listo para ser usado.

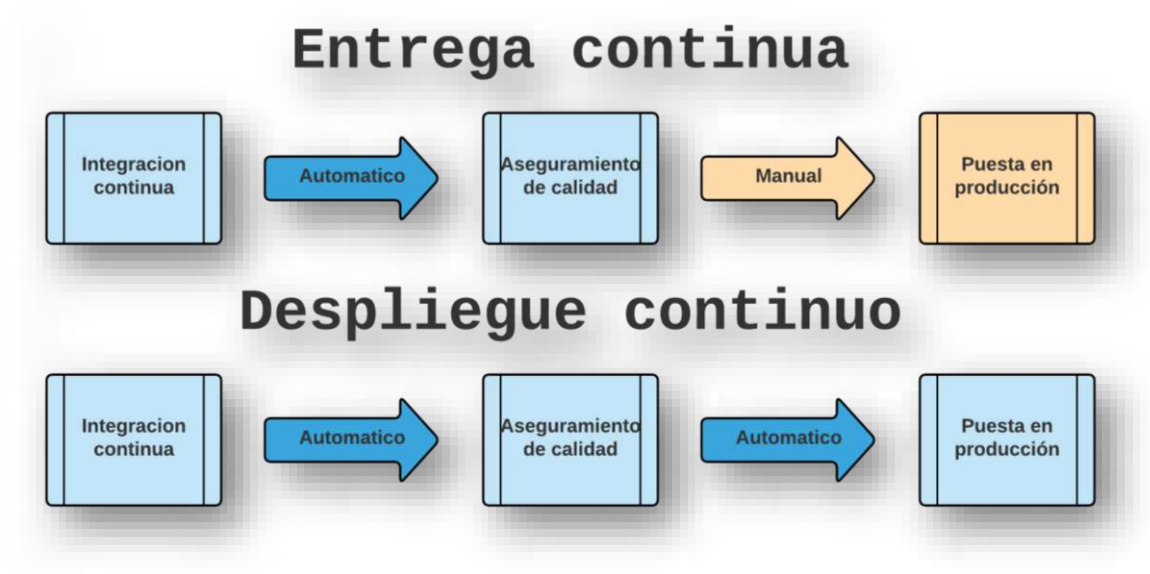

Ilustración 12 - Entrega y despliegue continuo

3.3 Realidad de la empresa

Si bien dentro de la empresa se considera como metodología de trabajo Scrum, la realidad de como una organización funciona muchas veces imposibilita lograr cumplir con todos los puntos que una metodología especifica. Uno de los puntos fundamentales de las metodologías ágiles es el trabajo en conjunto. Este es uno de los puntos importantes a resaltar ya que algunos de sus empleados trabajan de manera remota y todos a su vez trabajan con empleados de la empresa Emp-X, quienes no se encuentran alojados en Argentina.

Otro de los puntos importantes, es la coordinación horaria para reuniones diarias, como así también reuniones de *sprint*. Si bien cada uno de los empleados de Tecro colabora para su organización, muchas veces se ve comprometida debido a que unas de las políticas de la empresa es la flexibilidad horaria de trabajo. Por dicho motivo todos sus empleados trabajan en horarios diferentes y cambiantes, lo que dificulta la coordinación de las reuniones anteriormente mencionadas.

Si bien todo lo descripto en la sección 3.1 Scrum muchas veces no es respetado, la empresa local como así la empresa Emp-X y sus empleados colaboran para trabajar con metodologías ágiles, respetando los ideales originales del manifiesto por el desarrollo ágil de software [14].

Un término que comenzó a surgir en el último tiempo es la expresión de "post-agilismo". Si bien, es poca la documentación que se encuentra sobre el tema, y en su mayor parte proviene de blogs y sitios no oficiales, existe este movimiento que reconoce las limitaciones que tienen estas metodologías y *frameworks* que describen de manera específica como deben ser realizadas las cosas. Dave Thomas, uno de los autores del manifiesto por el desarrollo ágil de software expresa en la conferencia GOTO 2015 [15] , que las ideas iniciales del manifiesto no se alinean con las metodologías estrictamente detalladas como pueden ser *scrum, extreme programming, canvas*, etc. Sino, que la idea original del manifiesto era expresar las buenas

prácticas de una programación recursiva y retroalimentada que permita la inserción de cambios sin la necesidad de tener que desechar todo el trabajo desarrollado hasta el momento.

Capítulo 4 – Arquitectura del sistema y características

Una arquitectura básica y simplificada puede ser observada en la Ilustración 13 donde los componentes mínimos del circuito son un "cliente", el cual es el encargado de producir e iniciar las diferentes transacciones que pueden ser soportadas por el *switch*. Otro de los componentes es el servidor, el cual mantiene alojado el servicio que se encarga de procesar las operaciones anteriormente mencionadas, dicho servicio es el previamente denominado como *switch*. Por último, el *endpoint*, que en primera instancia y por fuera de la arquitectura mencionada, es el encargado de la impresión de las tarjetas de regalos y la distribución en las diferentes tiendas. Este último integrante de nuestra arquitectura es el encargado de validar las diferentes transacciones que le son enviadas, además de brindar una respuesta aprobando o no la transacción.

En la actualidad, la empresa Emp-X mantiene activos prácticamente tantos *switchs* como clientes posee. Muchos de estos *switchs* poseen características y/o funcionalidades comunes, las cuales son heredadas de servicios anteriormente desarrollados. Una de las metas principales de Emp-X es la creación de un servicio lo suficientemente flexible, que permita la conexión con una cantidad necesaria de *endpoints* para procesar transacciones de más de un cliente, creando así un *switch* multi-cliente. Una arquitectura simplificada de dicho sistema, que detalla cómo se conectan los diferentes actores del sistema puede observarse en la ilustración 14.

Si bien anteriormente se habló de un servidor que contiene el servicio alojado, en la realidad, dista mucho de ser simplemente un servidor que recibe peticiones de transacciones y devuelve respuesta a dichas peticiones. En la actualidad, debido a la sensibilidad de la información que se maneja, a la cantidad de transacciones que se manejan diariamente, como a la obligación de ofrecer un servicio de alta disponibilidad, la arquitectura del sistema posee una complejidad mayor.

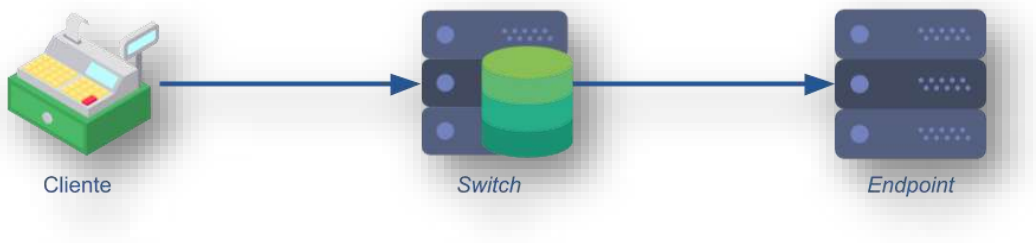

Ilustración 13 - Arquitectura básica del sistema

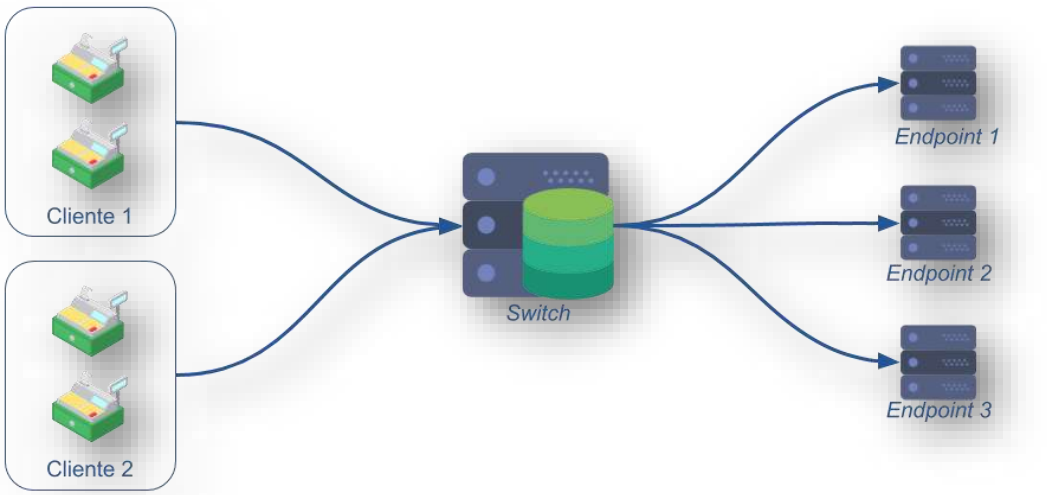

Ilustración 14 - Arquitectura de *switch* multi-cliente

Dicha arquitectura, reflejada en la Ilustración 15, está compuesta por 4 servidores de procesamiento de transacciones y dos servidores de *batch* o procesadores de archivos; los cuales pueden ser de ingreso o egreso de datos (generación de reportes). Si bien los 2 servidores procesadores de archivos no cuentan con conexión a ningún endpoint, los 4 servidores procesadores de transacciones cuentan con una conexión doble con cada *endpoint*. De esta manera, no solo se tiene una replicación de conexiones debido a la cantidad de servidores, sino que más aún, se cuenta con una replicación a nivel de servidores. Esto se produce debido a que cada servidor internamente mantiene "viva" al menos dos conexiones con cada uno de los *endpoints*.

Como el sistema está compuesto por múltiples servidores trabajando en paralelo, se debe contar con un balanceador de carga. Este tiene que ser capaz de asegurar la misma carga de trabajo en todos los servidores. También tiene que poseer la capacidad de detectar cuando alguno de estos servidores está fuera de servicio, para así poder desviar todas las transacciones recibidas al servidor que todavía permanece disponible. Por último, se puede observar en dicha ilustración que tantos los servidores procesadores de transacciones como los de *batch* se encuentran distribuidos en dos lugares físicos diferentes, es decir, existe una replicación física en dos ciudades. De esta manera, la replicación permite brindar un servicio con alta disponibilidad de uso (ver sección 4.5).

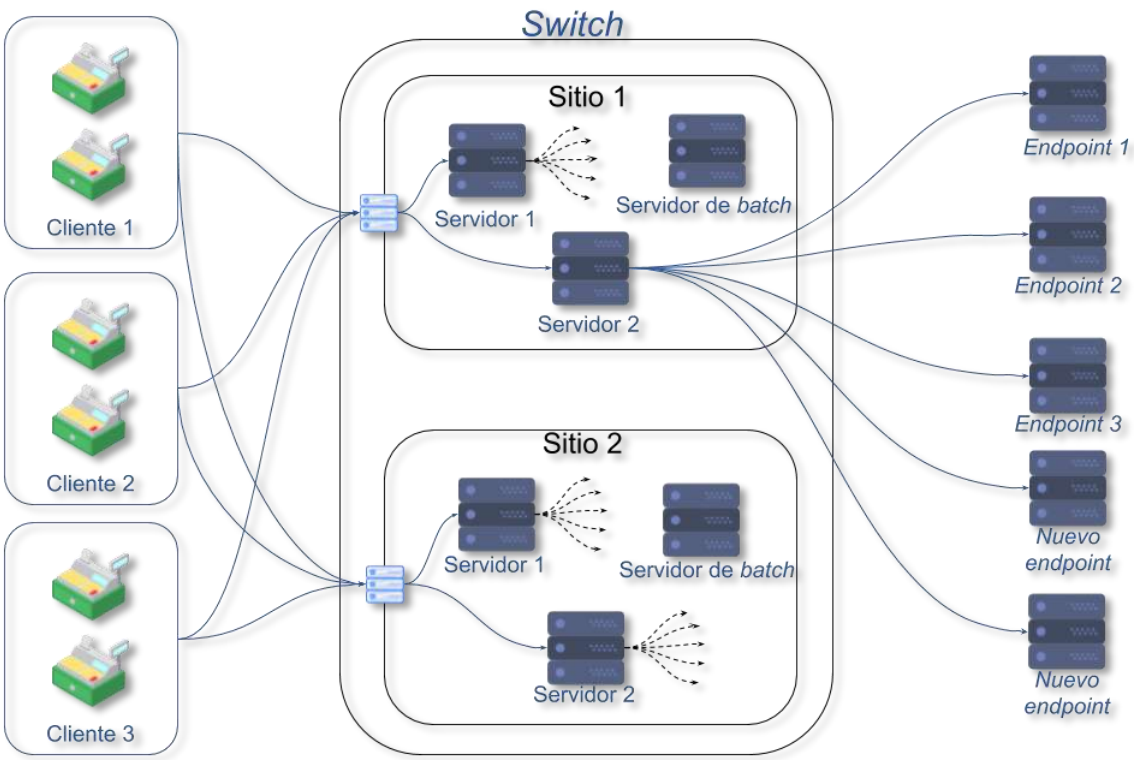

Ilustración 15 - Arquitectura completa del sistema

4.1. Servidores de transacciones

Los servidores de transacciones son la parte principal del sistema. Dichos servidores son los que mantienen alojado el servicio que procesará las diferentes transacciones. Este servicio es desarrollado en su mayoría por la empresa Tecro Ingeniería S.A, quien se encarga de colaborar en el mantenimiento del mismo. El software desarrollado es instalado como un servicio en cada uno de los servidores en cuestión, conectándose a una base de datos única compartida por todos los servidores procesadores de transacciones. Cabe destacar que, dicha base de datos se encuentra alojada en el sitio 1 (ver Ilustración 15). De esta manera, todos los servidores de ambas ciudades tienen acceso a la misma información. Para seguridad de los datos, la base de datos es completamente espejada en una base de datos alojada en el sitio 2, teniendo de esta manera un respaldo de la información.

Uno de los servicios más importantes con los que cuenta el software es con el sistema de SAF (*store and forward)*, traducido al español como sistema de almacenamiento y reenvío. Esto permite no solo asegurar la consistencia en las transacciones, sino que, a su vez, el almacenamiento de transacciones que no requieren de un inmediato procesamiento que serán procesadas más tarde. Estos casos de procesamiento diferenciales hacen que el *switch* sea quien brinde una respuesta inmediata al cliente sin haber tenido una respuesta definitiva por parte del *endpoint*, esto se debe a que la operación es almacenada en la base de datos para un posterior procesamiento. Un ejemplo de esto es cuando se desea revertir o cancelar una operación que ya ha sido concluida exitosamente, es decir, supongamos que la primera operación en concretarse correctamente fue la activación de una tarjeta de regalo, dicha transacción realizó el circuito normal. Inmediatamente después de dicha activación llega al

switch un pedido de cancelación de dicha transacción, es decir, deshacer la última operación efectuada, siendo en este caso la activación de la tarjeta. Esta transacción viene con datos de la transacción original, si dicha transacción es encontrada en la base de datos, se devuelve al cliente una respuesta afirmativa, y el pedido de cancelación es guardado en SAF para un futuro procesamiento. Esta operación queda almacenada y es retrasmitida hasta su correcto procesamiento por parte de *endpoint*. De esta manera no solo se tiene una respuesta rápida para el cliente, si no, que al mismo tiempo se tiene la certeza de que la operación será cancelada, ya que se intentará hasta tener éxito o haberlo intentado una cierta cantidad de veces, siendo en estos casos necesario la intervención humana.

4.2. Servidores de *batch* o reportes

El sistema en su totalidad no solo debe brindar soporte para el procesamiento de transacciones, si no que a su vez debe ser capaz de generar reportes que faciliten la conciliación entre el cliente, la empresa Emp-X y el determinado *endpoint* en cuestión. Estos servidores generadores de reportes tienen fundamentalmente dos trabajos, la generación de los *extracts* y los *spool* de carga de datos.

Los *extracts* son reportes personalizados que permiten obtener a través de un archivo, comúnmente de formato de valores separados por coma (*csv – comma separated value*), parte de la información que se encuentra en la base de datos. Esto no solo permite lograr la conciliación entre las diferentes empresas, sino que también es una manera de generar archivos de respaldo cada un determinado periodo de tiempo. Estos archivos son generados periódicamente cada una cierta cantidad de tiempo, que es configurada basada en la cantidad de transacciones diarias que son procesadas, las necesidades de las partes interesadas, etc. Estos valores son comúnmente pre-acordados con las partes involucradas. Una vez que dichas transacciones son extraídas de la base de datos, son marcadas para no volver a ser extractadas nuevamente. En cuando al archivo generado, éste es encriptado y enviado de manera automática vía SFTP a la ubicación que se le haya configurado previamente.

Por otra parte, los *spools* son servicios secundarios que están a la espera de la detección de un archivo para procesarlo. Este servicio a diferencia del anteriormente mencionado, no produce como salida un archivo de reporte, sino que, a partir de un archivo con un formato determinado, ingresa datos a la base de datos. Un claro ejemplo de esto son los *spool* de carga de sucursales. Los clientes que trabajan con la empresa Emp-X son en su mayoría grandes cadenas de negocios, por lo que la apertura de una nueva sucursal, como así también el cierre de alguna de ellas a lo largo de todo el país es algo que se da diariamente; es por ello que los listados de altas, bajas y modificaciones de tiendas son provistos a la empresa Emp-X periódicamente. A los fines de no hacer cambios en la base de datos de manera manual, es que estos archivos son ingresados a los *spools*. De esta manera, el servicio hace los cambios pertinentes en base de datos sin la intervención humana. Esto facilita el trabajo, pero principalmente, evita que alguien pueda cometer un error en el ingreso de los datos. Otra actividad frecuente donde, al igual que en las sucursales, se requiere de estos *spools* es la de mantener actualizados los productos disponibles a la venta, como por ejemplo el listado de nuevas tarjetas de regalo.

4.3. VIPs – IPs Virtuales

Los VIPs, acrónimo usado para "*Virtual* IP", traducidos al español como IPs virtuales, son los servidores que se encuentran distribuidos en cada uno de los sitios donde la Emp-X tiene servidores de transacciones. Dichos servidores tienen múltiples funciones, aunque la principal desde el punto de vista del servicio aquí analizado, es el re-direccionamiento de las transacciones. Por lo tanto, cada transacción entrante debe ser enviada a un servidor activo dentro del lugar, esto implica que el VIP debe contar con la capacidad de detectar cuales servidores están activos y cuáles no. Por lo tanto, el servidor puede reenviar transacciones a otro servidor, en caso de que alguno de los servidores presente problemas o se encuentre fuera de servicio.

Si bien la principal función es re-direccionar las transacciones, también opera como primera barrera de seguridad. Este servidor tiene múltiples reglas de conexión, dentro de las cuales está aceptar transacciones de determinados IP's con quienes mantiene conexiones seguras. De esta forma, garantiza que la información enviada de un extremo a otro no sea alterada o escuchada en el trayecto.

4.4. Múltiples conexiones a los *endpoints*

En la realidad otro de los puntos fundamentales para esta clase de servicios de alta disponibilidad es la habilidad para poder conectarse con múltiples servidores provistos por los diferentes *endpoints*. Para ello, no solo tenemos que poder distribuir las cargas de envío de mensajes, sino que también tener la capacidad de poder unir un mensaje de respuesta con su correspondiente mensaje de solicitud. En estos casos se cuenta con una de las funcionalidades provistas por el *framework* jPOS llamadas *MUX* y *MUX*Pool.

Los *MUX*s son las interfaces que permiten configurar a través de un archivo el canal por el cual el *switch* se va a comunicar con el *endpoint*. Dicho canal mantiene una conexión en una determinada dirección IP y un puerto. Además, es en este archivo de configuración de canal donde es especificado el formato con el que será transmitido el mensaje. Los *MUX*s a su vez, permiten configurar cuáles serán los campos del mensaje que permitan unir un mensaje de respuesta con un mensaje de solicitud de manera unívoca. A su vez, debido a que usualmente se cuenta con múltiples *MUX*s de conexión, jPOS brinda una herramienta llamada *MUX*pool que cumple la función de balanceador de cargas antes de enviar los mensajes. En este archivo se especifica todos los *MUX*s de conexión con los que se cuenta, y el mismo *MUX*pool es el encargado de distribuir los mensajes a través de los diferentes canales abiertos siguiendo algún método de planificación, para este caso se eligió *round-robin*. A su vez, cuenta con la capacidad de detectar cuáles de los canales previamente configurados están fuera de servicio, redistribuyendo la carga sobre los canales activos.

4.5. Replicación del servicio

La replicación es una pieza fundamental para cualquier servicio que quiera ser brindado con alta disponibilidad. Alta disponibilidad indica que la Emp-X tiene que tener la capacidad de estar funcional tanto a pesar de los inconvenientes rutinarios como por ejemplo la caída de un servidor como también así ante eventualidades mayores que puedan perjudicar a todos los servidores al mismo tiempo. Es por ello que la empresa Emp-X no solo tiene la

replicación de servidores en un mismo lugar, que le ofrece la posibilidad de lidiar con problemas como la necesidad de reiniciar un servidor sin quedar fuera de servicio, sino que más aún, tiene replicada toda la arquitectura en otro centro de datos en una ciudad diferente. Esto le permite lidiar con eventualidades más allá de lo convencional, por ejemplo, una ciudad sin energía por un periodo de tiempo demasiado extenso. La empresa Emp-X no solo presta servicio dentro de la ciudad donde se encuentran alojados del centro de datos, sino que lo hace alrededor de todo el país. Esto lleva al requerimiento de que un problema que afecte a una ciudad o una zona determinada en un caso extremo no debe impedir poder seguir operando de manera normal en el resto del país.

Capítulo 5 – Desarrollo de las actividades

De acuerdo a lo propuesto en el plan de trabajo "Sistema de conmutación de tarjetas de regalo (gift cards) basado en jPOS" [16] se llevaron a cabo las actividades que fueron detalladas en dicho documento. A través de los capítulos anteriormente descriptos se detalló sobre los lenguajes de programación, frameworks y tecnologías utilizadas; se explicó la metodología de trabajo que implementa la empresa y con la cual se llevaron a cabo las diferentes actividades propuestas, como así también la arquitectura actual de sistema sobre el que se estuvo trabajando. Esto permitió la familiarización de las tecnologías utilizadas, y de la metodología de trabajo de la empresa, lo cual facilitó poder llevar a cabo las actividades propuestas.

A lo largo de este capítulo se describirá en tres secciones las actividades realizadas dentro de la empresa, como así también, los conocimientos que fueron adquiridos para lograr la conexión con un nuevo *endpoint*, el soporte para el nuevo conjunto de transacciones que comenzaron a funcionar con la llegada de este *endpoint* y los test y pruebas que se utilizaron para asegurar el correcto funcionamiento del sistema.

5. 1 Conexión a un nuevo *endpoint*

Como se especificó en el plan de trabajo las actividades 1, 2 y 3 especifican la necesidad de establecer una conexión con un nuevo *endpoint*. La instancia de familiarización del proyecto como del framework utilizado fue llevada a cabo a través de reuniones con el tutor en la empresa, quién fue el encargado de transmitir las necesidades de nuestro cliente, en este caso, la Emp-X.

Dichas reuniones pusieron en contexto a todos los miembros de Tecro Ingeniería S.A afectados a dicho proyecto. De esta manera se explicó el proyecto, el fin de este, y las necesidades generales que poseía. En cuanto a las tareas específicas a desarrollar, estas fueron descriptas y asignadas a través de la página de asignación de tickets *YouTrack*[5]. En este caso se me especificó la tarea de implementar las clases y archivos necesarios, como así también sus correspondientes *test*, para establecer una nueva conexión entre el *switch* y un nuevo *endpoint*.

Para llevar a cabo esta actividad, se necesitó establecer la conexión a través de nuevos adaptadores de canales, como así también la especificación de nuevos *MUXs* que administren dichos canales.

5.1.1 Adaptador de canal

Para llevar a cabo la conexión, previa lectura de la especificación del framework [17], se debió especificar el adaptador de canal que será utilizado para conectarse con el endpoint. Este componente es propio del framework jPOS y ofrece la posibilidad de establecer una

[5] YouTrack: Es un software privativo de la empresa JetBrains donde su principal función es el seguimiento de tareas/errores a través de tickets. También cuenta con herramientas de gestión de proyectos orientadas a desarrolladores y gestores de proyectos.

comunicación a través de un canal y mantener esta conexión de manera permanente, incluso aun cuando no está siendo utilizada.

En el Algoritmo 2 se puede apreciar la descripción QBeans que es necesaria para establecer un canal entre el *switch* y el *endpoint*. En dicho algoritmo se especifica:

- *Name*. Esta variable define el nombre que se le es dado al adaptador del canal, este debe ser único. Dicho nombre es el que permite especificarle a un Q*MUX* que lo administre.
- *Channel*. Dentro de esta propiedad se puede especificar cuál será la clase java que se ejecutará cuando dicho servicio sea llamado, a saber:
    - *Packager*. Indica el *packager* con el que serán procesados los diferentes mensajes.
    - *Host*. La dirección IP del *endpoint*
    - *Port*. El puerto de escucha del *endpoint*
    - *Timeout*. El tiempo máximo por el que se espera una respuesta.
    - *Keep-alive*. Si se desea mantener la conexión establecida de manera constante.
    - *Filter*. Se pueden especificar diferentes tipos de filtros, tanto de entrada como de salida.
- *In*. Nombre de la cola de mensajes listos para ser transmitidos al *endpoint*.
- *Out*. Nombre de la cola donde se colocarán los mensajes entrantes desde el *endpoint*.
- *Reconnect-delay*. Si por algún motivo la conexión con el *endpoint* es finalizada, el canal intentará reconectarse después del tiempo en milisegundos especificado en este campo.

Una de las ventajas más significativas con la que cuenta dicho adaptador es la posibilidad de mantener "viva" la conexión, de lo contrario, tras un cierto periodo de inactividad entre las dos partes (*switch-endpoint*) la conexión entre ambos es cerrada. Esto puede causar problemas a la hora de ofrecer un servicio de alta prestación y rápida respuesta, ya que uno de los

```
1.  <channel-adaptor name='Canal-de-ejemplo' logger="Q2">
2.    <channel class="org.jpos.iso.channel.NACChannel"
3.      packager="org.jpos.iso.packager.GenericPackager">
4.      <property name="packager-config"
5.      value="jar:packager/iso87binary.xml" />
6.      <property name="host" value="127.0.0.1" />
7.      <property name="port" value="8001" />
8.      <property name="timeout" value="300000" />
9.      <property name="keep-alive" value="true" />
10.      <filter class="org.jpos.iso.filter.IncomingFilter"
11.         direction="incoming" />
12.      <filter class="org.jpos.iso.filter.OutgoingFilter"
13.         direction="outgoing" />
14.    </channel>
15.    <in>Canal-de-ejemplo-send</in>
16.    <out> Canal-de-ejemplo-receive</out>
17.    <reconnect-delay>10000</reconnect-delay>
18.    </channel-adaptor>
```

Algoritmo 2 – Descripción QBean de un adaptador de canal

factores por lo que una petición puede ser rechazada es cuando el tiempo que conlleva procesar toda la transacción es mayor al *timeout* establecido. Por lo tanto, para que la conexión se mantenga siempre activa, este adaptador de canal se encarga de enviar cada cierto periodo preestablecido de tiempo un mensaje al *endpoint*. Estos mensajes basados en el estándar ISO 8583 [9], llamados mensajes de red, tienen un indicador de tipo de mensaje particular, MTI por sus siglas en inglés, (0800 para peticiones y 0810 para respuestas) y cuentan con la información mínima necesaria para comprobar el estado del canal. De esta manera, a través de estos mensajes de red la conexión nunca es cerrada, y si el *endpoint* tuviese un problema podría a través de dichos mensajes comunicarlo sin necesidad de desconectarse.

5.1.2 *MUX*

Debido a que solo se puede establecer un solo *socket* de conexión por transacción, nuestro canal se encontraría totalmente ocupado mientras se lleva a cabo una transacción hasta el *endpoint*, es decir, que todos los mensajes entrantes deberían acumularse en una cola a la espera de poder tener su oportunidad. Un enfoque como este conllevaría a que la mayoría de las transacciones sean canceladas por el mismo *switch* debido al tiempo total que tomaría una transacción. Por lo tanto, el canal debería contar con un multiplexador que permita abrir múltiples sockets. Es por eso que el framework jPOS ofrece una interface que cumple esta función. Estas interfaces son llamadas *MUX*, y permiten recibir y manejar múltiples peticiones, enviándolas a través del canal previamente configurado.

Cada *MUX* está asociado a un canal previamente descripto. De esta manera en la cola de entrada (En inglés "*in*") se espera que sea enviado a través del *out* (cola de salida) del adaptador de canal. La misma idea aplica para el *out* del *MUX* que se convierte en el *in* del canal. En el Algoritmo 3 se puede apreciar la configuración de un *MUX* para un canal ya establecido, en este caso se especifica el *MUX* para el canal previamente definido en el algoritmo 2. La propiedad de *ready*, traducido al español como "disponible", indica la cola donde el *MUX* puede consultar sobre el estado del canal así saber si se encuentra disponible y listo para enviar un mensaje. Un modelo de una conexión *MUX*-Adaptador de canal puede ser apreciada en la Ilustración 16.

Debido a que el *MUX* recibe múltiples peticiones y las envía a través del canal, cuando recibe una respuesta debe saber exactamente con cual petición unir para no entregar una respuesta. errónea a una determinada petición. Por lo tanto, se debe especificar al *MUX* una *key*, (llave). Esto es un campo del mensaje o por lo general un conjunto de estos que conforman una llave única de información por mensaje. En consecuencia, cuando llega una nueva petición, el *MUX* reserva esa llave para el mensaje entrante y solo la libera cuando obtiene una

```
1. <MUX class="org.jpos.q2.iso.QMUX" logger="Q2" name="myMUX">
2.   <key>42 41 11</key>
3.   <key mti="0800">41</key>
4.   <in>Canal-de-ejemplo-receive</in>
5.   <out>Canal-de-ejemplo-send</out>
6.   <ready>Canal-de-ejemplo.ready</ready>
7.   <unhandled>mensajes-no-controlados</unhandled>
8. </MUX>
```

Algoritmo 3 - Descripción QBeans de un *MUX*

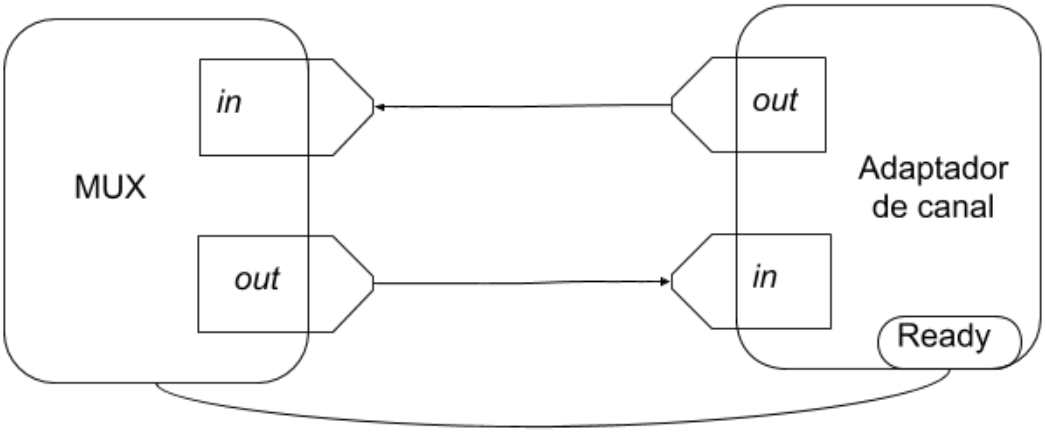

Ilustración 16 - Conexión de un *MUX* a un adaptador de canal

respuesta para este o se cumple su periodo de espera. Por lo tanto, la elección de estos campos debe forman un conjunto único dentro de cada mensaje, de lo contrario cuando un mensaje intenta registrarse en el *MUX* con una llave que ya fue utilizada y aún no fue liberada, la transacción abortará. Dado a que no todos los diferentes tipos de mensajes contienen exactamente los mismos campos, el *framework* provee una funcionalidad que permite establecer llaves especiales por MTI (Message Type Indicator, en español, indicador de Tipo de Mensaje). De esta manera, a los mensajes que contienen menos campos (por ej. los mensajes de red) es posible especificarles una llave compuesta diferente.

Por último, se especifica la cola de mensajes no controlados. Dichos mensajes son los que llegan al *MUX* pero que no se esperaba que arriben. Es decir, la llave de ese mensaje no coincide con ninguna de las llaves registradas en el *MUX*. Esto puede ser porque el *endpoint* modificó alguno de los campos que fueron seleccionados como campos llave, o bien, desde el *endpoint* se envió alguna petición que no estaba siendo esperada por el canal. En ambos casos, estos mensajes son enviados a la cola de mensajes no controlados, para luego ser procesados de manera especial. En el caso de la Emp-X la decisión es guardarlos, en una tabla diferente de la base de datos, para un futuro análisis de la situación.

5. 2 Soporte para nuevas transacciones

Muchas de las operaciones que se pueden hacer sobre una tarjeta de regalo son comunes en la mayoría de los proveedores de tarjetas sin importar que *endpoint* termine procesándolas. Por ejemplo, la activación de una tarjeta, la desactivación de ésta, o el pedido de balance de la cuenta que está asociada a dicha tarjeta. Sin embargo, no todos los *endpoints* tiene la capacidad de procesar las mismas operaciones y/u ofrecer las mismas prestaciones o facilidades sobre sus productos. Es por esto que el anexado de un nuevo *endpoint* al *switch* trae consigo la necesidad de incorporar el soporte para todas las transacciones que este ofrece.

Esto implica que una vez que la transacción entra al sistema, debe haber un flujo de participantes capaz de procesarlas. El agregado de un nuevo *endpoint* produce la necesidad del agregado de nuevos participantes al flujo normal de las transacciones. Para llevar a cabo esta tarea debemos conocer cómo funciona el *TransactionManager* de jPOS, cuáles son sus

características y cuáles son las características que se necesitan modificar para poder llevar a cabo nuestro objetivo.

5. 2 .1 TransactionManager

El "*TransactionManager*" es un servicio jPOS, cuya principal función es supervisar una cola declarada en un cierto espacio en la cual están las transacciones en espera de que se procesen. Se espera que estas transacciones sean un objeto *serializable*, siendo en la mayoría de las aplicaciones jPOS objetos de tipo "org.jpos.transaction.*Context*" una clase específica del *framework* jPOS que implementa la interfaz java *serializable*.

Por lo tanto, como puede ser observado en la Ilustración 17, el servicio que se encarga de recibir los mensajes entrantes (es este caso un servico QServer), crea un objeto *serializable*, en nuestro caso de tipo *Context*, y lo pone en la cola a la espera de ser procesado. El *TransactionManager* se encarga de tomar ese objeto y procesarlo a través de los diferentes participantes declarados. Estos participantes, como se puede apreciar en el Algoritmo 4 describen cuál es la clase java que los procesa, cuál será el archivo donde se escribirá el *log*, y de ser necesario especificarán todas las propiedades que dicho participante necesite. Estas propiedades están descriptas en un formato de llave-valor, donde *name* hace referencia al nombre de la propiedad, el cual debe ser único; por su parte *value* especifica el valor de dicha propiedad. Estas propiedades son específicas de los participantes, por lo tanto, un participante puede o no contener propiedades. Estos participantes pueden ser propios del *framework* o también pueden crearse basados en las necesidades de la empresa, esto puede implicar la edición de algunos de los *frameworks* o realizar un participante completamente independiente.

Una vez que la transacción entra al sistema, esta es procesada por un grupo de participantes. Sin embargo, no todas las transacciones son procesadas por el mismo grupo. Esto quiere decir, que no todas las transacciones siguen el mismo flujo, si no que algunas de ellas, son procesadas por más o menos participantes, dependiendo de la información que posean Uno de los participantes más utilizados, provistos por el *framework* jPOS, es el llamado "*switch*" un participante que implementa la clase *groupSelector*. Esta clase cuenta con la capacidad de modificar el flujo de la transacción. Dada cierta información que posee la transacción en sí,

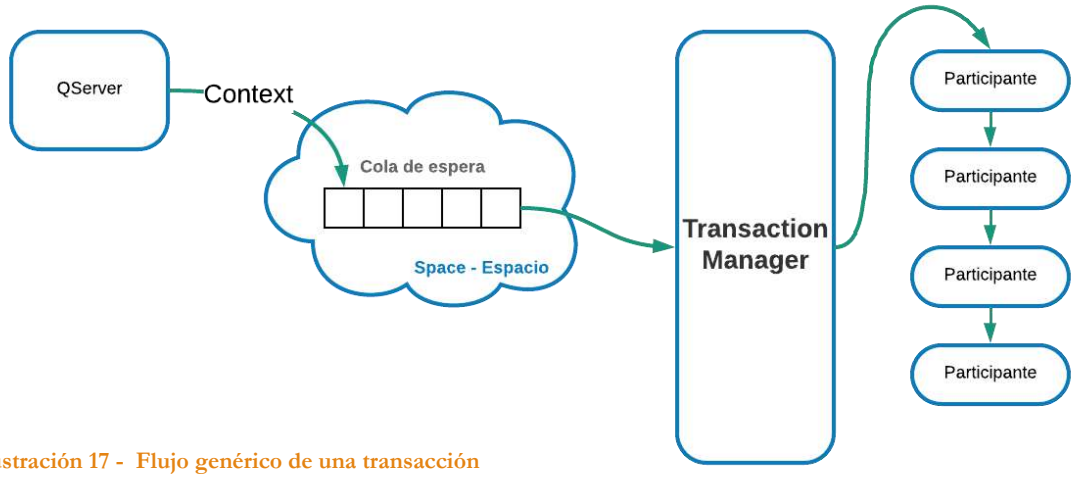

Ilustración 17 - Flujo genérico de una transacción

```
1.  <txnmgr name="MyTxn" logger="Q2"
2.   class="org.jpos.q2.transaction.TransactionManager">
3.    <property name="space"                  value="tspace:default" />
4.    <property name="queue"                  value="MyTxnQueue" />
5.    <property name="profiler"               value="true" />
6.    <property name="sessions"               value="32" />
7.    <property name="call-selector-on-abort" value="true" />
8.
9.    <participant class="com.my.company.ValidateMessage" logger="Q2"/>
10.   <participant class="com.my.company.FetchData"       logger="Q2"/>
11.   <participant class="com.my.company.QueryRemoteHost" logger="Q2"/>
12.   <participant class="com.my.company.LogTransaction"  logger="Q2"/>
13.   <participant class="com.my.company.SendResponse"    logger="Q2">
14.     <property name="my.optional.property"       value="abc" />
15.     <property name="my.other.optional.property" value="xyz" />
16.   </participant>
17. </txnmgr>
```

Algoritmo 4 - Descripción de un TransactionManager

esta clase devuelve al *TransactionManager* la lista de participantes o grupos de estos que deben ser ejecutados a continuación. De esta manera si, por ejemplo, arriba una nueva transacción podemos verificar si campos que son consideramos obligatorios se encuentran presentes. Si estos campos se encuentran presente, la transacción sigue un camino, es decir ejecuta un conjunto determinados de participantes, mientras que, si alguno de estos campos faltase, el conjunto de participantes a ejecutar es otro, donde se crea un mensaje específico de error y le es reenviado al comerciante. De esta manera un *TransactionManager*, no tiene un flujo único, y permite describir participantes especiales para ser ejecutados bajo solo ciertas circunstancias.

Cada participante extienda de la clase "*TxnSupport*" y deben sobrescribir tres métodos que son llamados por el mismo *TransactionManager*; estos son: "*doPrepare*", "*commit*" y "*abort*". Como es apreciado en la Ilustración 18, el flujo normal de una transacción se divide en dos partes, en el primero, el *TransactionManager* se encarga de llamar al método *doPreapare* de cada participante involucrado en la transacción, este método es el que prepara todo lo necesario para que luego, en la fase dos, se pueda confirmar la operación, si no hubo ningún inconveniente en el flujo. Este método permite devolver dos valores: *PREPARED* o *ABORTED*. El primero de estos indica que el método terminó de ejecutarse correctamente, mientras que el segundo de los casos presenta que algún error ocurrió durante el proceso. A su vez, estas dos salidas pueden estar acompañadas de los modificadores *NO_JOIN* y *READONLY*, el primero de estos indica que en la segunda fase (de aborto o confirmación) el participante no debe ser ejecutado. Por otra parte, el modificador *READONLY* permite especificarle al *TransactionManager* que ningún dato persistente en el *Context* fue editado, por lo que no se requiere guardar el *Context*, ya que por defecto y por cuestiones de seguridad y restauración de la transacción el *TransactionManager* guarda una captura del *Context* después de ejecutarse cada participante.

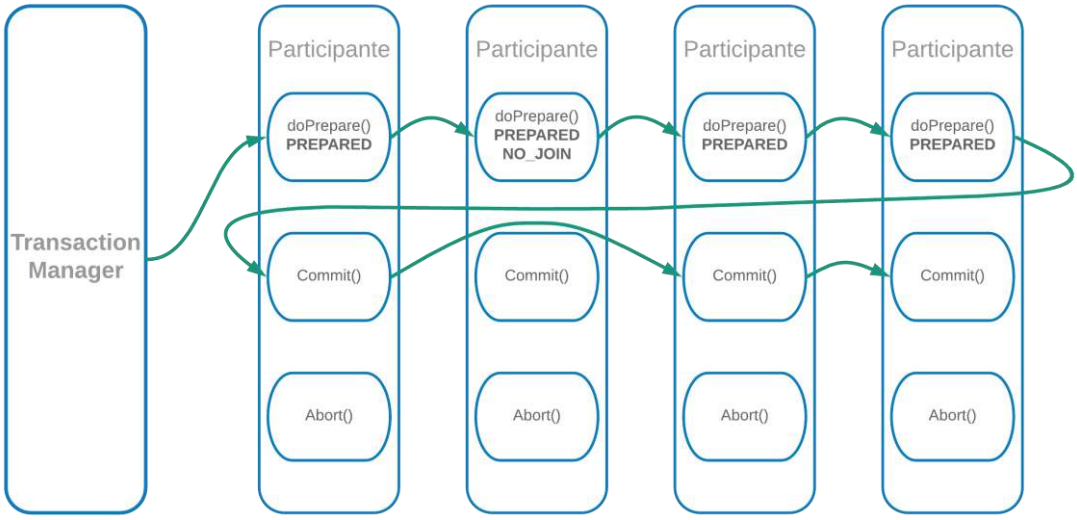

Ilustración 18 - Flujo de una transacción exitosa

Si al llegar al último participante de la lista, y todos estos arrojaron como salida el valor *PREPARED*, el *TransactionManager* comienza a llamar al método "*commit"* de todos los participantes que ejecutaron el *"doPrepare"* anteriormente. En el caso de que alguno de los participantes, cualquiera sea, haya presentado algún inconveniente durante la preparación de la transacción, es decir, haya arrojado como salida el valor *ABORTED,* el *TransactionManager* deja de llamar a los siguientes y comienza a llamar al método *"abort"* desde el primer participante en la lista hasta el participante que haya producido el "*abort*" salteando a todos los participantes que hayan devuelto "*NO_JOIN*", una representación del flujo de una transacción que aborta puede ser observado en la Ilustración 19.

Cada *TransactionManager* está definido para procesar todos los objetos *serializables* que están definidos en una determinada cola declarada en un espacio específico, los cuales son descriptos a través de las propiedades de este servicio. Como puede observarse en el

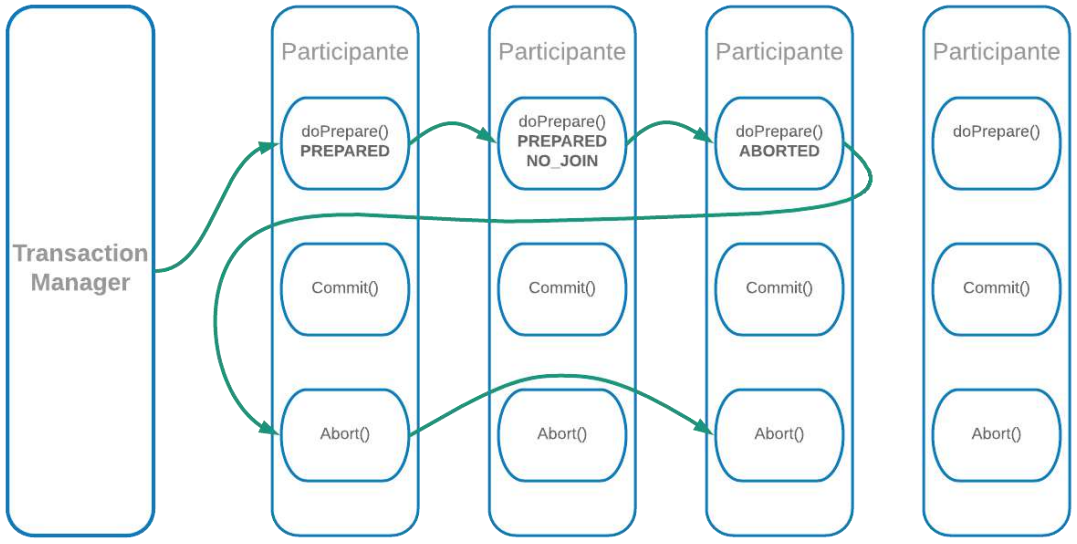

Ilustración 19 - Flujo de una transacción que aborta

algoritmo 4, a través de las propiedades es posible especificar en el *TransactionManager* las siguientes condiciones:

- *Space*. Es el espacio utilizado por el *TransactionManager* para manejar las transacciones que están activas en ese momento.
- *Queue*. Esta propiedad define la cola de la cual el *TransactionManager* tomará los objetos *seriablizables* de tipo *Context* para ser procesados. Esta es la única propiedad que debe estar presente de manera obligatoria a la hora de declarar un *TransactionManager*, ya que de no estar presente el *TransactionManager* no sabría de cual cola procesar.
- *Sessions*. Define la cantidad de sesiones simultaneas *(Threads)* utilizadas para procesar la transacción. De no estar presente esta propiedad se toma por defecto el valor 1. Es recomendable usar números de sesiones acorde a la cantidad de núcleos con las que el servidor está trabajando.
- *Debug*. Si esta propiedad es establecida como *true*, en español "verdadera", el *TransactionManager* registra un pequeño informe en el archivo de log después de cada transacción que indica qué participantes estuvieron involucrados en cada transacción, un ejemplo de este puede ser observado en el Algoritmo 5.
- *Profiler*. Esta propiedad al igual que la anterior de estar establecida como verdadera, además del informe que ofrece el *debug*, el *TransactionManager* registra los tiempos consumidos por cada uno de los participantes. De especificar esta propiedad como verdadera, la propiedad de *debug* se establecerá también como verdadera, sin importar lo que se haya especificado previamente. El algoritmo 6 muestra un ejemplo de como esto es descripto en el archivo de log.
- *Call-selector-on-abort:* El *TransactionManager* llama en cada uno de los participantes al método "doP*repare"* y luego, si el participante implementa la interfaz de *GroupSelector*, dicho participante llama al método "*select*" sin importar el resultado del método

```
1.  <debug>
2.   txnmgr-1:2
3.    prepare: org.jpos.jcard.Prepare Context NO_JOIN
4.    prepare: org.jpos.jcard.CheckVersion READONLY NO_JOIN
5.    prepare: org.jpos.transaction.Open READONLY NO_JOIN
6.    prepare: org.jpos.jcard.Switch READONLY NO_JOIN
7.    groupSelector: notsupported prepareresponse close sendresponse
8.    prepare: org.jpos.jcard.NotSupported NO_JOIN
9.    prepare: org.jpos.jcard.PrepareResponse NO_JOIN
10.   prepare: org.jpos.transaction.Close READONLY
11.   prepare: org.jpos.jcard.SendResponse READONLY
12.   prepare: org.jpos.jcard.ProtectDebugInfo READONLY
13.   prepare: org.jpos.transaction.Debug READONLY
14.   commit: org.jpos.transaction.Close
15.   commit: org.jpos.jcard.SendResponse
16.   commit: org.jpos.jcard.ProtectDebugInfo
17.   commit: org.jpos.transaction.Debug
18.  head=3, tail=3, outstanding=0, active-sessions=2/2, tps=0,
19.  peak=0,avg=0.00, elapsed=22ms
20. </debug>
```

Algoritmo 5 - Ejemplo de un *debug*

```
1.  <debug>
2.    ....
3.    ....
4.    <profiler>
5.      prepare: org.jpos.jcard.PrepareContext [0.0/0.0]
6.      prepare: org.jpos.jcard.CheckVersion [0.0/0.0]
7.      prepare: org.jpos.transaction.Open [0.5/0.6]
8.      prepare: org.jpos.jcard.Switch [0.0/0.6]
9.      prepare: org.jpos.jcard.NotSupported [0.1/0.7]
10.     prepare: org.jpos.jcard.PrepareResponse [5.8/6.6]
11.     prepare: org.jpos.transaction.Close [0.0/6.6]
12.     prepare: org.jpos.jcard.SendResponse [0.0/6.6]
13.     prepare: org.jpos.jcard.ProtectDebugInfo [0.0/6.7]
14.     prepare: org.jpos.transaction.Debug [0.0/6.7]
15.     commit: org.jpos.transaction.Close [1.0/7.7]
16.     commit: org.jpos.jcard.SendResponse [4.3/12.0]
17.     commit: org.jpos.jcard.ProtectDebugInfo [0.2/12.3]
18.     commit: org.jpos.transaction.Debug [9.3/21.7]
19.     end [22.8/22.8]
20.   </profiler>
21. </debug>
```

Algoritmo 6 - Ejemplo de un *profiler*

"*prepare*". En algunos casos, sin embargo, el desarrollador puede desear que si el participante actual, o alguno de los anteriores tuvo algún inconveniente y arrojaron como resultado un *abort* el método *"select"* no sea ejecutado. Este comportamiento puede ser configurado a través de esta propiedad.

5. 3 Test y pruebas sobre el *endpoint*

Para llevar a cabo los diferentes *test* sobre todo lo realizado se utilizaron principalmente dos instancias. La primera de estas basadas en simuladores y la segunda de ellas estableciendo una conexión con los servidores de *test* del nuevo *endpoint*.

Para la primera instancia, se desarrollaron simuladores. Estos son los encargados de imitar el comportamiento del *endpoint*. Estos simuladores son desarrollados basados en la documentación que la empresa con la que se desea establecer la nueva conexión provee a la empresa Emp-X. Estos ofrecen la posibilidad de poder inyectar sobre el *switch* mensajes como los que se esperan recibir provenientes de los clientes/comerciantes. Es en esta instancia donde se desarrollaron un conjunto de casos de pruebas, que imitan los diferentes escenarios con los que el sistema se va a ver envuelto. Es decir, se crean casos de pruebas para cada tipo de transacción. Estos casos de pruebas consisten en mensajes que son inyectados en el *switch* y a los cuales se les analiza la respuesta. No solo se imitan los casos de éxito, si no más aún se crean casos de pruebas para situaciones atípicas, donde las circunstancias exceden al manejo de la empresa Emp-X. De esta manera se puede también analizar el comportamiento del sistema frente a estas situaciones.

Para llevar a cabo estos test, se utilizó el *framework* de test TestNG [18]. Este es un *framework* diseñado para simplificar una amplia gama de necesidades de prueba, desde pruebas unitarias (probando una clase aisladamente de las demás) hasta pruebas de integración (probando

sistemas completos hechos de varias clases, varios paquetes e incluso otras *frameworks* externos, como servidores de aplicaciones).

Dichos test son utilizados de manera local por cada uno de los desarrolladores para corroborar el trabajo que se fue realizando, y asegurarse que la implementación de nuevas características no afectó las que el sistema ya poseía. A su vez, como el sistema con el cual se está trabajando, posee una integración continua, como fue explicado en la sección 3.2, el servidor de versiones dispara una notificación para que en primera instancia todos los desarrolladores sean notificados. Seguido a todo el código que se encuentra en el servidor, es compilado de manera automática, seguido a este una vez compilado el sistema se procede ejecutar todos los test que fueron descriptos anteriormente. Esto se produce en un servidor independiente, destinado exclusivamente para esto. De esta manera todos los test son probados siempre en el mismo entorno independientemente de quien lo haya desarrollado. Seguido a esto, todos los involucrados en el proyecto son notificados de los resultados obtenidos de los test.

Una vez que el sistema cuanta con una robustez que el gerente de proyecto considera apta para el ambiente de test, el sistema completo es instalado en los ambientes de test. Estos son servidores exclusivos donde el sistema queda funcional todo el tiempo a la espera del ingreso de algún mensaje al cual procesar, aquí no son ejecutados los test anteriormente descriptos. En este ambiente, se utilizan herramientas especiales desarrolladas por la compañía para inyectar sobre el sistema mensajes y analizar el comportamiento completo del sistema. Desde el punto de vista del *endpoint*, aquí todavía son utilizados los simuladores que fueron descriptos anteriormente.

Una vez que esta fase de prueba, es terminada, se procede a la instalación de *switch* en los ambientes de QA (*quality assurance*), en español aseguramiento de calidad. Este entorno de prueba se diferencia del anterior principalmente por dos puntos: El primero de ellos, es que no se instala en solo un servidor, si no que se replica el entorno de producción de manera completa, con tantos servidores trabajando en paralelo como los que se estén asignados a producción. El segundo punto importante es que se deja de trabajar con los simuladores desarrollados para establecer una comunicación con los servidores de test del *endpoint* en cuestión. De esta manera se analiza el comportamiento del sistema frente a una conexión en tiempo real.

Otro punto importante es que pasada esta fase de prueba, se habilita la conexión del entorno de QA con el nuevo cliente de la Emp-X. De esta manera, se le brinda al cliente la posibilidad de que puedan hacer sus casos de pruebas sobre el sistema hasta el momento desarrollado.

Estas fases de pruebas están acompañadas de comunicación constante entre todas las partes involucradas. Incluyendo en muchos de los casos, momentos donde las diferentes empresas prueban sus sistemas simultáneamente para asegurar el correcto funcionamiento de todos los sistemas y obtener respuestas de manera rápida ante situaciones atípicas que pudiesen surgir durante la fase de test.

Capítulo 6: Conclusiones

A través de los diferentes capítulos de este trabajo se pudo conocer, comprender, y trabajar con algunas de las tecnologías con las que el mercado laboral cuenta en la actualidad. La gama de diferentes tecnologías y *frameworks* que pueden ser encontrados en este contexto son amplias y diversas. Pudiendo ser estas pagas o gratuitas, de código abierto o privativas. Es por esto que instruirse en particular sobre las tecnologías con la que la empresa Tecro Ingeniería S.A trabaja en la actualidad resultó tan importante como enriquecedor. La utilización del lenguaje de programación java, uno de los más utilizados en el mercado, junto con la metodología de trabajo Scrum fueron los que permitieron llevar a cabo la realización de las actividades sin inconvenientes. Conocer la arquitectura del sistema sobre el que se estuvo trabajando fue lo que permitió tener un conocimiento general de la empresa y sus clientes, como así también, del servicio que prestan.

Esta práctica en empresa, permitió alcanzar los objetivos tanto generales como específicos propuestos en el plan de trabajo. Es decir que, las diferentes actividades realizadas permitieron que en la actualidad el *switch* cuente con conexión a un nuevo *endpoint* y con las características necesarias para el procesamiento de todos los tipos de transacciones que este soporte.

Si bien el presente trabajo fue desarrollado con la intencionalidad de agregar nuevas funcionalidades, basadas en la necesidad de una nueva conexión, en la realidad, los requerimientos de este sistema son más amplios y complejos. Esto habilita poder continuar en un futuro trabajando sobre nuevas funcionalidades para dicho proyecto. A su vez, dado que muchos de los otros sistemas en los que trabaja la empresa guardan semejanzas con este proyecto, y también utilizan muchas de las mismas tecnologías, permite en un futuro la realización o incorporación a otros proyectos sin mayores dificultades.

Enriquecimiento profesional

Poder llevar a cabo la experiencia de esta práctica en una empresa que se encuentra trabajando continuamente con compañías de otros países, trabajando con tecnologías y metodologías que son muy utilizadas en el común de las organizaciones, fue sin duda alguna, una de las experiencias más enriquecedoras. Esto no solo permitió la posibilidad de aplicar los diferentes conocimientos adquiridos a lo largo de la carrera, sino, más aún, permitió el crecimiento profesional, llevando a la práctica lo que fue adquirido en la teoría.

Referencias bibliográficas

[1] Oracle, «Java - Oracle,» Oracle, 14 febrero 2018. [En línea]. Disponible: https://www.java.com/es/about/whatis_java.jsp. [Último acceso: 14 febrero 2018].

[2] JAVA SE, «Oracle Java SE,» 18 febrero 2018. [En línea]. Disponible: http://www.oracle.com/technetwork/java/javase/documentation/index.html. [Último acceso: 18 febrero 2018].

[3] O. javaFX, «JavaFX,» Oracle, 24 Febrero 2018. [En línea]. Disponible: https://www.java.com/es/download/faq/javafx.xml. [Último acceso: 24 Febrero 2018].

[4] Oracle, «Java ME,» Oracle, 24 Febrero 2018. [En línea]. Disponible: http://www.oracle.com/technetwork/java/embedded/javame/java-mobile/overview/index.html. [Último acceso: 24 Febrero 2018].

[5] Oracle, «Java EE,» Oracle, 24 Febrero 2018. [En línea]. Disponible: http://www.oracle.com/technetwork/es/java/javaee/documentation/index.html. [Último acceso: 24 Febrero 2018].

[6] R. E. Johnson y . B. Foote, «Designing reusable classes,» *Journal of Object-Oriented Programming,* vol. 1, nº 2, pp. Pág. 22-35, 1988.

[7] F. S. Foundation, «GNU Operating System,» Free Software Foundation, 25 Febrero 2018. [En línea]. Disponible: https://www.gnu.org/licenses/lgpl.html. [Último acceso: 25 Febrero 2018].

[8] Hibernate, «Hibernate,» 25 Febrero 2018. [En línea]. Disponible: https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html_single/. [Último acceso: 25 Febrero 2018].

[9] ISO, «ISO - Web Site,» 03 diciembre 2017. [En línea]. Disponible: https://www.iso.org/obp/ui/#iso:std:iso:8583:-1:ed-1:v1:en. [Último acceso: 03 diciembre 2017].

[10] JPOS, «JPOS - Web Site,» 03 12 2017. [En línea]. Disponible: http://www.jpos.org/. [Último acceso: 03 12 2017].

[11] Apache, «Apache ANT,» 21 Marzo 2118. [En línea]. Disponible: http://ant.apache.org/. [Último acceso: 21 Marzo 2018].

[12] K. S. a. J. Sutherland, The Definitive Guide to Scrum: The Rules of the Game, 2017.

[13] M. Fowler, «Continuous Integration,» [En línea]. Disponible: https://martinfowler.com/articles/continuousIntegration.html. [Último acceso: 24 Junio 2018].

[14] D. T. &. otros, «Manifiesto por el Desarrollo Ágil de Software,» 01 Abril 2018. [En línea]. Disponible: http://agilemanifesto.org/iso/es/manifesto.html. [Último acceso: 01 Abril 2018].

[15] D. THOMAS, «GOTO Conference,» [En línea]. Disponible: http://gotocon.com/amsterdam-2015/presentation/EVENING%20KEYNOTE:%20Agile%20is%20Dead. [Último acceso: 20 Junio 2018].

[16] E. Bezzone, «Sistema de conmutación de tarjetas de regalo (gift cards) basado en jPOS,» Facultad de ingenería, UNLPam, General Pico, La Pampa, 2018.

[17] JPOS, «JPOS - Web Site,» 03 diciembre 2017. [En línea]. Disponible: http://www.jpos.org/. [Último acceso: 03 diciembre 2017].

[18] TestNG, «Framework TestNG,» 04 abril 2018. [En línea]. Disponible: https://testng.org/doc/index.html. [Último acceso: 04 abril 2018].