



Universidad de las Ciencias Informáticas

**Propuesta de Arquitectura de la Capa de Integración para
Sistemas de Información sobre tecnologías JEE.**

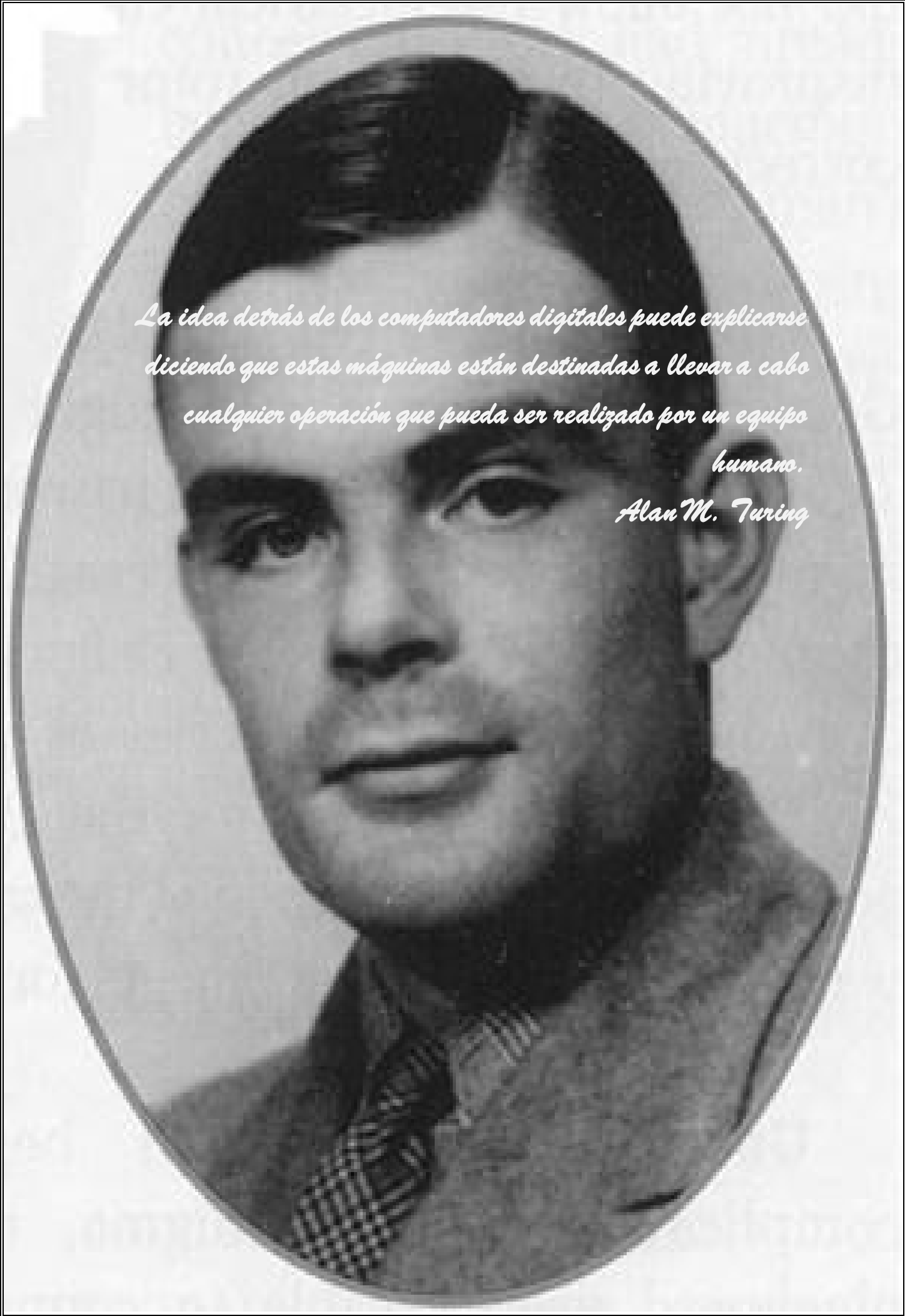
Trabajo de Diploma para optar por el título de Ingeniero en Ciencias
Informáticas.

Autor(es): Ray Williams Robinson Valiente

Tutor(es): Ing. Adolfo Miguel Iglesias Chaviano

Ciudad de la Habana, 20 de Mayo del 2011.

“Año 53 de la Revolución”

A black and white portrait of Alan Turing, a young man with dark hair, wearing a suit and tie. The portrait is enclosed in an oval frame. The background is a light, textured grey.

La idea detrás de los computadores digitales puede explicarse diciendo que estas máquinas están destinadas a llevar a cabo cualquier operación que pueda ser realizado por un equipo humano.

Alan M. Turing

Declaración de Autoría

Declaro ser el autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de junio del año 2011.

Ray Williams Robinson Valiente

Ing. Adolfo Miguel Iglesias Chaviano

Datos de Contacto

Tutor: Ing. Adolfo Miguel Iglesias Chaviano.

Síntesis del Tutor: Graduado en la Universidad de las Ciencias Informáticas con Título de Oro en el año 2007. Actualmente es Arquitecto del proyecto Quarxo perteneciente al Dpto. Soluciones Financieras de CEIGE.

Categoría Docente: Instructor.

Correo Electrónico: aiglesias@uci.cu .

Dedicatoria

Quiero dedicar la presente investigación a mi madre, sin la cual nada de lo que soy hubiera sido posible; a mi hermano, por ser el amigo eterno; y a mi novia, por existir.

Agradecimientos

La realización de la presente investigación no habría sido posible sin la colaboración de un buen número de personas, sin embargo sería injusto no hacer referencia a algunos cuya participación fue vital. Quiero por tanto agradecer explícitamente a:

- *Mi tutor, el Ing. Adolfo Miguel Iglesias Chaviano, por sugerir un tema de suma actualidad y envergadura, por el tiempo libre que encontró siempre para atenderme, por las indicaciones, las correcciones y en fin, todo el trabajo desplegado para llevar esta investigación a feliz término.*
- *Ing. Omar Antonio Díaz Peña, por sus valiosos consejos sin los cuales no habría llegado tal vez tan lejos.*
- *Armando Palacio Vázquez, mi compañero y amigo desde el preuniversitario, por permitirme usar parte de su tiempo durante las pruebas realizadas en la fase de evaluación.*
- *Mi novia, Yudelaine Maidelia Garcia Laborde por entenderme cuando no pude estar a su lado, por su apoyo y por las fuerzas que me dio siempre.*
- *Aquellos que en algún momento obstaculizaron el desarrollo de la presente investigación, gracias a ustedes también por poner a prueba mi capacidad de esfuerzo y llevarla al límite.*

A todos, sin distinción alguna, a los mencionados y a los que no lo están porque harían enorme la lista, por todo lo que hicieron, les estaré eternamente agradecido.

Resumen

Con la explosión del poder de cómputo del hardware que ha experimentado la informática desde los años '80 del siglo pasado, han ido quedando obsoletas gradualmente, viejas técnicas de desarrollo de software que generaban inmensas aplicaciones, sobre todo en el marco empresarial, cuyo entremado tanto conceptual como a nivel de implementación es poco menos que caótico. Este paradigma de aplicaciones monolíticas ha sido sustituido paulatinamente por un creciente auge de la modularización, lo cual unido a las exigencias actuales de las empresas, han originado la necesidad de integrar en un todo funcional, a aplicaciones distintas y dispares. La presente investigación presenta una propuesta de arquitectura para la sección del software construido sobre plataforma JEE encargada de manejar estas responsabilidades, basada en *Apache Camel*.

Palabras Claves: Integración, arquitectura, aplicaciones, *Apache Camel*.

Tabla de Contenido

DECLARACIÓN DE AUTORÍA	I
DATOS DE CONTACTO	II
DEDICATORIA	III
AGRADECIMIENTOS	IV
RESUMEN	V
ÍNDICE DE ILUSTRACIONES	VIII
ÍNDICE DE TABLAS	IX
INTRODUCCIÓN	1
OBJETO DE ESTUDIO	2
CAMPO DE ACCIÓN.....	3
OBJETIVO GENERAL	3
OBJETIVOS ESPECÍFICOS.....	3
IDEA A DEFENDER	3
TAREAS ESPECÍFICAS DE LA INVESTIGACIÓN.....	3
RESULTADOS ESPERADOS.....	4
ESTRUCTURA DEL DOCUMENTO.....	5
CAPÍTULO 1 – FUNDAMENTACIÓN TEÓRICA	6
INTRODUCCIÓN	6
DESCRIPCIÓN GENERAL DE LA INTEGRACIÓN DE SISTEMAS DE INFORMACIÓN.....	6
<i>¿Qué es la integración de sistemas?</i>	6
<i>Sistemas de Información</i>	8
<i>¿Es necesaria la integración?</i>	9
<i>Desafíos de la Integración de Sistemas</i>	9
NIVELES DE INTEGRACIÓN.....	12
<i>¿Qué son los niveles de integración?</i>	12
<i>Niveles de integración actuales</i>	12
ESTILOS DE INTEGRACIÓN.....	15
<i>¿Qué son los estilos de integración?</i>	15
<i>Estilos de integración actuales</i>	16
PRINCIPALES TECNOLOGÍAS DE INTEGRACIÓN SOBRE PLATAFORMA JEE	27
CONCLUSIONES PARCIALES.....	37
CAPÍTULO 2 – DESCRIPCIÓN DE LA ARQUITECTURA	39
INTRODUCCIÓN	39
DESCRIPCIÓN GENERAL	39
ESTRUCTURA INTERNA.....	43
<i>Lógica de Integración</i>	43
<i>Gestión de Errores</i>	53
<i>Transacciones</i>	56
<i>Seguridad</i>	60
CONCLUSIONES PARCIALES.....	65
CAPÍTULO 3 – EVALUACIÓN DE LA ARQUITECTURA	66
INTRODUCCIÓN	66
ATRIBUTOS DE CALIDAD.....	67
<i>Funcionalidad</i>	67
<i>Confiabilidad</i>	68

**Propuesta de Arquitectura de la Capa de Integración para
Sistemas de Información sobre tecnologías JEE**

<i>Usabilidad</i>	68
<i>Eficiencia</i>	68
<i>Mantenibilidad</i>	69
<i>Portabilidad</i>	69
CASO DE ESTUDIO	71
MÉTRICAS Y RESULTADOS	74
CONCLUSIONES PARCIALES.....	82
CONCLUSIONES	84
RECOMENDACIONES	86
GLOSARIO DE TÉRMINOS ANGLÓFONOS	87
GLOSARIO DE TÉRMINOS	88
REFERENCIAS BIBLIOGRÁFICAS	91
BIBLIOGRAFÍA	94

Índice de Ilustraciones

ILUSTRACIÓN 1: SISTEMA DE INFORMACIÓN. (J2EE CONNECTOR ARCHITECTURE AND ENTERPRISE APPLICATION INTEGRATION)	8
ILUSTRACIÓN 2: NIVELES DE INTEGRACIÓN (ENTERPRISE APPLICATION INTEGRATION)	13
ILUSTRACIÓN 3: TRANSFERENCIA DE ARCHIVOS GRÁFICAMENTE (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	17
ILUSTRACIÓN 4: APLICACIONES USANDO UNA BASE DE DATOS COMPARTIDA (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	19
ILUSTRACIÓN 5: COMUNICACIÓN MEDIANTE INVOCACIÓN DE PROCEDIMIENTOS REMOTOS (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	20
ILUSTRACIÓN 6: COMUNICACIÓN MEDIANTE MENSAJERÍA (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	21
ILUSTRACIÓN 7: LAS APLICACIONES SE COMUNICAN A TRAVÉS DE CANALES DE MENSAJE (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	23
ILUSTRACIÓN 8: LA INFORMACIÓN SE EMPAQUETA EN MENSAJES PARA SER TRANSMITIDA A TRAVÉS DE LOS CANALES (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	24
ILUSTRACIÓN 9: EL PATRÓN <i>PIPES AND FILTERS</i> PERMITE REALIZAR TRANSFORMACIONES COMPLEJAS SOBRE LOS MENSAJES (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	25
ILUSTRACIÓN 10: UN ENRUTADOR ENVÍA EL MENSAJE A UNO U OTRO CANAL DE SALIDA, DEPENDIENDO DE SU CONJUNTO DE CONDICIONES (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	26
ILUSTRACIÓN 11: UN TRADUCTOR TRANSFORMA EL FORMATO DE UN MENSAJE (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	26
ILUSTRACIÓN 12: UN PUNTO FINAL DE MENSAJE CONSTITUYE UNA ABSTRACCIÓN DEL SISTEMA DE MENSAJERÍA (ENTERPRISE INTEGRATION PATTERNS - DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS)	27
ILUSTRACIÓN 13: DISPOSICIÓN DE LAS CAPAS DE UN SISTEMA JEE Y FLUJO DE LA INFORMACIÓN ENTRE ELLAS	40
ILUSTRACIÓN 14: DISPOSICIÓN DE LA CAPA DE INTEGRACIÓN E INTERACCIÓN CON EL RESTO DEL SISTEMA	41
ILUSTRACIÓN 15: INTEGRACIÓN MEDIANTE UN <i>MESSAGE BROKER</i>	42
ILUSTRACIÓN 16: INTEGRACIÓN MEDIANTE COMUNICACIÓN DIRECTA	42
ILUSTRACIÓN 17: ESTRUCTURA INTERNA DE LA CAPA DE INTEGRACIÓN	43
ILUSTRACIÓN 18: ELEMENTOS INTERNOS DE LA SUBCAPA DE LÓGICA DE INTEGRACIÓN	44
ILUSTRACIÓN 19: UN OBJETO <i>MESSAGE</i> DE <i>APACHE CAMEL</i> (<i>CAMEL IN ACTION</i>)	48
ILUSTRACIÓN 20: UN OBJETO <i>EXCHANGE</i> DE <i>APACHE CAMEL</i> (<i>CAMEL IN ACTION</i>)	48
ILUSTRACIÓN 21: PATRÓN <i>SERVICE ACTIVATOR</i> EN <i>CAMEL</i> (<i>CAMEL IN ACTION</i>)	51
ILUSTRACIÓN 22: RELACIÓN ENTRE <i>EXCHANGE</i> , <i>UNITOFWORK</i> Y <i>SYNCHRONIZATION</i> (<i>CAMEL IN ACTION</i>)	59
ILUSTRACIÓN 23: DIAGRAMA DE DESPLIEGUE DEL ESCENARIO ACTUAL DEL CASO DE ESTUDIO	72
ILUSTRACIÓN 24: DIAGRAMA DE DESPLIEGUE FINAL DEL CASO DE ESTUDIO	74
ILUSTRACIÓN 25: RESULTADOS GRÁFICOS DE LAS PRUEBAS DE ESCALABILIDAD	83

Índice de Tablas

TABLA 1: ATRIBUTOS DESCARTADOS EN LA CONFORMACIÓN DEL MODELO DE CALIDAD.....	71
TABLA 2: MÉTRICAS UTILIZADAS. ADECUACIÓN FUNCIONAL.....	75
TABLA 3: MÉTRICAS UTILIZADAS. PRECISIÓN COMPUTACIONAL.....	75
TABLA 4: MÉTRICAS UTILIZADAS. INTERCAMBIABILIDAD.....	76
TABLA 5: MÉTRICAS UTILIZADAS. CAPACIDAD DE CONTROL DE ACCESO.....	76
TABLA 6: MÉTRICAS UTILIZADAS. DENSIDAD DE FALLAS CONTRA CASOS DE PRUEBA.....	76
TABLA 7: MÉTRICAS UTILIZADAS. PREVENCIÓN DE COLAPSOS.....	77
TABLA 8: MÉTRICAS UTILIZADAS. CAPACIDAD DE REINICIO.....	77
TABLA 9: MÉTRICAS UTILIZADAS. CAPACIDAD DE RESTAURACIÓN.....	77
TABLA 10: MÉTRICAS UTILIZADAS. CAPACIDAD DE SER AUDITADO.....	78
TABLA 11: MÉTRICAS UTILIZADAS. COMPLEJIDAD DE MODIFICACIÓN.....	78
TABLA 12: MÉTRICAS UTILIZADAS. LOCALIZACIÓN DEL IMPACTO DE LA MODIFICACIÓN.....	78
TABLA 13: MÉTRICAS UTILIZADAS. ADAPTABILIDAD AMBIENTAL DEL SISTEMA A NIVEL DE SOFTWARE.....	79
TABLA 14: MÉTRICAS UTILIZADAS. USO CONTINUADO DE LOS DATOS.....	79
TABLA 15: MÉTRICAS UTILIZADAS. CAÍDA DEL TIEMPO DE RESPUESTA.....	80
TABLA 16: MÉTRICAS UTILIZADAS. CAPACIDAD DE RESTAURACIÓN EN AMBIENTES TRANSACCIONALES.....	80
TABLA 17: RESULTADOS POR MÉTRICA OBTENIDOS EN LAS PRUEBAS.....	81

Introducción

El mundo de hoy se encuentra dominado por las tecnologías de la informática y sus ramas afines. Tanto es así que resulta difícil imaginar algún aspecto de nuestro entorno cotidiano, que no esté vinculado de alguna forma con las ciencias de la información, esto es, que no posea en alguno de sus ambientes o espacios propios, una aplicación informática que gestione total o parcialmente el quehacer interno.

Las empresas actuales no escapan de esta realidad y, es común encontrar dentro de un mismo ambiente empresarial, un sistema gestionando todo lo concerniente a capital humano, al tiempo que otro se encarga de la información financiera, cuentas, créditos, etcétera, por sólo citar un par de ejemplos. Con el paso del tiempo, tanto las empresas y por consiguiente, los sistemas que soportan y gestionan su información, evolucionan hacia estados de una complejidad creciente. La complejización de las organizaciones y sus sistemas subyacentes, ha traído como consecuencia, que en la mayoría de los casos, estos sistemas hayan experimentado una cierta especialización, especialización esta que, en conjunto con el enorme progreso de la computación empresarial en los últimos años y el advenimiento y asimilación de las tecnologías web, ha hecho que, para casi cualquier empresa, la integración de sus procesos y, en definitiva, de las aplicaciones que los soportan se haya convertido en un tema casi imperativo y de gran importancia para el desarrollo de software contemporáneo.

La integración de sistemas, no obstante los nuevos bríos adquiridos, no es para nada un tema nuevo. En realidad, ha estado teniendo lugar tal vez, desde los inicios mismos de la construcción de aplicaciones. Desafortunadamente es, aunque ampliamente estudiado, un tema sumamente complejo. Para mayor infortunio, tiende a crecer gradualmente en dificultad, dada las nuevas condiciones en las que se desenvuelve. En un esfuerzo de acercar al mundo a formas en las que resolver los desafíos que impone, muchos han sido los autores que, ya sea para una tecnología de desarrollo en particular, o desde una óptica independiente de una plataforma específica, han descrito posibles enfoques a soluciones de integración, que en mayor o menor medida, responden a las necesidades actuales.

Introducción

Siendo una problemática propia del desarrollo de software no dependiente de alguna plataforma particularmente, el desarrollador de software sobre tecnologías JEE¹, se plantea una interrogante vital: ¿cómo obtener una solución de integración, que sin ignorar los principios básicos de ser altamente cohesiva y poseer un acoplamiento relativamente bajo con el o los sistemas a integrar, sea ajustable al dominio de integración particular con el que se va a enfrentar? Si bien la respuesta a esta pregunta dista mucho de ser totalmente concluyente, existe hoy un número nada despreciable de opciones a elegir, lo cual eventualmente pone nuevamente al desarrollador frente a una disyuntiva: ¿cuál de estas maneras responde con mayor eficacia y eficiencia, acorde a las restricciones actuales que implican integrar sistemas de información? La presente investigación no es más que un modesto esfuerzo de despejar estas y otras incógnitas para el desarrollador sobre tecnologías JEE.

A las ya expuestas necesidad de comunicación entre sistemas para el intercambio ya sea de información o procesos de negocio comunes e inexistencia de claridad al definir una arquitectura de la capa de integración sobre sistemas de información JEE, se suman los hechos no menos importantes de que la información que se comparte tiene estructura y formatos variables de un sistema a otro, o lo que es más, existen conflictos semánticos sobre dicha información, esto es, dos sistemas pudieran estar hablando de “cuenta”, cada uno con acepciones o significados no necesariamente iguales; así como también, que las soluciones de integración que se implementan hoy, se encuentran generalmente tan enmarcadas a sus dominios de integración respectivos, que resultan muy difíciles de reutilizar en alguna medida.

Por tanto, esta investigación se plantea como **problema a resolver** la siguiente interrogante: ¿cómo construir la capa de integración para sistemas sobre tecnología JEE?, y para la misma se identificaron como:

Objeto de estudio

- ✓ La integración de sistemas de información.

¹ *Java Enterprise Edition. Constituye la especificación del lenguaje Java para la construcción de aplicaciones empresariales.*

Introducción

Campo de Acción

- ✓ La capa de integración de sistemas desarrollados sobre JEE.

Objetivo General

- ✓ Definir una arquitectura para la capa de integración de sistemas de información JEE.

Objetivos específicos

- ✓ Realizar la fundamentación de la investigación, a través de la creación del Marco Teórico.
- ✓ Definir los componentes de una arquitectura para la capa de integración, su funcionamiento, y las tecnologías asociadas.
- ✓ Brindar la posibilidad de usar varias formas de integración.
- ✓ Demostrar la validez de la arquitectura propuesta.

Idea a defender

- ✓ Una arquitectura de la capa de integración para sistemas de información JEE, facilitaría el proceso de integración de estos sistemas y proveería mayor grado de reutilización y flexibilidad a las soluciones de integración.

Tareas específicas de la investigación

- ✓ Caracterización y evaluación de los patrones y modelos de integración existentes.
- ✓ Evaluación de las principales formas o tecnologías de integración para sistemas de información sobre JEE.

Introducción

- ✓ Evaluación de arquitecturas existentes para otras capas de sistemas de información sobre JEE u otras tecnologías.
- ✓ Caracterización y evaluación de los componentes de software a utilizar.
- ✓ Definición de una propuesta de arquitectura para la construcción de la capa de integración para sistemas de información sobre JEE.
- ✓ Evaluación de la arquitectura propuesta para la capa de integración.

Resultados esperados

- ✓ Obtención de una arquitectura para la capa de integración de sistemas de información JEE.

Estructura del Documento

Capítulo 1: Se establecen las bases teóricas del tema abordado por esta investigación, enfocándolo desde las perspectivas del desarrollador y del usuario final. Se realiza un estudio de los niveles de integración de sistemas, la evolución histórica de sus estilos asociados, así como un breve análisis del estado del arte de las tecnologías disponibles a los efectos de integración sobre la plataforma seleccionada.

Capítulo 2: Describe desde el punto de vista técnico la arquitectura de la capa de integración para sistemas sobre la plataforma especificada. Se expone su disposición general en el sistema, así como sus componentes internos, de los cuales se ofrecen detalles acerca de su funcionamiento e implementación.

Capítulo 3: Se enfoca en los aspectos relacionados con la demostración de la validez de la arquitectura presentada en el presente documento.

Capítulo 1 – Fundamentación Teórica

Introducción

En aras de obtener una arquitectura para la capa de integración de sistemas sobre plataforma JEE, la presente investigación se sumerge en el mundo de la integración de aplicaciones empresariales.

Con el objetivo de lograr una vista panorámica general de la integración, así como de comprender por qué es necesaria, para posteriormente lograr un entendimiento de la arquitectura propuesta, se exponen en el presente capítulo los principales conceptos y definiciones asociados al tema, cuáles son sus desafíos actuales, así como los distintos niveles de integración.

Son presentados además los estilos de integración desarrollados hasta la fecha. Particular énfasis se hará con el estilo de *Mensajería*, debido a que constituye el estilo entorno al cual giran las principales tendencias de las soluciones de integración actuales, marco que será aprovechado para discutir los principales patrones de integración asociados a dicho estilo. Por último se hará un esbozo del estado del arte de las tecnologías que en la actualidad, permiten el desarrollo de soluciones de integración sobre la plataforma JEE.

Descripción general de la Integración de Sistemas de Información

¿Qué es la integración de sistemas?

Pese a la relativa popularidad que ha cobrado este término en numerosos contextos durante los últimos años, existen diversas ideas en torno a qué es, en fin, la integración de sistemas de información, ideas que, sin embargo, giran en torno a un consenso generalizado.

Capítulo 1 – Fundamentación Teórica

Algunas definiciones dadas por diferentes autores son:

- “... el compartimiento no restringido de información y procesos de negocio entre cualesquiera aplicaciones conectadas y fuentes de datos dentro de la empresa...” (1).
- “...implica integrar aplicaciones y fuentes de datos empresariales tal que puedan fácilmente compartir procesos de negocio e información...” (2).
- “La integración es la tarea de hacer trabajar juntas a aplicaciones dispares para producir un conjunto unificado de funcionalidad” (3)

Independientemente de que la naturaleza y objetivo de las definiciones anteriores es distinto, lo cual da lugar a que tengan algunas diferencias, dejan ver con claridad que la integración de sistemas involucra igualmente a información y procesos del negocio en función de proveer funcionalidad común. En aras de conceptualizar estos aspectos, puede decirse que la integración de sistemas de información **es la conformación de una unidad funcional lógica a partir del uso compartido de procesos de negocio e información correspondiente a dos o más sistemas concebidos para funcionar de manera independiente entre sí.**

Siendo realistas, la integración de sistemas de información es algo muy común con lo que se lidia casi a diario. Desde el punto de vista de las definiciones dadas y, de acuerdo con los autores consultados, operaciones como la conexión a un sistema de gestión de bases de datos, es, por definición, integración; o lo que es lo mismo, la capa de persistencia o acceso a datos de cualquier aplicación de hoy en día, es en sí misma, una capa de integración. La presente investigación considera la integración entre dos o más sistemas concebidos inicialmente como entes independientes entre sí.

Un concepto relacionado, y que en ocasiones es utilizado (de forma incorrecta) en sustitución de integración es el de interoperabilidad entre sistemas. Interoperabilidad (4) es la habilidad de dos o más sistemas o componentes para intercambiar información y para usar la información intercambiada. Desde esta óptica pudiera pensarse en interoperabilidad como una parte (una muy importante) dentro del concepto de integración, dado que está solamente dirigida al intercambio de datos .

Sistemas de Información

Uno de los conceptos más profundamente vinculados a la integración de sistemas es, precisamente el de sistema de información. Un sistema de información (2) es una aplicación o sistema empresarial que provee la estructura de la información para una empresa. Un sistema de información consiste en una o más aplicaciones desplegadas en un sistema empresarial, que brinda un conjunto de servicios a sus usuarios en diferentes niveles de abstracción, que pueden ser nivel de sistema, de datos, de función o al nivel de objetos o procesos del negocio. En un ambiente menos formal, cuando se habla de un sistema de información, se habla simplemente de una o más aplicaciones circunscritas a algún contexto empresarial, esto es, a algún marco en el que esta aplicación resulta de valor.

Gráficamente hablando, un sistema de información pudiera lucir así:

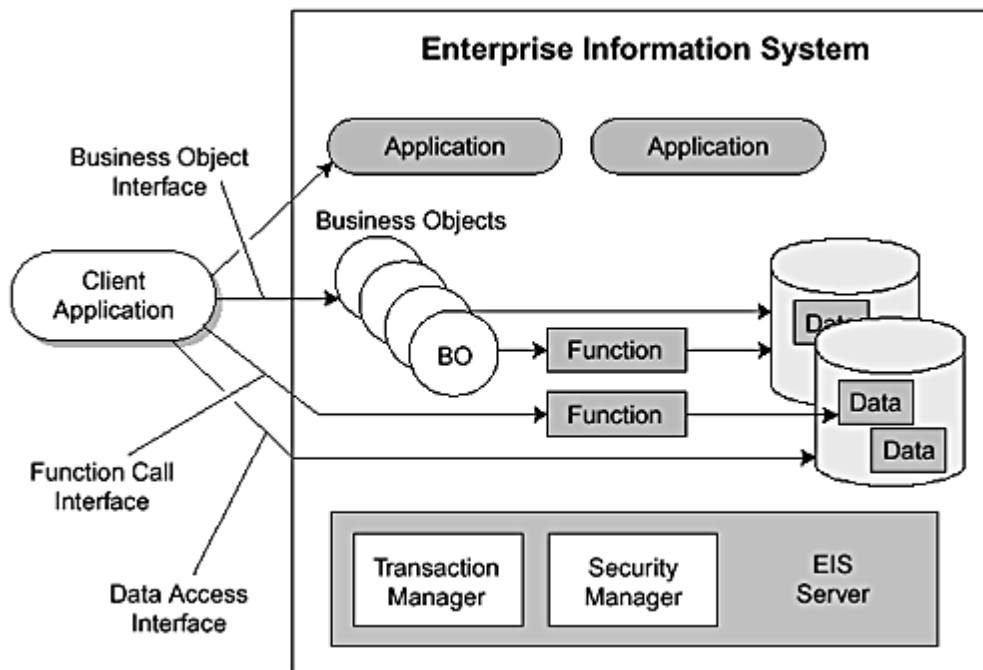


Ilustración 1: Sistema de Información. (J2EE Connector Architecture and Enterprise Application Integration)

En lo adelante, se usarán indistintamente los términos aplicación y sistema.

¿Es necesaria la integración?

La complejidad de organizaciones actuales, ha ocasionado que sus sistemas de información subyacentes, estén compuestos por más de una aplicación en la mayoría de los casos. Aunque en un estado de panacea, una sola aplicación capaz de gestionar todos los procesos sería la solución perfecta, lo cierto es que en estos momentos, esa solución está lejos de ser alcanzada. ¿Por qué?

Escribir aplicaciones empresariales es un proceso complejo para cualquier equipo de desarrollo. Si ya es difícil crear una aplicación para manejar sólo una de las ramas que interesan, *el desarrollo de una que soporte todas las ramas que de alguna manera sean de interés para una empresa resulta sumamente difícil en no pocos casos*. Por otro lado, dividir funciones de negocio generales como pudieran ser, la gestión bancaria y la del capital humano, en aplicaciones independientes, permite a los equipos de desarrollo, encontrar la mejor forma de administrar cada una de estas funciones. Esta separación, resulta práctica no solamente por lo anterior, sino porque también facilita la administración de requisitos, que la propia dinámica de las empresas los hace estar en constante evolución. Por tanto, la existencia de más de una aplicación dentro de un entorno empresarial, se convierte en una necesidad real hoy.

Sin embargo, dentro de esta dura realidad, para proveer no pocas funcionalidades al usuario, se hace necesario entonces involucrar a más de una aplicación. Por ejemplo, un cliente pudiera querer consultar sus datos personales y financieros. Desde el punto de vista de este usuario, la obtención de estos datos es una sola y monolítica transacción del negocio, pero desde la perspectiva del desarrollador, esta funcionalidad pudiera fácilmente expandirse entre dos o más aplicaciones. Con el objetivo de permitir que compartan ya sea información o procesos del negocio comunes, estas aplicaciones necesitan ser integradas.

Desafíos de la Integración de Sistemas

Como ya se ha dicho, la integración es un problema complejo. ¿Qué hace a algo tan necesario hoy ser tan problemático?

- Integrar aplicaciones usualmente requiere cambios en las políticas de las empresas. Esto viene dado por el hecho de que las aplicaciones están enfocadas en un área funcional particular, por lo que una solución efectiva de

Capítulo 1 – Fundamentación Teórica

integración, necesita no solamente establecer comunicación entre varios sistemas, sino también eventualmente, entre unidades de negocio distintas; además de que una vez integradas, cada unidad deja de controlar su antes única aplicación, puesto que esta pasa a ser parte del flujo general de información y servicios integrados.

- A causa de su extenso ámbito, la integración tiene serias implicaciones en los sistemas involucrados. Una vez integrada una aplicación, cualquier proceso importante del negocio mantenido por esta, se convierte en vital para todo el conjunto, por lo que una falla en este implicaría la falla de más de un sistema, lo cual pudiera tener consecuencias graves.
- Tal vez la restricción más importante es el limitado control que usualmente tienen los desarrolladores de una solución de integración sobre el resto de las aplicaciones participantes. Aunque muchas veces, el problema de la integración se podría resolver mucho más fácilmente implementando parte de la solución en los puntos finales dentro de cada uno de los sistemas involucrados, esta solución casi nunca está disponible debido a restricciones técnicas o políticas.
- Independientemente de la ampliamente expandida necesidad de soluciones de integración, solamente algunos pocos estándares se han establecido seriamente en este entorno. Aún cuando el advenimiento de XML², XSL³ y los servicios web ciertamente marcan el avance más significativo de las características estándar en las soluciones de integración, se han ido sentando las bases para una nueva fragmentación, a partir de la aparición de un sinnúmero de “extensiones” e “interpretaciones” de estos estándares.
- La existencia de estándares basados en XML solamente ayuda a solucionar una parte del problema. XML propone un dialecto común, pero aún escritas en el

² *Extensible Markup Language: es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación del SGML y permite definir la gramática de lenguajes específicos.*

³ *Extensible Stylesheet Language: es una familia de lenguajes basados en el estándar XML que permite describir cómo la información contenida en un documento XML cualquiera debe ser transformada o formateada para su presentación en un medio.*

Capítulo 1 – Fundamentación Teórica

mismo alfabeto, son diferentes las lenguas y dialectos los que se hablan. Esto es, las soluciones de integración tienen que lidiar con la diferencia semántica entre las aplicaciones, o sea, no necesariamente “hablan” de lo mismo todas las aplicaciones que manejan el término “cliente”, lo cual resulta una de las tareas más desafiantes, y por ende, una de las que mayor cantidad de esfuerzo requiere.

- En muchas soluciones de integración se requiere acceso transaccional garantizado, dado que muchos sistemas y por ende sus empresas, no pueden afrontar el hecho de que una aplicación externa comprometa la integridad de su información o cause inconsistencia dentro de esta.
- Seguridad dentro de las operaciones, es también un requerimiento importante. En pocas palabras, un sistema debería ser capaz de confiar en la información que posee, por lo que un acceso no autorizado podría ser sumamente costoso.
- Las soluciones de integración necesitan ser escalables, dado que las relaciones con otras aplicaciones de un mismo sistema tienden a crecer en número con el paso del tiempo.
- Si bien el simple desarrollo, si así puede llamarse, de una solución de integración, es en sí un gran problema, operarla y mantenerla puede ser aún más difícil, debido a la mezcla de tecnologías y a la naturaleza distribuida de la misma, lo cual ocasiona que el despliegue y monitorización de tal solución, sea mucho más que un camino espinoso.

En fin es evidente que, aún cuando una solución de integración pudiera ser crítica en más de un entorno, es más un gran dilema que un gran regalo, haciendo la tarea del desarrollador mucho más compleja y no simple, como pudiera pensarse.

Niveles de Integración

¿Qué son los niveles de integración?

A la hora de integrar, los desarrolladores deben entender a la vez no solamente la suma del contenido de la información y los procesos del negocio, sino también, cómo son automatizados estos procesos y cuál es la importancia real de cada uno de ellos, con el objetivo primario de seleccionar cuál o cuáles procesos necesitan ser integrados, y cuál es la información relevante para el proceso de integración. Los niveles de integración, definen o conceptualizan de alguna manera, las posibles dimensiones que este proceso pudiera abarcar, esto es, enmarcan en un ámbito definido la integración de dos o más aplicaciones.

Niveles de integración actuales

Independientemente de su plataforma y tecnologías de implementación, y de los dominios en que se enmarcan, todas las soluciones de integración pueden ser circunscritas, en uno o varios de los siguientes niveles de integración (1):

- Nivel de Datos
- Nivel de Interfaz de Aplicación
- Nivel de Método
- Nivel de Interfaz de Usuario

Gráficamente, esta sería su disposición:

Capítulo 1 – Fundamentación Teórica

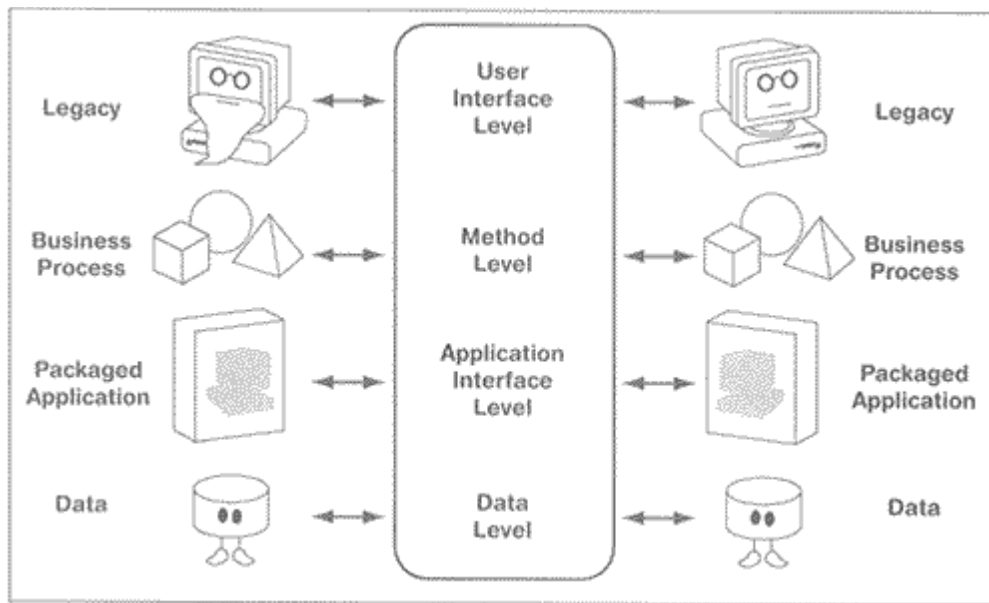


Ilustración 2: Niveles de Integración (Enterprise Application Integration)

No es objetivo de esta investigación, hacer una discusión detallada de las particularidades de cada nivel. A continuación se hará una breve descripción de cada uno de ellos.

- **Integración al nivel de datos**

Comprende el proceso, y por consiguiente, las técnicas y tecnologías, de mover información entre almacenes de datos. En simples palabras se puede describir como extraer información de una base de datos, posiblemente realizar algún tipo de procesamiento sobre ella, y actualizarla en otra base de datos. Pese a lo simple que este proceso luce a simple vista, el mismo pudiera potencialmente, involucrar un gran número de tablas, así como también realizar ciertas transformaciones y aplicar alguna lógica de negocio sobre la información en cuestión.

La gran ventaja de la integración a este nivel es su bajo costo. Debido a que las aplicaciones no son modificadas (no se introduce código debido a que todo sucede entre almacenes de datos), no es necesario incurrir en costos de cambios, pruebas y despliegues. Además, las tecnologías que proveen formas para este intercambio de información, así como su posible reformateo, son relativamente baratas en comparación con las que están disponibles en otros niveles de integración.

- **Integración al nivel de interfaz de aplicación**

Se refiere al uso de las interfaces de las aplicaciones para llevar a cabo el proceso de integración. Estas interfaces no son más que conjuntos de servicios expuestos por los desarrolladores dentro de las aplicaciones, usualmente en forma de API⁴, con el objetivo de brindar acceso a información o procesos del negocio. Este tipo de integración se encuentra limitado solamente por los servicios y funciones de las interfaces. Su funcionamiento se basa en el siguiente esquema: las interfaces se usan para acceder a datos y lógica de negocio, extraer información (la cual será dispuesta en algún formato comprensible por la aplicación de destino), procesarla y transmitirla.

A pesar de la existencia de variadas tecnologías para llevar a cabo la integración en este nivel, los *message brokers* han ganado una enorme aceptación.

- **Integración al nivel de método**

Es el intercambio de lógica de negocio existente dentro de las aplicaciones. Aunque los mecanismos para su realización son varios, existen dos soluciones básicas: o bien se crea un conjunto compartido de servidores de aplicaciones que existen dentro de un mismo servidor físico compartido, o bien se usan tecnologías de intercambio de métodos como los objetos distribuidos.

- **Integración al nivel de interfaz de usuario**

Esta es la más antigua de las formas de integración, pero aún continúa siendo necesaria en algunos ambientes, e incluso puede llegar a ser la única solución disponible. La idea aquí es usar la interfaz de usuario de las aplicaciones como punto de integración. Obviamente, esto ocasiona que los sistemas involucrados necesiten interactuar con otras interfaces de usuario de forma similar a como lo haría un usuario, lo cual presupone no pocas tareas de cierta complejidad. Felizmente, muchos de los problemas que pueden aparecer ya han sido resueltos a lo largo de los años.

⁴ *Application Programming Interface*

Estilos de integración

¿Qué son los estilos de integración?

Los estilos de integración son, a grandes rasgos, las respuestas con las que cuenta hoy por hoy cualquier equipo de desarrollo cuando se plantea la siguiente pregunta: ¿cómo llevar a cabo la integración de sistemas? Más formalmente hablando, describen las técnicas básicas que constituyen el fundamento de cualquier solución de integración. De acuerdo con lo anterior, cada estilo de integración es entonces un patrón, aunque todos, en primer lugar están orientados a resolver el mismo problema. Existen una serie de criterios (3) que ayudan a determinar qué estilo usar en cada momento, los cuales son:

- *Acoplamiento de aplicaciones*: las aplicaciones integradas deberían minimizar las dependencias entre ellas, de forma tal que cada una pueda evolucionar sin afectar a las demás, es decir, se deben minimizar las suposiciones sobre el funcionamiento de los sistemas externos. Por tanto, las interfaces de integración deberían ser lo suficientemente específicas para implementar funcionalidad útil, pero a la vez lo suficientemente generales para permitir cambios en la implementación según sean requeridos.
- *Intromisión*: se debería minimizar la cantidad de código necesaria para llevar a cabo la integración y este debería tener el mínimo impacto posible en el resto del sistema.
- *Selección de tecnología*: debería existir un balance entre las tecnologías a utilizar y lo que el equipo de desarrollo necesita implementar por sí mismo. Un elevado número de tecnologías en el escenario, elevaría considerablemente la curva de aprendizaje, mientras que por el otro lado, construir la solución “desde cero”, podría llevar al equipo a “reinventar la rueda”.
- *Formato de los datos*: las aplicaciones integradas deben acordar el formato de la información que comparten o, alternativamente, usar algún “traductor” que medie entre formatos que persisten en ser distintos.

Capítulo 1 – Fundamentación Teórica

- *Oportunidad de los datos:* la integración debe minimizar el tiempo que transcurre entre la decisión de una aplicación de compartir cierta información con otro sistema y el momento en que aquel efectivamente recibe dicha información. En la medida en que compartir información se demore, aumenta la complejidad de las soluciones de integración en función de evitar que las aplicaciones pierdan sincronización.
- *Datos o funcionalidad:* la integración debe considerar el hecho de que las aplicaciones pueden no solamente desear compartir información, sino también lógica de negocio.
- *Comunicación remota:* los sistemas funcionan internamente de forma síncrona en la generalidad de los casos. Sin embargo, en el caso de la llamada a un procedimiento remoto, invocaciones asíncronas parecen mucho más factibles, dado que, en primer lugar, llamar a un procedimiento remoto puede ser muchas veces más lento que una función local y, en segundo lugar, la aplicación que invoca el procedimiento remoto no necesariamente desea esperar a que este termine para continuar su procesamiento.

Estilos de integración actuales

Como se ha expuesto, los estilos de integración son los diferentes enfoques para soluciones de integración con que cuenta hoy, cualquier desarrollador. Aunque diferentes autores proponen diferentes categorías, en sentido general dichos estilos son (3):

- Transferencia de Archivos
- Bases de Datos Compartidas
- Invocación de Procedimientos Remotos
- Mensajería

A continuación se realiza una discusión de cada estilo. Con toda intención serán abordados en el orden anterior, con el objetivo de hacer notar como cada uno constituye una evolución hacia un estado de mayor sofisticación y complejidad de los anteriores, en busca de solucionar las dificultades de estos y proveer soluciones más elegantes.

- **Transferencia de Archivos**

Los archivos son un mecanismo de almacenamiento universal, integrado nativamente a cualquier sistema operativo y disponible en cualquier lenguaje de programación empresarial. Por tanto constituyen la manera más simple en la que, de alguna forma, dos o más sistemas pueden integrarse.

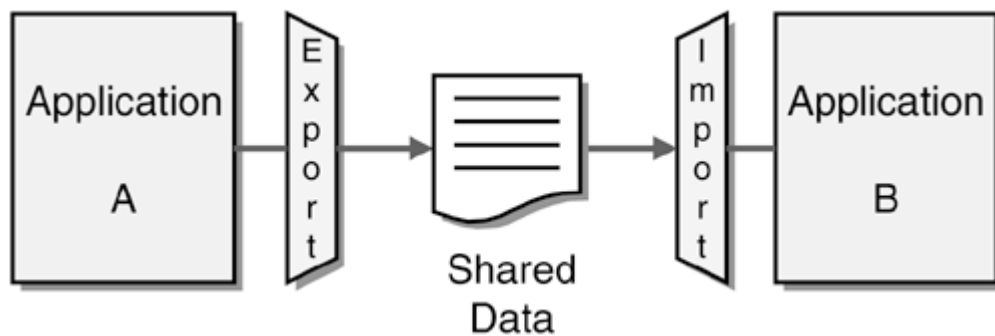


Ilustración 3: Transferencia de archivos gráficamente (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

Un punto importante aquí es el formato de los datos a utilizarse. Dado que rara vez, la salida que produce cada uno de los sistemas involucrados, es exactamente igual a la que espera el resto, es muy probable que se necesite no solamente leer desde el archivo compartido, sino también aplicar cierto procesamiento sobre lo leído. Como resultado, los formatos estándares han evolucionado gradualmente, siendo hoy los basados en XML los más utilizados.

Otro obstáculo lo constituye los momentos de producción y consumo del archivo, o en términos formales, la generación y transmisión de información y la recepción efectiva de la misma. El hecho de que crear archivos o procesarlos requieran un cierto esfuerzo, hace que el trabajo con ellos no sea muy frecuente, sino más bien dentro de un ciclo de operaciones más o menos definido dentro de la aplicación.

La gran ventaja de este estilo es que las aplicaciones que se integran no necesitan saber absolutamente nada del resto, o sea, se encuentran en un nivel de acoplamiento ideal. Dado que cada una de ellas, solamente hará uso de los archivos compartidos, esto permite hacer cambios internos dentro de cualquiera

Capítulo 1 – Fundamentación Teórica

de los sistemas sin afectar a los otros. Otro de sus puntos fuertes, es el hecho de que pocas o casi ninguna tecnología o herramientas extras son necesarias para usarlo, lo cual disminuye radicalmente la curva de aprendizaje de los desarrolladores. Esto, por supuesto que implica que los desarrolladores tendrán que hacer por sí solos casi todo el trabajo: se deben acordar las convenciones de nombres de archivos y sus directorios de ubicación, así como delimitar las responsabilidades sobre tareas críticas como son creación, eliminación, actualización de los archivos, así como cuál(es) sistema(s) deberá(n) conocer acerca de cuándo un archivo es obsoleto. Igualmente será responsabilidad de los desarrolladores definir algún mecanismo de bloqueo que impida que una aplicación lea un archivo a la vez que esté siendo modificado por otra.

Una desventaja clara del uso de archivo es que las actualizaciones tienden a ocurrir infrecuentemente, lo cual puede dar lugar a la pérdida de sincronización entre aplicaciones e inconsistencias que en ocasiones pueden ser muy difíciles de resolver. Una nueva debilidad aparece cuando se considera el hecho que no existe ninguna razón para un sistema en particular no produzca muchos archivos. El problema aquí es que garantizar que cada uno de ellos es leído y que ninguno se pierda, puede resultar demasiado costoso en términos de recursos del sistema y tener un impacto serio en la eficiencia de la aplicación, sobre todo si se necesita producir múltiples archivos rápidamente.

- **Bases de Datos Compartidas**

El uso de transferencia de archivos aunque permite compartir información tiene, como se explicó anteriormente, serios problemas con la oportunidad de los datos, así como que tampoco fuerza o restringe demasiado su formato. ¿Qué pasaría si las aplicaciones integradas descansaran sobre la misma base de datos, como se muestra en la siguiente ilustración?

Capítulo 1 – Fundamentación Teórica

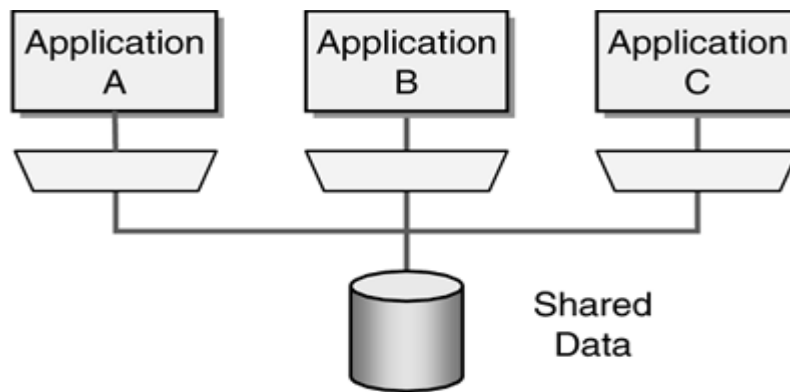


Ilustración 4: Aplicaciones usando una base de datos compartida (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

Inmediatamente se eliminan los problemas de consistencia, y desaparece la necesidad de crear un mecanismo de bloqueo dado que a estos efectos, existen sistemas de administración de transacciones que evitarán múltiples operaciones contraproducentes entre sí, ocurrir al mismo tiempo.

Una gran ventaja del uso de bases de datos compartidas es que hoy en día existen un buen número de sistemas de bases de datos relacionales basadas en SQL ampliamente difundidas, lo cual hace que desaparezca por completo el problema de los múltiples formatos. Sin embargo, esto hace que cobre mucha importancia la diferencia semántica de la información. Esta dificultad necesita ser resuelta incluso antes de que las aplicaciones sean desarrolladas y desplegadas, para evitar la recolección de muchos datos incompatibles. La superación de este obstáculo presupone una de las mayores desventajas de este estilo, necesidad de obtener un esquema relacional que sea compatible con todos los sistemas que se va a integrar. Obtener este esquema, en caso de ser posible, resulta sumamente complejo, y una vez obtenido, es fácil percatarse que resulta en muchos casos muy difícil de usar para los desarrolladores.

Aún cuando compartir la base de datos tiene ciertos beneficios, tener múltiples aplicaciones que intentan frecuentemente leer o modificar datos en ella, puede resultar en un “cuello de botella” y por ende, largas esperas para realizar las operaciones a causa de los bloqueos. Una posible alternativa al respecto, sería una base de datos distribuida, pero esto tiende a generar confusión acerca de dónde está realmente almacenada la información, además de que no resulta factible tener igualmente problemas de bloqueos en entornos distribuidos.

- **Invocación de Procedimientos Remotos**

Los estilos anteriores permiten, cada uno en su manera particular, el intercambio de datos entre aplicaciones, lo cual representa un punto importante en la integración, pero como ya ha sido explicado, compartir información no lo es todo en integración. Ciertos cambios y operaciones sobre los datos, pudieran extenderse a través de más de un sistema, pero estos detalles son solamente conocidos por la aplicación que implementa directamente estas funcionalidades. Tener, por tanto, a otra aplicación invocando directamente dichas acciones, esto es, implementándolas directamente como propias, requeriría en muchos casos que se conozca demasiado acerca del funcionamiento interno de las otras. Esto además de que incrementa notablemente el acoplamiento entre los sistemas, está en franca oposición al principio de encapsulación.

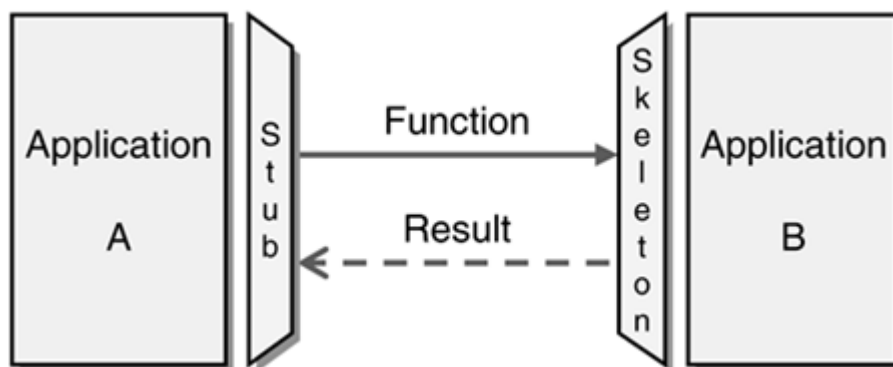


Ilustración 5: Comunicación mediante invocación de procedimientos remotos (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

La invocación de procedimientos remotos aplica el principio antes mencionado a la integración. En términos simples, si una aplicación necesita ejecutar alguna operación contenida dentro de otra, se lo solicita en la misma forma en la que se realizaría una llamada a una función local, aunque como se verá existen aquí diferencias fundamentales. Esto claramente permite a los sistemas conservar la integridad de sus datos. Además, este estilo provee una forma elegante de tratar las disonancias semánticas, puesto que como la información está encapsulada en métodos, se pudieran exponer varias interfaces para los mismos datos, aunque cada sistema tendría que negociar estas interfaces con sus vecinos.

Capítulo 1 – Fundamentación Teórica

La gran desventaja de la invocación remota es que difiere dramáticamente de una llamada a un procedimiento local en términos de eficiencia, comportamiento y confiabilidad. El no entendimiento de esta situación lleva en la mayoría de los casos al desarrollo de sistemas demasiado lentos y nada confiables. Además aunque este estilo reduce el nivel de acoplamiento que genera el compartir grandes estructuras de datos, deja ciertamente ciertas ataduras entre aplicaciones, puesto que las llamadas remotas, son insertadas en ciertas secuencias de acciones, lo cual dificulta en no pocos casos, realizar cambios en los sistemas independientemente del resto.

- **Mensajería**

El estilo de mensajería, y en particular la mensajería asíncrona, es una reacción sumamente pragmática a los problemas de los sistemas distribuidos.

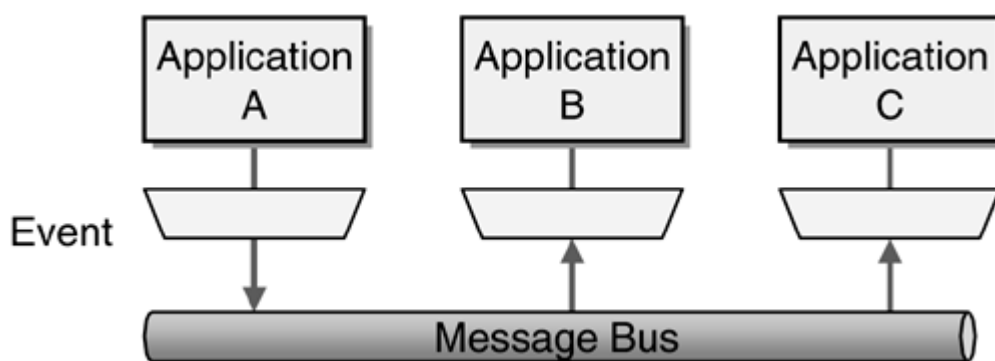


Ilustración 6: Comunicación mediante mensajería (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

Aquí la principal observación es que el envío de un mensaje no requiere que los sistemas integrados trabajen al mismo tiempo, además de que, desde el punto de vista del desarrollo, una mirada asíncrona a la comunicación entre sistemas, fuerza al desarrollador a reconocer que trabajar con sistemas remotos es mucho más lento, lo cual estimula la construcción de componentes con un alto grado de cohesión y bajo nivel de acoplamiento.

Los mensajes una vez enviados, pueden ser transformados, incluso si es necesario, sin que las aplicaciones involucradas tengan conocimiento de esto en la práctica. Esto permite diseñar soluciones de integración orientadas a manejar las disonancias semánticas en lugar de tratar de evadirlas, lo cual se traduce en

Capítulo 1 – Fundamentación Teórica

una mayor facilidad para trabajar con sistemas con modelos conceptuales diferentes. Este bajo nivel de acoplamiento permite entonces difundir un mensaje a muchos receptores, o enviarlos a uno en particular, entre otras topologías posibles; así como separa en general las decisiones de integración de las propias del funcionamiento interno de las aplicaciones.

La gran independencia entre aplicaciones que provee este estilo, ocasiona en la mayoría de los casos, que los desarrolladores tengan que escribir una cantidad de código extra, para unir ciertas cosas. Además, el diseño asíncrono dista mucho de ser el más común en la concepción de sistemas, lo cual pone en el escenario una gran cantidad de reglas y técnicas, incrementando esto a su vez la curva de aprendizaje. Realizar pruebas y corregir errores es también mucho más complejo en este nuevo ambiente.

Patrones de integración en Mensajería

¿Cómo resuelve en fin, la mensajería el problema de la integración? ¿Cómo integrar aplicaciones usando mensajería? Para responder estas y otras interrogantes, se impone una mirada más en detalle.

Mensajería nombra no solamente un estilo de integración, sino también a un patrón de integración, que es a su vez un sistema de sub – patrones que responden a las posibles preguntas que pueden surgir al abordar la integración desde la óptica del paso de mensajes. Este sistema se encuentra compuesto por los siguientes seis patrones fundamentales⁵ (3):

- ✓ *Message Channel*
- ✓ *Message*
- ✓ *Pipes and Filters*
- ✓ *Message Router*
- ✓ *Message Translator*
- ✓ *Message Endpoint*

⁵ Se utilizarán los nombres en inglés de los patrones para ser fieles a las fuentes consultadas

Capítulo 1 – Fundamentación Teórica

Estos patrones constituyen además, los elementos principales de un sistema de mensajería. A continuación se realiza una breve presentación de cada uno de ellos. Para una descripción completa y detallada, ver (3) a partir del capítulo 3.

¿Cómo se comunican las aplicaciones usando mensajería?

Cuando una aplicación necesita enviar información a otra, no la lanza al vacío de un sistema de mensajería y, de la misma manera, pensar en que la aplicación receptora recibe la información revisando al azar dicho sistema, dista mucho de la realidad. Dentro de la mensajería, la primera agrega su información a un particular canal de mensaje, y de igual forma, la última recibe dicha información desde un canal.

¿Qué es entonces, un canal de mensajes? Un canal de mensajes, es en pocas palabras, una tubería virtual que conecta un origen y un destino. Los canales son direcciones lógicas dentro de los sistemas de mensajería y su implementación real depende del propio sistema de mensajería que se use.

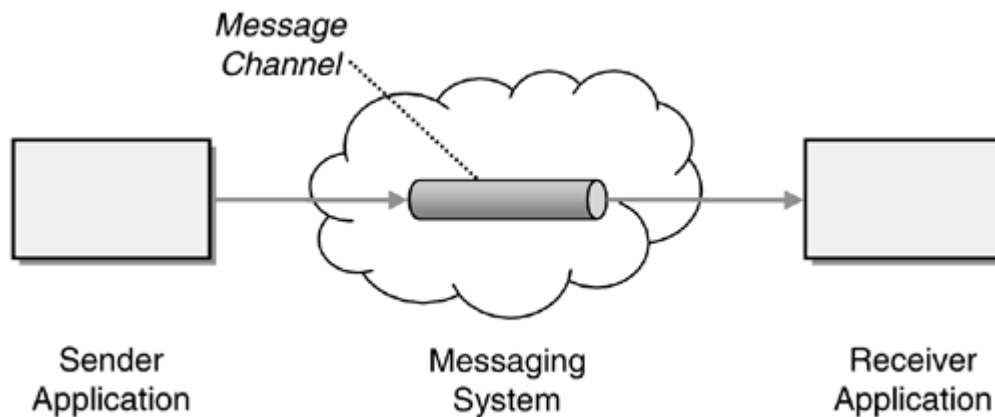


Ilustración 7: Las aplicaciones se comunican a través de canales de mensaje (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

¿Cómo pueden aplicaciones conectadas por un canal de mensaje intercambiar una pieza de información?

En vista de que los sistemas integrados en el caso más general residen en estaciones independientes, cada una de las unidades de información que se deseen transmitir necesita ser convertida a un flujo de bytes, y posteriormente

Capítulo 1 – Fundamentación Teórica

ser devuelta a su forma original. Muy útil sería una forma de encapsular estas unidades de información en un mecanismo apropiado para transmitirla a través de un canal de mensaje. Ese mecanismo es el mensaje, lo cual implica necesariamente que cualquier conjunto de datos que vaya a ser transmitido, tiene que ser convertido a una o más de estos entes.

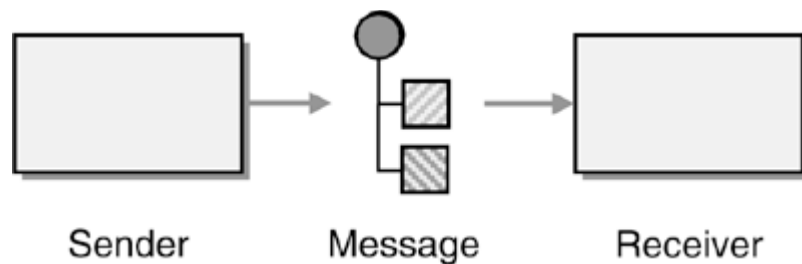


Ilustración 8: La información se empaqueta en mensajes para ser transmitida a través de los canales (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

Un mensaje consta básicamente de dos partes, un encabezado, con información relevante en general al sistema de mensajería, que describe el mensaje en términos de origen, destino, contenido, etcétera; y un cuerpo, que constituye el contenido real del mensaje.

Este patrón, rompe con la suposición, de que a semejanza de en las interacciones humanas, un mensaje solamente contendrá datos atómicos. Desde el punto de vista conceptual, va más allá y engloba dentro del mismo término de información, a datos simples, a una secuencia de estos que componen un elemento de mayor tamaño y, también, a la invocación de un procedimiento remoto. Esta versatilidad es la causa de la existencia en el mundo de la mensajería de más de un tipo de mensaje, cada uno orientado a una de las situaciones que pueden darse en el proceso de integración.

¿Cómo ejecutar procesamiento complejo sobre un mensaje sin afectar la independencia y la flexibilidad?

En el entorno más simplista, el sistema de mensajería entrega directamente un mensaje desde el origen hasta el destino. Sin embargo, en no pocos escenarios, es necesario aplicar ciertas transformaciones sobre el mensaje enviado, las cuales llegan a ser en varias ocasiones, muy complejas. Dentro del mundo de la mensajería, la solución a este problema la constituye el patrón *Pipes and Filters*.

Capítulo 1 – Fundamentación Teórica

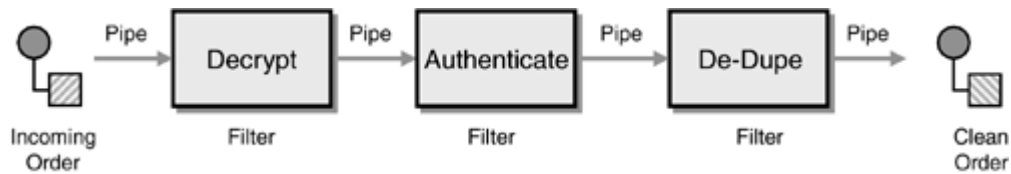


Ilustración 9: El patrón *Pipes and Filters* permite realizar transformaciones complejas sobre los mensajes (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

Lo que propone este patrón es la implementación en componentes (filtros) separados de cada una de las operaciones individuales que serán realizadas sobre el mensaje. Posteriormente, estos componentes son enlazados mediante canales (tuberías), en el orden deseado para la ejecución de las transformaciones. Las posibilidades de reutilización que brinda este diseño son elevadas, dado que cada componente es implementado de manera independiente sin tener conocimiento en lo absoluto de la existencia de los restantes, lo cual implica que será un componente altamente cohesivo capaz de ser transportado de un escenario a otro cuando sea requerido, sin afectar su funcionalidad.

Una de las desventajas potenciales que se presentan es, el número de canales que son necesarios. Puesto que cada canal necesita independientemente de su naturaleza, alguna cantidad de recursos del sistema para *buffers* internos, un número disparado de canales pudieran agotar el espacio de memoria disponible. Además, una cadena muy larga de filtros puede tener cierto impacto en el rendimiento del sistema, debido a la ligera sobrecarga que suponen las transformaciones dentro de cada filtro.

¿Qué hacer cuando es necesario pasar el mensaje hacia diferentes canales en función de algún conjunto de condiciones?

Aún cuando la noción de filtro más común es aquella que los presenta con un único punto de salida, esto dista mucho de ser una regla. De hecho, en más de un ambiente se hace necesario, tener alguna clase especial de filtro, que dependiendo de ciertas condiciones, entregue el mensaje a canales distintos. Estos filtros “especiales” responden al patrón *Message Router*, de manera que en estos puntos de la cadena de filtros, se colocaría un enrutador.

Capítulo 1 – Fundamentación Teórica

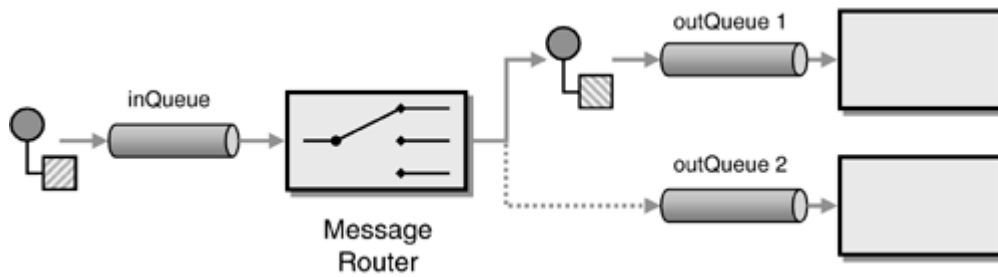


Ilustración 10: Un enrutador envía el mensaje a uno u otro canal de salida, dependiendo de su conjunto de condiciones (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

La ventaja clave con este patrón es que centra las decisiones de encaminamiento en un único lugar.

¿Cómo pueden comunicarse sistemas con diferentes formatos de datos?

Integrar sistemas no puede implicar de ninguna manera, que alguna de las aplicaciones involucradas pierda su “identidad”. Por tanto, enfrentar esta interrogante desde la perspectiva de la mensajería, descarta de antemano posibilidades como realizar cambios en el formato de la información dentro de algunas aplicaciones (lo cual es riesgoso, difícil y provocaría cambios en las funciones de negocio inherentes a dicha información), ajustar este formato para acercarlo al de otra aplicación (proceso que implicaría una violación del principio de bajo acoplamiento) o, realizar la transformación del formato en los puntos finales, o sea, en cada una de las aplicaciones (repercutiendo esto en modificaciones al código de los sistemas externos, cosa que casi nunca es posible). El patrón *Message Translator*, brinda una solución mucho más elegante y cohesiva.

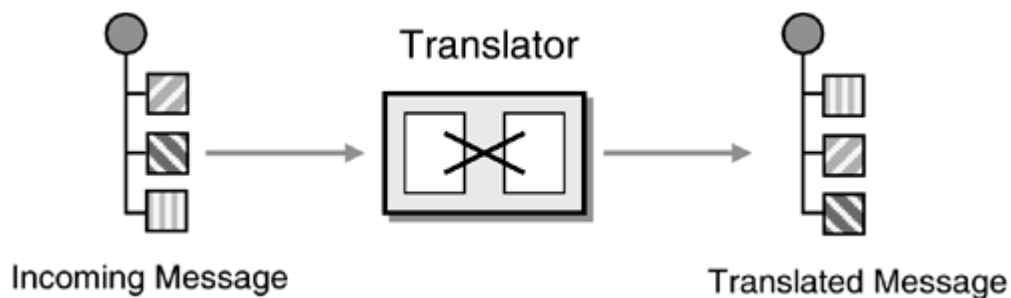


Ilustración 11: Un traductor transforma el formato de un mensaje (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

Un traductor no es más que otra clase de filtro “especial” encargado de modificar el formato de la información contenida en un mensaje. Representa el equivalente

Capítulo 1 – Fundamentación Teórica

del patrón *Adapter* presentado en (5) para mensajería y constituye la manera más efectiva de tratar las disonancias semánticas entre aplicaciones.

¿Cómo se conectan las aplicaciones a un canal de mensajes para el envío y recepción de los mismos?

Partiendo del hecho de que la aplicación y el sistema de mensajería son secciones de software independientes, resulta obvio pensar que debe existir alguna forma de conectarlos y hacerlos trabajar juntos. El patrón *Message Endpoint* resuelve esta necesidad de la manera tal vez más intuitiva y simple.

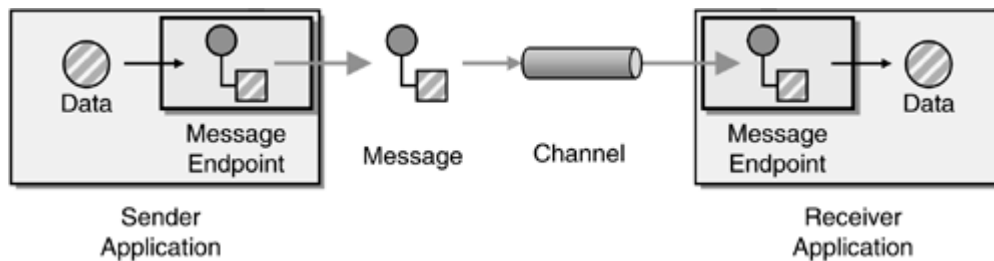


Ilustración 12: Un punto final de mensaje constituye una abstracción del sistema de mensajería (Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions)

La idea general es conectar la aplicación al sistema de mensajería a través de un cliente interno, cuya implementación dependerá de la propia aplicación en cuestión y del API del sistema de mensajería. Este cliente interno encapsula al sistema de mensajería del resto de la aplicación y a su vez particulariza un API de mensajería general, a los efectos de una aplicación y tarea específica. Debería además ser diseñado como un *gateway* de forma tal que oculte los detalles de la mensajería y su código asociado del resto de la aplicación, lo cual permitiría concentrar en él todo lo que concierne a la integración, de modo que un cambio en esta faceta del sistema debería influir solamente sobre este punto y no sobre ningún otro.

Principales tecnologías de integración sobre plataforma JEE

JEE ha demostrado ser de las más prometedoras y poderosas plataformas para la computación empresarial. Por lo tanto, es de esperarse que existan una amplia gama de tecnologías, principalmente enmarcadas en los estilos de *Invocación de*

Capítulo 1 – Fundamentación Teórica

Procedimientos Remotos y Mensajería cuya finalidad tribute en mayor o menor medida a la integración de sistemas de información, de manera que esta investigación resultaría minúscula en el intento de describirlas a todas. A continuación se ofrecen de manera sintetizada, descripciones de aquellas que son consideradas como fundamentales mundialmente.

✓ **JCA**⁶

JCA constituye la primera especificación que, para la plataforma Java, estuvo dirigida a proveer una arquitectura estándar para la integración de sistemas de información heterogéneos. Está basada en las tecnologías definidas y estandarizadas en JEE y, es a su vez, parte de dicha plataforma. Básicamente, JCA provee contenedores para aplicaciones clientes, componentes web basados en *Servlets* y *JSP*⁷ y componentes *EJB*⁸, los cuales brindan soporte de despliegue y ejecución, además de una vista federada de los servicios provistos por el servidor de aplicaciones subyacente para los mencionados componentes.

Usar JCA implica que o bien la aplicación será extendida (léase modificada) para soportar directamente dicha arquitectura, lo cual redundaría en una conectividad transparente a múltiples sistemas de información, o bien se proveerá un mecanismo estándar de conexión capaz de ser usado en cualquier servidor de aplicaciones que soporte dicha arquitectura, que en el lenguaje de la misma se conoce como adaptador de recursos.

JCA administra todo lo referente a conectividad, transacciones y seguridad, pero al no estar basada en patrones de integración, deja en manos del desarrollador lo que en materia de lógica de negocio aplicada al proceso de integración pudiera resultar necesario, es decir, no propone un mecanismo estándar para realizar operaciones como transformaciones de tipos de datos, reformato de la información, filtrado, entre otras. No obstante, liberar al desarrollador de las responsabilidades antes mencionadas, promueve un desarrollo más fácil y rápido de aplicaciones empresariales escalables, seguras y transaccionales que requieran conectividad con múltiples sistemas de información.

⁶ *JEE Connector Architecture*

⁷ *Java Server Pages*

⁸ *Enterprise Java Beans*

✓ **JMS⁹**

JMS es una API creada por *Sun Microsystems* para el uso de colas de mensajes, por lo que constituye el estándar sobre la plataforma JEE para la creación, envío, recepción y lectura de mensajes y es, por consiguiente, parte integral de dicha plataforma.

JMS propone dos modelos para la comunicación:

- **Punto a punto:** en este modelo, un remitente envía los mensajes a una cola en particular y el receptor lee los mensajes de la cola. En este caso, el remitente conoce el destino del mensaje y lo postea directamente a la cola del receptor. El modelo se caracteriza principalmente por:
 - Solo un consumidor recibe el mensaje.
 - Ni el receptor tiene necesariamente que estar corriendo en el momento en que el remitente envía el mensaje, ni el remitente necesita estar disponible en el momento en que el receptor consume el mensaje.
 - Cada mensaje procesado con éxito es reconocido por el consumidor.
- **Publicador/Suscriptor:** este modelo permite la publicación de mensajes para un tema de mensajes en particular y son los suscriptores los que deben registrar su interés en recibir mensajes de este tópico particular, de manera que, ni el sistema publicador, ni el o los sistemas suscriptos saben cada uno de la existencia de los otros. Las características principales del modelo son:
 - Múltiples consumidores o potencialmente ninguno recibirán el mensaje.
 - Existe dependencia de sincronización entre publicadores y suscriptores, que se encuentra dada por lo siguiente: el publicador tiene que crear los tópicos a los que se suscribirán los sistemas interesados, mientras que los suscriptores necesitan estar activos

⁹ *Java Messaging Service*

Capítulo 1 – Fundamentación Teórica

constantemente para recibir los mensajes, a menos que hayan establecido una suscripción duradera, caso en el cual, los mensajes publicados cuando algún suscriptor no se encuentra disponible, serán redistribuidos cuando este se reconecte.

Ambos modelos pueden ser síncronos o no, pero usualmente la variante asíncrona es preferida.

Para usar JMS, se debe contar con algún proveedor que gestione tanto las sesiones como las colas. Existe actualmente una amplia variedad de estos proveedores, entre los que destacan *Apache ActiveMQ*, *WebSphere MQ*, y *Oracle AQ*, por solo citar algunos. En (6) se encuentra disponible una matriz de comparación histórica de los proveedores de JMS desde 2005.

✓ **RMI¹⁰**

RMI es un mecanismo que brinda Java como plataforma para invocar un método de manera remota. Dicho de otra forma, es la propuesta de dicho entorno para la invocación de procesos remotos anteriormente discutida. RMI es parte del entorno estándar de ejecución de Java y proporciona un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en el mencionado lenguaje.

El principal rasgo distintivo de RMI es su facilidad de uso, dado que está específicamente diseñado para Java. Entre sus más notables características se encuentra paso de objetos por referencia, recolección de basura distribuida y paso de tipos arbitrarios.

A través de RMI, un programa Java puede exportar un objeto, con lo que dicho objeto estará accesible a través de la red y el programa permanece a la espera de peticiones en un puerto TCP. A partir de ese momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto.

La invocación se compone de los siguientes pasos:

- Encapsulado de los parámetros a través del mecanismo de serialización de Java.

¹⁰ *Remote Method Invocation*

Capítulo 1 – Fundamentación Teórica

- Invocación del método, proceso durante el cual el invocador se queda a la espera de una respuesta.
- Al terminar la ejecución, el servidor serializa el valor de retorno en caso de existir y lo envía al cliente.
- El código cliente recibe la respuesta y continúa como si la invocación hubiera sido local.

Spring Framework, uno de los *frameworks* más populares para el desarrollo de aplicaciones empresariales de los últimos tiempos brinda soporte entre otras, para las tecnologías anteriores, es decir, JCA, JMS y RMI. Como es de suponer, Spring no modifica la filosofía de trabajo con dichas tecnologías, pero sí representa una mejora sustancial en la manera de hacerlo, permitiendo al desarrollador explotar con mayor facilidad y en todas sus extensiones, las potencialidades de las mencionadas tecnologías. Para una descripción más detallada al respecto, ver (7), Parte IV.

✓ **SOAP**

SOAP (siglas en inglés de *Simple Object Access Protocol*) está diseñado básicamente para proveer un mecanismo simple y ligero de compartir información estructurada en un ambiente descentralizado y distribuido. Es esencialmente, un modelo para la codificación de datos en un formato XML estandarizado, para ser usado ya indistintamente en Mensajería o Llamadas de Procedimientos Remotos. Para la plataforma Java, los principales exponentes son:

JAX – RPC/JAX – WS

JAX – RPC (siglas en inglés de *Java API for XML – based Remote Procedure Call*) es, como su nombre lo indica, un API que permite la invocación remota mediante XML. Aunque en principio está diseñada para los mismos objetivos que

Capítulo 1 – Fundamentación Teórica

Java IDL (tecnología que permite invocación remota basada en CORBA¹¹) y RMI, la diferencia respecto a estos es que JAX – RPC está orientada a los servicios web.

JAX – RPC continúa con el modelo de las tecnologías RPC, en el cual el mapeo en ambos sentidos de los tipos de datos, así como el empaquetado y desempaquetado, ocurren de forma transparente al usuario, lo que significa que un cliente no necesita trabajar con XML o hacer ningún mapeo directamente

JAX – RPC hace sencillo el uso de servicios web, así como también facilita el desarrollo de servicios web. Un servicio web basado en RPC básicamente es una colección de procedimientos que pueden ser llamados por un cliente remoto desde Internet. El propio servicio es una aplicación servidora desarrollada sobre un contenedor del lado del servidor que implementa procedimientos que están disponibles para llamadas de clientes.

Un servicio web necesita estar disponible para clientes potenciales, lo que se puede hacer, por ejemplo, describiéndose a sí mismo usando el WSDL¹². Un consumidor (un cliente Web) puede entonces buscar el documento WSDL para acceder al servicio. Un consumidor usando el lenguaje Java usa JAX – RPC para enviar su petición al servicio, que podría o no, estar desarrollado en una plataforma Java.

Aunque JAX – RPC implementa una llamada a procedimiento remoto como una petición y una respuesta de mensaje SOAP, un usuario JAX – RPC está aislado de este nivel de detalle. Por eso, bajo la superficie, JAX – RPC realmente es una forma especializada de mensajería SOAP.

¹¹ *Common Object Request Broker Architecture: es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.*

¹² *Web Services Description Language: formato XML que se utiliza para describir servicios Web.*

Capítulo 1 – Fundamentación Teórica

JAX – RPC es la mejor elección para aplicaciones que desean evitar la complejidad de la mensajería SOAP y donde la comunicación usando el modelo RPC es una buena idea.

Con el advenimiento de su versión 2.0 JAX – RPC fue renombrada a JAX – WS para reflejar el movimiento desde el estilo RPC hacia el de los servicios web. JAX – WS se ha convertido en una tecnología fundamental no solamente para el desarrollo de SOAP, sino también para el de aquellos servicios web que usan herramientas de transferencia de estado representacional.

Apache Axis 2

Axis 2 es un completo rediseño y reescritura del ampliamente usado motor de servicios web Apache Axis. No solamente brinda la posibilidad de agregar interfaces de servicios web a las aplicaciones, sino que puede funcionar igualmente como un servidor de aplicaciones. Entre sus principales características se encuentran:

- *Velocidad*: usa su propio modelo de objetos y su propia API para el tratamiento de XML, alcanzándose velocidades significativamente mayores que las versiones anteriores.
- *Bajo consumo de memoria*.
- *Modelo de objeto ligero propio*: viene con su propio modelo de objeto ligero para el procesamiento de mensajes, el cual es extensible, optimizado para el rendimiento y simplificado para los desarrolladores.
- *Despliegue en caliente*: está equipado con la capacidad de desplegar nuevos servicios web y controladores cuando el sistema está corriendo, es decir, sin necesidad de reiniciar la aplicación.
- *Servicios web asíncronos*: soporta tanto servicios web como invocaciones a estos, de manera asíncrona, usando clientes y transportes libres de bloqueos.
- *Soporte para patrones de intercambio de mensajes*: soporta los patrones de intercambio de mensajes definidos en WSDL 2.0.

Capítulo 1 – Fundamentación Teórica

- *Flexibilidad*: permite al desarrollador introducir extensiones para el tratamiento personalizado de las operaciones llevadas a cabo por el motor.
- *Estabilidad*: define un conjunto de interfaces que cambian relativamente lento en comparación con el resto de los componentes.
- *Despliegue orientado a componentes*: se pueden definir fácilmente redes reutilizables de controladores para patrones comunes en el procesamiento dentro de la aplicación.
- *Marco de Transporte*: brinda una sencilla y limpia abstracción para integrar y usar transportes, al tiempo que el núcleo es independiente del transporte utilizado.
- *Soporte de WSDL*: soporta las versiones 1.1 y 2.0 de WSDL, lo que le permite fácilmente talones para acceder a servicios remotos, y también para exportar automáticamente las descripciones de los servicios desplegados.
- *Agregados*: han sido incluidas varias especificaciones de servicios webs como *WSS4J*¹³ (8), para la seguridad y *Sandeha* (9) para la mensajería confiable, entre otros.

✓ **Spring WS**

Spring WS es un producto de la comunidad *Spring* centrado en la creación de servicios web manejados por documentos. Tiene como objetivo facilitar el desarrollo de servicios SOAP de primero – el – contrato, permitiendo la creación de servicios web flexibles usando una de las tantas formas que existen para manipular contenidos XML. Está basado en *Spring* mismo, lo cual implica que conceptos de *Spring* como Inyección de Dependencias pueden ser usados como parte integral de los servicios web implementados.

¹³ *Web Services Security for Java: conjunto de tecnologías implementadas para proveer los estándares primarios de seguridad en servicios web para Java.*

✓ ***Spring Integration***

Spring Integration es un nuevo miembro en el portafolio de *Spring*, es decir, es un nuevo sub – proyecto de *Spring* motivado por las mismas metas y principios que el resto de los existentes sub – proyectos. Extiende el modelo de programación de *Spring* hacia el dominio de la mensajería y se erige sobre el soporte que este brinda a la integración empresarial, para proveer un nivel de abstracción mayor. Permite la creación de arquitecturas dirigidas por mensajes donde la inversión del control aplica a situaciones en tiempo de ejecución tales como cuando cierta lógica de negocio debe ser ejecutada y hacia donde debe ser enviada la respuesta. Soporta enrutamiento y transformación de mensajes lo cual implica que diferentes transportes y formatos de datos pueden ser integrados sin impacto alguno en la funcionalidad de la solución. En otras palabras, los problemas de la mensajería y la integración son manejados por el *framework*, por lo que los componentes del negocio pueden ser aislados de dicha infraestructura y los desarrolladores son liberados de complejas responsabilidades de integración.

El diseño de *Spring Integration* está inspirado por el reconocimiento de una fuerte afinidad entre patrones comunes dentro del modelo de *Spring* y bien conocidos patrones de integración (3), algunos de los cuales fueron previamente objetos de una breve discusión.

✓ ***Apache Camel***

Camel es un *framework* de integración dirigido a hacer los proyectos de integración productivos, focalizado en simplificar la integración. El proyecto *Camel* comienza a principios de 2007 y pese a su relativa juventud, es ya un proyecto de código abierto maduro, disponible bajo la liberal licencia Apache 2, que cuenta con una muy fuerte comunidad.

Camel puede ser visto como un motor de enrutamiento, o más precisamente un motor constructor de enrutamiento. Permite definir reglas propias de ruteo, decidir de cuáles fuentes aceptar mensajes y determinar cómo procesar y enviar esos mensajes hacia otros destinos. *Camel* usa un lenguaje de integración que permite definir complejas reglas de ruteo, de manera similar a como son definidos los procesos de negocio.

Capítulo 1 – Fundamentación Teórica

Uno de los principios fundamentales de *Camel* es que no asume nada al respecto de los tipos de datos que se necesitan procesar. Este es un punto importante, en tanto brinda al desarrollador la posibilidad de integrar cualquier tipo de sistema sin la necesidad de convertir los datos a un formato canónico, es decir, universalmente válido.

Camel ofrece abstracciones de alto nivel que permiten interactuar con varios sistemas usando el mismo API, sin importar el protocolo o los tipos de datos que dichos sistemas están usando. Los componentes en *Camel* proveen implementaciones específicas del API para diferentes protocolos y tipos de datos, contándose con soporte para más de 80 de estos. Su arquitectura extensible y modular permite implementar y transparentemente conectar el soporte para protocolos propios, sean propietarios o no. Estas opciones arquitectónicas eliminan la necesidad de ciertas conversiones, lo cual hace a *Camel* no solamente más rápido, sino también muy ligero. Otros proyectos de código abierto como *Apache ServiceMix* y *Apache ActiveMQ*, ya están usando *Camel* para llevar a cabo la integración empresarial.

De manera similar a *Spring Integration*, *Camel* es una implementación directa de los patrones de integración descritos en (3).

Conclusiones Parciales

Este capítulo muestra el trasfondo teórico que a lo largo de los años, se ha desarrollado en torno al problema de la integración de sistemas de información. Por cuanto *Mensajería*, constituye la evolución de los restantes estilos de integración, dicho estilo constituye el fundamento de la presente propuesta de arquitectura, lo cual necesariamente implica que la tecnología a utilizar deberá enfocar la integración desde la óptica del paso de mensajes. Entre las aquí expuestas, aparecen como fuertes candidatas, *Apache Camel* y *Spring Integration*, por los siguientes motivos:

- ✓ Son proyectos de código abierto.
- ✓ Están fundamentadas en bien conocidos patrones de integración.
- ✓ Permiten acceso transaccional basado en el soporte para transacciones del *Spring Framework*. Para más detalles al respecto ver (7) Capítulo 9.
- ✓ Gestionan seguridad a varios niveles.
- ✓ Brindan posibilidades de administración y monitoreo similares o superiores a las de las restantes tecnologías.

Sin embargo, se decide usar *Apache Camel* como tecnología subyacente a la propuesta de arquitectura. Las razones por las que *Spring Integration* se descarta son expuestas a continuación:

- ✓ *Spring Integration* puede ser utilizado solamente en aplicaciones basadas en *Spring Framework*, lo cual fuerza el desarrollo de la aplicación sobre dicho *framework*, hecho este que, aún considerando las ventajas que esto pudiera representar, constituye una fuerte imposición para la arquitectura de todo el sistema que no tiene por qué ser aceptada en todos los casos. *Apache Camel* en cambio, no asume absolutamente nada acerca de cómo está construido el resto del sistema, es decir, no fuerza la arquitectura del resto de la aplicación en ninguna dirección en particular.
- ✓ Otro de los efectos colaterales de usar *Spring Integration* es que obliga a un manejo declarativo, es decir, vía XML, en el mejor de los casos, de toda la

Capítulo 1 – Fundamentación Teórica

configuración de los aspectos de la integración, hecho este que excluye del escenario a todas aquellas aplicaciones en las que simplemente no se desee usar configuración XML alguna. *Apache Camel* en cambio, provee varios DSL¹⁴ que no solamente brindan la posibilidad de describir programáticamente todo lo que a la integración concierne, sino que en no pocos casos, dicha descripción resulta mucho más simple y comprensible.

- ✓ Las posibilidades en cuanto al manejo de errores con *Spring Integration* se encuentran limitadas al uso de un canal predeterminado al cual se enviarán mensajes cuyo contenido será el error ocurrido. *Apache Camel*, por su parte, define una mejor y más amplia estructura para la gestión de errores. Dicha estructura cuenta con diferentes *handlers* para los posibles errores, políticas de reenvío de mensajes, y un control tan específico como se desee para el manejo o descarte de las excepciones.
- ✓ *Spring Integration* gestiona seguridad solamente mediante chequeos basados en roles sobre el envío y recepción de mensajes sobre los canales, a través del uso de *Spring Security* (10), mientras que *Apache Camel*, propone un esquema de gestión de seguridad a varios niveles que resulta mucho más completo. Estos niveles son ruta, contenido, punto final y configuración.

Resumiendo, y de acuerdo con lo aquí expuesto, *Apache Camel* resulta un marco de trabajo capaz de ofrecer niveles de flexibilidad y robustez superiores, así como manejar los aspectos más relevantes a considerar en cualquier solución de integración.

¹⁴ *Domain Specific Language: es un lenguaje de programación o de especificación dedicado al dominio particular de un problema, una técnica particular de representación de problemas y/o una técnica de solución particular.*

Capítulo 2 – Descripción de la arquitectura

Introducción

El presente capítulo presenta y describe una propuesta de arquitectura para la capa de integración de aplicaciones sobre tecnologías JEE, la cual constituye una especificación técnica acerca de cómo implementar la mencionada sección en el marco seleccionado, cuyo diseño está orientado a lograr un aislamiento de las decisiones y asuntos referentes a la integración, del resto del sistema y, resolverlos con la mayor elegancia posible.

Como se ha mencionado anteriormente, la propuesta se enfoca en el estilo de *Mensajería*, es decir, en enfrentar el problema de la integración desde la óptica del paso de mensajes, por lo que, como es de esperar, se basa en patrones de integración (3); y utiliza *Apache Camel* como tecnología subyacente, por los motivos que en el capítulo fueron detallados. Siendo consecuente con esto último, la presente descripción no solamente introduce los componentes de la propuesta, sino que también muestra de manera general o a través de ejemplos, cómo implementarlos con el mencionado *framework*. Respecto a esto último, pese a que en las descripciones se utilizarán indistintamente los DSL de Java o de Spring, cabe señalar que ambos son lenguajes equivalentes y por tanto es posible usar cualquiera de ellos dentro de un sistema JEE.

Descripción general

En la gran generalidad de los casos, la arquitectura de un sistema de información sobre tecnologías JEE responde a un diseño multicapa, donde figuran al menos tres niveles bien definidos:

- ✓ Acceso a datos.
- ✓ Lógica de negocio.

Capítulo 2 – Descripción de la arquitectura

✓ Interfaz de usuario.

La disposición de dichos niveles dentro de la aplicación sigue un marcado orden jerárquico, de acuerdo con el cual las capas superiores se comunican solamente con las inmediatas inferiores. Dicho de otra forma, cada una de estas capas exporta ciertos servicios que constituyen su cara visible al resto del sistema y son consumidos de una forma u otra por la capa inmediatamente superior en la jerarquía. La siguiente figura muestra de manera gráfica lo anterior.

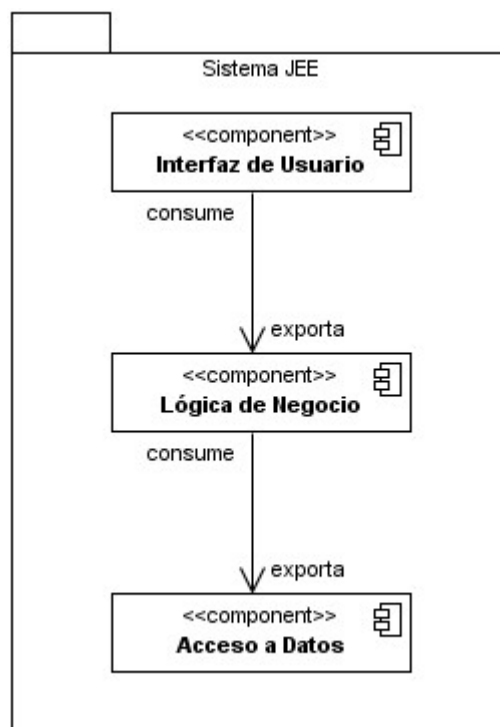


Ilustración 13: Disposición de las capas de un sistema JEE y flujo de la información entre ellas.

¿Dónde juega su papel dentro de este esquema, la capa de integración? La capa de integración queda dispuesta como una capa transversal al resto del sistema y su punto de contacto con el mismo, será a través de la capa de lógica de negocio.

La disposición transversal obedece al hecho de que el estado funcional del sistema no debe ser dependiente de la capa de integración, es decir, la aplicación en su totalidad, o al menos aquellas funcionalidades no involucradas en el proceso de integración, deberían seguir disponibles si el sistema se aísla; así como a que la capa de integración constituirá la interfaz de comunicación entre el sistema JEE y los sistemas externos, independientemente de la naturaleza de estos y del canal de comunicación a utilizar. La siguiente imagen muestra esta ubicación para la capa de integración.

Capítulo 2 – Descripción de la arquitectura

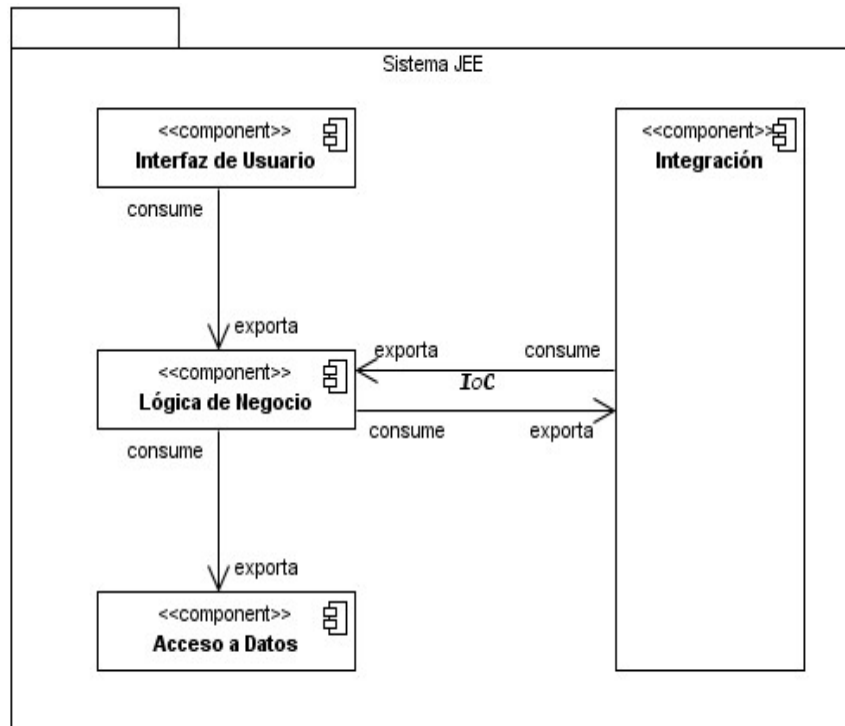


Ilustración 14: Disposición de la capa de integración e interacción con el resto del sistema.

Por otra parte, la capa de lógica de negocio es el punto de contacto, dado que esta constituye en términos generales, el núcleo de cualquier aplicación, al menos en lo que a funcionalidad concierne, y por ende resulta natural que sean exclusivamente los servicios que exporta dicha capa los utilizados para el acceso tanto a las funciones de negocio como a los datos. Esta comunicación entre las capas de lógica de negocio e integración cuenta con dos puntos a destacar: primero, a diferencia de los anteriormente mostrados, aquí el flujo es en ambos sentidos, es decir, es perfectamente posible que en ambos lados se consuman servicios exportados por el otro; segundo, es a través de la inyección de dependencias, lo que significa que su fundamento básico es la delegación de responsabilidades.

Un elemento fundamental en la presente arquitectura de integración y que debe ser entendido es que la comunicación entre el sistema JEE y los sistemas externos se llevará a cabo a través de un *message broker*, esto quiere decir, que existirá algún servidor de mensajes intermedio entre los sistemas que se integran, el cual será el encargado de gestionar las colas de mensajes y servirá de almacén temporal de datos cuando el sistema destino de algún mensaje no se encuentre disponible. Como es de esperarse, existen escenarios donde resulta impracticable el uso de tal mediador,

Capítulo 2 – Descripción de la arquitectura

como son por el ejemplo, el acceso a servidores FTP¹⁵ o a servicios web, pero se propone su inclusión en todos aquellos ambientes que así lo permitan, debido a que esto reduce drásticamente el nivel de acoplamiento entre los sistemas integrados, siendo responsabilidad de cada sistema involucrado administrar el acceso al servidor de mensajes. En sentido general el principio regente es el de evitar el uso de comunicaciones directas, puesto que dicho enfoque carece de escalabilidad al necesitar en el peor de los casos, un número de conexiones mucho mayor. Queda claro que en aquellos casos traumáticos, la comunicación directa es el único camino posible. Las siguientes figuras ilustran ambos enfoques para la arquitectura de integración.

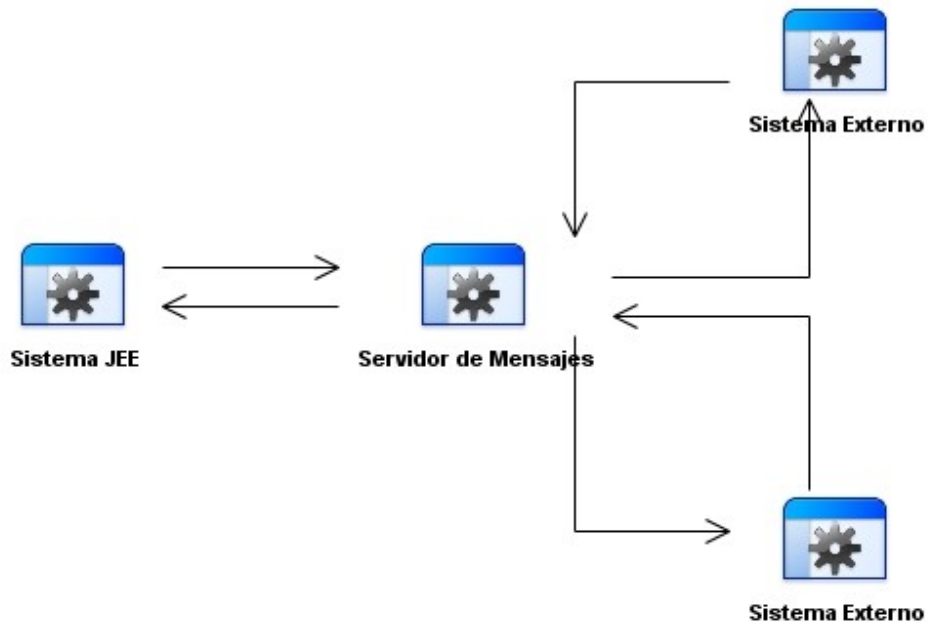


Ilustración 15: Integración mediante un *message broker*.

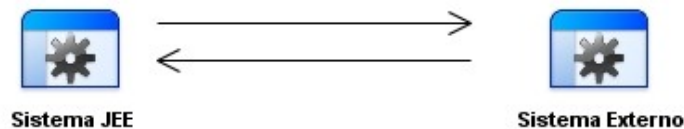


Ilustración 16: Integración mediante comunicación directa.

¹⁵ *File Transfer Protocol: protocolo de red para la transferencia de archivos entre sistemas conectados a una red TCP (Transmission Control Protocol), basado en la arquitectura cliente-servidor.*

Capítulo 2 – Descripción de la arquitectura

Estructura Interna

Desde una perspectiva global la estructura interna de la arquitectura puede dividirse en cuatro subcapas: lógica de integración, seguridad, transacciones y gestión de errores. El siguiente diagrama muestra la interacción de las mismas. A continuación se son introducidas cada una de estas subcapas con los elementos que la componen.

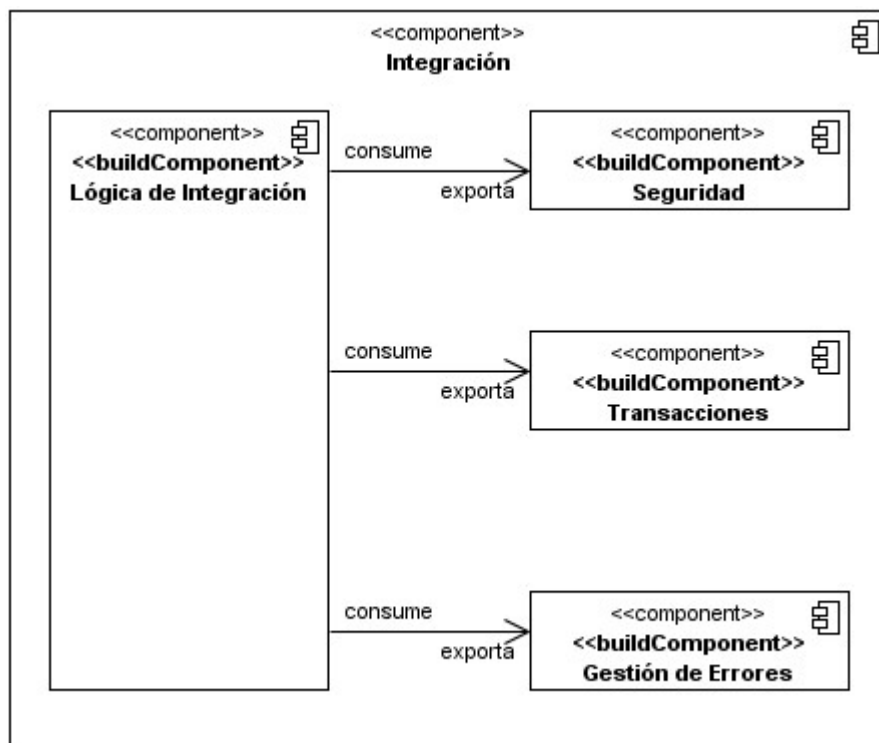


Ilustración 17: Estructura interna de la capa de integración.

Lógica de Integración

Esta subcapa es la de mayor peso respecto a las tareas de integración, dado que en ella residen las configuraciones correspondientes a la conectividad con los sistemas externos, así como todo el cúmulo de funcionalidad extra que el escenario donde se encuentre el sistema JEE requiera. De manera general, agrupa a sus elementos en procesadores o filtros, rutas y componentes o puntos finales. El siguiente diagrama muestra cuál es la relación que existe entre los mismos. Los subsiguientes acápites están destinados a sus respectivas descripciones.

Capítulo 2 – Descripción de la arquitectura

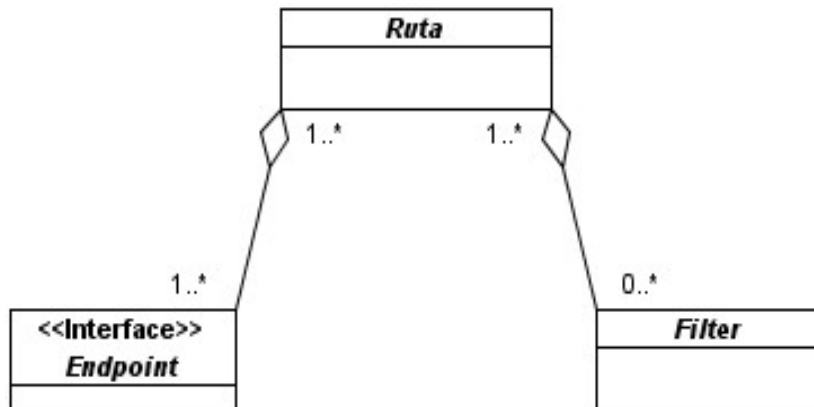


Ilustración 18: Elementos internos de la subcapa de lógica de integración.

Endpoint

Un *endpoint* es un punto de extensión que agrega conectividad con otros sistemas, es decir, representa un canal de comunicación entre el sistema JEE y algún sistema externo que pudiera no estar sobre dicha plataforma. Constituye una implementación del patrón *Message Endpoint* (3) al que ya se ha hecho referencia.

Dentro del lenguaje de *Camel* un punto final recibe el nombre de *Component* y está concebido en forma de *gateway*, lo que significa que esconde los detalles internos del establecimiento y administración de las conexiones, liberando al desarrollador de esta tarea y permitiéndole concentrarse en los aspectos funcionales de la integración.

¿Cómo definir un *endpoint* con *Camel*? Dependiendo del canal de comunicación a utilizar, esta definición pudiera incluir por ejemplo, la URL¹⁶ de un *message broker*, o la dirección del WSDL y demás parámetros necesarios para la invocación de un servicio web. Por ejemplo, si se necesitara conexión con un servidor de mensajes ubicado en 10.32.12.14, escuchando por el puerto 61610, la configuración sería la siguiente:

Usando Spring DSL (basado en XML):

```
<bean id="jms" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL" value="tcp://10.32.12.14:61610"/>
</bean>
```

¹⁶ *Uniform Resource Locator*: es una secuencia de caracteres, de acuerdo a un formato modélico y estándar, que se usa para nombrar recursos en Internet para su localización o identificación.

Capítulo 2 – Descripción de la arquitectura

`</bean>`

Usando Java DSL (programáticamente):

```
CamelContext context = new DefaultCamelContext();  
ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://10.32.12.14:61610");  
context.addComponent("jms", JmsComponent.jmsComponentAutoAcknowledge(factory));
```

Luego de realizada esta acción, el *endpoint* declarado podrá ser usado en rutas y su URI¹⁷ deberá comenzar por “*jms*”, es decir, “*from(jms:...)...*” o “*...to(jms:...)...*”.

En aras de no perder el foco de esta descripción, no es objetivo ejemplificar la declaración de los más de 80 componentes soportados por *Apache Camel* (11), sin embargo cabe destacar que en ellos, están presentes si no todas, al menos muchas de las principales vías de comunicación entre sistemas.

Ruta

Como su nombre lo indica, una ruta no es más que una especificación de origen y destino(s) para determinados mensajes. Dentro de la capa de integración, las rutas configuran y determinan el flujo de los mensajes entre los sistemas. Dicho en otras palabras, este elemento es el encargado de enlazar coherentemente dos o más *endpoints* y, eventualmente, las operaciones intermedias requeridas por el escenario de integración, en función de que entre los mismos la comunicación sea efectiva, lo cual implica necesariamente que todo el intercambio entre un sistema JEE y cualquier otro sistema externo, será especificado en uno o varios de estos elementos.

La existencia de este elemento dentro de la capa de integración, está justificada en gran medida por las no pocas ventajas que esto implica:

- ✓ Permite decidir dinámicamente, qué sistema externo invocar.
- ✓ Provee una muy flexible manera de agregar procesamiento extra en las tareas de integración.
- ✓ Permite al resto del sistema JEE ser desarrollado independientemente de los sistemas externos con los cuales será integrado y viceversa.

¹⁷ *Uniform Resource Identifier*: es una cadena de caracteres corta que identifica inequívocamente un recurso.

Capítulo 2 – Descripción de la arquitectura

No obstante la gran funcionalidad que provee, definir una ruta en *Camel* es una tarea relativamente sencilla. Puesto que esta es, a grandes rasgos un puente entre origen y destino, en principio solamente son necesarios estos elementos. Considérese el siguiente ejemplo, escrito en Java DSL:

```
from("file:data/inbox").to("jms:queue:order")
```

Como es posible imaginarse, con la ruta anterior se indica el envío de los archivos ubicados en el directorio “data/inbox” hacia una cola de mensajes. Un elemento a destacar es que la conversión del tipo de datos necesaria para que los datos viajen de un punto a otro, es realizada automáticamente por el *framework*, lo cual, como será expuesto más adelante, no significa que no existan ciertas transformaciones que necesiten ser explícitamente realizadas por el desarrollador. Otro punto importante es que las rutas funcionan bajo un esquema reactivo, es decir, un mensaje nuevo dentro del extremo definido por el elemento *from* causará que este sea enviado automáticamente hacia el(los) destino(s) especificado(s).

Existen dos posibles configuraciones básicas para cada sección de ruta que responden a las posibles situaciones a enfrentar: en la primera los *endpoints* están dispuestos secuencialmente en forma de cadena, implicando esto que los mensajes resultantes de cada componente, serán pasados como entrada al siguiente en la cadena. Dicha configuración luce en general de la manera siguiente:

```
from(...).to(...).to(...)...
```

La otra posibilidad es que desde un mismo nodo, se envíen mensajes a varios lugares, es decir, se envíe una copia del mismo mensaje a varios puntos. En ese caso, la ruta tiene un aspecto similar a este:

```
from("...").multicast().to("...", "...", ...)
```

Opcionalmente, se puede indicar si es necesario que el envío se realice en paralelo de la siguiente forma:

```
...multicast().parallelProcessing().to...
```

La gama de posibles topologías para una ruta que estas opciones permiten es sumamente amplia. No obstante el carácter introductorio de la presente descripción,

Capítulo 2 – Descripción de la arquitectura

es válido señalar que *Camel* brinda en el marco de estas dos variantes básicas, otras posibilidades de configuración con las cuales reporta gran flexibilidad a la capa de integración.

Filter

Los elementos *Filter* son los encargados, dentro de la arquitectura de integración de agregar básicamente todas las operaciones intermedias, extras o colaterales, que sean requeridas por un escenario de integración. A *grosso modo* pudiera decirse que un *Filter* es un elemento de software, usualmente una clase, altamente cohesiva, que realiza alguna operación en particular sobre un mensaje o parte de este. Constituyen por tanto, una de las piezas que mayor nivel de portabilidad aporta dentro de la arquitectura, puesto que al ser elementos implementados de forma independiente al resto, pueden ser trasladados de un sistema JEE a otro donde se requiere, sin prácticamente ningún cambio. Estos elementos manejan o dan soporte, en posiciones intermedias entre los *endpoints* a:

- ✓ Patrones de integración
- ✓ Enrutamiento
- ✓ Transformación
- ✓ Mediación
- ✓ Validación
- ✓ Intercepción

No es objetivo de esta investigación, hacer referencia a todos los patrones de integración soportados por *Apache Camel* (12), sino más bien, dar una panorámica general de aquellos que son considerados fundamentales. Antes de entrar a exponer cómo manejar cada uno de estos asuntos con *Camel*, es importante que se conozca cuál es el modelo de mensaje de dicho *framework* (13). A continuación se expone brevemente en qué consiste dicho modelo.

En *Apache Camel* existen dos abstracciones para modelar mensajes, las clases *org.apache.camel.Message* y *org.apache.camel.Exchange*. *Message* constituye una implementación directa del patrón (3) que lleva el mismo nombre y es por tanto la entidad fundamental que contiene la información transportada. A nivel estructural muestra tres elementos fundamentales: *headers*, que son valores asociados al mensaje como pueden ser la identidad del que lo envía, información de autenticación,

Capítulo 2 – Descripción de la arquitectura

etc.; *attachments*, valores opcionales típicamente usados por algunos componentes en particular; *body*, el cual constituye el contenido principal del mensaje. Cada uno de los valores almacenados son objetos de tipo *java.lang.Object* lo cual naturalmente implica que *Camel* no fuerza ningún tipo de dato en particular, dejando en manos del desarrollador la responsabilidad de que el receptor entienda el contenido íntegro del mensaje. Por su parte *Exchange*, es una abstracción del intercambio de mensajes, y constituye un contenedor de mensajes durante el enrutamiento. Expone un mensaje de entrada, un mensaje de salida como respuesta, un identificador unívoco, entre otros valores de interés. Las figuras siguientes ilustran lo anteriormente expuesto.



Ilustración 19: Un objeto *Message* de Apache Camel (Camel In Action).

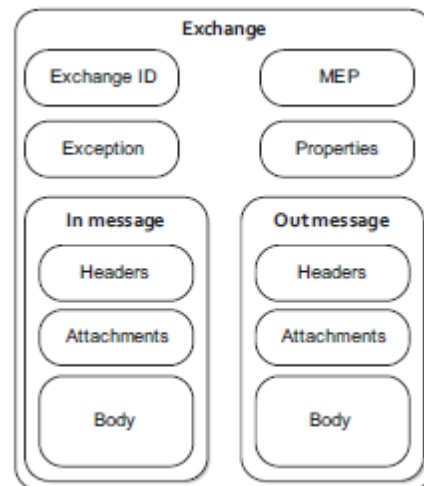


Ilustración 20: Un objeto *Exchange* de Apache Camel (Camel In Action).

- **Validación, Enrutamiento e Intercepción**

Bajo *Camel*, la validación de un mensaje consiste simplemente en dejarlo pasar o no en un punto de la ruta, de acuerdo con determinadas condiciones. Esta operación es la que más se asemeja a la idea que intuitivamente aparece cuando se habla de “filtro”. Para ello, *Camel* utiliza el método *filter* en el punto de la ruta donde desee realizar el filtrado. Por ejemplo:

```
from("jms:xmlOrders").filter(xpath("/order[not(@test)]"))...
```

Capítulo 2 – Descripción de la arquitectura

Dentro del método *filter* se utilizará siempre una expresión acorde con el lenguaje de expresiones definido por el *framework* (13). En el ejemplo anterior se utilizó la expresión *xpath* la cual es útil para crear condiciones para mensajes basados en formato XML.

Una generalización a la operación de filtrado, la constituye el enrutamiento, la cual es utilizada cuando a partir de determinadas condiciones, el mensaje es enviado a un punto u otro de la ruta. Esto, como se puede constatar, es exactamente la esencia del patrón *Message Router* (3) y es lo que *Camel* propone. Una de las vías posibles es basada en el contenido del mensaje, de la siguiente manera:

```
from("...").choice()  
  .when(predicate)  
  .to("...")  
  .when(predicate)  
  .to("...")  
  .otherwise()  
  .to("...");
```

donde *predicate* es o bien una expresión válida en el lenguaje de expresiones anteriormente mencionado, o bien es una instancia de la siguiente *interface*:

```
public interface Predicate {  
    boolean matches(Exchange exchange);  
}
```

Aquí el método *matches* contendrá la lógica necesaria para determinar si se cumple o no la condición requerida.

Otra de las variantes posibles es a través de enrutadores dinámicos, es decir, enrutadores cuyas implementaciones recaen en clases y que, basados en determinadas condiciones determinarán el punto al que será enviado el mensaje. Esto como se verá difiere ligeramente del ejemplo anterior:

```
public class DynamicRouterBean {  
    public String route(String body, @Header(Exchange.SLIP_ENDPOINT) String  
    previous) {  
        return whereToGo(body, previous);  
    }  
    private String whereToGo(String body, String previous) {
```

Capítulo 2 – Descripción de la arquitectura

```
        if (previous == null) {
            return "mock://a";
        } else if ("mock://a".equals(previous)) {
            return "language://simple:Bye ${body}";
        } else {
            return null;
        }
    }
}
```

Luego este enrutador puede ser utilizado en una forma similar a la siguiente basada en Spring DSL.

```
<bean id="myDynamicRouter" class="package.DynamicRouterBean"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="..." />
    <dynamicRouter>
      <method ref="myDynamicRouter" method="route"/>
    </dynamicRouter>
    <to uri="..." />
  </route>
</camelContext>
```

Para una descripción de los elementos no abordados aquí acerca de enrutamiento, ver (13), Capítulo 8, Secciones 4, 5 y 6, así como (12), sección *Message Routing*.

La intercepción por su parte, tiene dos objetivos primordiales, realizar alguna operación intermedia que no caiga dentro de las otras categorías mencionadas, o bien, desviar completamente el curso de algún mensaje. Esto último adquiere un valor notable en ambientes de prueba. En cuanto al primer objetivo, es solamente una cuestión de ubicar en el lugar de la ruta deseado una instancia de la *interface Processor* definida por *Camel*:

```
public interface Processor{
    void process(Exchange exchange) throws Exception;
}
```

He aquí un ejemplo de cómo hacerlo:

Capítulo 2 – Descripción de la arquitectura

```
public class DownloadLogger implements Processor {
    public void process(Exchange exchange) throws Exception {
        ...
    }
}

<bean id="downloadLogger" class="package.DownloadLogger"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="..." />
        <process ref="downloadLogger" />
        <to uri="..." />
    </route>
</camelContext>
```

- **Mediación, Transformación y Enriquecimiento**

Una pregunta que pudiera plantearse ahora es ¿qué hacer cuando la operación intermedia a ejecutar es uno de los servicios expuestos en la capa de lógica de negocio del sistema? La respuesta inmediata pudiera ser, inyectar dicho servicio dentro una instancia de la mencionada *interface Processor*, pero como casi siempre sucede dentro de los patrones de integración, existe una vía mucho más limpia de resolver esta situación. Esta vía responde al patrón *Service Activator* (3), y bajo *Camel*, puede representarse esquemáticamente de la siguiente forma:

Asúmase que existe una clase del negocio cuyo nombre es *HelloBean* cuyos servicios expuestos son requeridos dentro de una ruta.

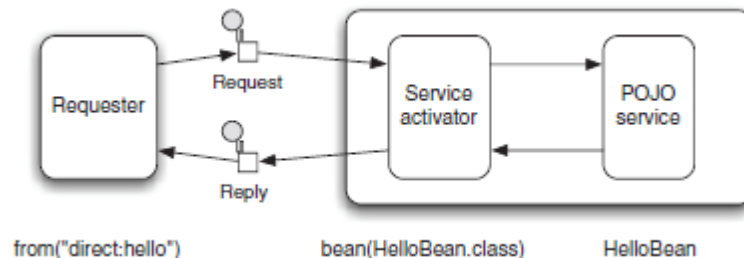


Ilustración 21: Patrón *Service Activator* en *Camel* (Camel in Action)

Capítulo 2 – Descripción de la arquitectura

Como puede suponerse, la implementación del patrón recae dentro del nodo *bean* de la definición de la ruta. Por tanto en general la invocación a un servicio del negocio se llevará a efecto de esta forma:

```
<bean id="helloBean" class="package.HelloBean"/>
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="..." />
    <bean ref="helloBean" method="hello"/>
  </route>
</camelContext>
```

Con lo cual se asegura la invocación del método *hello* del *bean helloBean* que existe dentro del contexto de la aplicación. Un detalle importante es que el contenido del mensaje es pasado automáticamente como parámetro del método y por tanto, es responsabilidad del que envía el mensaje que el mismo pueda ser interpretado por el receptor. Además, la salida del método se convertirá en el contenido del mensaje que recibirá el próximo nodo de la ruta. Similarmente, mediante Java DSL, es posible realizar la mediación anterior. Véase el siguiente ejemplo:

```
from("...").beanRef("helloBean", "hello");
```

Las posibilidades brindadas por el *framework* utilizado son numerosas en este sentido. Más detalles al respecto pueden ser encontrados en (13), Capítulo 4.

Uno de los grandes problemas con los que debe lidiar cualquier arquitectura de integración como ya ha sido expuesto, lo constituyen las disonancias en la información intercambiada entre los sistemas, las cuales en general o bien son en cuanto al formato de la información, o bien en cuanto al tipo de datos. Una de las posibles soluciones a este dilema es conceptualizar el cambio del formato de la información como una operación intermedia, o en otras palabras, asignar a uno de los elementos ya presentados (*bean* o *Processor*) la responsabilidad de realizar dicha transformación, con lo que se estará haciendo clara alusión al patrón *Message Translator* (3) discutido en acápites anteriores. Otra posibilidad es el uso del elemento *transform*, he aquí un ejemplo:

Capítulo 2 – Descripción de la arquitectura

```
from("...").transform(expression).to("...");
```

donde *expression* es o bien una expresión válida en el lenguaje de expresiones de *Camel* o es una instancia de la *interface Expression* definida por el *framework* como sigue:

```
public interface Expression{  
    <T> T evaluate(Exchange exchange, Class<T> type);  
}
```

cuya responsabilidad es la de devolver cuál será el nuevo contenido del mensaje a transformar.

Otras formas más sofisticadas de cambio del formato de un mensaje, así como problemas de esta índole aquí aludidos como la sustitución del tipo de dato del contenido y algunos no mencionados como son agregar información extra al mensaje en ciertos puntos de la ruta, dividir mensajes cuyo contenido es muy extenso en partes más pequeñas y luego unificar esas partes, balance de carga entre los *endpoints* de una ruta, y muchos más que hacen un gran total de alrededor de sesenta, encuentran una elegante respuesta dentro de *Apache Camel*, basada en uno o varios patrones de integración (3). Claramente esta investigación resultaría minúscula en un intento de abordar incluso de manera resumida, cada una de estas potencialidades, pese a que todas están confinadas a la subcapa de lógica de integración. El lector puede encontrar explicaciones mucho más profundas y abarcadoras al respecto en (13).

Gestión de Errores

Hasta este punto la sección funcional de la arquitectura de integración ha sido presentada, asumiendo que todas las operaciones funcionan correctamente, es decir, obviando la ocurrencia de errores. Sin embargo, como ya se ha expuesto en puntos anteriores, la naturaleza distribuida de la integración de sistemas, convierte a cualquiera de sus escenarios en ambientes rara vez libres de errores. En vista de la inminente presencia de fallas, se considera pertinente la inclusión de una subcapa, encargada de la gestión de las mismas, en aras de dotar de robustez a la capa de

Capítulo 2 – Descripción de la arquitectura

integración, es decir, permitir que el sistema en la medida de lo posible, pueda recuperarse de errores inesperados.

El mecanismo de gestión de errores cuenta con cinco *handlers* posibles a utilizar:

- ✓ **DefaultErrorHandler**: manejador de excepciones por defecto que se encuentra habilitado automáticamente, en caso de que ningún otro sea utilizado.
- ✓ **DeadLetterChannel**: manejador que implementa el patrón *Dead Letter Channel* (3), el cual básicamente establece que si un mensaje no puede ser procesado, debe ser movido hacia algún canal especial para un análisis ulterior.
- ✓ **TransactionErrorHandler**: extiende al primero y ofrece sus mismas funcionalidades para rutas transaccionales.
- ✓ **LogErrorHandler**: simplemente almacenará un log con el error ocurrido.
- ✓ **NoErrorHandler**: no realizará ninguna acción en particular.

los cuales serán invocados solamente si la falla ocurrida es una instancia de *java.lang.Exception*. En el caso contrario, se necesita especificar que cualquier otro problema sea igualmente tratado especificando lo siguiente:

```
getContext().setHandleFault(true);
```

con lo cual se estará haciendo a nivel global o

```
from("...").handleFault()  
.beanRef("...", "...")  
...
```

lo que tendrá un efecto similar, pero confinado solamente a la ruta en cuestión.

En sentido general, la estrategia seguida es la capturar el error lanzado y enviarlo al inicio de la ruta, hacia algún destino en particular (*LogErrorHandler* y *DeadLetterChannel*), o intentar el reenvío del mensaje. Para establecer el *handler* a utilizar, se deberá usar el elemento *errorHandler* en una forma similar a la que a continuación se muestra:

```
errorHandler(defaultErrorHandler()  
.maximumRedeliveries(2)  
.redeliveryDelay(1000)  
.retryAttemptedLogLevel(LoggingLevel.WARN));
```

Capítulo 2 – Descripción de la arquitectura

donde cada uno de los métodos posteriores a la especificación del *handler* corresponde a una de las posibles catorce opciones de reenvío del mensaje disponibles en (13), Tabla 5.3. Similarmente a como ocurre con el caso anterior, estos elementos pueden ser declarados para el uso exclusivo de una ruta, o para todo el contexto. Véanse los siguientes ejemplos mostrando una y otra variante.

```
errorHandler(defaultErrorHandler()  
.maximumRedeliveries(2)  
.redeliveryDelay(1000)  
.retryAttemptedLogLevel(LoggingLevel.WARN));  
  
from("...")  
.beanRef("...", "...")  
...  
  
from("...")  
errorHandler(deadLetterChannel("log:DLC")  
.maximumRedeliveries(5).retryAttemptedLogLevel(LoggingLevel.INFO)  
.redeliveryDelay(250).backOffMultiplier(2))  
...
```

La primera de las rutas definidas utilizará el *handler* global, mientras que la segunda, hará uso de aquel que fue definido en su interior.

En función de proveer un control con mayor nivel de especificidad pueden ser usados los elementos *onException*. Estos elementos, no solamente redefinen todos los valores del *handler*, sino que también cuentan con capacidades de reenvío y pueden constituir rutas alternativas cuando hay fallas. Lo anterior se puede corroborar en el fragmento siguiente:

```
onException(IOException.class).maximumRedeliveries(3)  
handled(true)  
.to("...");
```

Es importante hacer notar que el elemento *handled(true)* debe existir dado que por defecto las políticas definidas por *onException* no son aplicadas. Si se desea ignorar el error, simplemente debe usarse *continued* de manera similar a la siguiente:

```
onException(ValidationException.class)  
continued(true);
```

Capítulo 2 – Descripción de la arquitectura

Por último, también es posible monitorear errores dentro de una sección particular de la ruta, usando los elementos *doTry ... doCatch ... doFinally* de modo similar al que se hace con el bloque *try... catch... finally...*:

```
from("...")  
.doTry()  
.process(...)  
.to("...")  
.process(...);  
.doCatch(JmsException.class)  
.process(...)  
.end();
```

Como puede apreciarse la gestión de errores revierte una gran importancia para el *framework* utilizado como tecnología subyacente, lo cual dota a la arquitectura de integración en general de una poderosa herramienta en el hostil ambiente distribuido en el que debe existir. El Capítulo 5 de (13), puede ser consultado por el lector para conocer detalles no abordados por la presente descripción.

Transacciones

Si bien la subcapa de lógica de integración aquí descrita es la encargada de gestionar el carácter funcional de la arquitectura de integración, en la mayoría de los casos, como ya se ha discutido, esto no es suficiente. En la generalidad de los ambientes en los que coexisten más de un sistema de información, se necesita que las operaciones que involucren a más de uno de estos sistemas, tengan un carácter transaccional. No es objetivo de la presente investigación hacer una discusión detallada acerca del concepto de transacción, pero en aras de ganar en claridad, se puede considerar una transacción como un conjunto de operaciones que conforman un todo atómico, consistente, aislado y perdurable, el cual se considera terminado satisfactoriamente si y solo si cada una de estas operaciones han tenido una culminación efectiva. Enfocar las operaciones antes mencionadas desde una óptica transaccional se hace necesario, en tanto un sistema no puede permitir que su información quede en un estado inconsistente a causa de las acciones de aplicaciones externas.

Capítulo 2 – Descripción de la arquitectura

La subcapa de transacciones es, en pocas palabras, un elemento colocado dentro de la arquitectura de integración con el objetivo de añadir rasgos transaccionales a las tareas de integración. Constituye una derivación de la sección anterior, en tanto en sentido general, las transacciones están estrechamente vinculadas a la gestión de errores. Con *Apache Camel* el manejo de transacciones se divide en dos grandes categorías:

- ✓ Mediante el uso de *Transaction Managers*.
- ✓ A través del concepto de *UnitOfWork*.

Transaction Managers

La gestión transaccional vía *Transaction Managers* está basado en *Spring Framework* (7) Capítulo 9. Su uso se encuentra limitado sólo a aquellos componentes que lo soportan, mayoritariamente los relacionados con JMS y JDBC¹⁸ y es esencialmente gestión declarativa.

¿Cómo agregar carácter transaccional de acuerdo con esta perspectiva?

La respuesta varía dependiendo si se requieren transacciones locales, es decir, restringidas al marco de solo un componente dentro de la ruta, o globales, donde se quiere involucrar a todos los componentes de la ruta, asumiendo que todos la soporten. En el primer caso, es necesario declarar primeramente que el *endpoint* se comporta de forma transaccional, así como asignarle un *Transaction Manager*, por ejemplo:

```
<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">  
  <property name="transacted" value="true"/>  
  <property name="transactionManager" ref="txManager"/>  
</bean>  
  
<bean id="txManager"  
class="org.springframework.jms.connection.JmsTransactionManager">  
  <property name="connectionFactory" ref="jmsConnectionFactory"/>  
</bean>
```

¹⁸ *Java Database Connectivity*: es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.

Capítulo 2 – Descripción de la arquitectura

Un elemento a destacar es que existen algunas propiedades más a configurar para el comportamiento transaccional como son tiempos de espera y estrategias de *rollback*. Más detalles al respecto están disponibles en la cita anterior.

Una vez hechas estas configuraciones, se debe declarar como transaccional la ruta donde se utilice el *endpoint* definido, exactamente después del elemento *from* de la ruta, como en el siguiente ejemplo:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">  
  <route id="partnerToDB">  
    <from uri="activemq:queue:partners"/>  
    <transacted/>  
    ...  
  </route>  
</camelContext>
```

En el caso de que se requieran transacciones globales, la principal diferencia radica en el *Transaction Manager* a utilizar, dado que se necesita uno que soporte todos los componentes involucrados. La configuración es muy similar, pero cambian algunos detalles en función del escenario en particular.

Independientemente de si las transacciones son locales o no, es posible configurar la política a seguir por estas, o sea, se puede definir si la transacción deberá formar parte de una transacción anterior aún en curso, en caso de que exista (PROPAGATION_REQUIRED), o deberá iniciarse una nueva independientemente del escenario (PROPAGATION_REQUIRES_NEW). Véase el siguiente ejemplo:

```
<bean id="required" class="org.apache.camel.spring.spi.SpringTransactionPolicy">  
  <property name="transactionManager" ref="txManager"/>  
  <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/>  
</bean>
```

luego de lo cual, el elemento *transacted* queda de la siguiente forma:

```
<transacted ref="required"/>
```

UnitOfWork

Capítulo 2 – Descripción de la arquitectura

Para aquellos casos en los que no existe un *Transaction Manager*, aparece el concepto de *UnitOfWork*. La idea general detrás del mismo es la de agrupar un conjunto de tareas, como una unidad coherente. Está representado dentro del *framework* por la *interface* *org.apache.camel.spi.UnitOfWork* la cual está a su vez relacionada con *org.apache.camel.spi.Synchronization* que constituyen *callbacks* a ejecutar ya sea cuando se completen todas las tareas, o cuando falle alguna de ellas. *Camel* establece automáticamente los límites de las instancias de *UnitOfWork* en el inicio y fin de las rutas, y estas estarán disponibles para su manipulación a través de los objetos *Exchange* que son transportados. Las mencionadas *interfaces* cuentan con las siguientes definiciones:

```
public interface UnitOfWork{
    void addSynchronization(Synchronization synchronization);
    void removeSynchronization(Synchronization synchronization);
    void done(Exchange exchange);
}
```

```
public interface Synchronization{
    void onComplete(Exchange exchange);
    void onFailure(Exchange exchange);
}
```

y la relación que entre ellos existe es como sigue:

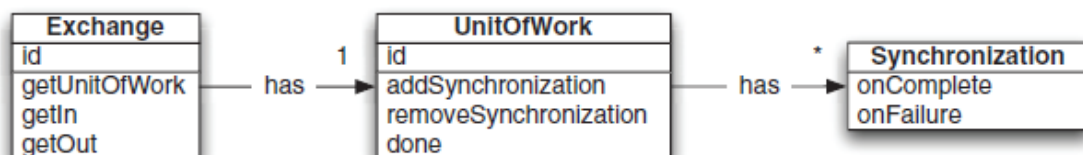


Ilustración 22: Relación entre *Exchange*, *UnitOfWork* y *Synchronization* (Camel in Action)

Sobre el presente esquema, el código necesario a ejecutar es colocado en los métodos *onComplete* y *onFailure* los cuales serán invocados dependiendo de la existencia o no de fallas. Posteriormente, las instancias de *Synchronization* pueden ser añadidas directamente al objeto *Exchange* mediante un *Processor* o ser utilizadas con el elemento *onCompletion*, ya sea globalmente o localmente dentro de una ruta en particular como se muestra a continuación:

```
public class FileRollback implements Synchronization {
```

Capítulo 2 – Descripción de la arquitectura

```
public void onComplete(Exchange exchange) {
    ...
}
public void onFailure(Exchange exchange) {
    ...
}
}

<bean id="fileRollback" class="package.FileRollback"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <onCompletion onFailureOnly="true">
        <bean ref="fileRollback" method="onFailure"/>
    </onCompletion>
    <route>
        <from uri="..." />
        ...
    </route>
</camelContext>

from("...")
.onCompletion().onFailureOnly()
.bean(FileRollback.class, "onFailure")
.end()
.bean(..., "...")
...
```

Seguridad

La gestión de la seguridad es una tarea casi omnipresente en cualquier sistema de información. Para la integración, la seguridad adquiere una significación adicional debido a que en principio, ningún sistema desea ver su información o procesos de negocio en caso de que existan, comprometidos por el acceso de aplicaciones externas. Por ende, no es posible considerar una arquitectura de integración sin capacidades de administración de seguridad.

Bajo la presente arquitectura la gestión de la seguridad es posible a cuatro niveles distintos:

- ✓ Nivel de Ruta
- ✓ Nivel de contenido de mensaje (*payload*).

Capítulo 2 – Descripción de la arquitectura

- ✓ Nivel de *endpoint*.
- ✓ Nivel de configuración.

Nivel de Ruta

La gestión de seguridad al nivel de ruta es básicamente un control de acceso basado en roles a la manera de Spring – Security (10). La idea general consiste en la definición de políticas de seguridad que luego serán aplicadas a las rutas dentro del contexto de la aplicación. Tales políticas requieren la especificación de ciertos elementos propios del mencionado *framework* de seguridad, cuya discusión escapa del marco de la presente investigación. En la cita anterior, pueden ser encontrados todos los detalles necesarios para la comprensión de su funcionamiento. De vuelta al establecimiento de las políticas de seguridad, el siguiente fragmento de configuración muestra cuáles son las especificaciones a considerar.

```
<bean id="accessDecisionManager"
class="org.springframework.security.access.vote.AffirmativeBased">
  <property name="allowIfAllAbstainDecisions" value="true"/>
  <property name="decisionVoters">
    <list>
      <bean class="org.springframework.security.access.vote.RoleVoter"/>
    </list>
  </property>
</bean>
<spring-security:authentication-manager alias="authenticationManager">
  <spring-security:authentication-provider user-service-ref="userDetailsService"/>
</spring-security:authentication-manager>
<spring-security:user-service id="userDetailsService">
  <spring-security:user name="..." password="..." authorities="ROLE_USER,
ROLE_ADMIN"/>
  <spring-security:user name="..." password="..." authorities="ROLE_USER"/>
</spring-security:user-service>
<authorizationPolicy id="admin" access="ROLE_ADMIN"
authenticationManager="authenticationManager"
accessDecisionManager="accessDecisionManager"
xmlns="http://camel.apache.org/schema/spring-security"/>

<camelContext id="myCamelContext" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="..." />
    <policy ref="admin">
      <to uri="..." />
    </policy>
  </route>
```


Capítulo 2 – Descripción de la arquitectura

</camelContext>

La configuración anterior especifica que solamente aquellos mensajes autenticados con rol **ROLE_ADMIN** pueden ser enviados en esa ruta hacia el componente que se especifique en el elemento **to**. La forma en la que se llevará a cabo dicha autenticación depende en gran medida de cada ambiente en particular pero debe hacerse notar que independientemente de la vía escogida, las credenciales correspondientes deben encontrarse o bien dentro del objeto **Exchange** específicamente en el objeto **Message** que representa la entrada o bien dentro del **SecurityContextHolder**.

Nivel de contenido de mensaje

Al nivel de contenido de mensaje, el mecanismo de seguridad consiste en la aplicación de algoritmos de encriptación sobre dicho contenido. Aunque existen opciones avanzadas como son la inicialización del vector a utilizar por el algoritmo, en principio simplemente se debe especificar el nombre del algoritmo a utilizar de acuerdo con JCE¹⁹ (14) como se muestra a continuación:

```
KeyGenerator generator = KeyGenerator.getInstance("DES");  
CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES", generator.generateKey());  
  
from("...")  
  .marshal(cryptoFormat)  
  .to("...")  
  .unmarshal(cryptoFormat)  
  .to("...");
```

Nótese que es necesario que el objeto utilizado en las operaciones de cifrado y descifrado sea el mismo.

Existe además una opción alternativa de cifrado cuando el contenido es basado en XML. En estos casos es posible usar el elemento **secureXML** el cual puede aplicar un cifrado parcial o total sobre el contenido del XML, y puede utilizar parámetros opcionales como el algoritmo a utilizar (**XMLCipher.TRIPLEDES**, **XMLCipher.AES_128**,

¹⁹ *Java Cryptographic Extension: tecnología que provee un framework e implementaciones para algoritmos de encriptación, generación y acuerdo de claves, y código de autenticación de mensajes.*

Capítulo 2 – Descripción de la arquitectura

XMLCipher.AES_192, *XMLCipher.AES_256*). Por ejemplo, si se desea hacer un cifrado completo, se debe escribir algo similar a:

```
from("...").  
    marshal().secureXML().  
    unmarshal().secureXML().  
to("...");
```

Mientras que las siguientes líneas:

```
String tagXPath = "//cheesesites/italy/cheese";  
boolean secureTagContent = true;  
String passPhrase = "Just another 24 Byte key";  
String algorithm = XMLCipher.TRIPLEDES;  
from("...").  
    marshal().secureXML(tagXPath, secureTagContent, passPhrase, algorithm).  
    unmarshal().secureXML(tagXPath, secureTagContent, passPhrase, algorithm).  
to("...");
```

ocasionarán un cifrado sobre el *tag* especificado del siguiente XML con el algoritmo y *password* especificados

```
<?xml version="1.0"?>  
<cheesesites>  
    <italy>  
        ...  
        <cheese>  
        </cheese>  
        ...  
    </italy>  
</cheesesites>
```

Nivel de endpoint

Determinados componentes (15) presentan capacidades de gestión de seguridad de sus correspondientes *endpoints* por lo que garantizan no solamente el aseguramiento del contenido de los mensajes, sino que además proveen algún esquema de autenticación. Esto quiere decir que el solo hecho de utilizar *endpoints* de dichos componentes incluye de manera automática las potencialidades de los mismos relativas a la seguridad.

Capítulo 2 – Descripción de la arquitectura

Nivel de configuración

Pese a que toda la configuración aquí presentada escribe directamente los valores, cabe agregar que es posible externalizar dichos valores en archivos *.properties*. Estos archivos pudieran contener en la mayoría de los casos, información sensible que en ambientes de alta seguridad, se desee proteger. La gestión de la seguridad al presente nivel es la encargada de llevar a cabo esta tarea.

Esencialmente, lo que se propone es la conversión de los valores de texto plano a valores cifrados utilizando el componente de *Jasypt* (16), y luego el propio componente se encargará de hacer las operaciones de descifrado correspondientes. Las conversiones iniciales se deben hacer manualmente, usando el *set* de opciones presentadas en (17). Supóngase que ya se tienen dichos valores y que fueron obtenidos utilizando la palabra “secret” como *password*, entonces una sección del archivo *.properties* pudiera lucir así:

```
cool.result=mock:{{cool.password}}  
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

Nótese que es necesario especificar cuáles valores han sido cifrados, a través del elemento *ENC*. Luego, se debe hacer la configuración correspondiente, indicando donde se encuentra el archivo *.properties*, así como el *parser* de *Jasypt*:

```
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">  
  <property name="password" value="secret"/>  
</bean>  
  
<camelContext xmlns="http://camel.apache.org/schema/spring">  
  <propertyPlaceholder id="properties" location="..." propertiesParserRef="jasypt"/>  
  <route>  
    <from uri="..." />  
    <to uri="{{cool.result}}"/>  
  </route>  
</camelContext>
```

Capítulo 2 – Descripción de la arquitectura

Conclusiones Parciales

Dando cumplimiento a los objetivos inicialmente planteados el presente capítulo ha presentado una propuesta de arquitectura para la capa de integración para sistemas de información sobre tecnologías JEE, o en otras palabras, a lo largo del mismo, se ha abordado la disposición de la capa en el sistema, su estructura interna, así como los componentes más relevantes que la componen y su funcionamiento. Es medular reafirmar además que *Apache Camel* constituye el fundamento tecnológico de la misma, lo cual quiere decir que en ella se muestra al mencionado *framework* como una poderosa herramienta con la que integrar aplicaciones de cualquier envergadura construidas sobre plataforma Java. Se hace énfasis en el hecho de que la capa de integración aquí descrita no es meramente un mecanismo de conexión para aplicaciones sobre la referida plataforma, sino que es más bien, una interfaz que permite la unificación de disímiles sistemas de información que no necesariamente son aplicaciones, con una aplicación escrita sobre la tecnología en cuestión, y manejar desde un punto de vista práctico las dificultades que esto implica, en función de crear desde la perspectiva del usuario final, un todo único capaz de responder coherentemente a sus necesidades.

Capítulo 3 – Evaluación de la arquitectura

Introducción

Los capítulos anteriores fundamentan desde el punto de vista teórico y describen a nivel técnico una propuesta de arquitectura de integración para sistemas de información desarrollados sobre tecnologías JEE. Sin embargo, la misma pudiera considerarse incompleta, en ausencia de parámetros concretos que avalen su validez para resolver los problemas de la integración de manera pragmática y efectiva. El presente capítulo muestra los elementos y criterios que evalúan a la presente propuesta, los cuales a su vez sirven de base para demostrar su carácter válido, dando cumplimiento así a uno de los objetivos más importantes de esta investigación.

Pese a la existencia de métodos formalmente definidos para la evaluación de arquitecturas de software, como son el caso de SAAM²⁰ (18), ATAM²¹ (19) y ARID²² (20), el carácter esencialmente práctico de la propuesta presentada aleja a sus escenarios potenciales del alcance de dichos métodos, cuya aplicación tiene mayor fuerza en fechas tempranas del ciclo de vida del software. Por tanto el análisis de lo expuesto en acápites precedentes se realizó sobre la base de su aplicación a un caso de estudio real. Primeramente se presenta el modelo de calidad utilizado en el análisis, es decir, los atributos de calidad que rigieron el proceso; luego es introducido el caso de estudio antes mencionado, y por último se muestran las métricas seguidas y los resultados obtenidos por medio de estas. A lo largo de este capítulo se utilizará el término software para hacer referencia indistinta a un producto completo o a una sección de este.

²⁰ *Software Architecture Analysis Method*

²¹ *Architecture Tradeoff Analysis Method*

²² *Active Reviews for Intermediate Designs*

Capítulo 3 – Evaluación de la arquitectura

Atributos de Calidad

Uno de los puntos críticos de cualquier evaluación de un software es sin dudas la elección de cuáles son los criterios básicos que regirán dicha revisión. Tanto es así que una selección errónea de estos criterios o atributos, conllevará necesariamente a consideraciones erróneas respecto al objeto evaluado, pudiendo esto tener serias consecuencias ulteriores en ambientes reales. La norma ISO/IEC 9126 en la parte primera de su revisión del 2001²³, establece un conjunto muy bien definido de atributos de calidad (21) que responden a características mundialmente aceptadas como deseables dentro de cualquier software y son introducidos a continuación.

Funcionalidad

La funcionalidad es el propósito inmediato de cualquier producto, servicio o sección de este. A grandes rasgos describe la cualidad de ejecutar la función o las funciones bajo su responsabilidad. Sin este atributo, carece de sentido práctico hablar de los demás. Engloba los siguientes aspectos:

- ✓ Idoneidad: característica esencial referida a la adecuación de las funciones del software a los propósitos especificados.
- ✓ Precisión: se refiere al carácter correcto de las funciones.
- ✓ Interoperabilidad: concierne la habilidad de un componente de software de interactuar con otros componentes o sistemas.
- ✓ Cumplimiento: referida a la capacidad del software de funcionar bajo las leyes y normas requeridas por una organización.
- ✓ Seguridad: relacionada con el acceso no autorizado a las funciones de un software.

²³ La norma cuenta con 4 partes desde el año 2001 y fue reemplazada en marzo de 2011 por ISO/IEC 25010

Capítulo 3 – Evaluación de la arquitectura

Confiabilidad

Define la capacidad de un software de mantener los servicios que provee durante un tiempo determinado y bajo condiciones bien definidas. Dentro de este atributo se enmarcan:

- ✓ Madurez: referida a la frecuencia de fallo del software.
- ✓ Tolerancia a fallos: habilidad del software de mantenerse en funcionamiento y recuperarse ante una falla interna o externa.
- ✓ Recuperabilidad: capacidad de volver a un estado funcional completo, luego de una falla general.

Usabilidad

La usabilidad se refiere a la facilidad de uso de las funciones del software y agrupa a los siguientes elementos:

- ✓ Comprensibilidad: determina la facilidad con la que las funciones del software pueden ser entendidas.
- ✓ Facilidad de aprendizaje: caracteriza la curva de aprendizaje de diferentes usuarios con el software.
- ✓ Operatividad: facilidad con la que el software es operable por un usuario dado en un ambiente determinado.

Eficiencia

Esta característica a veces referenciada como “rendimiento” se concentra en los recursos del sistema utilizados para proveer cierta funcionalidad. Como generalmente ocurre engloba a dos factores muy bien definidos:

- ✓ Comportamiento del tiempo: caracteriza los tiempos de respuesta para unas entradas dadas.
- ✓ Comportamiento de los recursos: caracteriza los recursos usados como memoria, ciclos de reloj, uso de la red, etc.

Capítulo 3 – Evaluación de la arquitectura

Mantenibilidad

Determina el grado en que es posible identificar y corregir una falla dentro del software. Se subdivide en:

- ✓ Analizabilidad: caracteriza la habilidad para identificar la causa raíz de una falla dentro del software.
- ✓ Variabilidad: caracteriza la cantidad de esfuerzo necesario para cambiar algún elemento del sistema.
- ✓ Estabilidad: caracteriza el impacto negativo que puede ser causado por cambios en el sistema.
- ✓ Capacidad de prueba: caracteriza el esfuerzo necesario para probar un cambio en el sistema.

Portabilidad

Determina qué tan bien el software adopta cambios en su ambiente o de sus requisitos. Este atributo considera los siguientes aspectos:

- ✓ Adaptabilidad: expresa la habilidad del sistema de cambiar a nuevas especificaciones o ambientes operativos.
- ✓ Capacidad de instalación: define el esfuerzo requerido para instalar el software.
- ✓ Conformidad: caracteriza el grado con que el software es capaz de adaptarse a normas y leyes regentes en otra organización.
- ✓ Sustituibilidad: determina qué tan fácil es cambiar un componente de software dado dentro de un ambiente especificado.

El modelo de calidad empleado en la evaluación de la arquitectura de integración propuesta por la presente investigación, no considera algunos de los atributos anteriormente mencionados, en función de las características de la misma. La siguiente tabla resume dicha información, especificando en cada caso las razones correspondientes:

Capítulo 3 – Evaluación de la arquitectura

Atributo de Calidad	Característica(s) excluida(s)	Motivo
Funcionalidad	Cumplimiento	No es responsabilidad de la capa de integración manejar que todo el software se rija por ciertas normas o políticas de alguna organización en particular.
Usabilidad	Todas	En tanto la capa de integración constituye una sección interna del software que no interactúa con el usuario final, carece de sentido medir a este nivel la facilidad con la que se lleva a cabo dicha interacción.
Eficiencia	Todas	El impacto de cualquier solución de integración en el rendimiento del software como un todo es siempre negativo, independientemente de qué tan bien se administren los recursos del sistema. Puede además asumirse axiomáticamente que dicha administración se realiza con alto grado efectividad, teniendo en cuenta el <i>framework</i> utilizado como tecnología subyacente.
Mantenibilidad	Capacidad de prueba	El <i>framework</i> utilizado como tecnología subyacente cuenta con un <i>kit</i> para pruebas basado en <i>JUnit</i> ²⁴ , lo cual presupone elevadas capacidades en este sentido.
Portabilidad	Conformidad	No es responsabilidad de la capa de integración manejar que todo el software se rija por ciertas normas o políticas de alguna organización en particular.
	Capacidad de instalación	El esfuerzo requerido para agregar los componentes necesarios para la

²⁴ *JUnit* es un *framework* cuyo objetivo central es permitir el desarrollo de pruebas unitarias sobre plataforma Java. En los últimos años se ha convertido en poco menos que un estándar en este campo y es soportado por los principales entornos de desarrollo para dicho lenguaje.

Capítulo 3 – Evaluación de la arquitectura

		implementación de la arquitectura de integración propuesta es mínimo, dado que solamente se necesitan las bibliotecas de clases correspondientes y sus dependencias, todo lo cual siempre está disponible para las distintas versiones de la tecnología subyacente en el sitio oficial de la misma (http://camel.apache.org/download-archives.html).
--	--	---

Tabla 1: Atributos descartados en la conformación del modelo de calidad.

Se incluyen además como atributos de calidad al modelo los siguientes no definidos por ISO/IEC 9126:

- ✓ Transaccionalidad: capacidad de garantizar que los procesos se administren como operaciones individuales e indivisibles, las cuales debe terminan exitosamente o fallar completamente, sin dar cabida a estados intermedios.
- ✓ Escalabilidad: define la habilidad para manejar el crecimiento del volumen de trabajo de manera adecuada, con el menor esfuerzo posible.

Caso de Estudio

A continuación se presenta el caso de estudio sobre el cuál versó el análisis de la arquitectura aquí descrita. El mismo corresponde a una situación real del proyecto Quarxo²⁵ perteneciente al centro CEIGE²⁶ de la Facultad 3 de la Universidad de las Ciencias Informáticas.

²⁵ Nombre que recibe el producto desarrollado por el Proyecto de Modernización del Sistema Bancario Nacional, anteriormente denominado SAGEB (Sistema Automatizado de Gestión Bancaria)

²⁶ Centro de Informatización de la Gestión de Entidades

Capítulo 3 – Evaluación de la arquitectura

Uno de los requisitos funcionales actuales de Quarxo, es el de permitir la construcción de mensajes SWIFT²⁷ para aquellas operaciones que así lo requieran. Sin importar la procedencia del mensaje, durante su construcción se necesita de la ejecución de ciertas operaciones que en principio, por razones de estandarización, deberían tener lugar en un sistema externo. Este mismo sistema externo debería ser responsable del procesamiento y edición de los mensajes una vez construidos, así como de las operaciones de recepción y envío al servidor de destino final. Por cuestiones que quedan fuera del marco de esta investigación, las tareas que debería realizar este sistema externo son soportadas hoy por un componente interno de Quarxo. Dicho componente además maneja la entrega de los mensajes de la siguiente manera: los mensajes son persistidos en una base de datos SiSCOM²⁸ para posteriormente ser retirados de esta por una cadena de aplicaciones SiSCOM que los transportan usando comunicación por *sockets* a su destino final, el cual soporta otras vías de comunicación como son el acceso a un servidor de mensajes. La siguiente figura muestra la configuración actual del escenario descrito:

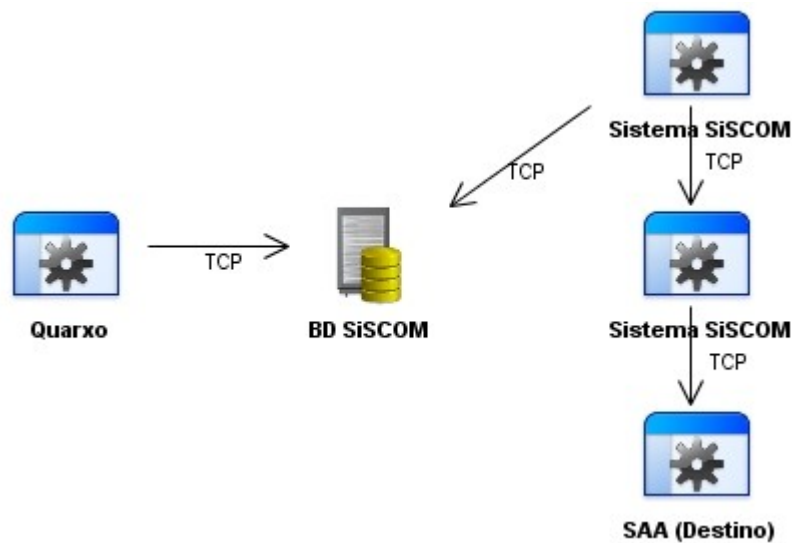


Ilustración 23: Diagrama de despliegue del escenario actual del caso de estudio.

No obstante el estado funcional del escenario descrito, esta configuración tiene serios inconvenientes. En primer lugar, la decisión de embeber dentro de otra aplicación como un componente lo que debería ser un sistema externo, no solamente está en

²⁷ Society for Worldwide Interbank Financial Telecommunications. Es un estándar internacional de mensajería inter – bancaria organizado bajo las leyes belgas.

²⁸ Sistema de Comunicación

Capítulo 3 – Evaluación de la arquitectura

franca oposición al principio de encapsulación, ya que Quarxo para la ejecución de sus tareas necesita “conocer” todos los detalles acerca de funciones que originalmente no fueron concebidas como suyas, sino que además compromete el carácter reutilizable de las funcionalidades de dicho componente, debido a que la disyuntiva para futuras aplicaciones que las necesiten es o bien forzar su arquitectura para que acepte la inclusión del mencionado componente, o bien implementarlas nuevamente de acuerdo con su arquitectura, ninguna de las cuales es una solución demasiado atrayente. Por otra parte, el esquema de envío de los mensajes además de altos grados de rigidez y acoplamiento, puesto que necesita que todo el conjunto de aplicaciones implicadas esté funcionando para que los mensajes lleguen correctamente a su destino, además de que ofrece una única vía posible de integración con sistemas externos la base de datos SiSCOM. Este acceso directo a la mencionada base de datos, presenta además riesgos potenciales de seguridad, puesto que esta se expone abiertamente a la acción de entes externos, lo cual pudiera resultar en operaciones indebidas, pérdidas o inconsistencias en los datos, entre otros problemas. Aún cuando esta situación es soluble a través del establecimiento de políticas de seguridad estrictas en cuanto al acceso dentro de la propia base de datos, esta solución requiere entonces de la existencia de un especialista en el tema, complejizando el despliegue de todo el mecanismo. Por lo anteriormente expuesto, una solución mucho más elegante basada en integración de sistemas sería:

- ✓ Separar en una aplicación independiente aquellas operaciones referentes a la mensajería SWIFT. Este nuevo sistema como es de suponer debe exponer entonces una o varias interfaces que le permitan integrarse con aquellas aplicaciones potenciales que requieran de sus funcionalidades. Nótese como esto permite no solamente liberar a nuevos sistemas de las responsabilidades para con la mensajería SWIFT que le son ajenas, sino que permite además la estandarización concreta y concentración de las mismas en un único sistema. Es de hacerse notar el hecho de que la arquitectura de este nuevo sistema no queda atada a restricción alguna, excepto por la condición de que debe exponer los servicios que así lo requieran. Dicho de otra forma, esta aplicación puede ser implementada perfectamente sobre una plataforma que no tiene por qué ser JEE. En aras de mostrar las potencialidades de la arquitectura de integración propuesta, y por motivos netamente prácticos, para esta investigación dicho sistema fue implementado sobre JEE y expone sus funcionalidades vía RMI.

Capítulo 3 – Evaluación de la arquitectura

- ✓ Agregar en Quarxo una capa de integración con el nuevo sistema.
- ✓ Reemplazar todo el sistema SiSCOM de recepción y envío de mensajes por un servidor de mensajes intermedio entre el nuevo sistema y el destino.

El lector debe percibir el notable impacto de estas decisiones sobre el acoplamiento de los sistemas involucrados, así como en la escalabilidad del escenario en general.

De acuerdo con las modificaciones propuestas el escenario final sobre el que se aplica la arquitectura de integración presentada en el capítulo previo se ilustra en la siguiente figura:



Ilustración 24: Diagrama de despliegue final del caso de estudio.

La propuesta de arquitectura descrita se aplicó sobre el nuevo sistema²⁹ para la publicación del servicio RMI anteriormente referido, así como para administrar y controlar el acceso al servidor de mensajes.

Métricas y Resultados

Una vez definido el conjunto de atributos de calidad a considerar en la evaluación, el próximo elemento en importancia a tener en cuenta es la lista de métricas que han de ser usadas para establecer el comportamiento de los mismos. De acuerdo con (22) una métrica es una medida cuantitativa de la evaluación, el control o la selección de una persona, proceso, evento, o institución, junto con los procedimientos para llevar a cabo las medidas y los procedimientos para la interpretación de la evaluación a la luz de las evaluaciones comparables o anteriores. El listado aquí empleado fue construido a partir de las métricas externas definidas por la norma ISO 9126 (23) bajo el principio de tener al menos una para cada atributo de calidad a controlar. Es de destacar además que siendo el objeto a evaluar una sección potencial dentro de un software, no son aplicables todas las métricas definidas en la norma, lo cual no resta validez a las pruebas realizadas si se tiene en cuenta que la propia norma no fuerza la

²⁹ Sistema SWIFT en la figura.

Capítulo 3 – Evaluación de la arquitectura

aplicación exhaustiva del mencionado conjunto. A continuación se exponen las métricas seleccionadas, aclarando en cada caso el factor de calidad involucrado.

Nombre: Adecuación funcional
Factor de calidad: Idoneidad
Propósito: ¿Qué tan adecuadas son las funciones evaluadas?
Método de aplicación: Comparación entre el número de funciones que resultan idóneas para la ejecución de las tareas especificadas y el total de funciones evaluadas.
Medida y fórmula: $X = 1 - A/B$ A = número de funciones en las que se detectaron problemas durante su evaluación B = total de funciones evaluadas
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.

Tabla 2: Métricas utilizadas. Adecuación funcional

Nombre: Precisión computacional
Factor de calidad: Precisión
Propósito: ¿Qué tan seguido los resultados finales no son exactos?
Método de aplicación: Registrar el número de resultados inexactos de acuerdo con las especificaciones.
Medida y fórmula: $X = A/T$ A = número de computaciones inexactas detectas T = tiempo total de operación
Interpretación: $0 \leq X$, la calidad es superior en tanto X es más cercano a 0.

Tabla 3: Métricas utilizadas. Precisión computacional.

Nombre: Intercambiabilidad de los datos
Factor de calidad: Interoperabilidad
Propósito: ¿Qué tan correctamente fueron implementadas las interfaces de intercambio para los datos especificados?
Método de aplicación: Comparación entre el número de formatos intercambiados efectivamente con otros sistemas y el total de formatos que se necesitan intercambiar.
Medida y fórmula: $X = A/B$

Capítulo 3 – Evaluación de la arquitectura

A = número de formatos intercambiados exitosamente
B = total de formatos a intercambiar
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.

Tabla 4: Métricas utilizadas. Intercambiabilidad.

Nombre: Capacidad de control de acceso
Factor de calidad: Seguridad
Propósito: ¿Qué tan controlable es el acceso al sistema?
Método de aplicación: Comparación entre el número de operaciones ilegales detectadas y el total de operaciones ilegales especificadas.
Medida y fórmula: $X = A/B$ A = número de operaciones ilegales detectadas B = total de operaciones ilegales especificadas
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.

Tabla 5: Métricas utilizadas. Capacidad de Control de Acceso.

Nombre: Densidad de fallas contra casos de prueba
Factor de calidad: Madurez
Propósito: ¿Cuántas fallas se detectan durante un determinado tiempo de prueba?
Método de aplicación: Contar el número de fallas detectadas y el total de casos de prueba.
Medida y fórmula: $X = A/B$ A = número de fallas detectadas B = total de casos de pruebas
Interpretación: $0 \leq X$, la calidad es superior en tanto X es más cercano a 0 conforme B crece.

Tabla 6: Métricas utilizadas. Densidad de fallas contra casos de prueba.

Nombre: Prevención de colapsos
Factor de calidad: Tolerancia a fallos
Propósito: ¿Qué tan seguido ocurre un colapso general de todo el ambiente de producción a causa de una falla del sistema?
Método de aplicación: Contar el número de colapsos respecto al número de fallas.

Capítulo 3 – Evaluación de la arquitectura

<p>Medida y fórmula: $X = 1 - A/B$</p> <p>A = número de colapsos</p> <p>B = número de fallas</p>
<p>Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.</p>

Tabla 7: Métricas utilizadas. Prevención de colapsos.

<p>Nombre: Capacidad de reinicio</p>
<p>Factor de calidad: Recuperabilidad</p>
<p>Propósito: ¿Qué tan seguido puede ser reiniciado el sistema y proveer servicios dentro de un tiempo requerido?</p>
<p>Método de aplicación: Contar el número de veces que el sistema reinicia y provee servicios dentro de un límite de tiempo y compararlo con el número total de reinicios del sistema.</p>
<p>Medida y fórmula: $X = A/B$</p> <p>A = número de reinicios exitosos durante el período de pruebas</p> <p>B = número total de reinicios</p>
<p>Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.</p>

Tabla 8: Métricas utilizadas. Capacidad de reinicio.

<p>Nombre: Capacidad de restauración</p>
<p>Factor de calidad: Recuperabilidad</p>
<p>Propósito: ¿Qué tan capaz es de restaurarse el sistema a sí mismo luego de eventos anormales?</p>
<p>Método de aplicación: Contar el número de restauraciones exitosas y compararlo con el total de restauraciones requeridas.</p>
<p>Medida y fórmula: $X = A/B$</p> <p>A = número de casos de restauración exitosos</p> <p>B = número de restauraciones requeridas</p>
<p>Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.</p>

Tabla 9: Métricas utilizadas. Capacidad de restauración.

<p>Nombre: Capacidad de ser auditado</p>
<p>Factor de calidad: Analizabilidad</p>

Capítulo 3 – Evaluación de la arquitectura

Propósito: ¿Pueden ser encontradas fácilmente las operaciones fallidas?
Método de aplicación: Observar el comportamiento de aquel que trata de resolver fallas.
Medida y fórmula: $X = A/B$ A = número de información realmente registrada durante la falla B = número de información planeada para ser registrada y que es suficiente para monitorear el sistema
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.

Tabla 10: Métricas utilizadas. Capacidad de ser auditado.

Nombre: Complejidad de modificación
Factor de calidad: Variabilidad
Propósito: ¿Puede el sistema ser cambiado fácilmente para resolver un problema?
Método de aplicación: Observar el comportamiento del probador que trata de realizar cambios en el sistema.
Medida y fórmula: $X = \text{Sum}(A/B)/N$ A = tiempo empleado en el cambio B = tamaño del cambio N = número de cambios
Interpretación: $0 < X$, la calidad es superior en tanto X es menor conforme la cantidad de cambios no es muy grande.

Tabla 11: Métricas utilizadas. Complejidad de modificación.

Nombre: Localización del impacto de la modificación
Factor de calidad: Estabilidad
Propósito: ¿Puede el sistema ser operado sin fallas luego del mantenimiento?
Método de aplicación: Contar el número de fallas ocurridas luego de un cambio que debieron verse afectadas por dicho cambio.
Medida y fórmula: $X = A/N$ A = número de fallas emergidas luego del cambio que debió resolverlas N = número de fallas resueltas
Interpretación: $0 \leq X$, la calidad es superior en tanto X está más cerca de 0.

Tabla 12: Métricas utilizadas. Localización del impacto de la modificación.

Capítulo 3 – Evaluación de la arquitectura

Nombre: Adaptabilidad ambiental del sistema a nivel de software
Factor de calidad: Adaptabilidad
Propósito: ¿Es capaz el sistema de adaptarse a sí mismo a diferentes ambientes operativos?
Método de aplicación: Observar el comportamiento del probador que trata de adaptar el sistema a un ambiente operativo distinto.
Medida y fórmula: $X = 1 - A/B$ A = número de funciones operacionales fallidas durante el cambio de ambiente B = número de funciones probadas
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X está más cerca de 1.

Tabla 13: Métricas utilizadas. Adaptabilidad ambiental del sistema a nivel de software.

Nombre: Uso continuado de los datos
Factor de calidad: Sustituibilidad
Propósito: ¿Si el sistema se reemplaza a su versión anterior es posible seguir utilizando los mismos datos?
Método de aplicación: Observar el comportamiento del probador que trata de instalar la versión anterior del sistema.
Medida y fórmula: $X = A/B$ A = número de datos usados en el software a ser reemplazado confirmados como disponibles a usarse de manera continuada B = número de datos usados en el software a ser reemplazado planeados para continuar en uso
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X está más cerca de 1.

Tabla 14: Métricas utilizadas. Uso continuado de los datos.

Nombre: Caída del tiempo de respuesta ^{*30}
Factor de calidad: Escalabilidad
Propósito: ¿Puede el sistema mantener sus niveles de respuesta en ambientes de alta concurrencia?
Método de aplicación: Medir el tiempo de respuesta del sistema en ambiente aislado

³⁰ Estas métricas no forman parte del estándar ISO 9126.

Capítulo 3 – Evaluación de la arquitectura

y compararlo con el tiempo de respuesta en ambientes concurrentes.
Medida y fórmula: $X = N/(A/B)$
A = peor tiempo de respuesta del sistema en ambiente concurrente
B = tiempo de respuesta del sistema en ambiente aislado
N = número de peticiones concurrentes efectuadas
Interpretación: $0 < X \leq N$, la calidad es superior en tanto X está más cercano a N.

Tabla 15: Métricas utilizadas. Caída del tiempo de respuesta.

Nombre: Capacidad de restauración en ambientes transaccionales*
Factor de calidad: Transaccionalidad
Propósito: ¿Qué tantas veces puede el sistema restablecer su estado anterior ante transacciones fallidas?
Método de aplicación: Contar el número de reversiones efectivas en transacciones fallidas.
Medida y fórmula: $X = A/B$
A = número de reversiones efectivas en transacciones fallidas
B = número de transacciones fallidas
Interpretación: $0 \leq X \leq 1$, la calidad es superior en tanto X es más cercano a 1.

Tabla 16: Métricas utilizadas. Capacidad de restauración en ambientes transaccionales.

Se considera importante señalar en este punto que fue obviada la inclusión de una tabla de priorización de atributos, dado que para este caso particular todos los atributos son considerados como de alto peso. Para las métricas especificadas se obtuvieron los siguientes resultados:

Nombre de la métrica	Valor obtenido
Adecuación funcional	1,0
Precisión computacional	0,0
Intercambiabilidad de los datos	1,0
Capacidad de control de acceso	1,0
Densidad de fallas contra casos de prueba	0,0
Prevención de colapsos	1,0
Capacidad de reinicio	0,9

Capítulo 3 – Evaluación de la arquitectura

Capacidad de restauración	1,0
Capacidad de ser auditado	1,0
Complejidad de modificación	10
Localización del impacto de la modificación	0,0
Adaptabilidad ambiental del sistema a nivel de software	1,0
Uso continuado de los datos	1,0
Caída del tiempo de respuesta	1,750291715
Capacidad de restauración en ambientes transaccionales	1,0

Tabla 17: Resultados por métrica obtenidos en las pruebas

Se considera oportuno esclarecer algunos elementos correspondientes a las pruebas realizadas. Referente a la métrica *Complejidad de modificación*, la medición de este factor se realizó con la aplicación de 10 cambios de variada complejidad, el tiempo fue medido en minutos y el tamaño del cambio en líneas de código necesarias. Por otra parte los resultados concernientes a la escalabilidad fueron recogidos en el ambiente descrito a continuación: se utilizaron 30 máquinas clientes y un servidor; los clientes tuvieron las siguientes características: procesador *Intel Pentium (HT) 3.0 GHz*, *motherboard ASUS P5LD2 – VM*, 1GB RAM, sistemas operativos *Windows XP Profesional Service Pack 2/Ubuntu 11.04*. Las características del servidor fueron: procesador *Intel Pentium Dual Core 2.6 GHz*, *motherboard Pegatron*, 1 GB RAM, sistema operativo *Ubuntu 11.04*. Los tiempos de respuesta fueron medidos para la generación de un millón de números aleatorios y la aplicación de operaciones de complejidad lineal sobre los valores generados.

Capítulo 3 – Evaluación de la arquitectura

Conclusiones Parciales

Llegado a este punto se considera demostrada la validez de la arquitectura de integración propuesta, tomándose como constancia no solamente las consideraciones teóricas que pudieran derivarse acerca de su estructura basada en bien conocidos patrones de integración, sino también esgrimiéndose como argumentos los resultados obtenidos durante la fase de evaluación, los cuales ratifican desde el punto de vista técnico lo planteado en apartados anteriores, aspecto con el cual se le da cumplimiento a uno de los objetivos inicialmente trazados por la presente investigación.

Es de suma importancia realizar un análisis crítico del valor obtenido durante las pruebas en el factor de escalabilidad, por la importancia que revierte el mismo en la generalidad de los entornos de integración. Para que se tenga una idea clara de lo que representa, el resultado expuesto de acuerdo con la métrica definida a estos efectos, indica que el factor de crecimiento de la cantidad de peticiones concurrentes es alrededor de 1,75 veces mayor al del tiempo de respuesta de estas peticiones concurrentes, como es obvio, para escenarios comparables con el utilizado. Esto puede ser visualizado de la siguiente forma: de cada 7 peticiones concurrentes, en el peor de los casos 4 son evaluadas de manera secuencial, es decir una después de la otra y las restantes son evaluadas instantáneamente, lo cual da una medida del considerable nivel de escalabilidad presente. Téngase en cuenta que RMI, es basado en el mecanismo de serialización y deserialización de Java, mecanismo a través del cual tiene que transitar toda la información que sea intercambiada entre los sistemas. En la gráfica que aparece a continuación se muestra la relación entre el valor esperado del tiempo de respuesta del peor caso (línea azul), y el valor real obtenido (línea roja).

Capítulo 3 – Evaluación de la arquitectura

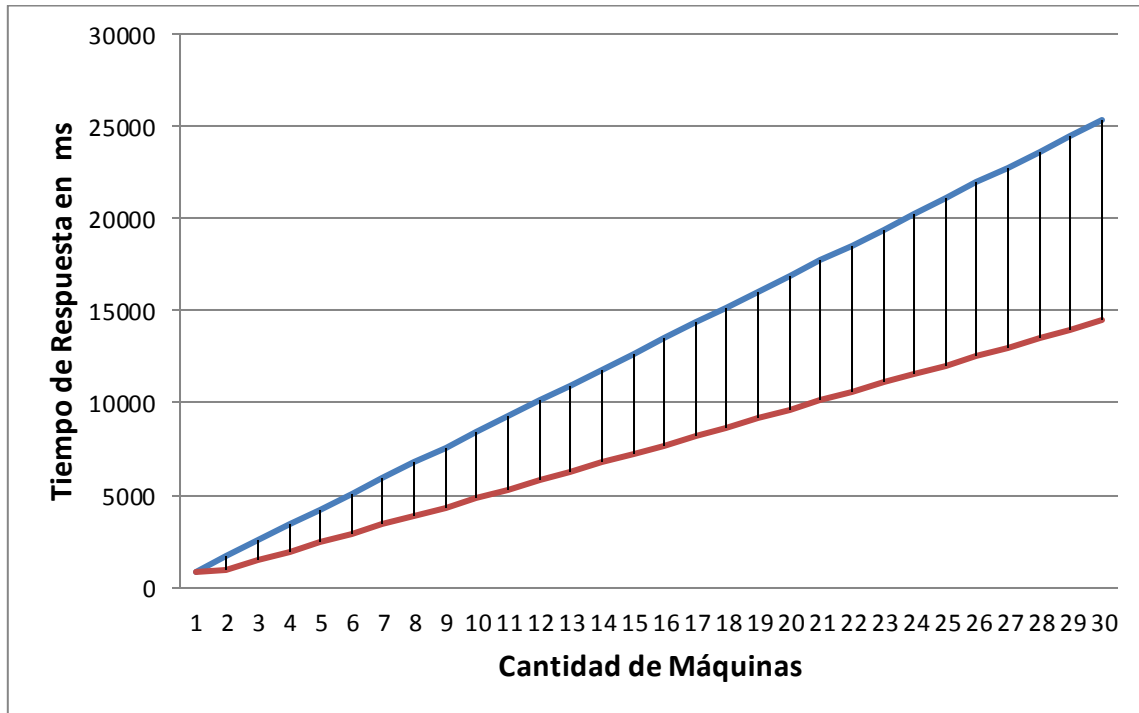


Ilustración 25: Resultados gráficos de las pruebas de escalabilidad.

El uso de esta arquitectura de integración, tiene como *trade-off point*³¹ la estrecha relación que existe entre la flexibilidad que permite imprimirle a la integración y el rendimiento, por lo cual se deben evitar implementaciones sobrecargadas en las cuales se sacrifique demasiado el rendimiento del sistema.

³¹ Dícese de aquellos elementos que fungen como línea divisoria entre dos o más aspectos generalmente relevantes, de forma que priorizar uno de ellos tiene implicaciones negativas en el desempeño de los restantes. Si dichos aspectos son considerados como valores numéricos a , b y c , por ejemplo, entonces sobre ellos se cumple que $a*b*c = 1$.

Conclusiones

Conclusiones

Llegado a este punto, es menester resaltar ciertos elementos que resultan de interés general. El presente trabajo, no intenta en ningún modo opacar las enormes potencialidades que brinda *Apache Camel* como *framework* de integración, sino que por el contrario, hace un uso exhaustivo de las mismas, exponiéndolas en una estructura coherente y aplicable en sistemas desarrollados sobre plataforma Java. Por otro lado, la arquitectura presentada no constituye una extensión al mencionado *framework*, sino más bien, una sección a agregar a cualquier sistema de información a implementar con JEE. Además, el basamento en el estilo de *Mensajería* no implica que se descarten totalmente, como queda evidenciado en el caso de estudio analizado, la invocación de procesos remotos, o cualquiera de los estilos discutidos en acápites anteriores. La idea general es utilizar sus puntos fuertes, conjuntamente con todo el sistema de patrones de integración subyacente en pos de lograr una interfaz capaz de integrar a una aplicación JEE con la mayor cantidad posible de sistemas externos, en las más disímiles formas, sin que el esquema propuesto resulte demasiado impositivo ni fuerce en modo alguno, decisiones no concernientes a la integración de sistemas.

A manera de colofón, se considera pertinente hacer un recuento de los resultados más relevantes alcanzados por la presente investigación:

- ✓ Fueron introducidas las bases teóricas fundamentales sobre las que descansa la integración de sistemas de información.
- ✓ Se identificó una amplia gama de tecnologías disponibles con las cuales llevar a efecto esta difícil tarea.
- ✓ Se definieron la estructura y componentes internos e interacción entre los mismos de una propuesta de arquitectura de la capa de integración para sistemas de información sobre tecnologías JEE, exponiéndose además los detalles técnicos más relevantes acerca de la instrumentación de la misma ya como parte de una aplicación propiamente dicha y, dando cumplimiento con ello a uno de los objetivos centrales inicialmente trazados.

Capítulo 3 – Evaluación de la arquitectura

- ✓ Fue demostrada la validez de la arquitectura presentada a partir de su aplicación en un caso de estudio real sobre el cual fue rigurosamente medurado un conjunto de atributos de calidad, arrojándose resultados sumamente satisfactorios durante esta fase de evaluación.

La gran cantidad de aspectos considerados en la definición de la arquitectura da una clara medida de cuán veraz es la afirmación realizada en los inicios de este documento, acerca de la complejidad de las tareas de la integración. La presente investigación ha intentado responder a muchas de las interrogantes que para llevarlas a efecto, surgen en los ambientes de desarrollo de aplicaciones sobre la plataforma JEE, pero esto para nada limita la posible extrapolación de las ideas aquí abordadas a otras plataformas de desarrollo. Sirvan pues las descripciones ofrecidas, no solamente como material de referencia al desarrollador de Java, sino también como fuente de posibles soluciones a escenarios de integración bajo otras tecnologías.

Recomendaciones

Recomendaciones

En aras de extender el alcance de esta investigación se sugiere:

- ✓ Explorar las nuevas posibilidades que brinden versiones futuras de la tecnología usada para definir la arquitectura propuesta por la presente investigación.
- ✓ Considerar las posibilidades reales de extender la arquitectura propuesta a otras plataformas de desarrollo.
- ✓ Realizar un estudio de viabilidad para la generación dinámica en tiempo de ejecución de aquellos parámetros de configuración que así lo permitan.
- ✓ Explorar las potencialidades para desarrollar una plataforma completa de integración sobre tecnologías JEE basada en la presente arquitectura de integración.

Glosario de Términos Anglófonos

Glosario de Términos Anglófonos

- ✓ Application Programming Interface (API): (sp. *interfaz de programación de aplicaciones*). Es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizada por otro software como capa de abstracción.
- ✓ Attachment: (sp. *archivo adjunto*). Dícese de aquellos archivos colaterales asociados a un mensaje.
- ✓ Buffer: cualquier zona de la memoria interna de un sistema de cómputo, utilizada por un software como almacenamiento, generalmente temporal.
- ✓ Framework: (sp. *marco*) en programación orientada a objetos, se refiere a un conjunto de clases que encarna y abstrae el diseño de soluciones para un número de problemas relacionados.
- ✓ Gateway: (sp. *puerta de enlace*) dispositivo que permite el flujo de datos entre redes distintas.
- ✓ Handler: (sp. *manipulador*) persona o cosa encargada de manejar cierto evento. Rutina de software encargada de realizar una determinada tarea ante la ocurrencia de un evento dado.
- ✓ Header: (sp. *encabezado*) se refiere a información suplementaria al inicio de un bloque de datos a transmitir o enviar
- ✓ Kit: anglicismo que denomina a una distribución completa de algún software o componente de software.
- ✓ Message broker: (sp. *corredor de mensajes*) software intermedio que traduce el lenguaje de un sistema de un lenguaje internacionalmente reconocido a otro a través de algún medio de telecomunicación. Se refiere además a un patrón arquitectónico para la validación, transformación y encaminamiento de mensajes.

Glosario de Términos

Glosario de Términos

- ✓ Arquitectura: diseño de más alto nivel de la estructura de un sistema. Consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información.
- ✓ Capa: referencia a cómo un software es segmentado desde el punto de vista lógico. Constituye cada una de las divisiones lógicas en las que se dividen las tareas centrales de una aplicación.
- ✓ Cohesión: es una medida de qué tan fuertemente está relacionada la funcionalidad expresada por el código fuente de una aplicación. A mayor cohesión, menor es el nivel de dependencias entre las divisiones modulares existentes dentro del software. Los componentes con alto grado de cohesión tienden a ser preferibles debido a que cohesión está usualmente asociada a otras características deseables como robustez, confiabilidad, nivel de reutilización, entre otras.
- ✓ Encapsulación: dicese del ocultamiento del estado, es decir, de los atributos de una clase. La idea es extensible al funcionamiento de un software, del cual debería solamente conocerse en caso necesario, las funcionalidades expuestas al exterior.
- ✓ Filtro: dentro de este contexto, se refiere a rutina de software encargada de realizar alguna operación en particular sobre un mensaje.
- ✓ Inversión del Control: se refiere a una técnica de programación que define la forma en la que un objeto utiliza a otro. Es un principio abstracto que describe un aspecto de algunos diseños arquitectónicos en los cuales el flujo del control es invertido en comparación con la programación procedural y bajo el cual código genérico y reutilizable controla la ejecución de código más específico.
- ✓ Invocación asíncrona: consiste en la invocación de algún servicio local o remoto y no esperar a la terminación de dicha llamada para ejecutar la siguiente instrucción.

Glosario de Términos

- ✓ Invocación síncrona: consiste en la invocación de algún servicio local o remoto y esperar a la terminación de dicha llamada para ejecutar la siguiente instrucción.
- ✓ Inyección de Dependencias: término acuñado por Martin Fowler³² en 2004 como forma más explícita de referirse a la inversión del control.
- ✓ Lógica de negocio: es término utilizado generalmente para describir los algoritmos funcionales que manipulan el intercambio de datos dentro de una aplicación.
- ✓ Patrón: es una solución efectiva y reutilizable a un problema bien identificado dentro del diseño de software.
- ✓ Procedimiento remoto: es cualquier servicio cuya invocación se realiza de manera similar a la de un servicio local, pero se ejecuta realmente en un sistema externo, es decir, fuera del contexto físico desde dónde fue llamado.
- ✓ Requisito: condición o capacidad que tiene que ser alcanzada o poseída por un sistema o componente de un sistema para satisfacer un contrato, estándar, u otro documento impuesto formalmente.
- ✓ Serialización: consiste en el proceso de codificación de un objeto en algún formato con el fin de transmitirlo a través de una conexión como una serie de bytes o humanamente más legible como XML o JSON³³, entre otros.
- ✓ Servicio web: un sistema de software diseñado para soportar la interacción interoperable máquina a máquina sobre una red. Tiene una interfaz descrita en un formato procesable por máquina (WSDL). Otros sistemas interactúan con el servicio Web de una manera prescrita por su descripción usando mensajes SOAP, típicamente transmitido a través de HTTP con una serialización XML en conjunción con otras normas relacionadas con la Web (24).

³² *Martin Fowler es actualmente uno de los mayores gurús del desarrollo de software ágil y orientado a objetos. Se especializa además en análisis y diseño orientado a objetos, UML y patrones.*

³³ *JavaScript Object Notation: es la notación propuesta por el lenguaje Javascript para la representación de objetos como cadenas de caracteres.*

Glosario de Términos

- ✓ Servlet: es un objeto cuya existencia se encuentra confinada a un contenedor de aplicaciones web, que existe con la intención de extender la funcionalidad de este.
- ✓ Sistema de Mensajería: es un sistema que permite la comunicación asíncrona entre aplicaciones no acopladas de manera confiable (25).
- ✓ Transferencia de estado representacional: es un enfoque para desarrollar servicios web y una alternativa a otras especificaciones de computación distribuida tales como CORBA o DCOM³⁴ (26). Aunque el término inicialmente estaba referido a un conjunto de principios de arquitectura, en la actualidad se usa en un sentido más amplio para describir cualquier interfaz web simple que utiliza XML y HTTP, sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes como el protocolo de servicios web SOAP

³⁴ *Distributed Component Object Model: es una tecnología propietaria de Microsoft para desarrollar componentes software distribuidos sobre varios ordenadores y que se comunican entre sí*

Referencias Bibliográficas

Referencias Bibliográficas

1. **Linthicum, David S.** *Enterprise Application Integration*. s.l. : Addison – Wesley, 1999. 0201615835.
2. **Sharma, Rahul, Stearns, Beth y Ng, Tony.** *J2EE Connector Architecture and Enterprise Application Integration*. s.l. : Addison – Wesley, 2003. 0201775808.
3. **Hohpe, Gregor y Woolf, Bobby.** *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. s.l. : Addison - Wesley, 2003. 0321200683.
4. **Institute of Electrical and Electronics Engineers.** *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York : s.n., 1990.
5. **Gamma, Erich, y otros, y otros.** *Design Patterns: Elements of Reusable Object-Oriented Software*. s.l. : Addison - Wesley, 1995. 0201633612.
6. **TechTarget Corporate.** The ServerSide Application Server Matrix. [En línea] 2011. [Citado el: 16 de Marzo de 2011.] <http://www.theserverside.com/reviews/matrix.tss>.
7. **Johnson, Rod, y otros, y otros.** *The Spring Framework – Reference Documentation Version 2.5.4*. s.l. : SpringSource Incorporated, 2008.
8. **The Apache Software Foundation.** Apache WSS4J - Web Services Security for Java. *Apache WSS4J*. [En línea] The Apache Software Foundation, 2011. [Citado el: 17 de Marzo de 2011.] <http://ws.apache.org/wss4j/>.
9. —. *Apache Sandesha2*. [En línea] The Apache Software Foundation, 2011. [Citado el: 18 de Marzo de 2011.] <http://ws.apache.org/sandesha/sandesha2/index.html>.

Bibliografía

10. **SpringSource**. *Spring Security Project Home Site*. [En línea] SpringSource, 2011. [Citado el: 17 de Marzo de 2011.] <http://static.springframework.org/spring-security/site/>.

11. **The Apache Software Foundation**. Components Supported. *Apache Camel*. [En línea] 2010. [Citado el: 18 de Abril de 2011.] <http://camel.apache.org/components.html>.

12. —. Enterprise Integration Patterns. *Apache Camel*. [En línea] 2010. [Citado el: 20 de Abril de 2011.] <http://camel.apache.org/enterprise-integration-patterns.html>.

13. **Ibsen, Claus y Anstey, Jonathan**. *Camel In Action*. s.l. : Manning Publications Company, 2011. 9781935182368.

14. **Oracle Corporation**. Java SE Security. *Oracle Technology Network*. [En línea] [Citado el: 22 de Abril de 2011.] <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html>.

15. **The Apache Software Foundation**. Camel Security Offerings and Capabilities. [En línea] 2010. [Citado el: 15 de Marzo de 2011.] <http://camel.apache.org/security.html>.

16. **The Jasypt Team**. *Jasypt: Java Simplified Encryption*. [En línea] The Jasypt Team, 2011. [Citado el: 22 de Abril de 2011.] <http://www.jasypt.org/>.

17. **The Apache Software Foundation**. Jasypt component. *Apache Camel*. [En línea] 2010. [Citado el: 22 de Abril de 2011.] <http://camel.apache.org/jasypt.html>.

18. **Kazman, Rick, Len Bass, Gregory Abowd y Webb, Mike**. *SAAM: A Method for Analyzing the Properties of Software Architectures*. 1995.

19. **Kazman, Rick, Klein, Mark y Clements, Paul**. *ATAM: Method for Architecture Evaluation*. Pittsburgh : Carnegie Mellon University, 2000. CMU/SEI-2000-TR-004.

20. **Clements, Paul C**. *Active Reviews for Intermediate Designs*. s.l. : Carnegie Mellon University, 2000. CMU/SEI-2000-TN-009.

Bibliografía

21. **Scottish Qualifications Authority.** ISO 9126 Software Quality Model. *ISO 9126 Software Quality Characteristics*. [En línea] 6 de Abril de 2010. [Citado el: 12 de Mayo de 2011.] <http://www.sqa.net/iso9126.html>.
22. **Theodoridis, Sergios y Koutrombas, Konstantinos.** *Pattern Recognition, Fourth Edition*. s.l. : Elsevier Academic Press, 2009.
23. **ISO/IEC.** *ISO/IEC 9126-2: Software engineering – Product quality – Part 2: External metrics*. 2002. ISO/IEC TR 9126-2.
24. **W3C.** Web Services Glossary. *Web Services Glossary*. [En línea] 11 de Febrero de 2004. [Citado el: 17 de Mayo de 2011.] <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/>.
25. **Wetherill, John.** Messaging Systems and the Java Message Service (JMS). *Sun Developer Network*. [En línea] 22 de Marzo de 2003. [Citado el: 17 de Mayo de 2011.] <http://java.sun.com/developer/technicalArticles/Networking/messaging/>.
26. **Thomas Fielding, Roy.** *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine : s.n., 2000.

Bibliografía

Bibliografía

1. **Linthicum, David S.** *Enterprise Application Integration*. s.l. : Addison – Wesley, 1999. 0201615835.
2. **Sharma, Rahul, Stearns, Beth y Ng, Tony.** *J2EE Connector Architecture and Enterprise Application Integration*. s.l. : Addison – Wesley, 2003. 0201775808.
3. **Hohpe, Gregor y Woolf, Bobby.** *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. s.l. : Addison - Wesley, 2003. 0321200683.
4. **Institute of Electrical and Electronics Engineers.** *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York : s.n., 1990.
5. **Gamma, Erich, y otros, y otros.** *Design Patterns: Elements of Reusable Object-Oriented Software*. s.l. : Addison - Wesley, 1995. 0201633612.
6. **TechTarget Corporate.** The ServerSide Application Server Matrix. [En línea] 2011. [Citado el: 16 de Marzo de 2011.] <http://www.theserverside.com/reviews/matrix.tss>.
7. **Johnson, Rod, y otros, y otros.** *The Spring Framework – Reference Documentation Version 2.5.4*. s.l. : SpringSource Incorporated, 2008.
8. **The Apache Software Foundation.** Apache WSS4J - Web Services Security for Java. *Apache WSS4J*. [En línea] The Apache Software Foundation, 2011. [Citado el: 17 de Marzo de 2011.] <http://ws.apache.org/wss4j/>.
9. —. *Apache Sandesha2*. [En línea] The Apache Software Foundation, 2011. [Citado el: 18 de Marzo de 2011.] <http://ws.apache.org/sandesha/sandesha2/index.html>.
10. **SpringSource.** *Spring Security Project Home Site*. [En línea] SpringSource, 2011. [Citado el: 17 de Marzo de 2011.] <http://static.springframework.org/spring-security/site/>.

Bibliografía

11. **The Apache Software Foundation.** Components Supported. *Apache Camel*. [En línea] 2010. [Citado el: 18 de Abril de 2011.] <http://camel.apache.org/components.html>.
12. —. Enterprise Integration Patterns. *Apache Camel*. [En línea] 2010. [Citado el: 20 de Abril de 2011.] <http://camel.apache.org/enterprise-integration-patterns.html>.
13. **Ibsen, Claus y Anstey, Jonathan.** *Camel In Action*. s.l. : Manning Publications Company, 2011. 9781935182368.
14. **Oracle Corporation.** Java SE Security. *Oracle Technology Network*. [En línea] [Citado el: 22 de Abril de 2011.] <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html>.
15. **The Apache Software Foundation.** Camel Security Offerings and Capabilities. [En línea] 2010. [Citado el: 15 de Marzo de 2011.] <http://camel.apache.org/security.html>.
16. **The Jasypt Team.** *Jasypt: Java Simplified Encryption*. [En línea] The Jasypt Team, 2011. [Citado el: 22 de Abril de 2011.] <http://www.jasypt.org/>.
17. **The Apache Software Foundation.** Jasypt component. *Apache Camel*. [En línea] 2010. [Citado el: 22 de Abril de 2011.] <http://camel.apache.org/jasypt.html>.
18. **Kazman, Rick, Len Bass, Gregory Abowd y Webb, Mike.** *SAAM: A Method for Analyzing the Properties of Software Architectures*. 1995.
19. **Kazman, Rick, Klein, Mark y Clements, Paul.** *ATAM: Method for Architecture Evaluation*. Pittsburgh : Carnegie Mellon University, 2000. CMU/SEI-2000-TR-004.
20. **Clements, Paul C.** *Active Reviews for Intermediate Designs*. s.l. : Carnegie Mellon University, 2000. CMU/SEI-2000-TN-009.
21. **Scottish Qualifications Authority.** ISO 9126 Software Quality Model. *ISO 9126 Software Quality Characteristics*. [En línea] 6 de Abril de 2010. [Citado el: 12 de Mayo de 2011.] <http://www.sqa.net/iso9126.html>.

Bibliografía

22. **Theodoridis, Sergios y Koutrombas, Konstantinos.** *Pattern Recognition, Fourth Edition.* s.l. : Elsevier Academic Press, 2009.

23. **ISO/IEC.** *ISO/IEC 9126-2: Software engineering – Product quality – Part 2: External metrics.* 2002. ISO/IEC TR 9126-2.

24. **W3C.** Web Services Glossary. *Web Services Glossary.* [En línea] 11 de Febrero de 2004. [Citado el: 17 de Mayo de 2011.] <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/>.

25. **Wetherill, John.** Messaging Systems and the Java Message Service (JMS). *Sun Developer Network.* [En línea] 22 de Marzo de 2003. [Citado el: 17 de Mayo de 2011.] <http://java.sun.com/developer/technicalArticles/Networking/messaging/>.

26. **Thomas Fielding, Roy.** *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine : s.n., 2000.

27. **Fisher, Mark, y otros, y otros.** *The Spring Integration Reference Manual Version 1.0.4.* s.l. : SpringSource Incorporated, 2010.

28. **Oracle Corporation.** *J2EE Connector Architecture Overview.* [En línea] 2010. [Citado el: 16 de Marzo de 2011.] <http://java.sun.com/j2ee/connector/overview.html>.

29. —. Java Message Service (JMS). [En línea] 2011. [Citado el: 17 de Marzo de 2011.] <http://www.oracle.com/technetwork/java/jms/index.html>.

30. **Oracle Corporation, Project Kenai, Cognisync.** The Standard Implementation for JAX-RPC. [En línea] 2011. [Citado el: 17 de Marzo de 2011.] <https://jax-rpc.dev.java.net/>.

31. —. JAX-RPC 2.0 renamed to JAX-WS 2.0. [En línea] Mayo de 2005. [Citado el: 16 de Marzo de 2011.] http://weblogs.java.net/blog/kohlert/archive/2005/05/jaxrpc_20_renam.html.

Bibliografía

32. **QuinStreet Inc.** Using SOAP with Java. *Java(TM) Boutique*. [En línea] 2010. [Citado el: 20 de Marzo de 2011.] <http://javaboutique.internet.com/tutorials/SOAP/>.
33. **Hermanos Carrero.** APIs de Java para XML. *Programación en Castellano*. [En línea] 2011. [Citado el: 18 de Marzo de 2011.] http://www.programacion.com/articulo/apis_de_java_para_xml_112/6.
34. **TechTarget Corporate.** JAX-WS (Java API for XML Web Services). *SearchSOA.com*. [En línea] 2011. [Citado el: 18 de Marzo de 2011.] <http://searchsoa.techtarget.com/definition/JAX-WS>.
35. **Oracle Corporation.** JSR-000914 Java™ Message Service (JMS) API. [En línea] 2011. [Citado el: 16 de Marzo de 2011.] <http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html>.
36. **The Apache Software Foundation.** *WebServices – Axis*. [En línea] The Apache Software Foundation, 2005. [Citado el: 19 de Marzo de 2011.] <http://axis.apache.org/axis/>.
37. **Poutsma, Arjen, Evans, Rick y Abed, Rabbo, Tareq.** *Spring Web Services - Reference Documentation Version 1.5.9*. s.l. : SpringSource Incorporated, 2007.
38. **Hohpe, Gregor.** Hub and Spoke [or] Zen and the Art of Message Broker Maintenance. *Enterprise Integration Patterns*. [En línea] 12 de Noviembre de 2003. [Citado el: 17 de Mayo de 2011.] http://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html.
39. **Dictionary.com.** Dictionary.com. [En línea] Dictionary.com. [Citado el: 17 de Mayo de 2011.] <http://dictionary.reference.com/>.