Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Itinerario de Ingeniería de Software. Código de Asignatura: 105128

# Comparación entre sistemas de integración

**Alumno:** Francisco de Borja García Moreno

**Directora:** Elena Ruiz Larrocha

**Curso académico:** 2015/2016

**Convocatoria:** Septiembre

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Itinerario de Ingeniería de Software. Código de Asignatura: 105128

# Comparación entre sistemas de integración

## Tipo: B

**Alumno:** Francisco de Borja García Moreno

**Directora:** Elena Ruiz Larrocha

**Impreso TFdM05_Autor. Autorización de publicación**

**y difusión del TFdM para fines académicos**

# Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

**Juan del Rosal, 16**
**28040, Madrid**

**Tel: 91 398 89 10**
**Fax: 91 398 89 09**

www.issi.uned.es

## Resumen del proyecto

Las organizaciones tienden a construir su software de integración utilizando software propietario y rechazan el software libre, debido a esto, surgió la idea de realizar este proyecto donde poder comparar las soluciones propietarias frente a las soluciones basadas en frameworks opensource.

El **Objetivo principal de este proyecto es comprobar si está justificada la preferencia de construir el servicio de integración utilizando software propietario en vez de software libre.**

## Palabras Clave

- Integración
- Modelo EAI
- SOA
- Patrón diseño
- JavaCAPS
- Camel
- Formato propietario
- Formato estándar
- Common Envelope
- Transformación
- Orquestación
- Comparativa

# Índice

Capítulo 1: Introducción ..... 9
Capítulo 2: Revisión crítica del estado de los sistemas. ..... 13
2.1 Definiciones ..... 13
2.2 El modelo EAI ..... 14
2.3 Criterios a tener en cuenta a la hora de realizar un proceso de integración .. 21
2.4 Tipos de arquitectura ..... 22
2.5 Patrones de integración ..... 24
2.5.1 Transferencia de ficheros ..... 24
2.5.2 Compartir la base de datos ..... 25
2.5.3 Invocación a procedimientos remotos ..... 25
2.5.4 Orquestación de servicios ..... 26
2.5.5 Mensajes ..... 27
2.6 Patrón de integración mediante de mensajes ..... 29
Capítulo 3: Especificación de requisitos ..... 63
3.1 Especificación de los formatos origen y destino ..... 63
3.2 Especificación de requisitos funcionales ..... 67
Capítulo 4: Diseño técnico del sistema ..... 75
4.1. Casos de uso ..... 75
4.2. Arquitectura y patrón EAI ..... 85
4.3. Herramientas que nos permitirán implementar la arquitectura propuesta y selección de las mismas ..... 92
4.3.1. Solución utilizando software propietario ..... 92
4.3.2. Solución utilizando software opensource ..... 93
Capítulo 5: Desarrollo del aplicativo ..... 95
5.1. Definición del mensaje común (Common Envelope) ..... 95
5.1.1. Header ..... 95
5.1.1.1. PhysicalData: ..... 96
5.1.1.2. BulkData ..... 96
5.1.1.3. TransformData ..... 96
5.1.2. Body ..... 98
5.2.1.1. OriginalOperations ..... 99
5.2.1.2. DestinationOperations ..... 99
5.2. Implementación utilizando software propietario ..... 100
5.2.1. Definición de formatos ..... 100

5.2.2. Introducción al desarrollo en JavaCAPS: Conceptos ..... 102
5.2.3. Uso de Endpoints ..... 108
5.2.4. Implementación de las piezas encargadas del proceso ..... 111
5.2.4.1. Entrada al sistema ..... 111
5.2.4.2. Validación de contenidos ..... 114
5.2.4.3. Transformación de contenidos ..... 116
5.2.4.4. Salida del sistema ..... 118
5.2.4.5. Comunicación de alertas ..... 120
5.2.5. Construcción del orquestador ..... 121
5.3. Implementación utilizando software libre ..... 124
5.3.1. Definición de formatos ..... 124
5.3.2. Introducción al desarrollo en Camel: Conceptos ..... 125
5.3.3. Uso de Endpoints ..... 129
5.3.4. Implementación de las piezas encargadas del proceso ..... 132
5.3.4.1. Entrada al sistema ..... 132
5.3.4.2. Validación de contenidos ..... 135
5.3.4.3. Transformación de contenidos ..... 137
5.3.4.4. Salida del sistema ..... 139
5.3.4.5. Comunicación de alertas ..... 142
5.3.5. Construcción del orquestador ..... 143
Capítulo 6: Comparación de los desarrollos ..... 146
6.1. Marco tecnológico ..... 146
6.2. Velocidad de procesado ..... 146
6.2.1. Test1. Transformación de BO a Sistema Central monobulk. ..... 148
6.2.2. Test2. Transformación de Sistema Central a BO monobulk. ..... 150
6.2.3. Test3. Transformación de BO a Sistema Central multibulk ..... 151
6.2.4. Test4. Transformación de Sistema Central a BO multibulk ..... 154
6.3. Consumo de recursos ..... 157
6.3.1. Software Propietario. ..... 158
6.3.2. Software Libre. ..... 163
6.4. Implementación ..... 170
6.4.1. Arquitectura de clases utilizada ..... 171
6.4.2. Número de clases que componen cada una de las soluciones. ..... 174
6.4.3. Número de líneas de código ..... 175
6.4.4. Componentes y utilidades disponibles ..... 176
6.4.5. Configuración ..... 179

6.4.6. Uso de un IDE de desarrollo ..... 180
6.4.7. Actualizaciones de Software ..... 180
6.4.8. Facilidad para realizar el Desarrollo ..... 181
6.4.9. Depuración y Pruebas del Sistema ..... 182
6.4.10. Aprendizaje y Soporte ..... 183
Capítulo 7: Conclusiones ..... 185
Capítulo 8: Mejoras ..... 188
Capítulo 9: Ámbito de negocio ..... 189
Bibliografía Utilizada ..... 190

## Índice de Figuras

Figura 1 - Comunicación entre organizaciones ..... 10
Figura 2 - Modelo EAI ..... 15
Figura 3 - Hub and Spoke ..... 17
Figura 4 - Bus ..... 18
Figura 5 - Enterprise Service Bus ..... 19
Figura 6 - Transferencia de ficheros ..... 24
Figura 7 - Compartir la base de datos ..... 25
Figura 8 - Invocación a procedimientos remotos ..... 26
Figura 9 - Orquestador de Servicios ..... 27
Figura 10 - Mensajería asíncrona ..... 28
Figura 11 - Canal de mensajes ..... 30
Figura 12 - Mensajes ..... 31
Figura 13 - Filtro-tubería ..... 32
Figura 14 - Enrutador de mensajes ..... 32
Figura 15 - Traductor de mensajes ..... 33
Figura 16 - Endpoint ..... 33
Figura 17 - Canal punto a punto ..... 35
Figura 18 - Canal Publish-Subscriber ..... 36
Figura 19 - Tipos de datos asociados al canal ..... 36
Figura 20 - Dead letter ..... 37
Figura 21 - Protección frente a caídas ..... 37
Figura 22 - Adaptador de comunicaciones ..... 38
Figura 23 - Sistema de mensajería en bus ..... 38
Figura 24 - Mensaje de tipo comando ..... 39
Figura 25 - Mensaje de tipo documento ..... 39
Figura 26 - Mensaje de tipo evento ..... 40
Figura 27 - Petición-Respuesta ..... 40
Figura 28 - Dirección de retorno de respuesta ..... 41
Figura 29 - Identificación de la respuesta ..... 41
Figura 30 - División en múltiples mensajes ..... 42
Figura 31 - Mensaje expirado ..... 43
Figura 32 - Enrutador basado en contenido ..... 43
Figura 33 - Filtro de mensajería ..... 44
Figura 34 - Enrutador dinámico ..... 45
Figura 35 - Lista de recipientes ..... 45
Figura 36 - Splitter ..... 46
Figura 37 - Agregador ..... 46
Figura 38 - Resecuenciador ..... 47
Figura 39 - Procesador de mensajes compuestos ..... 47
Figura 40 - Scatter-Gather ..... 48
Figura 41 - Tabla de enrutamiento ..... 48
Figura 42 - Administrador de proceso ..... 49
Figura 43 - Broker de mensajería ..... 49
Figura 44 - Reglas de decisión de patrones ..... 50
Figura 45 - Envolvente ..... 51

Figura 46 - Enriquecedor de contenido ..... 51
Figura 47 - Filtro de contenido ..... 52
Figura 48 - Claim Check ..... 52
Figura 49 - Normalizador ..... 53
Figura 50 - Modelo de datos canónico ..... 53
Figura 51 - Gateway de mensajería ..... 57
Figura 52 - Mapeador ..... 57
Figura 53 - Clientes transaccionales ..... 58
Figura 54 - Poll de consumidores ..... 58
Figura 55 - Consumidor basado en eventos ..... 59
Figura 56 - Competición de consumidores ..... 60
Figura 57 - Distribuidor de mensajes ..... 60
Figura 58 - Consumidor selectivo ..... 61
Figura 59 - Subscriptor con persistencia ..... 61
Figura 60 - Activador de servicio ..... 62
Figura 61 - Proceso de transformación ..... 67
Figura 62 - Transferencia de ficheros entre BO y el Sistema Central ..... 86
Figura 63 - Componente lector ..... 87
Figura 64 - Componente escritor ..... 88
Figura 65- Traductor de formatos ..... 89
Figura 66 – Arquitectura ..... 90
Figura 67 - Message Broker ..... 91
Figura 68 - Aplicación construida en JavaCAPS ..... 93
Figura 69 - Aplicación construida con software opensource ..... 94
Figura 70 - OTD Formato Plano ..... 101
Figura 71 - OTD Formato XSD ..... 101
Figura 72 - Creación de Colaboración ..... 102
Figura 73 - Selección de la Operación ..... 103
Figura 74 - Selección de otds y eWays adicionales ..... 104
Figura 75 - Conectivity Map ..... 105
Figura 76 - Definición de un Environment ..... 106
Figura 77 - Creación de un Deployment Profile ..... 107
Figura 78 - Asociación Software-Hardware ..... 107
Figura 79 - Funcionamiento básico del lector ..... 112
Figura 80 - Proceso lector ..... 113
Figura 81 - Funcionamiento de un Validador ..... 115
Figura 82 – Funcionamiento detallado de un Validador ..... 115
Figura 83 - Funcionamiento de un Transformador ..... 116
Figura 84 - Funcionamiento detallado del Transformador ..... 117
Figura 85 - Funcionamiento básico del Escritor ..... 118
Figura 86 - Funcionamiento del Escritor ..... 119
Figura 87 - Funcionamiento básico del Módulo de Alertas ..... 120
Figura 88 - Funcionamiento detallado del Módulo de Alertas ..... 120
Figura 89 - Orquestador ..... 121
Figura 90 - Funcionamiento detallado del Orquestador ..... 123
Figura 91 - Funcionamiento básico del lector ..... 133
Figura 92 - Proceso lector ..... 134
Figura 93 - Funcionamiento de un Validador ..... 136

Figura 94 – Funcionamiento detallado de un Validador ..... 137
Figura 95 - Funcionamiento de un Transformador ..... 138
Figura 96 - Funcionamiento detallado del Transformador ..... 139
Figura 97 - Funcionamiento básico del Escritor ..... 140
Figura 98 - Funcionamiento del Escritor ..... 141
Figura 99 - Funcionamiento básico del Módulo de Alertas ..... 142
Figura 100 - Funcionamiento detallado del Módulo de Alertas ..... 142
Figura 101 - Orquestador ..... 143
Figura 102 - Funcionamiento detallado del Orquestador ..... 145
Figura 103 - Datos iniciales de arranque del servidor Glassfish ..... 158
Figura 104 - Datos servidor Glassfish tras pasar Test1 ..... 159
Figura 105 - Datos servidor Glassfish tras pasar Test2 ..... 160
Figura 106 - Datos servidor Glassfish tras pasar Test3 ..... 161
Figura 107 - Datos servidor Glassfish tras pasar Test4 ..... 162
Figura 108 - Datos iniciales de arranque del servidor Glassfish4 ..... 163
Figura 109 - Datos servidor Glassfish4 tras pasar Test1 ..... 164
Figura 110 - Datos servidor Glassfish4 tras pasar Test2 ..... 165
Figura 111 - Datos servidor Glassfish4 tras pasar Test3 ..... 166
Figura 112 - Datos servidor Glassfish4 tras pasar Test4 ..... 167
Figura 113 - Arquitectura usada para la gestión de paquetes ..... 171
Figura 114 - Arquitectura del aplicativo utilizando Software Propietario ..... 172
Figura 115 - Arquitectura utilizada en la implementación usando Software Libre ..... 173

## Índice de Tablas

Tabla 1 - Cabecera mensaje BackOffice ..... 64
Tabla 2 - Registro de datos ..... 64
Tabla 3 - Pie del documento ..... 64
Tabla 4 - Especificación requisito ER-001 ..... 68
Tabla 5 - Especificación requisito ER-002 ..... 69
Tabla 6 - Especificación requisito ER-003 ..... 69
Tabla 7 - Especificación requisito ER-004 ..... 70
Tabla 8 - Especificación requisito ER-005 ..... 71
Tabla 9 - Especificación requisito ER-006 ..... 71
Tabla 10 - Especificación requisito ER-007 ..... 72
Tabla 11 - Especificación requisito ER-008 ..... 73
Tabla 12 - Especificación requisito ER-009 ..... 73
Tabla 13 - Especificación requisito ER-010 ..... 74
Tabla 14 - Caso de uso CU-001 ..... 76
Tabla 15 - Caso de uso CU-002 ..... 78
Tabla 16 - Caso de uso CU-003 ..... 79
Tabla 17 - Caso de uso CU-004 ..... 79
Tabla 18 - Caso de uso CU-005 ..... 80
Tabla 19 - Caso de uso CU-006 ..... 81
Tabla 20 - Caso de uso CU-007 ..... 82
Tabla 21 - Caso de uso CU-008 ..... 83
Tabla 22 - Caso de uso CU-009 ..... 84
Tabla 23 - Tiempos Test1 con Software Propietario ..... 148
Tabla 24 - Tiempos Test1 con Software Libre ..... 149
Tabla 25 - Tiempos Test2 con Software Propietario ..... 150
Tabla 26 - Tiempos Test2 con Software Libre ..... 151
Tabla 27 - Tiempos Test3 con Software Propietario ..... 153
Tabla 28 - Tiempos Test3 con Software Libre ..... 154
Tabla 29 - Tiempos Test4 con Software Propietario ..... 156
Tabla 30 - Tiempos Test4 con Software Libre ..... 157
Tabla 31 - Número de líneas de código ..... 176

## Índice de documentos adjuntos

Adjunto 1 - Common Envelope ..... 99

# Capítulo 1: Introducción

Actualmente vivimos en un mundo en continua evolución e intercomunicado.

En la rama de las tecnologías, la premisa de continua evolución es una realidad, los sistemas hardware que se utilizaban hace unos años se encuentran totalmente desfasados con respecto a los actuales, los lenguajes de programación han evolucionado de una manera asombrosa, tendiendo principalmente a la reutilización de componentes, mejorar la eficiencia en el uso de recursos como memoria, procesador…

Además, gracias a la aparición de las redes, especialmente internet, se vive en una sociedad donde prima la comunicación y en el mundo empresarial o de negocios, el principio anterior se convierte en una máxima, ya que las diferentes entidades deben comunicarse y compartir información para poder hacer negocios.

Pese a lo que se ha comentado en los párrafos anteriores, todavía existen en bancos o diferentes corporaciones sistemas antiguos, escritos en lenguajes desfasados, pero que funcionan perfectamente, además, suelen ser sistemas críticos y abordar su cambio puede ser demasiado costoso, tanto en términos de:

- Costes del nuevo proyecto, ya que suelen ser aplicaciones muy grandes que requerirían mucho tiempo y personas para ser implementadas. En resumen, requerirá un gran proyecto y nuevas máquinas, más modernas para albergar el nuevo software.

- Riesgo. Al ser sistemas críticos, cualquier mínimo bug puede acarrear grandes costes ya que al manejar información sensible, un mínimo fallo puede ser desastroso.

- Falta de servicio. Pueden existir momentos, entre migraciones de parte del sistema, que determinados servicios no se encuentren disponibles, no permitiendo realizar determinadas operativas en el momento adecuado.

Debido a esto, se opta por mantener los antiguos sistemas ya que:

- No acarrean grandes costes, solo se requiere mantenimiento y además no es necesaria una gran inversión en hardware.

- Son fiables, no contienen grandes bugs ya que llevan mucho tiempo en explotación.

- No existe el problema de falta de servicio ya que son sistemas estables.

Aun así, el mantener estos sistemas antiguos hace que surja un problema entre las diferentes organizaciones y es como comunicarse entre ellas, debido a que cada sistema tiene, por así decirlo, su propio lenguaje, debido a que cuando se construyeron no existía un estándar definido para garantizar el mutuo entendimiento.

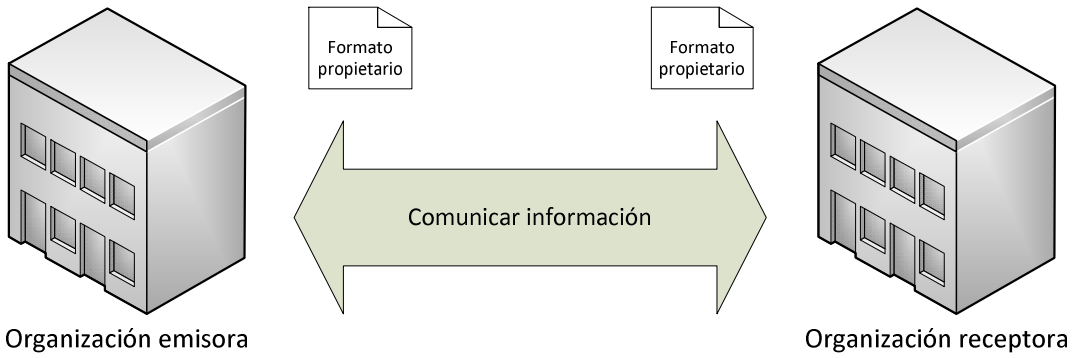



**Figura 1 - Comunicación entre organizaciones**

**[fuente: propia]**

Es en este punto donde adquieren gran importancia los sistemas de integración, los cuales se encargarán de traducir el formato de salida que genera el antiguo sistema, también denominado formato propietario, a un formato estándar (generalmente un XML) que entenderán el resto de organizaciones con las que la organización debe comunicarse.

Con esto se garantiza que la organización sabe “hablar con otros”, pero para que la comunicación sea correcta, la organización también debe “saber escuchar”, es decir, el sistema de integración deberá poder transformar del formato estándar al formato propietario de la organización.

Una vez realizada la introducción al contexto y problemática general que se pretende cubrir, se intentará explicar el ámbito real de este proyecto, el cual está basado en los dos puntos anteriores, es decir, en los sistemas de integración.

Por definición, los sistemas de integración no deberían ser un único bloque software que se encargue de transformar en ambos sentidos, no sería efectivo ya que existen múltiples posibilidades de cambios, estándares, formato propietario o las normativas propias de cada país, además, también existen numerosos formatos que adaptar, es decir, debe ser software que se adapte muy bien y rápido a cambios.

Es por esto que los sistemas de integración se construyen utilizando una arquitectura SOA, la cual garantiza el cumplimiento de los requisitos anteriores, es decir, el sistema estará compuesto por diferentes piezas colaborando entre sí para garantizar la funcionalidad, pero ante la necesidad de un cambio, se permitirá

cambiar la pieza o piezas necesarias de forma rápida, garantizando el correcto funcionamiento del conjunto.

Existen dos maneras de construir estos sistemas de integración:

1. Utilizar una suite comercial para el desarrollo como por ejemplo, JavaCAPS, Sterling Commerce de IBM, Oracle SOA Suite…

2. Desarrollar el sistema de integración basándonos en software libre, como por ejemplo Camel o Mule.

Las organizaciones siempre tienden a querer construir su software de integración utilizando software propietario y rechazan el software libre, debido a esto, surgió la idea de realizar este proyecto en el cual se pretenden comparar las soluciones propietarias frente a las soluciones basadas en frameworks opensource.

La idea sobre la que nace el proyecto cubre las siguientes fases:

- Proponer la construcción de un sistema de integración completo basándonos en una arquitectura SOA.
- Desarrollar la solución propuesta utilizando un software propietario, por ejemplo JavaCAPS.
- Desarrollar la solución utilizando tecnologías opensource como Camel.
- Comparar objetivamente las dos soluciones.
    - ✓ Tiempos de procesamiento.
    - ✓ Simplicidad en los desarrollos.
    - ✓ Conectores con sistemas externos disponibles.
    - ✓ Facilidad de abordar cambios.
    - ✓ …
- Una vez comparadas las dos soluciones, saber si es justificable que las organizaciones prefieran utilizar software propietario o realmente no tiene sentido esa preferencia.

Por tanto, el **Objetivo principal de este proyecto es comprobar si está justificada la preferencia de construir el servicio de integración utilizando software propietario en vez de software libre.**

En base a la introducción anterior, el proyecto se dividirá en los siguientes capítulos:

- **Capítulo 1: Introducción**. En él se expone el problema existente y como se pretende abordar.
- **Capítulo 2: Revisión crítica del estado de los sistemas.** En este capítulo se verá cómo el pasado, presente y futuro del campo sobre el que estamos estudiando.
- **Capítulo 3: Especificación de requisitos**. En este apartado se describirán a alto nivel que requisitos funcionales que debe cumplir la solución. Incluye la definición de formatos, exposición completa del problema…

- **Capítulo 4: Diseño de la solución.** En este apartado se definirán las diferentes piezas que componen la solución y como deben interactuar entre ellas.
- **Capítulo 5: Desarrollo de la solución.** En este apartado se verá cómo se implementó la solución, usando tanto software propietario como libre, que problemas surgieron y como se solucionaron utilizando las herramientas que nos proporcionan en la suite de desarrollo.
- **Capítulo 6: Comparación de los desarrollos.** Se obtendrán las medidas necesarias para comparar los desarrollos. Una vez obtenidas, se comentarán y compararán de la forma más objetiva posible.
- **Capítulo 7: Conclusiones.** A la vista de las comparaciones realizadas se contestará a la pregunta **¿Es mejor usar software propietario?**
- **Capítulo 8: Mejoras.** Se enumerarán posibles mejoras a realizar en el sistema implementado.
- **Capítulo 9: Ámbito de negocio.** Definirá en detalle donde son aplicables las soluciones estudiadas y porque.
- **Bibliografía utilizada.**

# Capítulo 2: Revisión crítica del estado de los sistemas.

El presente capítulo pretende ser una revisión de la historia que acompaña a los sistemas de integración. El capítulo será dividido en los siguientes puntos.

- **Definiciones.** En esta sección se intentará definir que es un sistema de integración, así como los diferentes estilos de integración que se pueden encontrar.
- **El modelo EAI.** En el cual pretendemos mostrar una pequeña introducción a la integración de aplicaciones.
- **Criterios**: Criterios a tener en cuenta a la hora de abordar un proceso de integración.
- **Tipos de arquitectura:** Explicaremos brevemente los tipos de arquitectura disponibles, centralizada y distribuida.
- **Patrones de integración.** Clasificación de los diferentes patrones utilizados para la integración de aplicaciones.
- **Patrón de integración mediante mensajes.** Definiremos más en detalle el patrón más utilizado para realizar la integración de aplicaciones y sus conceptos básicos.

Antes de continuar, debemos indicar que la mayoría de las ilustraciones que acompañan las diferentes definiciones y conceptos expuestos en este capítulo están extraídas de la web **Enterprise Integration Patterns** (http://www.enterpriseintegrationpatterns.com).

## 2.1 Definiciones

En esta sección veremos algunas de las definiciones que se pueden leer sobre la integración de sistemas.

Unas de las primeras definiciones la podemos encontrar en la web Wikipedia, según dicha web, podemos definir la integración de sistemas como:

> “En ingeniería, la integración de sistemas se define como el proceso de reunir los diferentes subsistemas existentes, asegurándonos que funcionen como un sistema único.” [Wikipedia]

> “Para la tecnología de la información, la integración de sistemas es el proceso de vincular diferentes sistemas de computación y aplicaciones software tanto física y como funcionalmente, para actuar como un todo coordinado.”

Otras definiciones sobre la integración de aplicaciones y sistemas (EAI) son:

> "Enterprise application integration (EAI) es el uso de tecnologías y servicios a través de una empresa para habilitar la integración de aplicaciones software y sistemas hardware." [Tecnopedia]
>
> Otra definición de EAI centrada más en términos de negocio es la siguiente, "EAI (Enterprise application integration), es un término utilizado en informática empresarial para los planes, métodos y herramientas destinadas a modernizar, consolidar y coordinar las aplicaciones informáticas en una empresa." [TechTarget ]
>
> También es definida como, "La traducción de datos y otros comandos de un formato de aplicación a otro. La EAI es un proceso continuo entre dos sistemas incompatibles. Esto permite a diferentes aplicaciones financieras compartir de manera eficaz datos o transacciones." [Investorpedia]

Existen otras muchas definiciones del concepto EAI (Integración de aplicaciones empresariales), pero básicamente, de todas las definiciones se puede extraer la misma interpretación, la integración de aplicaciones será la forma de que múltiples aplicaciones diferentes, independientemente de cómo fuesen realizadas, trabajen juntas como un todo.

## 2.2 El modelo EAI

Las aplicaciones desarrolladas dentro del ámbito empresarial pueden se pueden calificar de diferentes formas.

En base a cómo y para que se construye:

- Construidas por un proveedor externo pero adaptadas a las necesidades de la empresa.
- Construidas por empleados de la propia empresa.
- Aplicaciones de terceros, es decir, software comprado de ámbito general que utiliza nuestra empresa, como por ejemplo una base de datos.

En cuanto a las tecnologías que se utilizan:

- Aplicaciones open, utilizan para su construcción tecnologías de código abierto y el código del aplicativo también es abierto.
- Aplicaciones cerradas, las cuales son desarrolladas por un tercero y utilizando tecnologías propietarias o opensource, pero su código no es abierto.

Con lo cual y como vemos en las clasificaciones anteriores, dentro de una empresa podemos encontrar multitud de aplicaciones, cada una con un propósito determinado, construidas de manera diferente, pero que para conseguir un objetivo

único deben de ser coordinadas y organizadas, siendo este punto donde entra en juego el concepto de integración de aplicaciones o EAI.

En la siguiente figura, extraída de la web http://ngecacit.com/ podemos ver un esquema del modelo de integración de aplicaciones y sistemas.

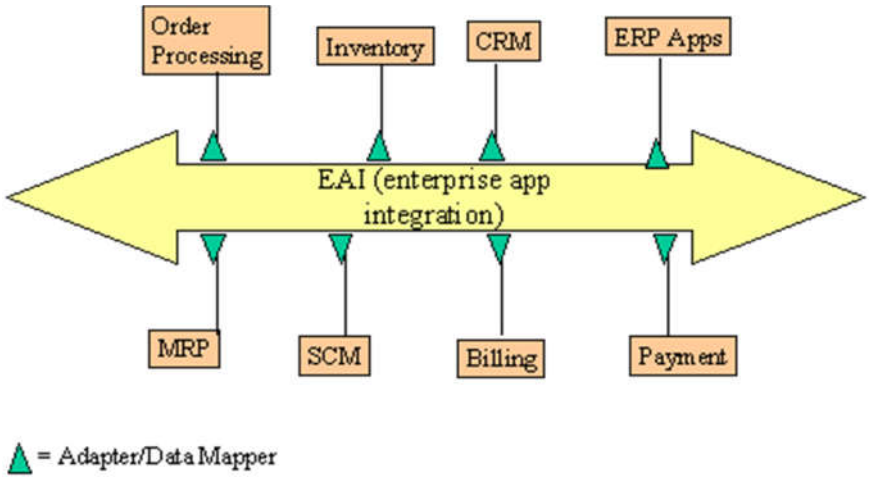



**Figura 2 - Modelo EAI**

**[fuente: http://ngecacit.com]**

La plataforma EAI no es una tecnología nueva, ha existido como término técnico desde el año 2000, aunque el problema que deseamos resolver con ella es mucho más antiguo.

Debemos indicar que cuando hablamos de EAI, realmente hablamos de una combinación de tecnologías bien conocidas que puede integrar múltiples aplicaciones. La plataforma en realidad será un servidor de integración, el cual nos provee de una colección de tecnologías (middleware como CORBA, adaptadores para conversión de protocolos, transformación de datos, manejo de transacciones y sistema de procesamiento) que permiten a diferentes aplicaciones hablar unas con otras.

Indicar que una solución EAI cobrará valor si implementa soluciones a los problemas existentes en la infraestructura como son:

- **Interoperabilidad**: Los diversos componentes de la infraestructura pueden utilizar diferentes sistemas operativos, formatos de datos, leguajes… que impiden la conexión a través de una interfaz estándar.
- **La integración de datos**: El fin último de un sistema distribuido, para ser funcional, necesita un método estándar que le permita manejar el flujo de datos entre aplicaciones y sistemas de forma no exista nunca problema en la integridad de los mismos (los datos).
- **Robustez, estabilidad y escalabilidad**: Debido a que son el pegamento que mantiene unida una infraestructura modular, una solución de integración debe ser robusta, estable y escalable.

Además, según el libro **Java CAPS Basic: Implementing Common EAI Patterns**, "La premisa fundamental de la integración de aplicaciones es que deberá ser no invasiva".

La anterior aproximación nos va a garantizar:

- Minimizar los cambios en las aplicaciones que queremos integrar. Ya que al no ser invasiva, los cambios que debemos realizar sobre las aplicaciones que deseamos integrar deberán ser nulos.
- Reducir por el coste asociado con la integración. Al no tener que modificar las aplicaciones para poder integrarlas el coste se ve reducido considerablemente.

También debemos destacar que existirá un gran abanico de aplicaciones a integrar, cada una de ellas con sus propias capacidades y requisitos, por tanto, deberá existir un gran abanico de estilos de integración que puedan ser utilizados. Realmente, una solución de integración no es cerrada a un único estilo, sino que puede usar tantos como capacidades tienen las diferentes aplicaciones que vamos a integrar.

Cuando hablamos de EAI debemos indicar que existen dos topologías principales, cada una de ellas con sus propias ventajas e inconvenientes, estas son:

- **Hub and Spoke**: Bajo esta topología, el sistema EAI actúa como el centro (concentrador), el cual interactúa con el resto de aplicaciones a través de los Spokes.
- **Bus**: Bajo esta topología, el sistema EAI actúa como un bus y el resto de aplicaciones que deben integrarse usan este bus para comunicarse.
- **ESB (Enterprise Service Bus)**: Es un modelo de arquitectura de software que gestiona la comunicación entre servicios web.

A continuación, intentaremos definir más en detalle cada uno de los dos modelos arquitectónicos que hemos comentado.

- **Hub and Spoke**

  Esta arquitectura utiliza un bróker centralizado (Hub) y adaptadores (Spoke) que son los encargados de conectar las aplicaciones al Hub.

  A continuación, mostraremos una figura que representa el concepto expresado en el párrafo anterior.

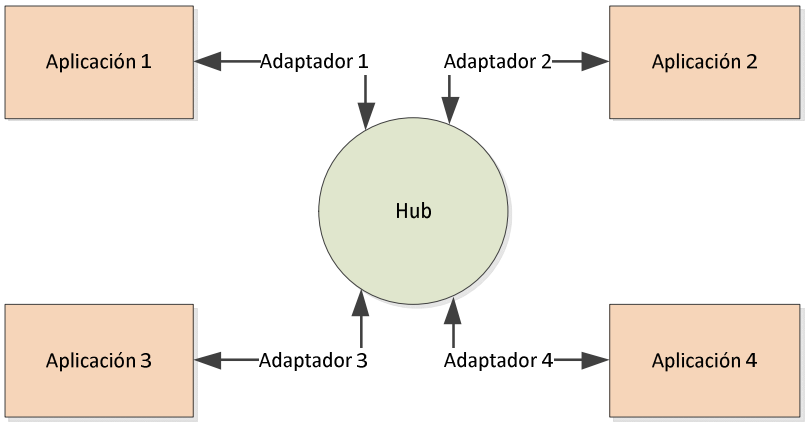



**Figura 3 - Hub and Spoke**

**[fuente: propia]**

El Spoke se conecta a una aplicación y convierte su formato de datos a un formato que entiende el Hub y viceversa, por otro lado, el Hub es un encaminador de mensajes, su forma de actuar es la siguiente:

- ✓ Recibe os mensajes.
- ✓ Analiza cuidadosamente cual es el destino de dicho mensaje.
- ✓ Transforma si fuera necesario el contenido del mensaje al formato que entiende el destino.
- ✓ Envía el mensaje al destino.

Por tanto, el funcionamiento completo puede resumirse en:

1. El adaptador obtiene los datos de la aplicación.
2. Crea y envían los mensajes con dichos datos al enrutador.
3. El enrutador transforma y envía los mensajes al adaptador destino.
4. El adaptador destino recibe los mensajes, extrae los datos y se los envía a la aplicación destino.

Tener un único hub hace que la arquitectura del sistema sea muy fácil de manejar, el problema es que el sistema no sería muy escalable ya si el número de mensajes se incrementa, la escalabilidad dependerá únicamente del hardware donde se encuentre la solución instalada y tener una máquina grande para permitir la escalabilidad nunca ha sido una solución ideal.

Para solucionar esto, en muchas ocasiones se recurre al concepto de arquitectura hub and spoke federado, donde en vez de existir un único hub, existen varios.

Cada hub tendrá metadatos locales y globales, si existen cambios en los metadatos globales, estos son propagados de forma automática al resto de hubs.

La arquitectura federada soluciona el problema de escalabilidad que existente en el caso de tener un único hub y al seguir manteniendo una

arquitectura “centralizada”, su manejo será igual de sencillo y por tanto, los costes en el soporte se verán reducidos.

Antes de continuar con la siguiente topología vamos a definir el concepto de arquitectura en **Bus**.

- **Bus**

  Una arquitectura en Bus utiliza un bloque de mensajería central (bus) para la propagación de mensajes.

  En la siguiente figura mostramos un esquema de la topología por bus.

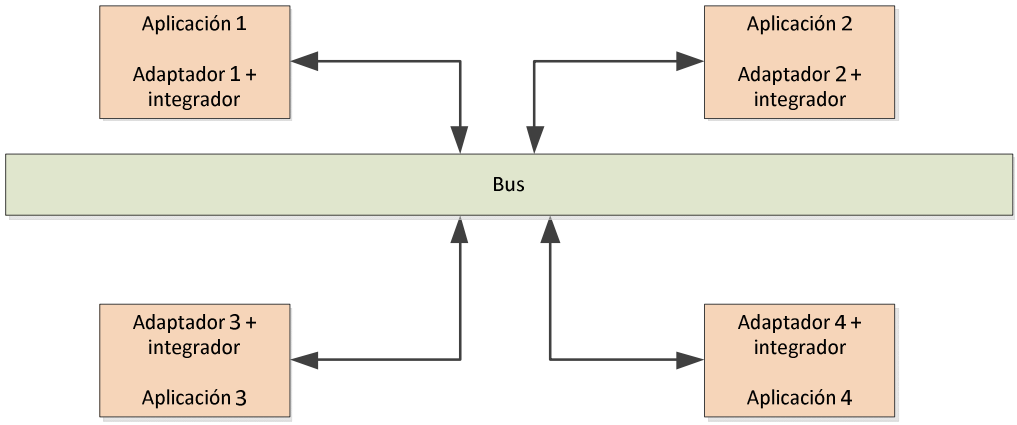



**Figura 4 - Bus**

**[fuente: propia]**

El funcionamiento de esta arquitectura es el siguiente:

1. Las diferentes aplicaciones publican mensajes al bus utilizando adaptadores.
2. Estos mensajes seguirán el flujo a través del bus hasta las aplicaciones subscritas.
3. Las aplicaciones subscritas tienen adaptadores que tomaran el mensaje del bus.
4. Una vez que el adaptador toma el mensaje este es transformado en el formato requerido por la aplicación.

Una diferencia clave entre esta topología y la topología **hub and spoke**, es que la lógica de transformación y enrutado está distribuido en los adaptadores asociados a la aplicación.

La arquitectura en bus requiere un adaptador de aplicación en la misma plataforma que las aplicaciones que vamos a integrar. Además, debido a que cada adaptador posee la lógica de transformación y enrutado, ejecutándose además en la misma plataforma que las aplicaciones origen y

destino, el aplicativo escalará mucho mejor, pero por contra, será más complejo de mantener que la topología anterior.

- **Enterprise Service Bus**

A diferencia de la anterior arquitectura, la topología **Enterprise Service Bus** se utilizará para gestionar la comunicación entre servicios web.

ESB es un acrónimo que viene definido por:

- ✓ Servicio: Ejecución de un programa no iterativa y autónoma que comunica con otros servicios mediante mensajes.
- ✓ Bus: Usado en analogía a un bus hardware.
- ✓ Empresa: El concepto inicialmente hacía referencia a la integración de aplicaciones dentro de una empresa, en la época actual este concepto es obsoleto ya que la actual integración no se limita únicamente a entidades corporativas.

Veremos una figura que representa esta topología.

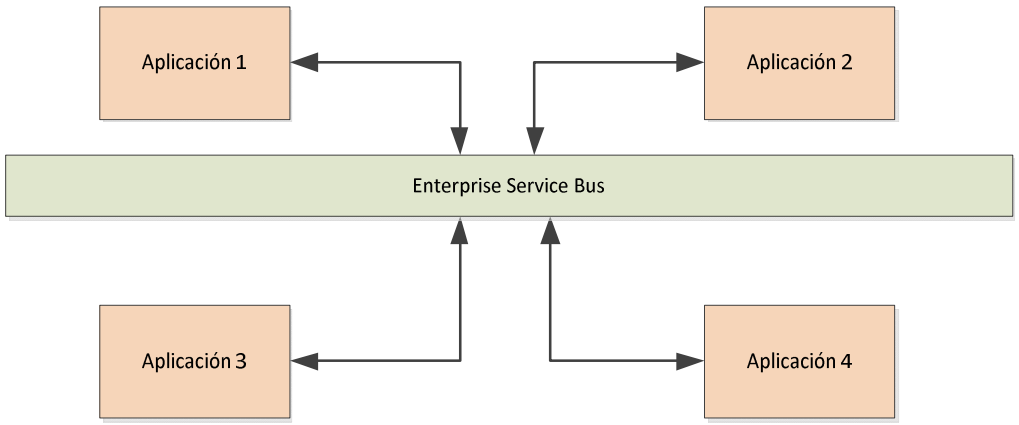



**Figura 5 - Enterprise Service Bus**

**[fuente: propia]**

Esta arquitectura provee un API, el cual puede ser utilizado para desarrollar e implementar servicios web que pueden interactuar unos con otros de forma fiable.

Entre las ventajas o beneficios que nos proporciona un ESB podemos encontrar:

- ✓ Los sistemas se acomodarán de forma más barata y sencilla.
- ✓ Mayor flexibilidad, es muy sencillo adaptarnos a nuevos requisitos.
- ✓ Basado en normas y estándares.
- ✓ Posibilidad real de escalar el sistema, un ESB nos servirá para dar desde soluciones puntuales hasta implementaciones complejas.
- ✓ Tipos de servicio listos para funcionar predefinidos.
- ✓ Mayor configuración en vez de tener que codificar la integración.
- ✓ Sin motor de normas central.

- ✓ Parches incrementales, tiempo de apagado instantáneo.

Entre las desventajas que podemos encontrar al utilizar un ESB están:

- ✓ Normalmente requiere un modelo de mensajería, lo cual implica una gestión adicional.
- ✓ Se necesita una administración constante en las versiones de los mensajes para asegurar un bajo acoplamiento. Una administración incorrecta, insuficiente o incompleta de las versiones de mensajes puede ocasionar un alto acoplamiento en lugar de uno bajo.
- ✓ Normalmente precisa más hardware que sistema de mensajes punto a punto.
- ✓ Se precisan conocimientos adicionales para configurar, administrar y operar un ESB.
- ✓ Existirá una latencia causada por los mensajes que atraviesan la capa extra del ESB, especialmente si se compara con comunicaciones punto a punto. La mayor latencia también se originará por un procesamiento extra de XML ya que el ESB normalmente lo utiliza como lenguaje de comunicación.
- ✓ El ESB se convierte en un elemento único de fallo.
- ✓ Aunque los sistemas basados en ESB pueden requerir un esfuerzo significativo para ser implementados, no producen ningún valor si no se desarrollan los servicios que deben usar el ESB.

Técnicamente, el ESB es un bus de mensajería que posee conversión de protocolos, transformación de formatos de mensaje, enrutado, aceptación y entrega de mensajes a los servicios y aplicaciones que utilizan el ESB.

Visto el punto anterior, ¿cuál es realmente la diferencia entre un ESB y el bus definido anteriormente? En la actualidad no existen diferencias notables entre ellos, la principal referencia entre un ESB y la creación de un bus propietario es el coste, es significativamente inferior en el caso de usar un ESB.
Las razones de este coste inferior son las siguientes:

- ✓ Un bus propietario ofrece muchas funcionalidades construidas que deberían ser desarrolladas en implementaciones basadas en ESB.
- ✓ Los buses propietarios utilizan en ocasiones formatos propietarios para mejorar el rendimiento, pero el uso de estos formatos propietarios incrementa el coste.

Por otro lado, un ESB está basado en estándares y debido a esto, su coste es inferior al uso de las otras topologías **hub and spoke** o un **bus** propietario.

## 2.3 Criterios a tener en cuenta a la hora de realizar un proceso de integración

Según el libro **Enterprise Integration Patterns**, los principales criterios de decisión a la hora de proceder con un proceso de integración son los siguientes:

- **Acoplamiento:** Las aplicaciones integradas deben reducir al mínimo sus dependencias, de este modo, cada una puede evolucionar sin causar problemas a las demás.
  Aplicaciones con un fuerte acoplamiento deben realizar numerosas suposiciones acerca de cómo funciona el resto, por este motivo, cuando cambiamos la aplicación y no se cumplen las suposiciones realizadas, ya no existe la integración.
  La interfaz utilizada para la integración de aplicaciones será lo suficientemente específica para implementar la funcionalidad útil, pero lo suficientemente genérica para permitir si es necesario que la aplicación cambie.

- **La integración debe ser simple:** Cuando se realiza la integración de una aplicación en una empresa, los desarrolladores deben tener en cuenta los siguientes principios:
  - ✓ Reducir al máximo los posibles cambios que deben efectuarse en la aplicación.
  - ✓ Reducir al mínimo la cantidad de código necesaria para realizar la integración.

  Aun así, los cambios del aplicativo o la creación de nuevo código pueden tener un impacto negativo, no siendo por tanto no ser el mejor camino para realizar la integración.

- **Tecnologías:** Las diferentes técnicas de integración requieren diferentes cantidades de software, así como un hardware especializado.
  Estas herramientas especiales pueden ser costosas, incluso pueden conducirnos a depender de un proveedor, también pueden originar un aumento de carga sobre los desarrolladores ya que deben entender cómo usarlas para integrar aplicaciones.

- **Formato de datos**: Las aplicaciones integradas deben ponerse de acuerdo sobre el formato de los datos que van a intercambiar, en caso de que no sea posible dicho acuerdo, al menos debe existir un traductor intermedio que permita unificar los diferentes formatos de datos utilizados por las aplicaciones.
  Un problema asociado con el formato de datos es que puede cambiar con el tiempo, debido a esto, la integración de las diferentes aplicaciones se verá afectada.

- **Tiempo de entrega de los datos**: La integración debe reducir al mínimo el tiempo existente entre el momento en que una aplicación publica los datos y las otras aplicaciones consumen esos datos.
  Los datos deberán ser intercambiados de forma frecuente en bloques pequeños, en vez de en un bloque enorme de ellos.
  Así mismo, tan pronto estén listos para ser consumidos, las aplicaciones que los están esperando deberán ser informadas. Por tanto, la existencia de un tiempo entre intercambios deberá ser tenida en cuenta a la hora de diseñar la integración, ya que cuanto más tiempo se tarde en consumir los datos, estos pueden estar obsoletos.

- **Compartir datos o funcionalidad**: Las aplicaciones integradas pueden no solo compartir datos, es posible que también deban compartir la funcionalidad de tal manera una aplicación pueda acceder a la funcionalidad de las demás.
  Cuando se comparte funcionalidad, la invocación de funcionalidad remota puede ser difícil de conseguir, ya que aunque pueda parecer igual que invocar funcionalidad local, realmente es muy diferente, pudiendo tener consecuencias significativas en el correcto funcionamiento de la integración.

- **Asincronía**: El funcionamiento de un programa informático suele ser síncrono, es decir, un procedimiento espera mientras que otro se ejecuta.
  En programación, un procedimiento deberá de estar disponible para cuando otro lo desee invocar, pero un procedimiento no está continuamente esperando por ser invocado por otro para ser ejecutado. También podemos querer invocar al procedimiento de forma asíncrona y que este se ejecute en segundo plano.
  En aplicaciones integradas, donde la aplicación remota no se está ejecutando o la red puede no estar disponible, el concepto de asíncrono es importante ya que la aplicación origen puede:

  - ✓ Querer únicamente que los datos estén disponibles para otras aplicaciones.
  - ✓ Querer que se ejecute la aplicación remota cuando esta se encuentre disponible.

## 2.4 Tipos de arquitectura

Las soluciones de integración son las encargadas de conectar múltiples sistemas externos unos con otros, realizando actividades tan variadas como transformación de datos, auditoría, procesamiento de mensajes…

La arquitectura EAI utilizada al desarrollar una solución de integración viene determinada por los siguientes parámetros:

- Número de componentes que deben trabajar juntos.
- Escalabilidad.
- Tolerancia a fallos y capacidad de recuperación ante los mismos.

Estos dos últimos son factores no funcionales, pero muy importantes a la hora de elegir una arquitectura.

Tenemos dos posibilidades de elección:

- **Solución Centralizada:** Agrupar todos los componentes que conforman nuestro sistema de integración en un único punto (host).
  Cuando la integración a realizar es pequeña, esta opción es perfectamente válida.
  Con este tipo de arquitectura podemos garantizar:

  - ✓ Alta escalabilidad. El sistema podrá incrementarse hasta el máximo que proporcione la plataforma.

  Por el contrario, utilizando una solución centralizada no disponemos de:

  - ✓ Balanceo de carga. Al no existir el sistema replicado, todas las solicitudes pasarán por el mismo punto.
  - ✓ Tolerancia a fallos. Si el sistema es centralizado, si cae la plataforma no tenemos ningún tipo de contingencia.

- **Solución Distribuida:** Ubicar los diferentes componentes que conforman nuestro sistema entre distintas plataformas. Este tipo de arquitectura es preferible a la arquitectura centralizada ya que con él se podrá garantizar:

  - ✓ Balanceo de carga. Al encontrarse el sistema en diferentes nodos o plataformas, las solicitudes pueden repartirse entre ellos, liberándolos de carga de proceso y por tanto, permitiéndonos optimizar más los recursos.
  - ✓ Alta tolerancia a fallos. Al tener el sistema en múltiples plataformas, si una de ellas cae, las otras seguirán disponibles. Para el resultado final es como si no hubiera ocurrido nada.
  - ✓ Alta escalabilidad. El sistema, al encontrarse en diferentes plataformas permitirá crecer en cada una de ellas.

## 2.5 Patrones de integración

El siguiente apartado nos mostrará los principales estilos según la clasificación dada por el libro **Java CAPS Basic: Implementing Common EAI Patterns**, la cual coincide con la propuesta en el libro **Enterprise Integration Patterns**.

### 2.5.1 Transferencia de ficheros

Es el patrón de integración más simple y básico. La comunicación entre aplicaciones se realiza a través de ficheros en el sistema de ficheros (local, ftp, sftp...)

En la siguiente figura podemos ver un esquema de funcionamiento:

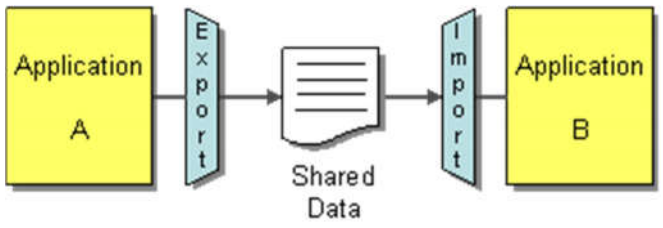



**Figura 6 - Transferencia de ficheros**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

Como podemos observar en la figura, existen dos componentes diferenciados en este patrón de integración:

- ✓ Productor: Genera los datos y una vez generados estos son escritos en el fichero.
- ✓ Consumidor: Recibe el fichero generado y procesa su contenido.

Para que este tipo de integración funcione, es muy importante que el fichero que se comparte sea entendible por las dos aplicaciones, es decir, el productor debe escribir los datos en un formato que entienda el consumidor.

Este estilo de integración puede ser válido y apropiado de ser usado bajo ciertas circunstancias, pero por naturaleza no es muy escalable ni robusto.

### 2.5.2 Compartir la base de datos

El uso de una base de datos compartida es otra técnica de integración sencilla que puede ser utilizada para la comunicación entre diferentes aplicaciones.

La siguiente figura muestra como sería el funcionamiento de este estilo de integración.

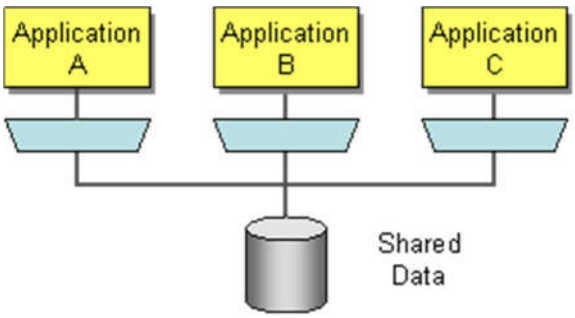



**Figura 7 - Compartir la base de datos**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

La idea de este estilo de integración es tener una sola base de datos compartida que actuará como un medio donde las diferentes aplicaciones puedan escribir (productores) y leer (consumidores).

El uso de una base de datos compartida es más adecuado para datos estructurales que la transferencia de archivos, en ella podremos definir la estructura de la tabla que actuará como una especie de formato de intercambio de mensajes.

Por el contrario, existe una limitación bastante importante en el uso de una base de datos compartida y es que esta puede convertirse en un cuello de botella, ya que múltiples aplicaciones estarán leyendo y escribiendo en la misma tabla, cada una dentro de su propia transacción, lo que puede generar un gran deadlock.

### 2.5.3 Invocación a procedimientos remotos

Distribuir la funcionalidad entre múltiples procesos independientes o hosts es lo que diferencia una aplicación monolítica de una distribuida.

La invocación de procedimientos remotos (RPC) es el mecanismo a través del cual, una aplicación puede invocar remotamente procedimientos expuestos por la otra aplicación.

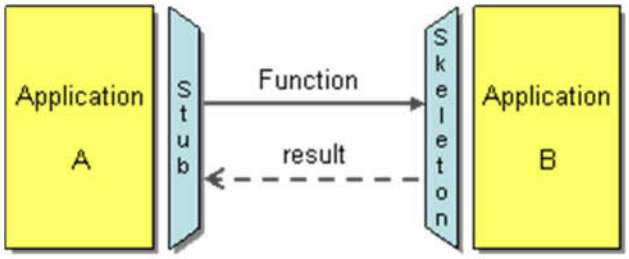



Figura 8 - Invocación a procedimientos remotos

[fuente: http://www.enterpriseintegrationpatterns.com]

El procedimiento al cual vamos a invocar deberá encapsulará algún tipo de funcionalidad de interés, la interfaz para invocarlo deberá estar publicada y ambas aplicaciones deberán usar el mismo protocolo, la misma representación de datos y la misma semántica.

En cuanto a los problemas que se presentan en la invocación a procedimientos remotos podemos encontrar:

- ✓ Baja tolerancia a fallos.
- ✓ Latencias en la invocación, ya que esta se realiza a través de la red.
- ✓ Escalabilidad.
- ✓ Disponibilidad, ya que puede estar caído el procedimiento al que vamos a invocar.

### 2.5.4 Orquestación de servicios

La orquestación de servicios se basa en la idea de construir servicios, publicar sus interfaces y usar protocolos estándar para encapsular y exponer alguna funcionalidad específica de la aplicación.

La idea es que los diferentes servicios pueden ser combinados a través de diferentes vías para conseguir la funcionalidad final deseada. Para el uso de dichos servicios, se utilizará HTTP como protocolo de comunicación lo cual facilita la invocación remota a través de la red, por esto son llamados Web Services.

Otro aspecto importante es que los datos serán representados utilizando XML (entrada, salida o error), así mismo, la invocación a un servicio siempre será síncrona.

Utilizando esta aproximación, para obtener la funcionalidad requerida por la empresa deberá ser a través de la invocación apropiada de los diferentes servicios publicados y para realizar esta tarea necesitaremos un orquestador.

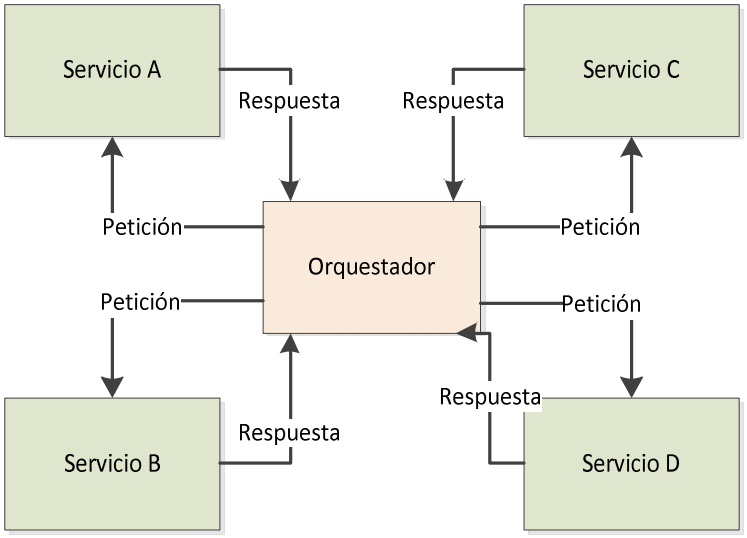



**Figura 9 - Orquestador de Servicios**

**[fuente: propia]**

Aun así, el uso de Web Services tiene el mismo problema que la invocación a procedimientos remotos, sobre todo:

- ✓ Baja tolerancia a fallos.
- ✓ Latencias en la invocación, ya que esta se realiza a través de la red.
- ✓ Escalabilidad.
- ✓ Disponibilidad, ya que puede estar caído el Web Service al que vamos a invocar.
- ✓ Integridad transaccional.
- ✓ Acoplamiento entre el cliente y el servicio que proporciona la funcionalidad.

Además, los servicios web se van a desplegar sobre redes no seguras, como Internet, lo cual añadirá problemas de seguridad e incrementará los problemas de latencia, veracidad y disponibilidad ya existentes.

Para intentar solventar los problemas anteriores, se ha intentado un complejo proceso de estandarización para proporcionar garantías de calidad de servicio en el uso de servicios web, las cuales no estaban incluidas en el diseño original de SOAP.

### 2.5.5 Mensajes

Uno de los principales inconvenientes de los estilos anteriormente comentados es que son síncronos, es decir, requiere que los dos sistemas estén preparados y funcionando para poder operar.

Debido a lo anterior, debemos concluir que la mensajería asíncrona es una aproximación más práctica a la integración de aplicaciones, ya que el envío

de un mensaje no requiere que ambos sistemas estén preparados y funcionando al mismo tiempo.

En la siguiente figura veremos el funcionamiento de un sistema de mensajería asíncrona.

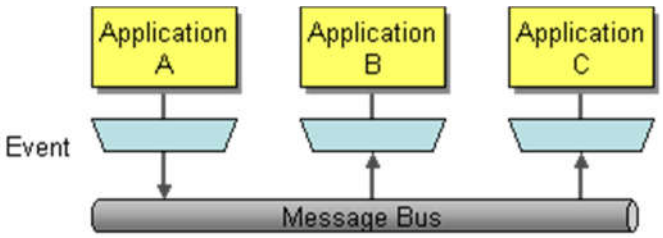



**Figura 10 - Mensajería asíncrona**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

Además, el pensar en comunicación asíncrona hace que los desarrolladores reconozcan que trabajar con aplicaciones remotas es más lento, lo que fomenta el diseño de componentes con alta cohesión y de baja adherencia.

Podemos concluir por tanto, que el estilo de integración por mensajes es el mejor estilo para resolver la mayoría de los problemas de integración, así como la base de muchas de las soluciones EAI existentes.

Cada uno de los patrones definido anteriormente intenta solucionar el problema de integrar aplicaciones y contextos muy similares.

Cada patrón intenta buscar la solución más elegante para realizar dicha integración, pero aún que tienen el mismo propósito, lo que realmente les diferencia es que cada patrón posee un enfoque más sofisticado que intenta hacer frente a las deficiencias de los otros, por tanto, el orden refleja un mayor grado de sofisticación.

La elección del patrón depende de cada tipo de integración, cada uno tiene sus ventajas y desventajas. Dos aplicaciones pueden ser integradas utilizando diferentes patrones, y cada integración, utiliza el patrón que se adapta mejor a sus necesidades. De igual forma, una aplicación puede elegir el patrón o patrones que mejor funciona para integrarse con otras aplicaciones, es decir, podemos tener un hibrido de patrones.

Los productos de integración o middleware EAI pueden emplear una combinación de estos patrones, los cuales estarán ocultos en el producto.

## 2.6 Patrón de integración mediante de mensajes

Como vimos en el apartado anterior, el patrón de integración mediante mensajes es el mejor para resolver la mayoría de los problemas de integración existentes, en nuestro caso, va a servir de base para construir el test sobre el que se basa este proyecto.

El contenido de este apartado ha sido tomado de la web http://www.enterpriseintegrationpatterns.com/, realizando únicamente ampliaciones del mismo cuando era necesario.

En primer lugar se definirán los conceptos básicos que se encuentran asociados al patrón.

- **Canales:** Cuando realizamos una integración entre aplicaciones, estás deben comunicar los mensajes a través de un medio que conecte al emisor y el receptor.

- **Mensajes:** El mensaje será la unidad de datos que se transmitirá a través del canal. Si los datos son muy grandes, el emisor podrá dividirlos en fragmentos más pequeños que emitirá a través del canal, cada uno de estos fragmentos es un mensaje.

  Al otro lado del canal estará el receptor, el cual recibe los mensajes, extrae los datos y los procesa.

- **Entrega indirecta**: El funcionamiento normal ante un envío de mensajes sería de entrega directa, es decir, el emisor manda el mensaje y el receptor lo lee directamente del canal.

  Esto sería la situación ideal, pero no es realmente así, en numerosas situaciones, entre el envío y la recepción del mensaje existen otros pasos intermedios como validación, transformación… los cuales deben preparar el mensaje antes de poder ser entregado en destino.

- **Enrutado**: En una gran empresa, con numerosas aplicaciones y canales que las conecta, los mensajes deberán ir a través de diferentes canales antes de llegar a destino. En este caso, el emisor no sabe a quién debe enviar el mensaje, además existen números canales a los que emitir, no sabiendo si el que escucha es el receptor real del mensaje.

  En este caso, el emisor envía el mensaje a un enrutador, esta pieza decidirá si envía el mensaje al receptor final o si debe enviarlo a algún paso intermedio como por ejemplo validación o transformación, esperando la respuesta de este paso intermedio y decidiendo a quien debe enviárselo después hasta que llegue a destino.

- **Transformación**: Las diferentes aplicaciones que se van a integrar seguramente no van a poseer el mismo formato de datos, el emisor enviará el mensaje con un formato determinado y el receptor espera otro diferente.

  Ante esta casuística, es imposible que podamos integrar dichas aplicaciones ya que los formatos que entienden son diferentes, para solventar este problema necesitamos una pieza adicional, un transformador, el cual se encargará de traducir el formato de la aplicación origen al de la de destino.

- **Endpoints**: Las aplicaciones, por si solas no pueden interactuar de forma directa con el sistema de mensajería, por tanto, deberá existir una capa que sepa cómo funciona la aplicación y el sistema de mensajería. El objetivo de esta capa intermedia es que los dos (aplicación y sistema de mensajería) funcionen juntos, permitiendo enviar y recibir mensajes.

A continuación veremos los detalles de un sistema de mensajería según la clasificación dada por el libro **Enterprise Integration Patterns**.

- **Canal de mensajes:** Cuando dos aplicaciones separadas que necesitan comunicarse, estas deberán ser conectadas utilizando un canal. Cuando se utiliza un canal podemos identificar los siguientes agentes:

  - ✓ Productor, será quien escribe el mensaje en el canal de comunicación.
  - ✓ Canal de transmisión del mensaje.
  - ✓ Receptor del mensaje, será el encargado de leerlo del canal de comunicación.

  En la siguiente figura podemos ver una representación simple un canal de mensajería.

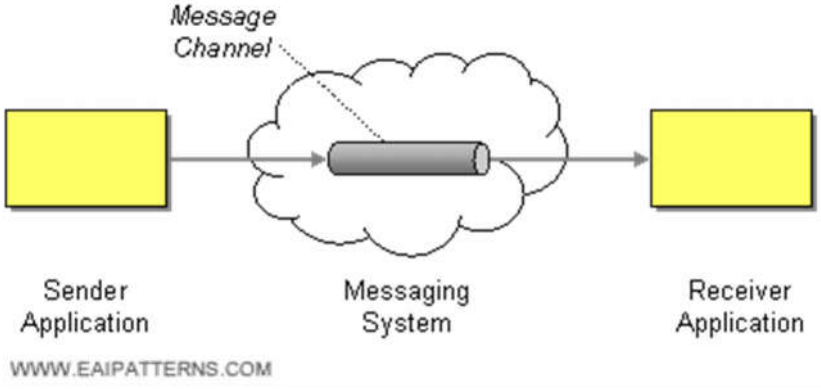



**Figura 11 - Canal de mensajes**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

- **Mensajes**: Dos aplicaciones que comparten información se comunicarán entre ellas utilizando mensajes y usarán un canal de comunicación para transmitirlos y/o recibirlos.

Cualquier tipo de datos que las aplicaciones a integrar vayan a compartir utilizando el patrón **Mensaje**, deberán ser transmitidos en uno o varios mensajes, bien sea fragmentando dichos datos (varios mensajes) o sin fragmentar (uno solo).

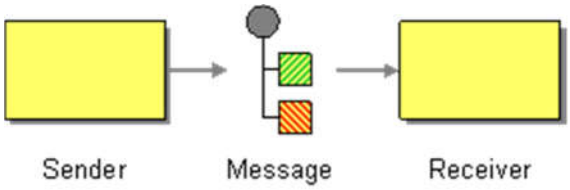



Figura 12 - Mensajes

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Filtros y tuberías**: En muchos escenarios de integración, podemos definir el concepto de filtro-tubería como una secuencia de pasos, cada uno con una función específica. Es decir, realizamos un procesamiento complejo, dividiendo dicho procesamiento en partes más sencillas.

  Un ejemplo de la afirmación anterior sería el siguiente proceso, ante una orden de transferencia bancaría en formato plano, se necesita que esta sea traducida a un formato XML para poder ser emitida. En este ejemplo sencillo podemos identificar los siguientes pasos:

  - ✓ Validación de la orden de transferencia.
  - ✓ Transformación de dicha orden de formato plano a formato xml.
  - ✓ Validación de la orden transformada.
  - ✓ Emisión de la transferencia.

  Como podemos observar, ante la recepción de la transferencia, el proceso que debemos ejecutar, es un proceso largo, compuesto de varios pasos y no podemos saltar al siguiente hasta que el anterior haya terminado.

  Visto el ejemplo anterior, si queremos dividir dicho proceso en otros más pequeños, podemos concluir que la aplicación de patrón filtro-tubería se adapta perfectamente. Cada filtro expondrá una interfaz sencilla para su utilización.

  El funcionamiento del patrón es muy sencillo, un filtro recibe el mensaje de entrada, lo procesa y envía el mensaje resultante al pipe, para que lo lea el siguiente filtro.

  La conexión entre el filtro y el pipe es llamado puerto, de forma sencilla podemos decir que cada filtro posee un puerto de entrada y otro de salida.

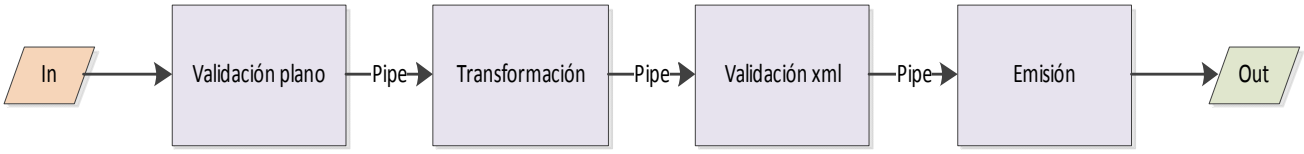



**Figura 13 - Filtro-tubería**

**[fuente: propia]**

- **Enrutador de mensajes**: La función principal de un enrutador es consumir un mensaje desde una entrada y en base a la evaluación de algún tipo de condición, ser capaz de enviar el mensaje al destino correspondiente.
  Es un concepto diferente al presentado anteriormente de filtro-tubería, en este caso también tendremos una entrada, pero en vez de tener un único punto de salida tenemos múltiples.

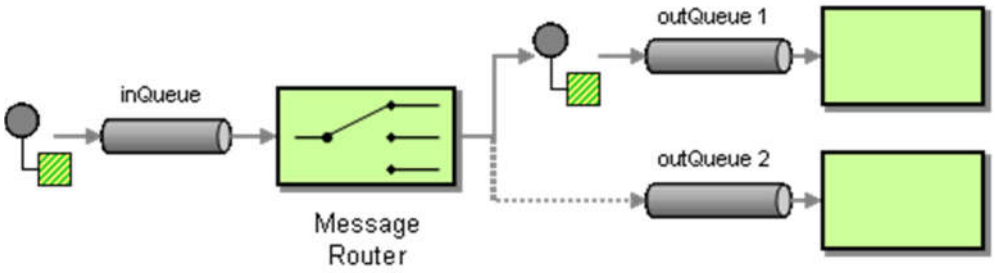



**Figura 14 - Enrutador de mensajes**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

Un ejemplo enrutador, basándonos en el ejemplo de transferencia comentado en el punto anterior podría ser, tener dos tipos de transacciones diferentes que se reciben por el mismo canal de comunicación. En ambos casos el proceso es similar, validación, transformación, validación de la transformación y emisión, pero necesitamos una pieza previa que identifique que tipo de transacción es para enviarla al proceso correcto.

- **Traductor de mensajes**: Las aplicaciones que deseamos integrar son muy diferentes entre sí, propietarias, de terceros, de propósito general…, en ocasiones, incluso aplicaciones externas que esperan datos nuestros para poder operar.

  El problema principal radica en que cada una de estas aplicaciones tiene su propio formato de datos, algunas pueden usar estándares, pero por lo normal cada una utilice el suyo propio.

  Los formatos que utilizan las aplicaciones son muy variados, ficheros, bases de datos, XML…

  En esta situación, debemos utilizar una pieza que se encargue de realizar la traducción de datos entre aplicaciones, de esta forma pueden entenderse entre ellas.

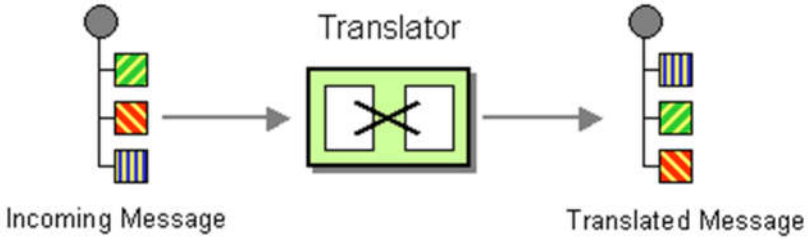



Figura 15 - Traductor de mensajes

[fuente: http://www.enterpriseintegrationpatterns.com]

Un ejemplo en el uso de diferentes tipos de datos utilizados entre aplicaciones pueden ser dos aplicaciones bancarias, una utiliza el CCC para solicitar la emisión de transferencias, pero tiene que solicitárselo a otra que utiliza el IBAN bancario para emitirlas, y ambas deben comunicarse para poder realizar la transferencia.

- **Endpoint**: Esta pieza será la encargada de conectar la aplicación con el sistema de mensajería, realmente, un Endpoint no es más que un fragmento de código existente en una aplicación que conoce como poder interactuar con el sistema de mensajería para poder emitir o recibir mensajes.

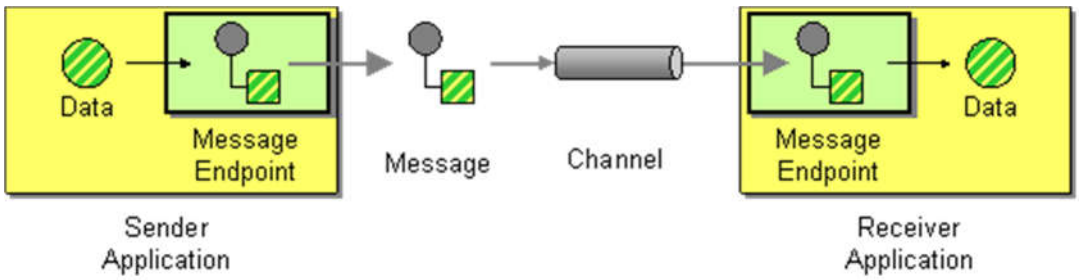



Figura 16 - Endpoint

[fuente: http://www.enterpriseintegrationpatterns.com]

Para finalizar este apartado, comentaremos como funciona un Endpoint, existen dos formas, para emitir y para recibir.
En una emisión, el Endpoint actúa de la siguiente forma:

1. Recibe los datos que deben ser emitidos.
2. Construye el mensaje con los datos.
3. Emite el mensaje al canal.

En una recepción, el funcionamiento varía:

1. Recibe el mensaje que fue emitido.
2. Extrae los datos.
3. Los entrega los datos a la aplicación para que los procese.

A la hora de definir un sistema de mensajería el primer punto del que hemos hablado han sido los **canales de mensajería**, los cuales son utilizados por las diferentes aplicaciones que necesitan intercambiarse datos, es decir, dos aplicaciones que necesitan intercambiarse datos están conectadas a través de un canal.

La aplicación que va a enviar los datos puede no conocer quién es la aplicación que los va a recibir, lo que si conoce es el canal a través del cual debe enviarlos para que los reciba la aplicación destino.

Decidir si vamos a utilizar o no canales de comunicación es sencillo, se usarán si dos aplicaciones tienen datos que comunicar a través de un canal, el reto real es saber cuántos vamos a necesitar.

- **Número fijo de canales**: Por lo general, el conjunto de canales que se utilizarán tiene a ser fijo. Cuando un desarrollador construye una aplicación conoce que tipos de datos se van a enviar y cuales se van a recibir desde otras aplicaciones.

  Los canales que se van a utilizar para esta emisión o recepción no pueden ser creados dinámicamente, deben de ser conocidos a la hora de realizar la implementación, por tanto, el número de canales a utilizar suele ser fijo.

  Existe un caso en el que si existe y es ventajoso el uso de canales dinámicos, esta situación es cuando se utilizan canales en modo petición-respuesta (request-reply).

- **Determinar el conjunto de canales**: El definir el conjunto de canales que se va a utilizar es un proceso iterativo.

  - ✓ En primer lugar, las aplicaciones serán las que marquen que canales van a utilizar para comunicarse entre ellas.
  - ✓ El resto de solicitudes adicionales intentarán usar los canales existentes, pero si esto no es posible, se añadirán nuevos canales.
  - ✓ Si tenemos un conjunto de aplicaciones que utilizan un conjunto de canales determinado, pero añadimos nuevas aplicaciones, estas puede que usen el conjunto existente, o bien, necesitan agregar nuevos.

- **Canales unidireccionales**: Una fuente de confusión es si un canal es unidireccional o bidireccional. Técnicamente en un canal las aplicaciones ponen datos y otras aplicaciones los leen.

  Los canales deberán ser unidireccionales, una aplicación envía los datos en la dirección que marca el canal y otra, un receptor, los leería. Si el canal fuera bidireccional, el emisor sería a su vez receptor con lo cual podría llegar a consumir sus propios mensajes, lo cual no tiene mucho sentido.

Debido al párrafo anterior, donde se indica que los canales son unidireccionales, si dos aplicaciones necesitan establecer una comunicación bidireccional, necesitan dos canales.

Ahora que conocemos como funcionan los canales de comunicación vamos a ver que decisiones debemos tomar a la hora de utilizarlos.

- **Canal uno a uno o uno a muchos**: Cuando las aplicaciones comparten datos, necesitamos saber si quien va a recibir estos datos es una aplicación fija o un conjunto de aplicaciones. Gracias a este conocimiento podemos definir el tipo de conexión a utilizar.

    - ✓ Si necesitamos que el receptor de los datos sea una única aplicación, utilizaremos canales de tipo punto a punto. Este tipo de canales no nos garantiza que los datos lleguen siempre a la misma aplicación, ya que el canal puede tener múltiples receptores, pero si nos garantiza que solo va a existir un receptor para los datos emitidos.

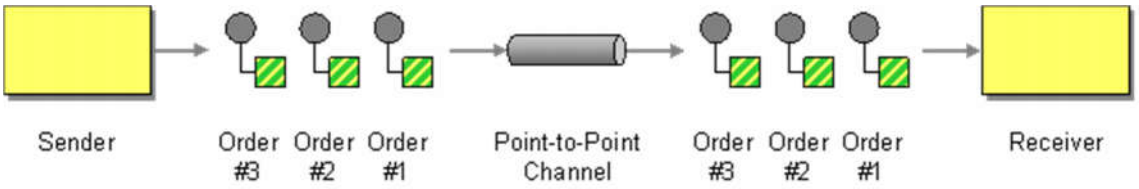



Figura 17 - Canal punto a punto

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ Si lo que necesitamos es que los datos enviados lleguen a muchos receptores a la vez, utilizaremos un canal de tipo publish-subscribe (topic). En este tipo de canales, existirá una aplicación emisora que envía los datos y múltiples receptores que reciben una copia de los mismos.

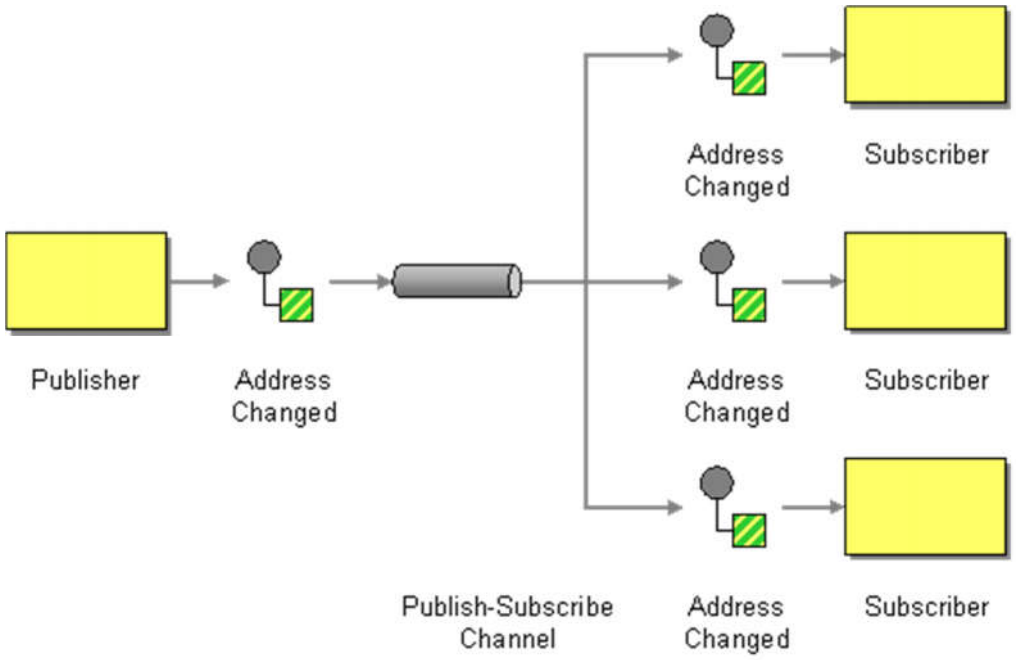



Figura 18 - Canal Publish-Subscriber

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Tipo de datos del canal**: En un sistema de mensajería, el contenido del mensaje que vamos a intercambiar entre las diferentes aplicaciones deberá poseer un tipo de estructura que sea capaz de entender el receptor.
  Existe un principio básico a la hora de mandar mensajes por el canal y que todos los mensajes que se emiten deben ser del mismo tipo. Esta es la principal razón de porque un sistema de mensajería necesita muchos canales; por otro lado, si los datos pudieran ser de cualquier tipo, el sistema únicamente necesitará un canal en cada sentido entre emisor y receptor.

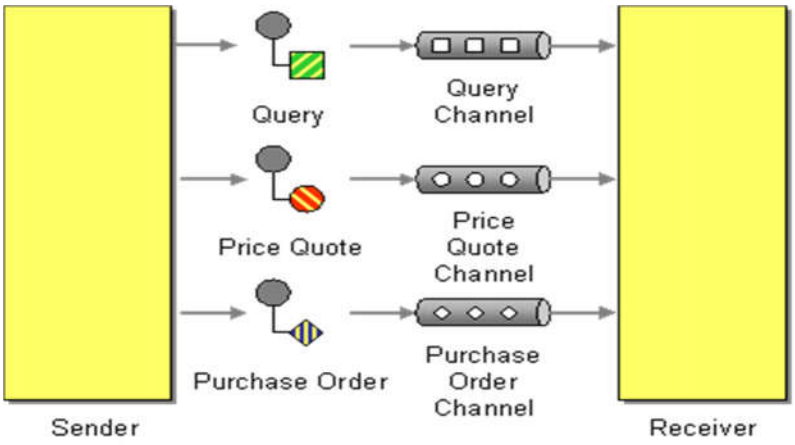



Figura 19 - Tipos de datos asociados al canal

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Mensajes inválidos o expirados**: El sistema de mensajes puede garantizar que un mensaje ha sido entregado, pero no puede garantizar que el receptor sepa procesarlo.

  Cuando un receptor no sabe qué hacer con un mensaje que ha recibido, en principio lo desecha, pero lo que se podría hacer el receptor es reenviar dicho mensaje a un canal dedicado exclusivamente para estos casos (Dead Letter).

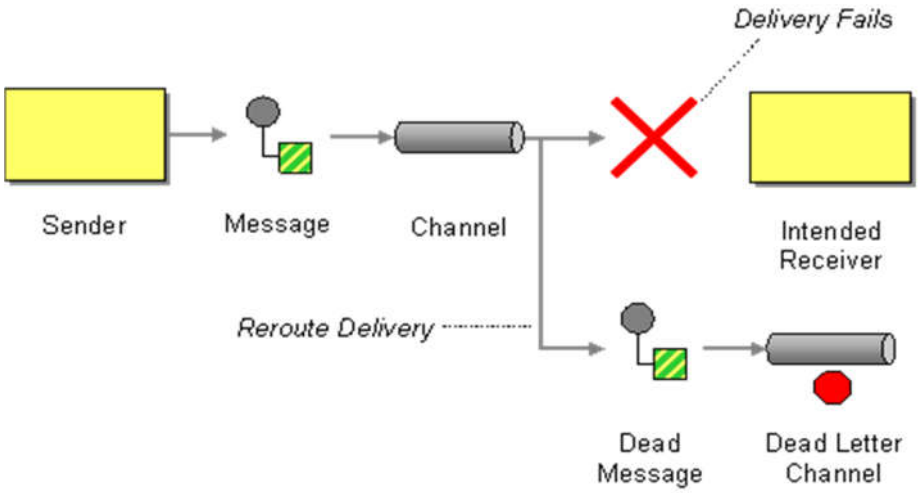



Figura 20 - Dead letter

[fuente: http://www.enterpriseintegrationpatterns.com]

Si utilizamos este mecanismo para tratar mensajes erróneos, necesitaremos tener un mecanismo de gestión que sepa qué hacer con estos mensajes erróneos.

- **Protección frente a caídas**: ¿Si el sistema se cae o se bloquea, que ocurrirá con los mensajes en vuelo? En primera instancia, un sistema de mensajería no es seguro debido a que los mensajes únicamente están almacenados en memoria, sin embargo, si el sistema persistiera sus mensajes en disco o en base de datos evitaremos el problema de perder los mensajes afectando rendimiento, pero ganando fiabilidad.

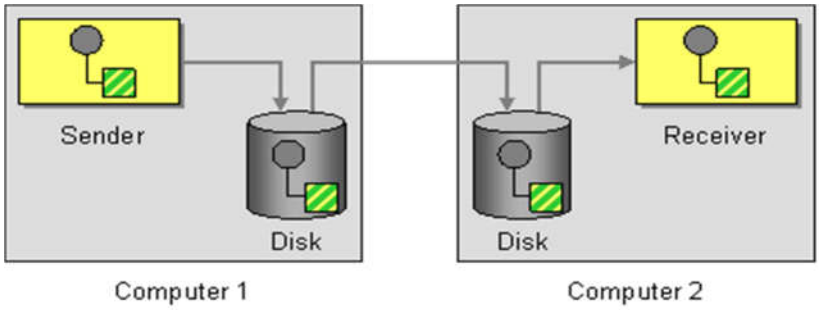



Figura 21 - Protección frente a caídas

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Clientes que no pueden acceder al sistema de mensajería**: ¿Qué ocurriría si una aplicación no puede conectarse al sistema pero quiere utilizarlo? Normalmente no existiría opción, pero si el sistema de mensajería publicara alguna forma de acceso, API, TCP/IP, base de datos, una interfaz…, un adaptador sobre el sistema de mensajería podría ser utilizado para conectar la aplicación sin necesidad de modificarla.

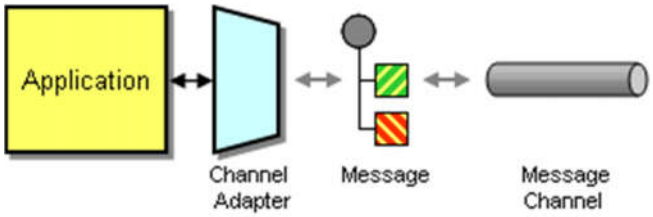



Figura 22 - Adaptador de comunicaciones

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Comunicaciones**: A medida que cada vez más aplicaciones se conectan al sistema de mensajería y comienzan a enviar mensajes para comunicarse, el sistema de mensajería comienza a ser una pieza central de la empresa. Una aplicación nueva que desee acceder al sistema, necesitará conocer que canales utilizar para solicitar la ejecución de una funcionalidad y de cuales debe leer los resultados.

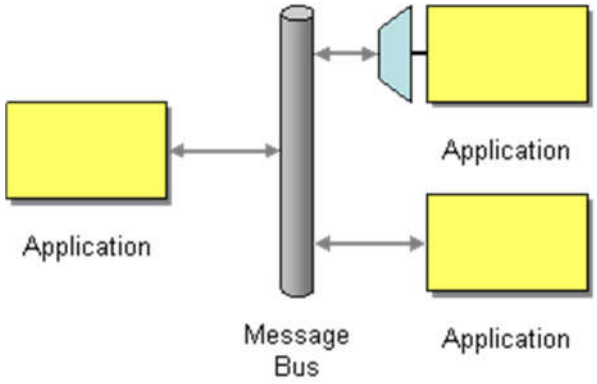



Figura 23 - Sistema de mensajería en bus

[fuente: http://www.enterpriseintegrationpatterns.com]

Podemos ver por tanto, que el sistema de mensajería se va a convertir en la herramienta permite acceder a la funcionalidad proporcionada por las distintas aplicaciones.

Como podemos observar, una integración basada en mensajes no es solo crear las aplicaciones y conectarlas a un sistema de mensajería, requiere tener en consideración otros muchos factores que van a influir en el funcionamiento.

El siguiente punto que vimos en la introducción de este patrón son los **Mensajes**. Un mensaje, es la unidad de intercambio de información entre dos aplicaciones que desean compartir datos.

El uso de mensajes para intercambiar datos entre aplicaciones hace que surjan una serie de cuestiones.

- **Intención del mensaje**: En última instancia, los mensajes contienen los datos que queremos intercambiar, pero este intercambio de datos puede tener diferentes intenciones como por ejemplo:

  ✓ Enviar un mensaje de tipo comando. El emisor envía un mensaje en el que se indica una función o método que debe ejecutar el receptor.

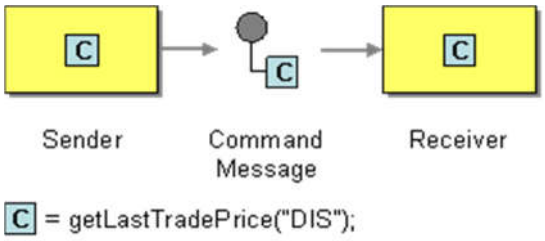



Figura 24 - Mensaje de tipo comando

[fuente: http://www.enterpriseintegrationpatterns.com]

  ✓ Enviar un mensaje de tipo documento. El emisor envía un mensaje que contiene una estructura de datos al receptor.

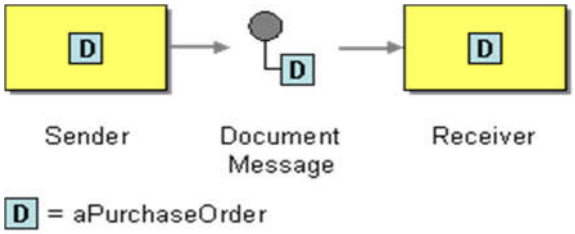



Figura 25 - Mensaje de tipo documento

[fuente: http://www.enterpriseintegrationpatterns.com]

  ✓ Enviar un mensaje de tipo evento. Mediante este tipo de mensaje el emisor notifica de un cambio al receptor pero no le indica cómo actuar ante ese cambio, simplemente le informa.

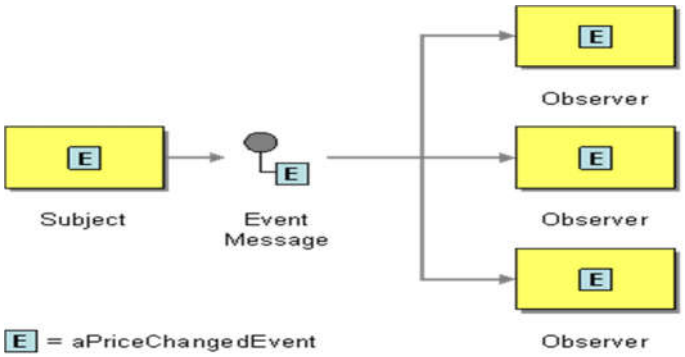



Figura 26 - Mensaje de tipo evento

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Retorno de una respuesta**: Cuando un emisor envía un mensaje, lo que en muchas ocasiones espera es una respuesta indicando que el mensaje ha sido procesado y el resultado de dicho procesamiento.

  Cuando ocurre esto, decimos que estamos ante un escenario solicitud-respuesta, donde el emisor generalmente envía un mensaje de tipo comando y espera un mensaje de tipo documento que contenga el resultado o la excepción que ha ocurrido durante el proceso.

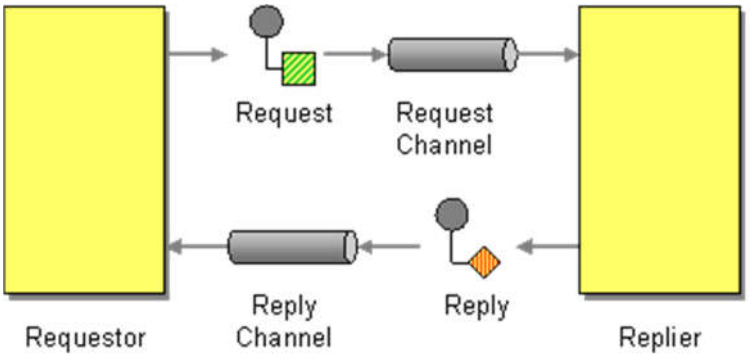



Figura 27 - Petición-Respuesta

[fuente: http://www.enterpriseintegrationpatterns.com]

El emisor deberá incluir en su mensaje una dirección, de esta forma, el receptor sepa donde enviar la réplica.

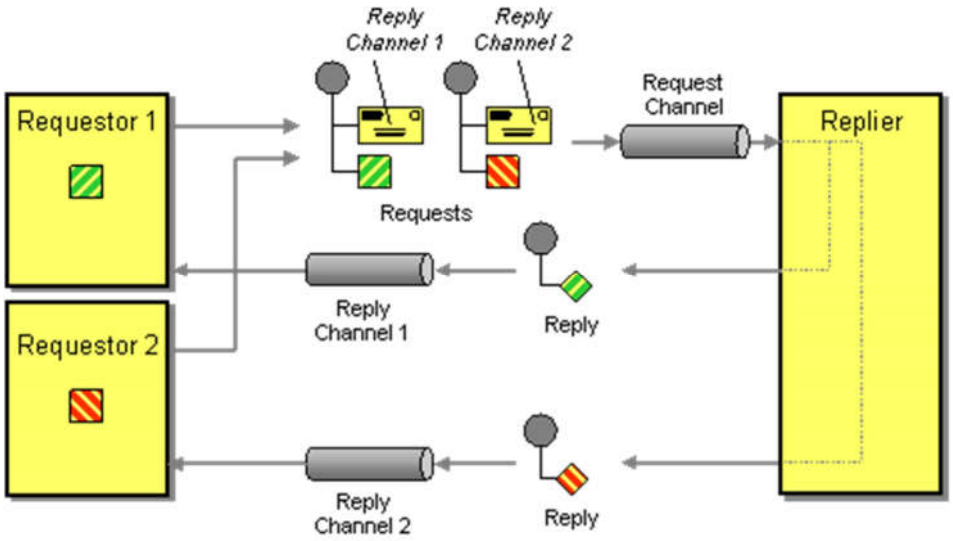



**Figura 28 - Dirección de retorno de respuesta**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

También puede ocurrir que el emisor tenga varias solicitudes pendientes y por tanto deberá recibir varias réplicas, así que la respuesta recibida debe indicar a que solicitud está respondiendo.

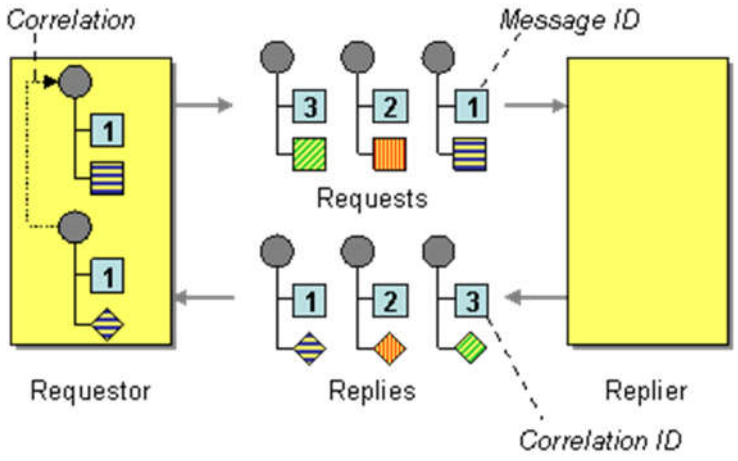



**Figura 29 - Identificación de la respuesta**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

Existen dos escenarios en un protocolo solicitud-respuesta:

- ✓ En el primero, mensajería RPC (Invocación a procedimiento remoto) el solicitante realiza la invocación de una función, pero no solo quiere que la función se ejecute, sino que también se le retorne el resultado de la misma.
- ✓ El segundo escenario, consulta, el emisor envía un mensaje con una consulta al receptor y espera que este le responda con un mensaje a dicha consulta.

- **Volumen de los datos**: Los mensajes son utilizados para que las diferentes aplicaciones intercambien datos entre ellas. Pueden existir dos situaciones:

  - ✓ La cantidad de datos es pequeña y cabe en un solo mensaje.
  - ✓ La cantidad de datos que se pretende intercambiar es tan grande que no cabe en un único mensaje.
    En un caso como el anterior, se debe proceder de la siguiente forma:

    1. Los datos deberán ser fragmentados en distintos mensajes.
    2. Los mensajes generados serán enviados de forma secuencial.
    3. Al ser enviados como secuencia, el receptor podrá recomponer la estructura original y recuperar los datos.

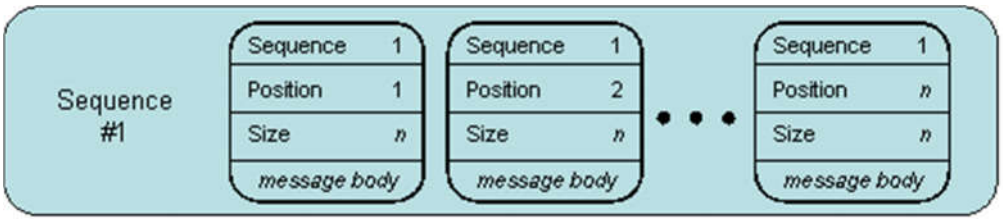



**Figura 30 - División en múltiples mensajes**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

- **Lentitud en la entrega de mensajes**: Uno de los mayores problemas que tenemos a la hora de trabajar con mensajes es que el emisor, no sabe cuándo va a recibir el receptor el mensaje, puede ocurrir el caso, donde los datos que contiene el mensaje solo son válidos en un determinado espacio de tiempo y si el mensaje no es recibido en este intervalo, debería ser ignorado en caso de recibirse posteriormente.

  Para resolver esta situación utilizamos la fecha de expiración, esta fecha marca el tiempo máximo de vida que poseerá el mensaje. Si un mensaje se encuentra expirado, bien el receptor o el sistema de mensajería lo ignorarán.

  A continuación se mostrará una ilustración en la que podremos visualizar este principio.

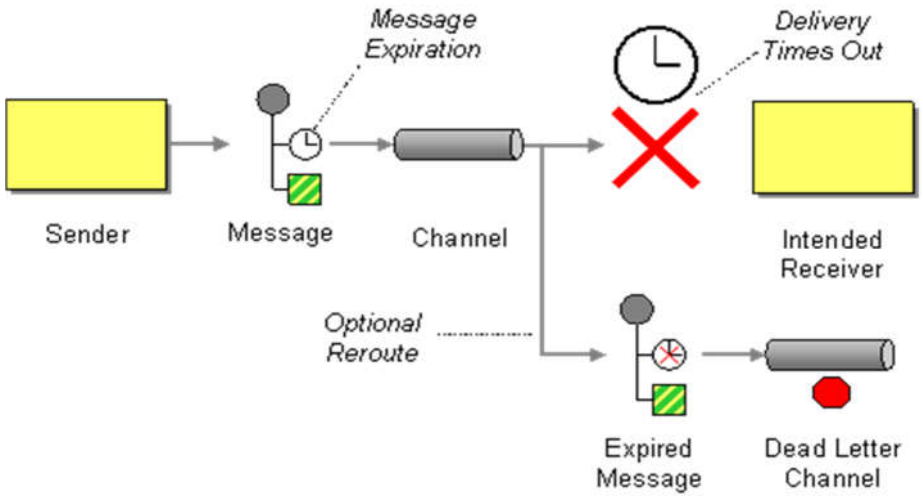



Figura 31 - Mensaje expirado

[fuente: http://www.enterpriseintegrationpatterns.com]

A continuación explicaremos otro de los componentes existentes en un sistema de integración mediante mensajes, este componente son los **Enrutadores**, para los cuales vamos a definir las siguientes categorías:

- Enrutadores simples: Son los encargados de distribuir los mensajes desde un canal de entrada a uno o varios canales de salida.
- Enrutadores compuestos: Son una combinación de múltiples enrutadores simples formando un flujo más complejo.
- Message Broker: El cual recibe mensajes de diferentes aplicaciones y los enruta hacia el destino correcto.

Continuaremos explicando cada una de las categorías y comentando que tipos de enrutadores existen dentro de ellas.

- **Enrutadores Simples:** Dentro de los enrutadores simples podemos encontrar:

  - ✓ **Enrutadores basados en el contenido:** Este tipo de enrutadores inspeccionan el contenido del mensaje y dependiendo de él, envían el mensaje a un canal determinado. El funcionamiento sería, el emisor envía el mensaje a un canal simple, el enrutador analiza su contenido y decide a que canal destino debe enviarlo, en ese canal destino estará escuchando el receptor.

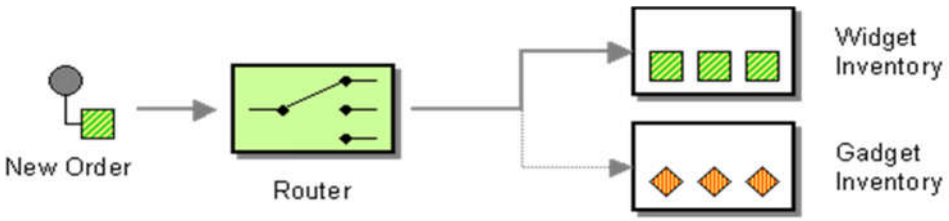



Figura 32 - Enrutador basado en contenido

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Filtro de mensajes**: Este tipo de enrutador podemos decir que es un tipo particular de enrutador basado en el contenido del mensaje. En este caso, el filtro examina el contenido y pasa el mensaje a otro canal si su contenido cumple cierto criterio, en otro caso, lo descarta.

  Un filtro de mensajes puede ser usado en un canal de tipo **Publish-Subscribe**, el cual enrutará los mensajes válidos a los destinatarios correspondientes y filtrará los mensajes irrelevantes.

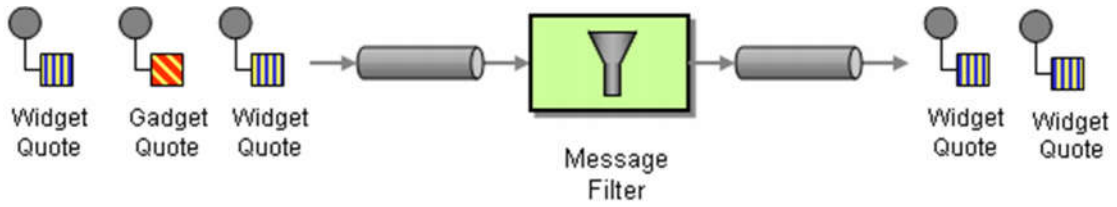



**Figura 33 - Filtro de mensajería**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

- ✓ **Enrutador dinámico:** Mientras que un enrutador básico usa reglas fijas para saber quién es el destinatario de un mensaje, pero cuando necesitamos una mayor flexibilidad, debemos utilizar un **enrutador dinámico**.

  Este tipo de enrutadores nos permite modificar la lógica de enrutamiento mediante el envío de mensajes de control a un puerto determinado.

  Cuando un receptor se encuentra activo, envía un mensaje de control indicando que está activo y la lista de condiciones que debe cumplir un mensaje para que este sea entregado a ese receptor. Estas condiciones son almacenadas en una base de reglas, las cuales son evaluadas cuando llega un mensaje para poder enrutarlo.

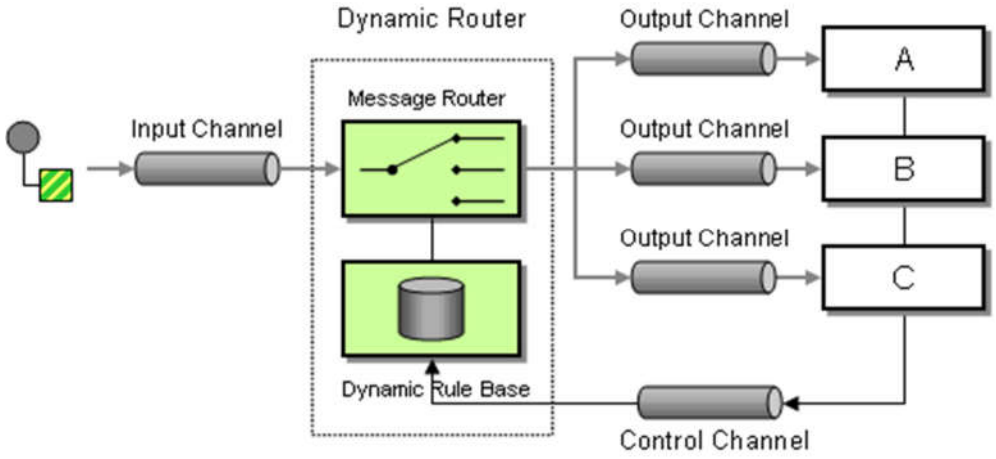



Figura 34 - Enrutador dinámico

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Lista de recipientes:** En ciertas situaciones, tenemos uno o más de un destinatario para un mensaje y queremos tener un control sobre estos destinatarios, para esto es para lo que necesitamos una lista de recipientes.

  Por lo tanto, una lista de recipientes no es más que un enrutador basado en contenido, el cual enrutará un mensaje simple a uno o más canales de destino.

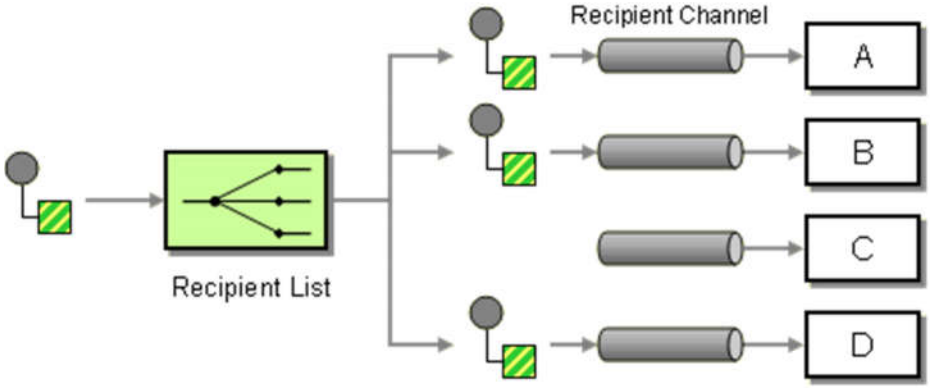



Figura 35 - Lista de recipientes

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Splitter:** Muchos mensajes contienen en su interior múltiples elementos, los cuales deben ir cada uno a un receptor distinto, es decir, un mensaje largo será procesado en partes individuales y estas entregadas a cada receptor correspondiente, es decir, el Splitter consume el mensaje, procesa cada elemento por separado y envía cada fragmento al destino correspondiente.

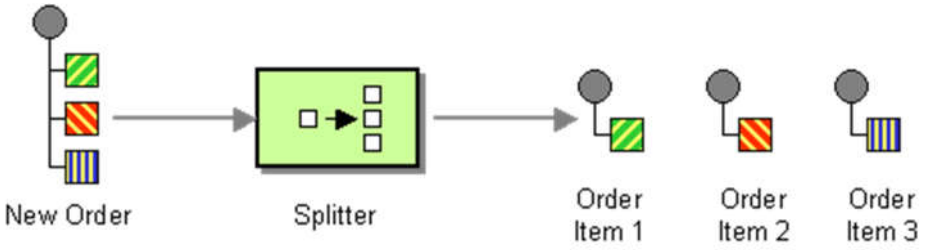



Figura 36 - Splitter

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Agregador:** Igual que tenemos un splitter, el cual se encarga de dividir el mensaje grande en fragmentos, en algún momento necesitaremos realizar el paso contrarío, es decir, coger cada uno de los fragmentos combinándolos en un mensaje simple.

  Para realizar esta acción utilizaremos un **Agregador**, el cual consumirá múltiples mensajes antes de publicar el mensaje real.

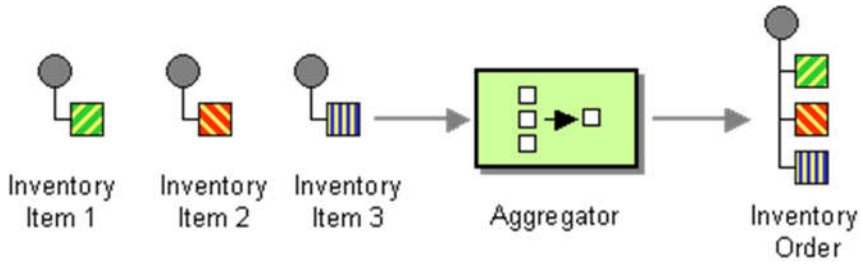



Figura 37 - Agregador

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Resecuenciador**: Al utilizar el modelo de mensajes para interconectar aplicaciones que se encuentran funcionando en múltiples entornos, con lo cual, estos mensajes pueden ser procesados en paralelo.

  Al procesar los mensajes en paralelo, algunos pueden ser procesados más rápido que otros, lo cual hará que el orden de dichos mensajes no sea mantenido, pero en ocasiones, se requiere que el orden de los mensajes sea mantenido.

  Para conseguir esto utilizaremos un **resecuenciador**, que será el encargado de volver a poner los mensajes en el orden correcto.

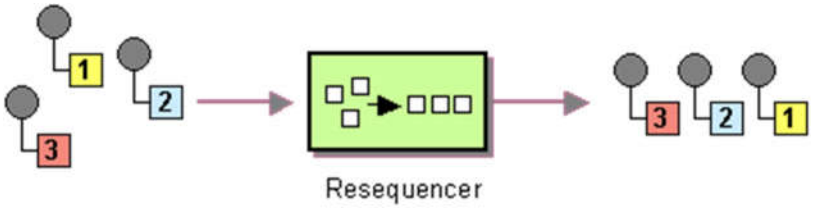



Figura 38 - Resecuenciador

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Enrutadores compuestos:** La ventaja del patrón filtro-tubería nos permite componer numerosos filtros para tener un procesamiento más largo.
  Vamos a definir tres variantes de este tipo de enrutador, donde ambos, reciben información de múltiples fuentes y generando un mensaje simple con ella.

  ✓ **Procesador de mensajes compuestos:** El procesador de mensajes compuestos se deberá utilizar para tratar mensajes simples que estén compuestos por diferentes sub-mensajes. Este tipo de enrutador estará compuesto generalmente por los siguientes elementos:

  - Un Splitter, que será el encargado de dividir el mensaje compuesto en los diferentes sub-mensajes que lo componen.
  - Un Enrutador, el cual se encargará de enviar cada uno de los diferentes sub-mensajes al destino adecuado.
  - Un Agregador, que recibirá los diferentes sub-mensajes que ya están procesados y los unificará en un nuevo mensaje simple.

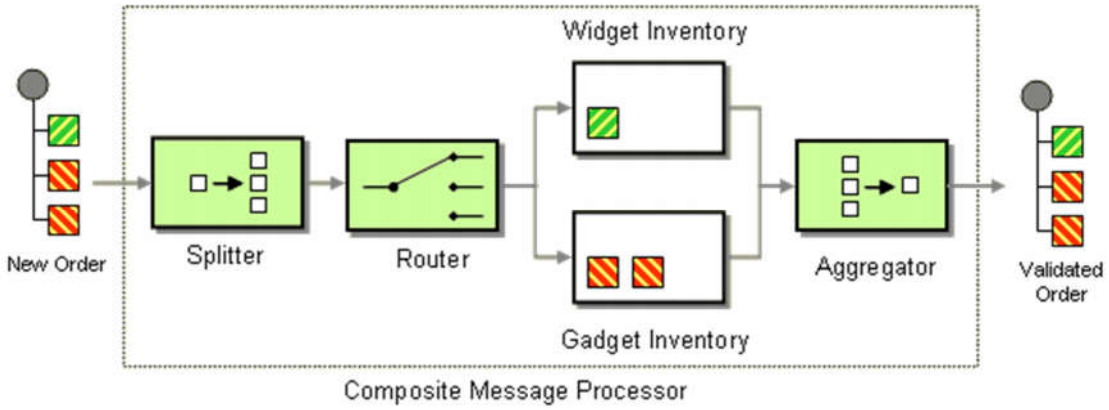



Figura 39 - Procesador de mensajes compuestos

[fuente: http://www.enterpriseintegrationpatterns.com]

✓ **Scatter-Gather:** Usaremos un Scatter-Gather cuando vamos a enviar un mensaje como broadcast a múltiples destinatarios y unificar posteriormente sus respuestas en un mensaje simple.

Por tanto, simplificando el texto anterior, el Scatter-Gather enviará un mensaje a varios receptores, que lo procesarán, para posteriormente utilizar un Agregador unificándolo en un solo mensaje.

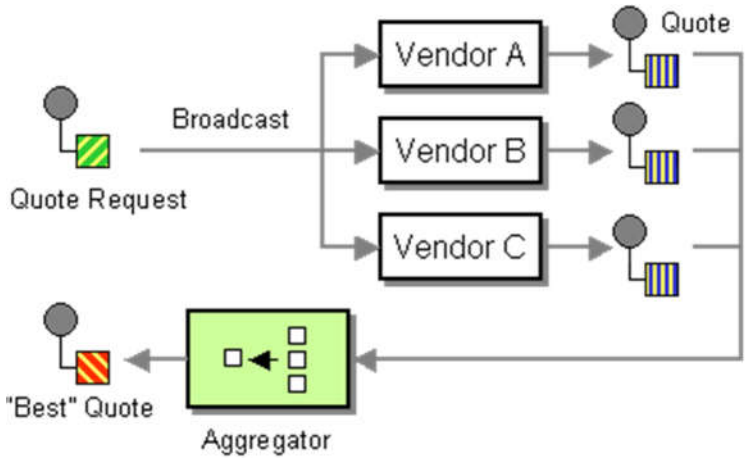



Figura 40 - Scatter-Gather

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Tabla de enrutamiento:** Hasta ahora, todos los tipos de enrutadores vistos, reciben un mensaje simple de entrada y lo envían de alguna manera a uno o a varios receptores.
  Pero existen casos en los que en vez de enviar el mensaje, debemos encauzarlo a través de diferentes componentes.
  En estos casos necesitaremos agregar una tabla de enrutamiento a cada mensaje, la cual especifica la secuencia de procesamiento que debe seguir el mensaje.

  Cada componente que trata el mensaje, una vez tratado, leerá la tabla de rutas y enviará el mensaje al siguiente paso de procesamiento.

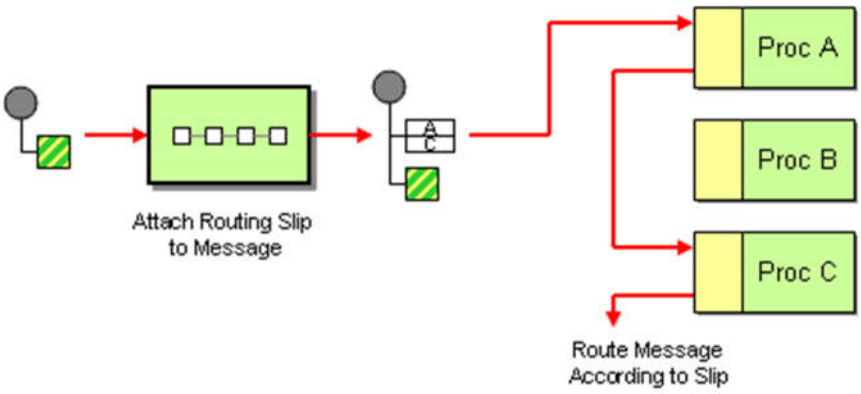



Figura 41 - Tabla de enrutamiento

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Administrador de proceso:** El tipo de enrutamiento anterior, nos muestra cómo podemos enviar un mensaje dinámicamente a través de diferentes pasos de procesamiento.
  El problema que tiene es que es lineal y está compuesto por una secuencia bien definida.

  Un **administrador de proceso** es diferente, utilizado también para enviar el mensaje a los diferentes pasos de procesamiento, pero permitiendo mantener la secuencia y pudiendo determinar el siguiente paso en base a los resultados intermedios de dicho proceso.

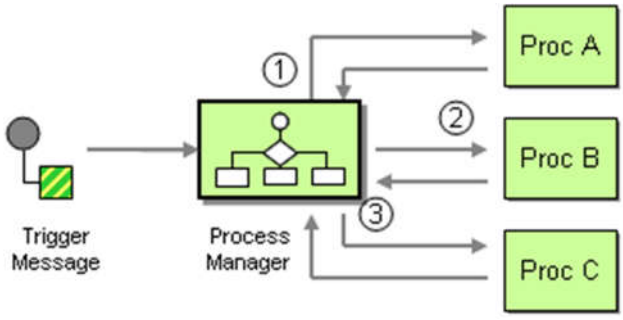



Figura 42 - Administrador de proceso

[fuente: http://www.enterpriseintegrationpatterns.com]

- ✓ **Message Broker**: Un bróker de mensajería es una pieza central que puede recibe mensajes de múltiples destinos, determina el destino correcto y enruta el mensaje al canal correcto.

  La implementación del bróker utilizará los principios explicados en los puntos anteriores. Este tipo de patrón resuelve un gran número de problemas de integración existentes.

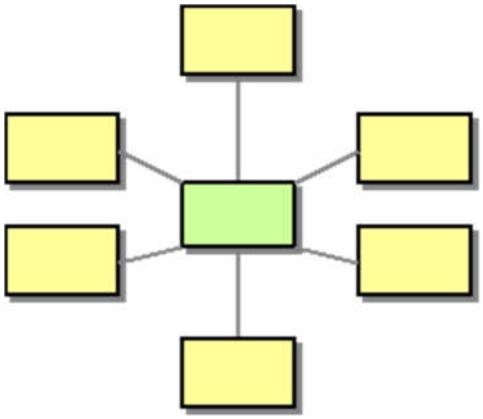


Figura 43 - Broker de mensajería

[fuente: http://www.enterpriseintegrationpatterns.com]

Para finalizar esta introducción a los diferentes tipos de enrutador, mostraremos una figura con las reglas de decisión que debemos aplicar a la hora de seleccionar un enrutador para resolver el problema.

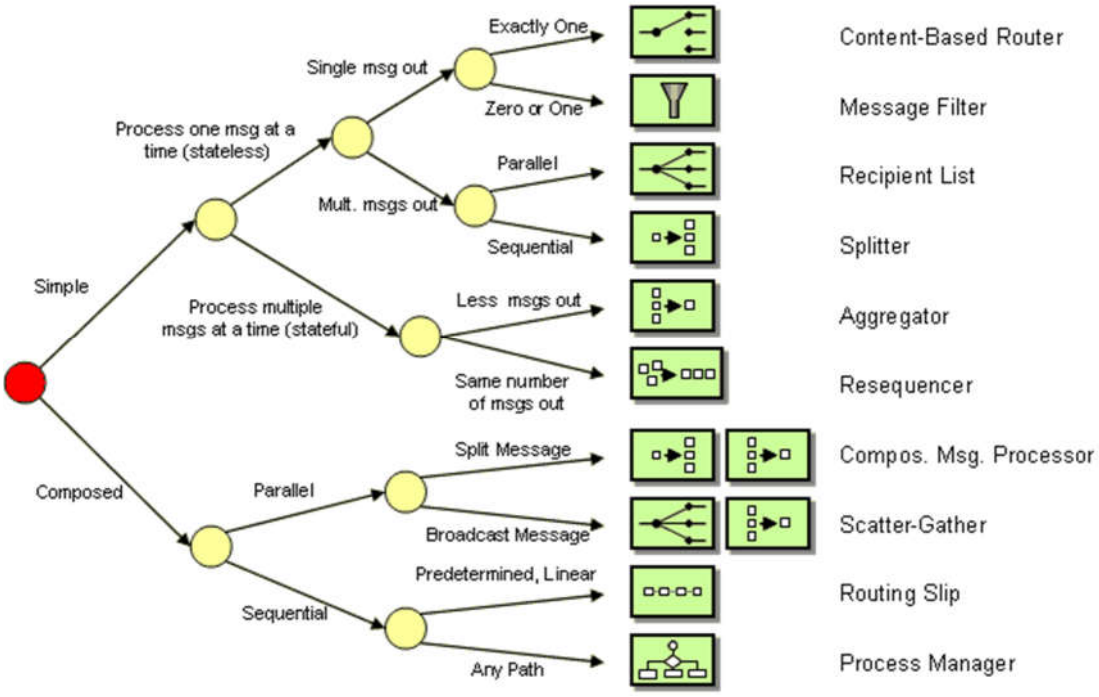



**Figura 44 - Reglas de decisión de patrones**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

El siguiente punto que debemos tener en cuenta a la hora de hablar de integración a través de mensajes, debemos tener un formato común que entiendan todas las aplicaciones que vamos a integrar.

El formato que las diferentes aplicaciones a integrar utilizan un formato interno que raramente va a coincidir con el formato común utilizado para realizar la integración, es en este punto donde entra en juego el traductor de mensajes, el cuál funciona genial en para solventar problemas de diferencia de formatos, ya que por lo general, las aplicaciones que se integrarán no permiten ser modificadas para adaptar el formato que usan internamente al formato común utilizado en el proceso.

Además, numerosas aplicaciones requiere que se ubiquen ciertos datos en la cabecera del mensaje.

Entre los diferentes elementos que podemos encontrar a la hora de trabajar con traductores de mensajes tenemos.

- **Envolvente del mensaje**: Un gran número de sistemas dividen el mensaje en cabecera y cuerpo. En estos casos, la cabecera contendrá datos relativos al manejo del flujo de procesamiento y aunque la mayoría de los aplicativos no sabrá de la existencia de estos datos adicionales, otros pueden considerar el mensaje erróneo, ya que no existen en el mensaje que espera el aplicativo.

Por otro lado, los aplicativos encargados de realizar el enrutado del mensaje utilizarán estos campos adicionales para realizarlo, considerando inválido el mensaje si no los tiene.

Lo que se hará será incluir los datos del aplicativo en una envolvente que es compatible con el sistema de mensajería, que es extraído del envolvente cuando llega a destino.

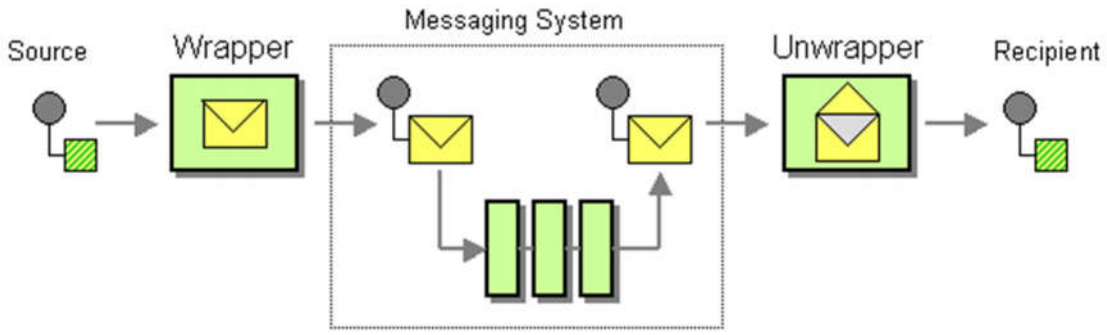



Figura 45 - Envolvente

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Enriquecedor de contenido**: Cuando enviamos un mensaje desde un sistema a otro, lo más normal es que el mensaje que debe llegar al sistema destino deba contener más información de la que es proporcionada por el sistema origen.

  Para solventar este problema utilizaremos un transformador especializado, el cual, conectará con una fuente de datos externa proporcionando al mensaje la información adicional que necesita al llegar a destino.

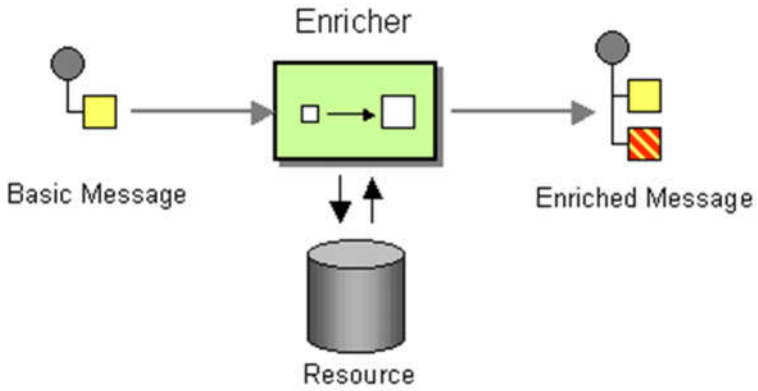



Figura 46 - Enriquecedor de contenido

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Filtro de contenido:** El enriquecedor de contenido nos sirve cuando un mensaje requiere más o diferentes datos de los que el creador provee. Pero existen situaciones en las que la situación es la inversa, es decir,

necesitamos eliminar datos del mensaje y para ello utilizaremos el filtro de contenido.

Un **filtro de contenido** no solo sirve para eliminar contenido innecesario del mensaje, sino que también se utilizará para simplificar la estructura del mensaje, explicaremos esto a continuación, generalmente, muchos mensajes tienen una estructura en forma de árbol, pero al pasar por las diferentes aplicaciones, se irán acumulando datos repetidos, niveles innecesarios…, debido a esto, puede que sea necesario aplanar la estructura con el fin de simplificarla y es por esto por lo que utilizaremos un **filtro de contenido**.

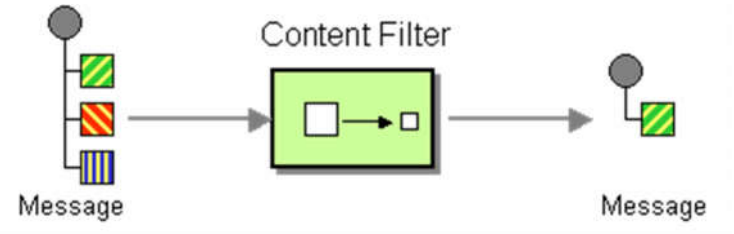



Figura 47 - Filtro de contenido

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Claim Check**: Un enriquecedor de contenido nos permitirá agregar datos que se encuentran olvidados o que son necesarios en un mensaje. El filtro de contenido nos permite eliminar datos superfluos del mismo, pero en ocasiones, únicamente queremos eliminar datos del mensaje de forma temporal, por ejemplo, porque estos datos pueden corromperse en un determinado paso de procesamiento.

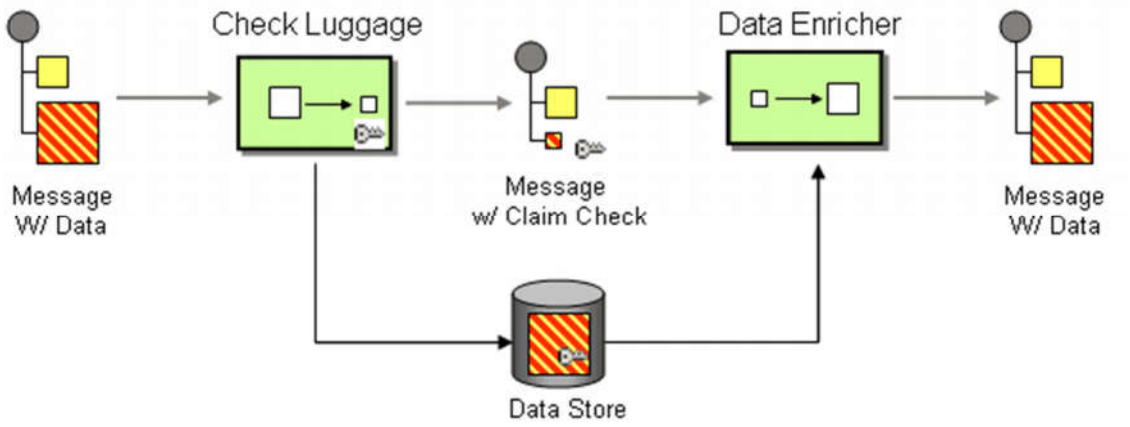



Figura 48 - Claim Check

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Normalizador**: En un escenario de integración es muy común recibir mensajes de terceros, los cuales pueden contener el mismo significado

pero tener diferentes formatos, dependiendo de los diferentes sistemas que poseen las empresas que envían dichos mensajes.

Utilizaremos un **Normalizador** para enrutar cada uno de los diferentes mensajes al traductor correspondiente, de forma que el resultado de esta traducción sea un mensaje en un formato común.

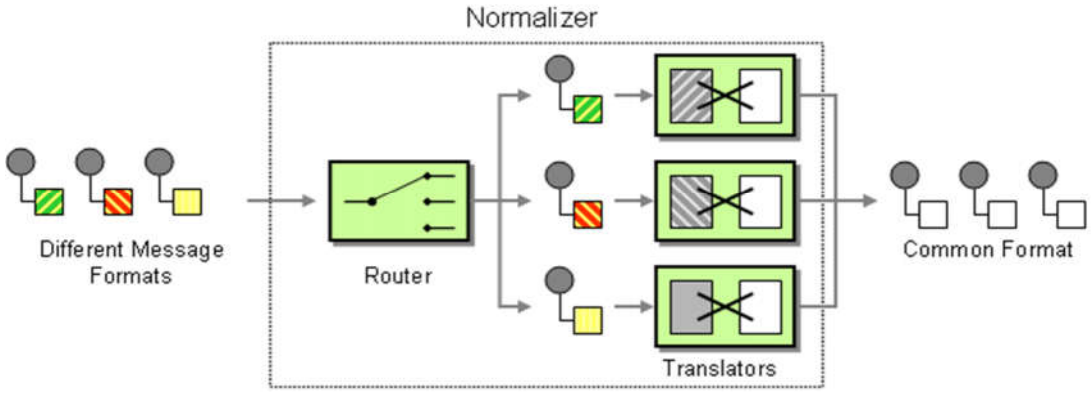



Figura 49 - Normalizador

[fuente: http://www.enterpriseintegrationpatterns.com]

Como podemos observar en la figura, el normalizador utiliza un enrutador, el cual, recibe el mensaje de entrada y decide a que traductor enviarlo para generar el mensaje en formato común correspondiente.

- **Modelo de datos canónico**: Cuando diseñamos aplicaciones para trabajar utilizando menajes, cada una de las cuales puede tener su propio formato interno.

  Por tanto, si vamos a diseñar un modelo canónico de datos, debemos conseguir que este sea independiente de cualquier aplicación que utilicemos, además, una vez definido, debemos conseguir que dada aplicación genere y consuma mensajes que cumplan el patrón común que hemos definido.

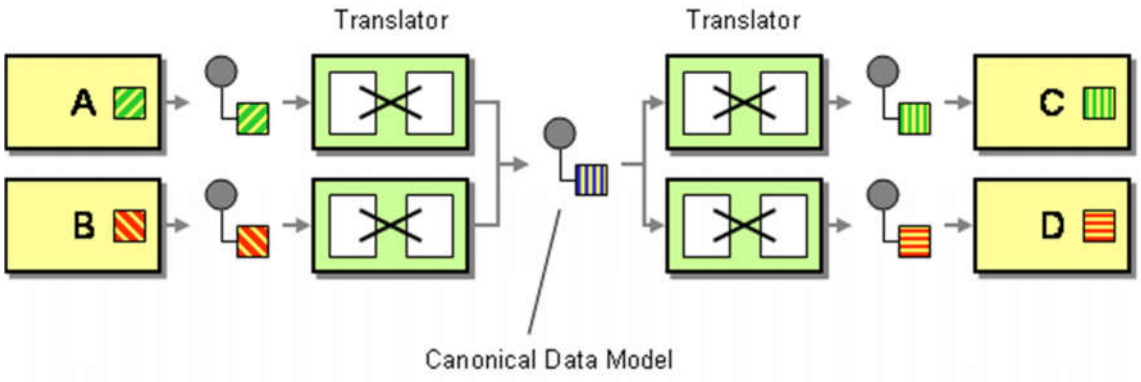



Figura 50 - Modelo de datos canónico

[fuente: http://www.enterpriseintegrationpatterns.com]

Como podemos observar en la imagen, el funcionamiento será el siguiente, la aplicación origen genera un mensaje en su formato propietario, este mensaje es traducido al modelo canónico a través de un traductor.

El mensaje en formato canónico, una vez tratado, generará también un mensaje en formato canónico que será enviado al traductor correspondiente para convertir el formato canónico al formato propietario que espera la aplicación destino.

El último punto vamos a comentar antes de finalizar esta sección son los **Endpoints**. Como comentamos antes, un endpoint es un mecanismo que permite conectar una aplicación al sistema de mensajería para que esta pueda enviar y recibir mensajes.

- **Patrones de envío y recepción**: Algunos patrones de endpoint pueden ser aplicados tanto a endpoints de emisión como de recepción.

  - ✓ **Encapsular el código del mensaje**: Una aplicación no debería ser consciente de que está usando mensajería para integrarse con otras aplicaciones. No se debería tener en cuenta que la aplicación va a utilizar mensajería para integrarse con otras.

    Cuando la integración es implementada utilizando mensajería, existirá una capa de software que unirá la aplicación con el sistema de mensajería.

  - ✓ **Traducción de datos**: Si las aplicaciones origen y destino compartieran el mismo formato interno y el mensaje a intercambiar también fuera el mismo, la integración sería muy simple.

    El problema es que raramente esto no es así, el emisor o el receptor usan formatos diferentes o el formato de mensaje que se intercambia es diferente de los formatos internos. En esta situación utilizaremos un mapeador de mensajes para convertir los datos entre las aplicaciones y el sistema de mensajería.

  - ✓ **Transacciones controladas externamente**: Los sistemas de mensajería utilizan sistemas de transacción internos.

    Externamente, cada emisión o recepción de mensajes tendrá su propia transacción, por otro lado, los consumidores y receptores de mensajes podrán utilizar un cliente transaccional para controlar estas transacciones externamente, lo cual es muy útil cuando queremos trabajar por lotes o para coordinar mensajes con otros servicios.

- **Patrones de consumo de mensajes**: Otros patrones únicamente pueden ser aplicados a receptores de mensajes.

Un punto muy importante es el ratio de consumo de mensajes, como se habló en la introducción del capítulo, es que un alto volumen de peticiones de mensajes puede hacer caer el servidor y la idea es intentar controlar ese ratio, pero con los sistemas de mensajería el servidor no puede controlar el volumen al que los clientes envían peticiones al servidor, pero si podemos controlar el ritmo al cual el sistema consumirá dichos mensajes.

El sistema tiene por tanto que consumir los mensajes a un ritmo sostenible, estableciendo una relación adecuada entre velocidad de proceso y carga aceptable, además, si el servidor requiriera más velocidad de proceso, tendría opción de incrementar la velocidad y procesar más mensajes.

- ✓ **Consumidor síncrono o asíncrono**: Una alternativa es si usaremos un poll de consumidores, o bien, un consumidor basado en eventos.

  Un poll nos produce un mayor cuello de botella, pero evita que el servidor no se sobrecargue, ya que si el poll de consumidores se encuentra exhausto, los mensajes se encolarán hasta que haya consumidores libres que puedan procesarlos.

  Un consumidor basado en eventos consumirá el mensaje tan rápido como llegue, por tanto la llegada de numerosos mensajes puede sobrecargar el servidor, pese a esto, cada consumidor únicamente puede consumir un mensaje, por lo tanto, una limitación en el número de consumidores genera un cuello de botella.

- ✓ **Asignación vs Grab de mensajes**: Cómo se va a manejar el procesamiento de los mensajes. Si cada consumidor obtiene un mensaje, estos pueden ser procesados de forma paralela.

  El enfoque más simple es que los diferentes consumidores compitan entre ellos por obtener los mensajes y procesarlos, si deseáramos controlar quién debe procesar cada mensaje, deberíamos incluir un distribuidor de mensajes, este nos permite crear un cuello de botella limitando el número de consumidores a los que entregar los diferentes mensajes que se van recibiendo.

- ✓ **Filtro de mensajes o aceptar todos los mensajes**: El funcionamiento por defecto es el siguiente, cuando un emisor pone un mensaje en un canal, cualquier receptor que este escuchando en ese canal puede recibirlo.

  Sin embargo, no todos los consumidores quieren recibir cualquier mensaje, algunos consumidores desearán únicamente consumir algunos mensajes, más en concreto aquellos que coincidan con un

determinado **Selector**, el cual actuará como filtro, no permitiendo recibir aquellos mensajes que no lo cumplan.

- ✓ **Subscripción mientras estas desconectado**: Un problema que existe en un sistema Publish-Subscriber, es el siguiente, ¿que ocurre si un subscritor que está interesado en los mensajes que se van a recibir, esta justamente desactivado en el momento de que el mensaje sea publicado?

  De forma predeterminada, estos mensajes no serán entregados a ese subscriptor, aun así, si declaramos el subscritor como Persistente solventaremos esta problemática.

- ✓ **Idempotencia**: En ocasiones, el mismo mensaje se enviará más de una vez, en ocasiones porque el sistema no sabe si el mensaje se ha entregado satisfactoriamente, o bien, porque ha bajado la calidad del canal.

  Por otro lado, el receptor suele estar construido para recibir el mensaje únicamente una vez, habiendo problemas en el caso de que el mensaje se reciba más de una vez. Un receptor idempotente, permite tratar estos mensajes duplicados y evitando que den problemas.

- ✓ **Servicios síncronos o asíncronos**: Se deberá también decidir si la aplicación expondrá los mensajes de forma síncrona (llamada a procedimiento remoto) o asíncrona (utilización de un sistema de mensajería). Pero no solo tenemos que quedarnos en usar uno u otro enfoque, podemos mezclarlos, es decir, un cliente síncrono puede activar el mensaje mediante la invocación remota mientas que un cliente asíncrono puede invocarlo enviando un mensaje.

A continuación explicaremos los diferentes conceptos que se han comentado en los párrafos anteriores.

- **Gateway de mensajería:** Un Gateway encapsula el código específico del tratamiento del mensaje y lo separa del resto del código del aplicativo. De esta forma logramos encapsular el código encargado de interactuar con el sistema de mensajería del resto de código que posee la aplicación.

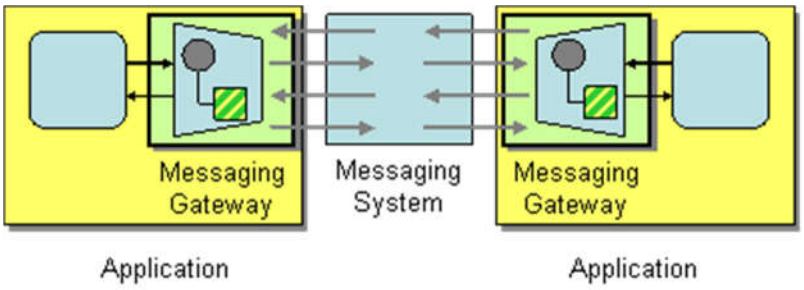



Figura 51 - Gateway de mensajería

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Mapeador:** Cuando integramos aplicaciones usando mensajería, los datos dentro del mensaje en ocasiones son derivados de objetos del dominio de la aplicación.

  Si usamos un Mensaje de tipo Documento, el mensaje en sí mismo puede directamente representar uno o varios objetos del dominio, si usamos un Mensaje de tipo Comando, algunos de los campos de datos es muy probables que también estén extraídos de los objetos del dominio. Aun así, existen diferencias entre los mensajes y los objetos de domino.

  Ahora existe la duda principal, ¿Cómo podemos mover datos entre los objetos del dominio y los mensajes, manteniendo a la vez la independencia entre ellos?

  Para evitar el problema, se deberá crear un mapeador separado de ambos ámbitos y que contiene la lógica de mapeo entre los mensajes y los objetos del dominio, pero ni los mensajes ni los objetos de dominio tendrán conocimiento de él.

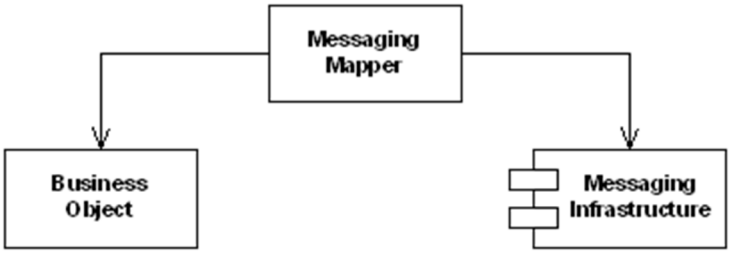



Figura 52 - Mapeador

[fuente: http://www.enterpriseintegrationpatterns.com]

El mapeador accederá a uno o más objetos de dominio y los convertirá en los mensajes que requiere el canal. También realizará la operación inversa, creará o actualizará los objetos del dominio basándose en mensajes de entrada.

- **Clientes transaccionales:** Un sistema de mensajería, por defecto, utiliza transaccionalidad de forma interna. Pero para un cliente externo puede ser útil controlar el ámbito de las transacciones que le afectan.

  La utilización de un cliente transaccional permitirá especificar los límites de la transacción.

  Ambos, emisor y receptor pueden ser transaccionales. Con un emisor, el mensaje no es realmente añadido al canal hasta que realice un commit de la transacción. Cuando el receptor recibe el mensaje, este no es realmente eliminado del canal hasta que el receptor no realice un commit de la transacción.

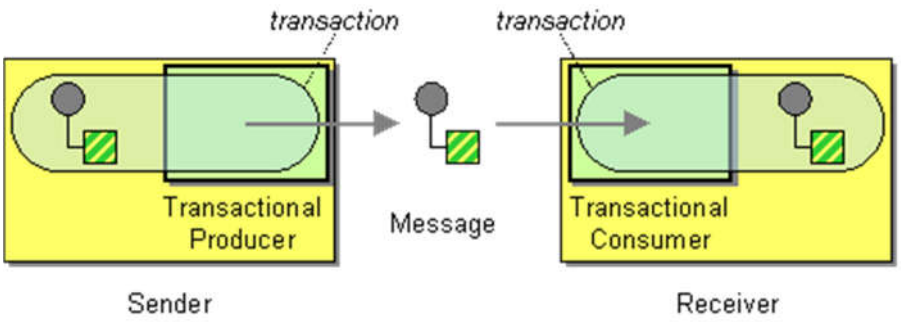



Figura 53 - Clientes transaccionales

[fuente: http://www.enterpriseintegrationpatterns.com]

Un emisor que usa transacciones explícitas puede ser usado con un receptor que usa transacciones implícitas y viceversa. Un canal simple puede ser una combinación de transacciones implícitas y explícitas.

- **Poll de consumidores:** Una aplicación necesita consumir mensajes, pero también queremos controlar cuando se consume cada mensaje.

  La aplicación debe realizar una invocación al poll explícitamente cuando quiere recibir un mensaje.

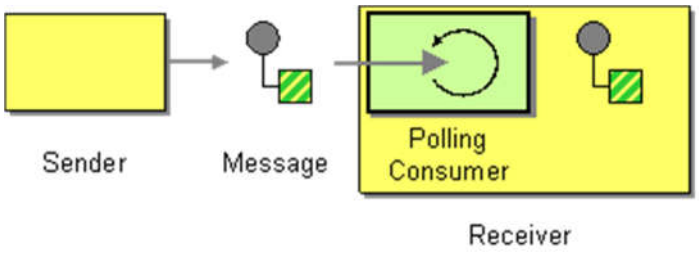



Figura 54 - Poll de consumidores

[fuente: http://www.enterpriseintegrationpatterns.com]

Un poll también es conocido como un receptor síncrono porque el receptor bloquea el hilo hasta que el mensaje sea recibido y procesado. Cuando

invocamos a un poll, este nos proporcionará un hilo que procesará el mensaje, retornará el resultado y devolverá el hilo de nuevo al poll.

- **Consumidor basado en eventos:** La aplicación consumirá mensajes tan pronto como ellos sean entregados.

  Un consumidor basado en eventos manejará los mensajes recibidos tan pronto como el mensaje este en el canal. Este tipo de receptores son considerados asíncronos porque el hilo no es lanzado hasta que el mensaje no es entregado a él, además decimos que es basado en eventos porque es la recepción de un mensaje quien hace que se active el consumidor.

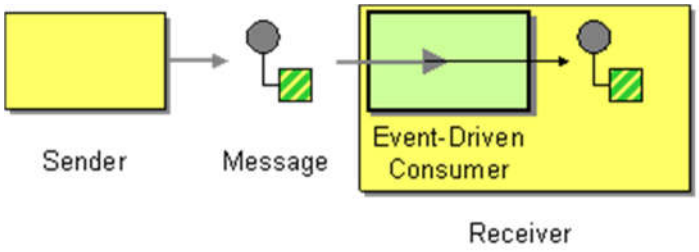



**Figura 55 - Consumidor basado en eventos**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

- **Competición de consumidores:** Una aplicación que recibe mensajes puede no poder procesarlos tan rápido como estos son recibidos.

  Se crearán diferentes consumidores sobre el mismo canal, de forma que los mensajes depositados en el puedan ser procesados de forma concurrente. Cuando el emisor envía un mensaje al canal, varios receptores pueden potencialmente recibirlo y por tanto, competirán entre sí para obtenerlo y procesarlo.

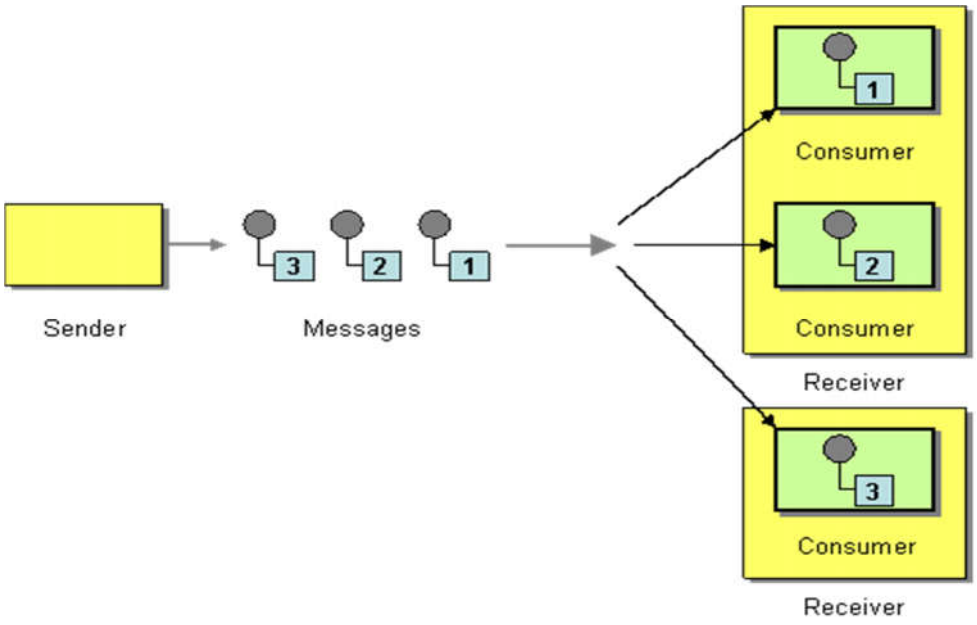



Figura 56 - Competición de consumidores

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Distribuidor de mensajes:** La aplicación tiene varios consumidores en un mismo canal, pero que necesitan trabajar de forma coordinada.

  Para solucionar este problema, se requerirá utilizar un despachador de mensajes, el cual será el encargado de consumir los mensajes del canal y entregárselos al receptor correspondiente.

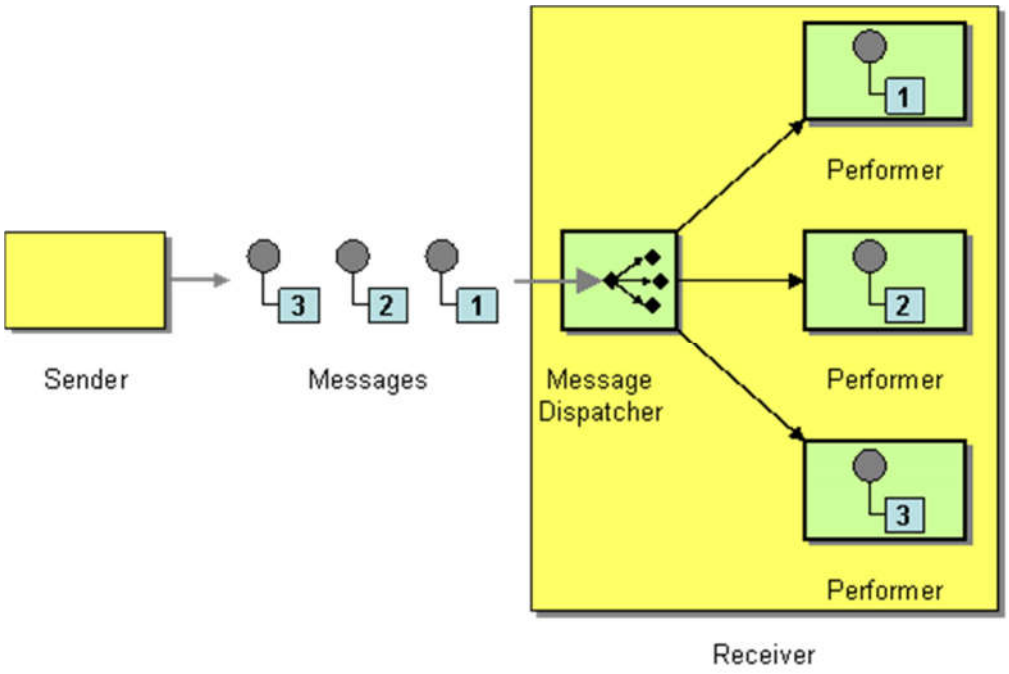



Figura 57 - Distribuidor de mensajes

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Consumidor selectivo:** Una aplicación encargada de consumir mensajes de un canal simple, no implica que tenga que consumir todos los mensajes que se publican en el canal.

  Para esto, el selector de mensajes filtrará todos aquellos mensajes que son entregados al canal entregando solo aquellos que cumplen los criterios de selección.

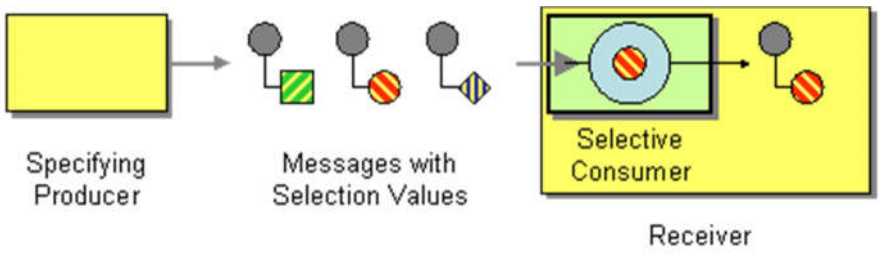



Figura 58 - Consumidor selectivo

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Subscriptor con persistencia:** Un subscriptor se encuentra desactivado pero desea recibir los mensajes que se están publicando porque son de interés para él, por esto se requiere que dicho subscriptor sea persistente.

  Cuando el subscriptor es persistente, los mensajes que deberían ser entregados al subscriptor desactivado son almacenados y serán entregados en el momento que el subscriptor vuelva estar activo, de esta manera, un subscriptor nunca perderá mensajes aunque se desconecte.

  Una subscripción persistente no tendrá diferencias con una no persistente mientras el subscriptor está activo, ya que en este caso actuará igual que si no fuera persistente, la diferencia recae en el comportamiento del sistema cuando esta desactivado el subscriptor.

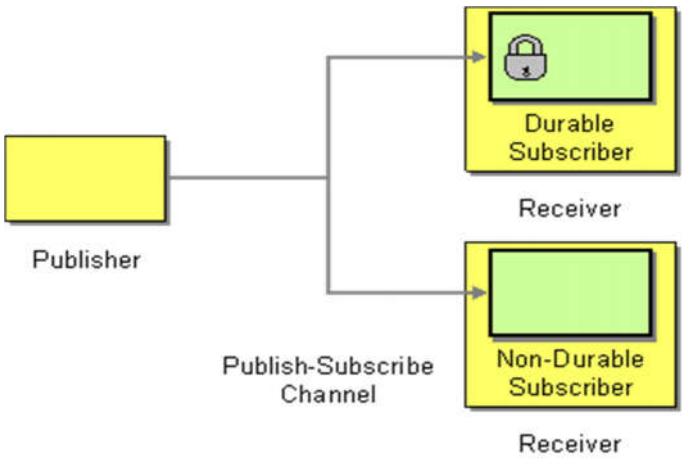



Figura 59 - Subscriptor con persistencia

[fuente: http://www.enterpriseintegrationpatterns.com]

- **Receptor Idempotente:** Un emisor envía un único mensaje, pero el receptor puede recibirlo más de una vez.

  Un receptor marcado como idempotente, puede recibir de manera segura el mismo mensaje múltiples veces que esto no hará que falle. Como nos indica el término idempotente, el resultado de recibir el mensaje una vez será el mismo que el de recibirlo múltiples veces.

- **Activador de servicio:** Una aplicación posee un servicio que está disponible para otras aplicaciones.

  El activador conectará los mensajes existentes en el canal con el servicio que se encuentra accesible.

  El activador puede ser unidireccional o bidireccional. El servicio puede ser tan simple como la llamada a un método (síncrono y no remoto), parte de la capa de servicio.

  Existen dos formas de cómo se puede encontrarse el activador:

  - ✓ Hardcodeado para que siempre invoque al mismo servicio.
  - ✓ Usarse reflection para invocar al servicio que se encuentra indicado en el mensaje, en este caso, el activador manejará los detalles del mensaje e invocará al servicio como cualquier otro cliente, por tanto el servicio no sabe que está siendo invocado a través de un mensaje.

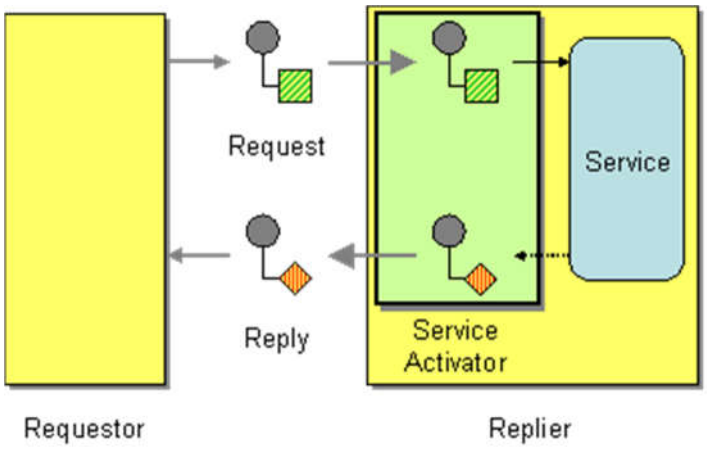



**Figura 60 - Activador de servicio**

**[fuente: http://www.enterpriseintegrationpatterns.com]**

# Capítulo 3: Especificación de requisitos

En el siguiente capítulo describiremos a alto nivel como debe ser el sistema que pretendemos construir para realizar la comparativa.

El apartado constará de dos partes:

- Especificación de los formatos origen y destino utiliza el sistema que pretendemos construir.
- Especificación de los requisitos funcionales que debe cumplir el sistema a construir.

## 3.1 Especificación de los formatos origen y destino

Por lo general, en muchas organizaciones existen sistemas antiguos, escritos en lenguajes ya en desuso, como por ejemplo Cobol, pero que debido a su gran fiabilidad todavía siguen estando activos.

El caso anterior es cierto por ejemplo en el mundo bancario. Muchas organizaciones poseen grandes sistemas, llamados en muchas ocasiones “Host”, escritos en la mayor parte de los casos en Cobol, que debido a la fiabilidad alcanzada a lo largo de los años de funcionamiento, son los encargados de controlar, por ejemplo, los diferentes sistemas de pagos.

Además, al estar escritos en lenguajes “antiguos”, el formato de salida, en muchos de ellos es un formato plano, es decir, una serie de registros que generan un formato de salida de tipo texto.

Para comenzar, vamos a definir por tanto un ejemplo de formato propietario de tipo plano, el cual poseerá una longitud de registro fija y que siempre acaba en un fin de línea.

Cada registro comienza por un identificador de registro, y luego los diferentes campos que se deben traducir. A continuación mostraremos el formato plano con el que se trabajará para construir el sistema.

- Registro de cabecera:

| Nombre | Id. Registro | Referencia | Fecha | Origen | Destino | Libre |
|---|---|---|---|---|---|---|
| Requerido | S | S | S | S | S | |
| Tipo Reg. | Numérico | Alfanumérico | Numérico | Alfanumérico | Alfanumérico | Espacios |
| Valor | 1 | | yyyyMMdd | | | |
| Longitud | 1 | 35 | 8 | 20 | 20 | 6 |
| Posición | 01 | 02-36 | 37-44 | 47-63 | 64-84 | 85-90 |

**Tabla 1 - Cabecera mensaje BackOffice**

**[fuente: propia]**

- Registro de datos

| Nombre | Id. Registro | Referencia Op. | Fecha Op. | Importe | Observaciones | Libre |
|---|---|---|---|---|---|---|
| Requerido | S | S | S | S | N | |
| Tipo Reg. | Numérico | Alfanumérico | Numérico | Numérico | Alfanumérico | Espacios |
| Valor | 2 | | yyyyMMdd | | | |
| Longitud | 1 | 35 | 8 | 15 | 25 | 6 |
| Posición | 01 | 02-36 | 37-44 | 45-59 | 60-84 | 85-90 |

**Tabla 2 - Registro de datos**

**[fuente: propia]**

- Pie

| Nombre | Id. Registro | Total Op. | Total Imp. | Libre |
|---|---|---|---|---|
| Requerido | S | S | S | |
| Tipo Reg. | Numérico | Numérico | Numérico | Espacios |
| Valor | 3 | | | |
| Longitud | 1 | 15 | 15 | 59 |
| Posición | 01 | 02-16 | 17-31 | 32-90 |

**Tabla 3 - Pie del documento**

**[fuente: propia]**

Como podemos observar en las tablas, el formato está dividido en tres tipos de registro.

El primer registro será una cabecera. Este registro únicamente aparecerá una vez y estará compuesto por los siguientes datos:

- Referencia que la entidad asigna al fichero completo.
- Fecha en la cual se ha generado dicho fichero.
- Entidad origen de las operaciones que contiene el fichero. Existen en este caso dos posibles opciones:
    - Si es un fichero a emitir, el origen de operaciones será la entidad.
    - Si es un fichero a recibir, el origen será la entidad que lo emitió.
- Entidad destino de las operaciones que contiene el fichero. En este caso, también existen dos opciones:
    - Si es un fichero a emitir, el destino será la entidad a la que van dirigidas las diferentes operaciones.
    - Si es un fichero recibido, el destino será la entidad que lo recibe.

El siguiente registro, cuya cardinalidad será 1..n, contendrá los datos relativos a una operación. El fichero estará compuesto por tanto por n de estos registros. Los datos que incluye son:

- Referencia de la operación. Referencia que asigna la entidad a la operación.
- Fecha de operación: Fecha en la cual deberá ser ejecutada dicha operación.
- Importe: El monto de la operación.
- Observaciones: Datos adicionales que sean requeridos.

Para finalizar, existirá un último registro, a modo de resumen, el cual formará el Pie del fichero, los datos que incluye son los siguientes:

- Total de operaciones: Número total de operaciones que incluye el fichero.
- Total Importes: Suma de los importes de las operaciones incluidas en el fichero.

Los campos Libre existentes en cada uno de los registros son reservados para uso futuro.

A la hora de comunicar operaciones, la entidad generalmente se comunica con un sistema central que se encarga de gestionar las órdenes procedentes de las diferentes entidades.

Los sistemas centrales, en orden cronológico, son más modernos y por tanto utilizan tecnologías más actuales. Generalmente, su formato propietario está definido como un xml.

A continuación veremos el formato que utiliza el sistema central para la comunicación de operaciones.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.fgarcia1763.org/envelope"
        xmlns:xsd="http://www.fgarcia1763.org/envelope"
         elementFormDefault="qualified">

   <complexType name="DefCabecera">
        <sequence>
             <element name="Referencia" type="string" />
             <element name="Fecha" type="date" />
             <element name="Origen" type="string" />
             <element name="Destino" type="string" />
        </sequence>
   </complexType>

   <complexType name="DefDatos">
        <sequence>
              <element name="ReferenciaOperacion" type="string" />
              <element name="FechaOperacion" type="date" />
              <element name="Importe" type="decimal" />
              <element name="Observaciones" type="string" minOccurs="0"
                                                  maxOccurs="1" />
        </sequence>
   </complexType>

   <complexType name="DefPie">
        <sequence>
               <element name="TotalOperaciones" type="integer" />
               <element name="TotalImportes" type="decimal" />
        </sequence>
    </complexType>

    <complexType name="DefDocument">
        <sequence>
            <element name="cabecera" type="xsd:DefCabecera"
                        minOccurs="1"  maxOccurs="1"/>
            <element name="datos" minOccurs="1" maxOccurs="1">
                 < complexType >
                        <sequence>
                              <element name="operacion" type="xsd:DefDatos"
                                       minOccurs="1" maxOccurs="unbounded"/>
                        </sequence>
                   </ complexType >
            </element>
            <element name="pie" type="xsd:DefPie" minOccurs="1"
                        maxOccurs="1"/>
        </sequence>
    </complexType>

    <element name="document" type="xsd:DefDocument" />
</schema>
```

Como podemos observar y viendo los mensajes que utiliza cada parte, nos encontramos ante las siguientes situaciones:

- Si una entidad va a comunicar sus operaciones al sistema central de forma directa no podría, ya que la entidad genera un formato plano con las operaciones a comunicar, pero la entidad central, realmente espera un xml.

- Si la entidad central tiene que comunicar operaciones procedentes de otras entidades, la va a enviar en formato xml, pero la entidad realmente espera un formato plano y por tanto, tampoco serán capaces de entenderse.

## 3.2 Especificación de requisitos funcionales

En el siguiente sub-apartado se especificará a alto nivel los requisitos que debe cumplir el aplicativo a desarrollar.

Para especificar los requisitos funcionales utilizaremos el lenguaje natural y la especificación deberá responder a las siguientes cuestiones.

- Que se espera que haga el sistema (¿qué?).
- La justificación (¿por qué ha de ser así? ¿quién lo propuso?)
- Criterios de aceptación aplicables (¿cómo se verifica su cumplimiento?).

En primer lugar y antes de entrar en el detalle se definirá a muy alto nivel la necesidad que se pretende cubrir.

La entidad desea comunicar al sistema central una serie de operaciones que este debe realizar, así mismo, desea recibir del sistema central todas aquellas operaciones que van dirigidas hacia ella.

Debido a que el sistema central y la entidad utilizan diferentes formatos, se requiere construir un sistema de traducción capaz de recibir el formato del sistema “Host” y traducirlo al formato xml que admite el sistema central para que las pueda enviar a las entidades destino.

Así mismo, el sistema a construir deberá poder recibir el formato xml procedente del sistema central y traducirlo al formato plano que admite el sistema “Host” de la entidad, de forma que este pueda conocer que operaciones han enviado otras entidades y actuar en consecuencia.

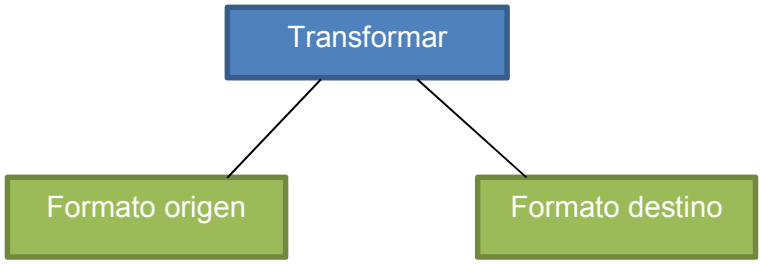



**Figura 61 - Proceso de transformación**

**[fuente: propia]**

Una vez especificado el problema a solucionar a muy alto nivel, se definirán los requisitos funcionales que debe cumplir el aplicativo a desarrollar.

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-001 | Emisión de operaciones | | 04/04/2016 | Esencial |
| **Descripción** | El sistema deberá recibir las operaciones a emitir procedentes del sistema host y enviarlas al sistema de transformación para su procesamiento. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un fichero plano que contiene las operaciones a emitir | Sistema Host de la entidad a través del sistema de ficheros | Formato entendible por el sistema de transformación | Sistema de transformación | Se deberá poder tratar cualquier fichero plano de operaciones, independientemente de su tamaño. |
| **Definición proceso** | El sistema "Host" de la entidad escribe un fichero plano con las operaciones que se deben emitir ese día. El sistema verificará que el fichero recibido contiene operaciones. En caso afirmativo, el sistema empaquetará bloques de operaciones y los enviará al sistema de transformación para su tratamiento. | | | |
| **Verificación** | Se deberá comprobar que se emiten al sistema de transformación las operaciones contenidas en el formato plano recibido. | | | |

**Tabla 4 - Especificación requisito ER-001**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-002 | Validación de operaciones formato plano emisión | | 04/04/2016 | Esencial |
| **Descripción** | El sistema deberá validar las operaciones procedentes de "Host" y descartar las inválidas. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un bloque de operaciones válidas.<br><br>Un fichero con las operaciones inválidas. | Sistema de transformación<br><br>Sistema de ficheros | Solamente se permitirán aquellas operaciones que sean válidas. |

| | |
|---|---|
| **Definición proceso** | El sistema recibe un bloque de operaciones. Se verificará que el bloque contenga operaciones. En caso afirmativo, se verificará el contenido de cada una de ellas descartando a un fichero de texto las que no cumplan los requisitos contenido o formato de campo. En caso de que existan operaciones inválidas se debe informar mediante un correo electrónico y adjuntar el fichero de inválidas generado. |
| **Verificación** | Se comprobará que ante un fichero con operaciones válidas e inválidas, las válidas continúan en el sistema y las inválidas son emitidas a un fichero plano. |

**Tabla 5 - Especificación requisito ER-002**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-003 | Transformación operaciones emitidas | | 05/04/2016 | Esencial |
| **Descripción** | El sistema deberá transformar las operaciones procedentes de “Host” al formato XML de la entidad central y descartar las inválidas. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un bloque de operaciones transformadas. | Sistema de transformación | Solamente se permitirán aquellas operaciones que sean válidas. |
| **Definición proceso** | El sistema recibe un bloque de operaciones. Se verificará que el bloque contenga operaciones. En caso afirmativo, se transformará el contenido de cada una de ellas al formato aceptado por el sistema central. En caso de que existan operaciones que no pueden ser transformadas, se debe informar mediante un correo electrónico. | | | |
| **Verificación** | Se comprobará que las diferentes operaciones en formato plano recibidas son transformadas al formato xml. | | | |

**Tabla 6 - Especificación requisito ER-003**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-004 | Validación de operaciones formato xml emisión | | 05/04/2016 | Esencial |
| **Descripción** | El sistema deberá validar las operaciones transformadas y descartar las inválidas. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un bloque de operaciones válidas.<br><br>Un fichero con las operaciones inválidas. | Sistema de transformación<br><br>Sistema de ficheros | Solamente se permitirán aquellas operaciones que sean válidas. |
| **Definición proceso** | El sistema recibe un bloque de operaciones transformadas. Se verificará que el bloque contenga operaciones. En caso afirmativo, se verificará el contenido de cada una de ellas descartando a un fichero las que no cumplan con el esquema proporcionado por la entidad central. En caso de que existan operaciones inválidas se debe informar mediante un correo electrónico y adjuntar el fichero de inválidas generado. | | | |
| **Verificación** | Se deberá comprobar que las operaciones son validadas contra el esquema. | | | |

**Tabla 7 - Especificación requisito ER-004**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-005 | Emisión de operaciones transformadas | | 06/04/2016 | Esencial |
| **Descripción** | El sistema deberá escribir las operaciones transformadas a un fichero para realizar su emisión. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un fichero con las operaciones transformadas que vamos a emitir. | Sistema de ficheros | Solamente se permitirán aquellas operaciones que sean válidas. |

| | |
|---|---|
| **Definición proceso** | El sistema recibe los diferentes bloques que contienen las distintas operaciones ya transformadas al formato reconocido por el sistema central. Una vez que tiene todos los bloques, generará un fichero único con la fusión de todos ellos y validará dicho fichero contra el esquema una vez más. Si la validación es correcta, escribirá el fichero en disco para su envío posterior, en caso contrario, se deberá enviar un correo electrónico indicando el error y adjuntando el fichero erróneo. |
| **Verificación** | Se deberá comprobar que las operaciones son escritas en un fichero para su posterior envío. |

Tabla 8 - Especificación requisito ER-005

[fuente: propia]

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-006 | Recepción de operaciones | | 07/04/2016 | Esencial |
| **Descripción** | El sistema deberá recibir las operaciones procedentes del sistema central y enviarlas al sistema de transformación para su procesamiento. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un fichero xml que contiene las operaciones recibidas | Sistema central a través del sistema de ficheros | Formato entendible por el sistema de transformación | Sistema de transformación | Se deberá poder tratar cualquier fichero xml de operaciones, independientemente de su tamaño. |
| **Definición proceso** | El sistema central de la entidad escribe un fichero xml con las operaciones recibidas ese día. El sistema verificará que el fichero contiene operaciones. En caso afirmativo, el sistema empaquetará bloques de operaciones y los enviará al sistema de transformación para su tratamiento. | | | |
| **Verificación** | Se deberá comprobar que se emiten al sistema de transformación las operaciones contenidas en el formato xml recibido. | | | |

Tabla 9 - Especificación requisito ER-006

[fuente: propia]

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-007 | Validación de operaciones formato xml recibidas | | 08/04/2016 | Esencial |
| **Descripción** | El sistema deberá validar las operaciones procedentes del sistema central y descartar las inválidas. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un bloque de operaciones válidas.<br><br>Un fichero con las operaciones inválidas. | Sistema de transformación<br><br>Sistema de ficheros | Solamente se permitirán aquellas operaciones que sean válidas. |
| **Definición proceso** | El sistema recibe un bloque de operaciones. Se verificará que el bloque contenga operaciones. En caso afirmativo, se verificará el contenido de cada una de ellas contra el esquema xml proporcionado por el sistema central, descartando a un fichero de las que no lo cumplan. En caso de que existan operaciones inválidas se debe informar mediante un correo electrónico y adjuntar el fichero de inválidas generado. | | | |
| **Verificación** | Se comprobará que ante un fichero con operaciones válidas e inválidas, las válidas continúan en el sistema y las inválidas son emitidas a un fichero. | | | |

**Tabla 10 - Especificación requisito ER-007**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-008 | Transformación operaciones recibidas | | 08/04/2016 | Esencial |
| **Descripción** | El sistema deberá transformar las operaciones procedentes del sistema central en formato XML al formato plano de la entidad y descartar las inválidas. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |

| | | | | |
|---|---|---|---|---|
| Un bloque de operaciones | Sistema de transformación | Un bloque de operaciones transformadas. | Sistema de transformación | Solamente se permitirán aquellas operaciones que sean válidas. |
| **Definición proceso** | El sistema recibe un bloque de operaciones. Se verificará que el bloque contenga operaciones. En caso afirmativo, se transformará el contenido de cada una de ellas al formato aceptado por el sistema "Host" de la entidad. En caso de que existan operaciones que no pueden ser transformadas, se debe informar mediante un correo electrónico. | | | |
| **Verificación** | Se comprobará que las diferentes operaciones en formato xml recibidas son transformadas al formato plano. | | | |

**Tabla 11 - Especificación requisito ER-008**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-009 | Validación de operaciones transformadas en formato plano | | 09/04/2016 | Esencial |
| **Descripción** | El sistema deberá validar las operaciones transformadas y descartar las inválidas. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un bloque de operaciones válidas.<br><br>Un fichero con las operaciones inválidas. | Sistema de transformación<br><br>Sistema de ficheros | Solamente se permitirán aquellas operaciones que sean válidas. |
| **Definición proceso** | El sistema recibe un bloque de operaciones. Se verificará que el bloque contenga operaciones. En caso afirmativo, se verificará el contenido de cada una de ellas descartando a un fichero de texto las que no cumplan los requisitos contenido o formato de campo. En caso de que existan operaciones inválidas se debe informar mediante un correo electrónico y adjuntar el fichero de inválidas generado. | | | |
| **Verificación** | Se deberá comprobar que las operaciones transformadas cumplen los requisitos de contenido y formato. | | | |

**Tabla 12 - Especificación requisito ER-009**

**[fuente: propia]**

| **Sistema de transformación – Especificación Requisitos Funcionales** | | | | |
|---|---|---|---|---|
| **Código** | **Nombre** | | **Fecha** | **Importancia** |
| ER-010 | Recepción de operaciones transformadas | | 10/04/2016 | Esencial |
| **Descripción** | El sistema deberá escribir las operaciones a un fichero para enviárselas al sistema “Host” de la entidad. | | | |
| **Entrada Esperada** | **Origen** | **Salida Esperada** | **Destino** | **Restricciones** |
| Un bloque de operaciones | Sistema de transformación | Un fichero con las operaciones transformadas que debemos recibir. | Sistema de ficheros | Solamente se permitirán aquellas operaciones que sean válidas. |
| **Definición proceso** | El sistema recibe los diferentes bloques que contienen las distintas operaciones ya transformadas al formato reconocido por el sistema “Host” de la entidad. Una vez que tiene todos los bloques, generará un fichero único con la fusión de todos ellos. El fichero será escrito en disco para su posterior recogida por el módulo de “Host” correspondiente. | | | |
| **Verificación** | Se deberá comprobar que las operaciones son escritas en un fichero para su posterior recogida por el sistema “Host” de la entidad. | | | |

**Tabla 13 - Especificación requisito ER-010**

**[fuente: propia]**

# Capítulo 4: Diseño técnico del sistema

En este apartado definiremos los aspectos técnicos a tener en cuenta para construir el sistema. La estructura del capítulo será la siguiente:

- Casos de uso. A través de ellos definiremos como deberá comportarse el aplicativo.
- Arquitectura y patrón EAI.
- Herramientas que nos permitirán implementar la arquitectura propuesta y selección de las mismas.

## 4.1. Casos de uso

A continuación definiremos los diferentes casos de uso, los cuales serán utilizados para especificar la funcionalidad real del sistema.

Nos basaremos en los requisitos definidos en el apartado anterior, pero llegando a un nivel de detalle más técnico, especificando por tanto, como debe ser implementado el aplicativo.

| **CU-001** | **Lectura de un fichero de BO** | |
|---|---|---|
| **Objetivo** | Leer el fichero procedente de BO y enviarlo al sistema para realizar la transformación del formato. | |
| **Requisitos asociados** | ER-001: Emisión de operaciones | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciba un fichero procedente del BO de la organización. | |
| **Precondición** | El fichero recibido debe contener operaciones y estar depositado en el directorio de lectura correcto. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso detecta la existencia del fichero y comprueba si la escritura en el directorio de entrada realizada por el proceso de BO ha finalizado. |
| | 2 | Se efectuará una copia del fichero a un directorio de backup para no perder información. |
| | 3 | El proceso realiza la inserción en base de datos del contenido del fichero, así como de los datos más representativos. |
| | 4 | Se comprueba la existencia de operaciones. |
| | 5 | Existen operaciones, se deberá comprobar cuantas existen para proceder diferente dependiendo del número de ellas. |

| | | |
|---|---|---|
| | 5.1 | Si el número de operaciones no excede el límite de procesamiento simultáneo, el sistema creará un mensaje común con dichas operaciones y este se enviará al sistema de transformación. |
| | 5.2 | El número de operaciones existente supera el límite de procesamiento simultáneo establecido por el sistema, por lo tanto mientras existan operaciones. |
| | 5.2.1 | Agregamos la operación al mensaje en formato común. |
| | 5.2.2 | Si el número de operaciones existente en el mensaje común es igual al límite de procesamiento simultaneo, enviamos dicho mensaje al orquestador. |
| | 5.2.3 | Se creará un mensaje común nuevo para continuar el procesamiento. |
| | 5.2.4 | Si es la última operación, se enviará el mensaje común al sistema de transformación. |
| **Postcondición** | El fichero se encuentra en proceso de transformación, teniendo guardada una copia del original en un directorio de backup. | |
| **Excepciones** | **Paso** | **Acción** |
| | 3 | Si no se puede realizar la persistencia física, el proceso enviará un correo indicando dicho error y se copiará el fichero original a un directorio de errores. |
| | 4 | Si no existen operaciones, se enviará un mensaje de correo indicando el problema. |
| | - | Ante una excepción en el procesamiento del fichero de BO, el fichero será copiado a un directorio de errores y se enviará un correo electrónico indicando la excepción. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |
| **Comentarios** | Al menos se tiene que esperar 2 minutos desde la última modificación del fichero de BO para poder leerlo. | |

**Tabla 14 - Caso de uso CU-001**

**[fuente: propia]**

| **CU-002** | **Lectura de un fichero del sistema central** |
|---|---|
| **Objetivo** | Leer el fichero procedente del sistema y enviarlo al sistema para realizar la transformación del formato. |
| **Requisitos asociados** | ER-006: Recepción de operaciones |

| | | |
|---|---|---|
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciba un fichero procedente del sistema central. | |
| **Precondición** | El fichero recibido debe contener operaciones y estar depositado en el directorio de lectura correcto. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso detecta la existencia del fichero y comprueba si la escritura en el directorio de entrada realizada por el proceso de recepción ha finalizado. |
| | 2 | Se efectuará una copia del fichero a un directorio de backup para no perder información. |
| | 3 | El proceso realiza la inserción en base de datos del contenido del fichero, así como de los datos más representativos. |
| | 4 | Se comprueba la existencia de operaciones. |
| | 5 | Se realizará una validación del formato del xml recibido (validación de estructura). |
| | 6 | Existen operaciones, se deberá comprobar cuantas existen para proceder diferente dependiendo del número de ellas. |
| | 6.1 | Si el número de operaciones no excede el límite de procesamiento simultáneo, el sistema creará un mensaje común con dichas operaciones y este se enviará al orquestador. |
| | 6.2 | El número de operaciones existente supera el límite de procesamiento simultáneo establecido por el sistema, por lo tanto mientras existan operaciones. |
| | 6.2.1 | Agregamos la operación al mensaje en formato común. |
| | 6.2.2 | Si el número de operaciones existente en el mensaje común es igual al límite de procesamiento simultaneo, enviamos dicho mensaje al sistema de transformación. |
| | 6.2.3 | Se creará un mensaje común nuevo para continuar el procesamiento. |
| | 6.2.4 | Si es la última operación, se enviará el mensaje común al sistema de transformación. |
| **Postcondición** | El fichero se encuentra en proceso de transformación, teniendo guardada una copia del original en un directorio de backup. | |
| **Excepciones** | **Paso** | **Acción** |
| | 3 | Si no se puede realizar la persistencia física, el proceso enviará un correo indicando dicho error y se copiará el fichero original a un directorio de errores. |

| | | |
|---|---|---|
| | 4 | Si no existen operaciones, se enviará un mensaje de correo indicando el problema. |
| | 5 | Si el fichero xml recibido no es válido estructuralmente, se copiará al directorio de errores y se enviará un correo electrónico indicando el error. |
| | - | Ante una excepción en el procesamiento del fichero, este será copiado a un directorio de errores y se enviará un correo electrónico indicando la excepción. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |
| **Comentarios** | Al menos se tiene que esperar 2 minutos desde la última modificación del fichero para poder leerlo. | |

**Tabla 15 - Caso de uso CU-002**

**[fuente: propia]**

| **CU-003** | **Validación de operaciones en formato plano** | |
|---|---|---|
| **Objetivo** | Validamos la operación en formato plano para determinar si es correcta o no | |
| **Requisitos asociados** | ER-002 y ER-009: Validación de operaciones en formato plano. | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciban operaciones en formato plano (de BO) que requieran validación. | |
| **Precondición** | El mensaje recibido debe contener operaciones. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso recibe el mensaje común con las operaciones que se desean validar. |
| | 2 | Por cada una de las diferentes operaciones. |
| | 3 | Se valida el contenido de cada uno de los campos obligatorios. |
| | 4.1 | Si la operación es correcta se marca dicha situación en el mensaje común. |
| | 4.2 | Si la operación es incorrecta, se marcará el error la operación en el mensaje común y se incluirá una descripción del error. |
| **Postcondición** | El mensaje contiene todas las operaciones existentes validadas. | |
| **Excepciones** | **Paso** | **Acción** |
| | 4.2 | Si una operación es inválida, se retornará el error, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |

| | | |
|---|---|---|
| | - | Ante una excepción en la validación operaciones, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |

Tabla 16 - Caso de uso CU-003

[fuente: propia]

| **CU-004** | **Validación de operaciones en formato xml** | |
|---|---|---|
| **Objetivo** | Validamos la operación en formato xml para determinar si es correcta o no | |
| **Requisitos asociados** | ER-004 y ER-007: Validación de operaciones en formato xml. | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciban operaciones en formato xml que requieran validación. | |
| **Precondición** | El mensaje recibido debe contener operaciones. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso recibe el mensaje común con las operaciones que se desean validar. |
| | 2 | Por cada una de las diferentes operaciones. |
| | 3 | Se valida el contenido de la operación contra el esquema xsd. |
| | 4.1 | Si la operación es correcta se marca dicha situación en el mensaje común. |
| | 4.2 | Si la operación es incorrecta, se marcará el error la operación en el mensaje común y se incluirá una descripción del error. |
| **Postcondición** | El mensaje contiene todas las operaciones existentes validadas. | |
| **Excepciones** | **Paso** | **Acción** |
| | 4.2 | Si una operación es inválida, se retornará el error, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| | - | Ante una excepción en la validación operaciones, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |

Tabla 17 - Caso de uso CU-004

[fuente: propia]

| CU-005 | Transformación de operaciones en Plano (BO) a XML | |
|---|---|---|
| **Objetivo** | Transformamos la operación de formato BO (plano) a XML. | |
| **Requisitos asociados** | ER-003: Transformación de operaciones emitidas. | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciban operaciones en formato plano (BO) que requieran ser transformadas a XML. | |
| **Precondición** | El mensaje recibido debe contener operaciones válidas. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso recibe el mensaje común con las operaciones que se desean transformar. |
| | 2 | Por cada una de las diferentes operaciones. |
| | 3 | Mapeamos los campos del formato plano hacía su correspondiente en el formato XML. |
| | 4.1 | Si la operación es transformada correctamente se almacena en el mensaje común. |
| | 4.2 | Si la operación no puede ser transformada, se marcará el error la operación en el mensaje común y se incluirá una descripción del error. |
| **Postcondición** | El mensaje contiene todas las operaciones válidas existentes transformadas. | |
| **Excepciones** | **Paso** | **Acción** |
| | 4.2 | Si una operación no puede ser transformada, se retornará el error, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| | - | Ante una excepción en la transformación de operaciones, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |

**Tabla 18 - Caso de uso CU-005**

**[fuente: propia]**

| CU-006 | Transformación de operaciones en XML a Plano (BO) |
|---|---|
| **Objetivo** | Transformamos la operación de formato XML a BO (plano). |
| **Requisitos asociados** | ER-008: Transformación de operaciones recibidas. |

| | | |
|---|---|---|
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciban operaciones en formato XML que requieran ser transformadas a plano (BO). | |
| **Precondición** | El mensaje recibido debe contener operaciones válidas. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso recibe el mensaje común con las operaciones que se desean transformar. |
| | 2 | Por cada una de las diferentes operaciones. |
| | 3 | Mapeamos los campos del formato XML hacía su correspondiente en el formato plano (BO). |
| | 4.1 | Si la operación es transformada correctamente se almacena en el mensaje común. |
| | 4.2 | Si la operación no puede ser transformada, se marcará el error la operación en el mensaje común y se incluirá una descripción del error. |
| **Postcondición** | El mensaje contiene todas las operaciones válidas existentes transformadas. | |
| **Excepciones** | **Paso** | **Acción** |
| | 4.2 | Si una operación no puede ser transformada, se retornará el error, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| | - | Ante una excepción en la transformación de operaciones, se enviará un correo electrónico indicando el fallo y se anulará el fichero totalmente. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |

**Tabla 19 - Caso de uso CU-006**

**[fuente: propia]**

| | | |
|---|---|---|
| **CU-007** | **Fichero Inválido** | |
| **Objetivo** | Invalidar el fichero a emitir o recibido ante un error en el proceso. | |
| **Requisitos asociados** | | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se reciba la orden de invalidar un fichero. | |
| **Precondición** | El fichero ha sido invalidado previamente en uno de los procesos de transformación. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | Se le comunica al proceso que debe invalidar el fichero que está tratando actualmente. |

| | | |
|---|---|---|
| | 2 | Realiza una copia del fichero existente en el directorio backup al directorio de errores. |
| | 3 | Comunica mediante correo electrónico que el fichero ha sido invalidado. |
| **Postcondición** | El fichero ha sido invalidado completamente y se ha informado de esta situación. | |
| **Excepciones** | **Paso** | **Acción** |
| | - | Ante una excepción en el proceso encargado de invalidar el fichero, se enviará un correo electrónico indicando el. |
| **Frecuencia esperada** | Mínima, no debería ser una situación frecuente. | |
| **Estabilidad** | Crítica. | |

**Tabla 20 - Caso de uso CU-007**

**[fuente: propia]**

| **CU-008** | **Escritura del fichero para el sistema central** | |
|---|---|---|
| **Objetivo** | Escribir el fichero para su envío al sistema central. | |
| **Requisitos asociados** | ER-005: Emisión de operaciones transformadas | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se vayan a recibir operaciones transformadas para emitir al sistema central. | |
| **Precondición** | El mensaje recibido deberá contener operaciones válidas y transformadas a XML. | |
| **Secuencia Normal** | **Paso** | **Acción** |
| | 1 | El proceso recibe un mensaje común con las operaciones transformadas en XML. |
| | 2 | Se comprueba si el escritor ya ha recibido todos los mensajes en los que se dividió el fichero original. |
| | 2.1 | Si únicamente había un mensaje en el sistema, el contenido transformado se escribirá en el directorio de salida y se realizará una copia a un directorio de backup. |
| | 2.2 | Si el fichero origen fue dividido en múltiples mensajes. |
| | 2.2.1 | Si no es se han terminado de procesar los mensajes en los que se dividió el fichero origen, se generará un fichero temporal con el contenido transformado. |

| | | |
|---|---|---|
| | 2.2.2 | Si se ha terminado el procesado de los mensajes en los que se dividió el fichero origen, se realizará la fusión de todos los temporales, escribiendo el resultado en el directorio de salida y realizando una copia en un directorio de backup. |
| | 3 | Comprobación de que el fichero generado es estructuralmente correcto. |
| | 4 | Actualización en base de datos con el contenido transformado. |
| **Postcondición** | El fichero generado se corresponde con la transformación correspondiente del fichero original, además se tendrá guardada una copia en un directorio de backup. | |
| **Excepciones** | **Paso** | **Acción** |
| | 3 | Si el fichero no es estructuralmente correcto, se enviará un correo electrónico indicándolo, adjuntado el fichero generado, así mismo, se eliminará el fichero del directorio de salida y se copiará el original al directorio de error. |
| | 4 | Si el fichero no puede ser persistido, se enviará un correo electrónico indicando el error, se eliminará el fichero del directorio de salida y se copiará el original al directorio de error. |
| | - | Ante una excepción en el proceso de escritura, el fichero original este será copiado a un directorio de errores y se enviará un correo electrónico indicando la excepción. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |

**Tabla 21 - Caso de uso CU-008**

**[fuente: propia]**

| **CU-009** | **Escritura del fichero para BO** | |
|---|---|---|
| **Objetivo** | Escribir el fichero para su envío a BO. | |
| **Requisitos asociados** | ER-010: Recepción de operaciones transformadas | |
| **Descripción** | El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando se vayan a recibir operaciones transformadas con destino BO. | |
| **Precondición** | El mensaje recibido deberá contener operaciones válidas y transformadas a formato BO. | |
| **Secuencia** | **Paso** | **Acción** |
| **Normal** | 1 | El proceso recibe un mensaje común con las operaciones transformadas en formato BO. |

| | | |
|---|---|---|
| | 2 | Se comprueba si el escritor ya ha recibido todos los mensajes en los que se dividió el fichero original. |
| | 2.1 | Si únicamente había un mensaje en el sistema, el contenido transformado se escribirá en el directorio de salida y se realizará una copia a un directorio de backup. |
| | 2.2 | Si el fichero origen fue dividido en múltiples mensajes. |
| | 2.2.1 | Si no es se han terminado de procesar los mensajes en los que se dividió el fichero origen, se generará un fichero temporal con el contenido transformado. |
| | 2.2.2 | Si se ha terminado el procesado de los mensajes en los que se dividió el fichero origen, se realizará la fusión de todos los temporales, escribiendo el resultado en el directorio de salida y realizando una copia en un directorio de backup. |
| | 3 | Actualización en base de datos con el contenido transformado. |
| **Postcondición** | El fichero generado se corresponde con la transformación correspondiente del fichero recibido del sistema central, además se tendrá guardada una copia en un directorio de backup. | |
| **Excepciones** | **Paso** | **Acción** |
| | 3 | Si el fichero no puede ser persistido, se enviará un correo electrónico indicando el error, se eliminará el fichero del directorio de salida y se copiará el original al directorio de error. |
| | - | Ante una excepción en el proceso de escritura, el fichero original este será copiado a un directorio de errores y se enviará un correo electrónico indicando la excepción. |
| **Frecuencia esperada** | Alta, varias veces al día | |
| **Estabilidad** | Crítica. | |

**Tabla 22 - Caso de uso CU-009**

**[fuente: propia]**

## 4.2. Arquitectura y patrón EAI

En este apartado del capítulo describiremos la arquitectura que construirá para dar solución a la problemática que se ha definido, tanto en la especificación de requisitos como en los casos de uso del apartado anterior.

Una vez que se ha definido la arquitectura, se decidirá que patrón EAI se debe aplicar para construirla.

Los requisitos que debe cumplir la arquitectura a construir son los siguientes:

- **Desacoplada:** Los diferentes módulos funcionales que se vayan a construir deben de actuar como piezas independientes, de esta forma obtenemos:

    - Simplicidad en el mantenimiento ya que si ocurre un error en el aplicativo, únicamente debemos corregir esa pieza y no tener que estar mirando y corrigiendo sobre enormes bloques de código, lo que generalmente introduce errores ante mínimas correcciones.

    - Posibilidad de realizar test unitarios sobre cada una de las piezas, gracias a esto, no se tiene que realizar un test completo del sistema cada vez que se modifique una de ellas.

    - Los dos puntos anteriores proporcionan otra ganancia y es la reducción de costes, reducción en costes de mantenimiento así como en implementación de pruebas.

    - Facilidad para cambiar la funcionalidad de alguna de las piezas. Si se requiere cambiar la funcionalidad de una de las piezas, simplemente habrá que sustituir esa pieza por otra con la nueva funcionalidad, no teniendo que recompilar el sistema completo.

- **Fácilmente escalable:** El sistema debería ser diseñado para que sea fácilmente escalable.
  Un montaje utilizando piezas software proporciona esta característica, ya que si se desea agregar nueva funcionalidad, únicamente se construirán las diferentes piezas que componen la nueva integración, orquestándolas a través del sistema.

- **Independencia del sistema de entrada y salida**: Se debe garantizar una cierta independencia del sistema que se utilizará para leer o escribir los ficheros que se van a intercambiar.
  Si no se garantizara esta independencia, aunque solo sea parcial, cuando se necesitara cambiar de sistema de intercambio habría que reconstruir parte del sistema y esto no es viable por dos motivos:

- La reconstrucción del sistema sería costosa y acarrearía la inclusión de nuevos bugs, lo cual no es tolerable en un sistema que ya estaba funcionando.

- Debido a que el cambio implica una reconstrucción de parte del sistema, esto lleva asociado un alto coste, si a esto se le suma el incremento debido a la realización de pruebas complejas y solución de nuevos problemas, el coste podría ser desmesurado.

En primer lugar se tendrá en cuenta el patrón a seguir para construir el aplicativo. Si se consideran como aplicaciones integrar el BO y el sistema central, se aplicará el patrón por transferencia de ficheros, existiendo una comunicación en ambas direcciones, es decir, desde el punto de vista de los roles productor-consumidor:

- **Productor:** Como productores actuarán tanto el BO cuando tiene que emitir operaciones al sistema central, como el sistema central cuando tiene que emitir operaciones para el BO.

- **Consumidor:** Igual que en el caso anterior, tanto el BO como el sistema central recibirán las operaciones que el otro envía.

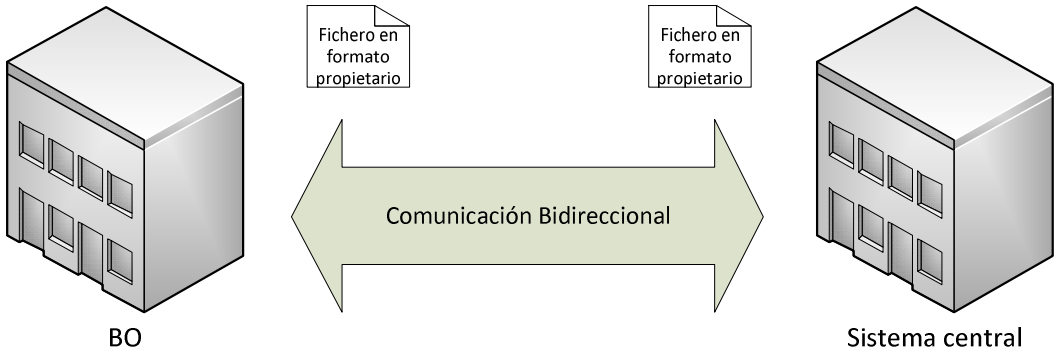



**Figura 62 - Transferencia de ficheros entre BO y el Sistema Central**

**[fuente: propia]**

A continuación, sabiendo que se van a comunicar a través de un sistema de intercambio de ficheros, se definirá como se va a construir el sistema de transformación encargado de realizar la conversión de formatos.

A la hora de definir cómo será el sistema de transformación, se considerarán los requisitos que debe cumplir la arquitectura.

- **Desacoplada:** La arquitectura a construir estará formada por diferentes piezas que se comunicarán entre sí, de forma que trabajen como un todo.

  Dado que el sistema estará compuesto por diferentes piezas, necesitamos un mecanismo estándar que las permita compartir información.

Además, las diferentes piezas deberán actuar de forma asíncrona, por ejemplo, en el caso de que llegue un fichero muy grande y se fragmente, el procesado de un fragmento no dependerá del procesado de los otros.

Leyendo de nuevo el apartado Patrones de integración, concretamente el sub-apartado Mensajes, se puede observar que el uso de este patrón se adaptará perfectamente a este requisito ya que:

- El uso de mensajes va a permitir a las diferentes piezas comunicarse entre sí.

- Existe un bus que se utiliza para el intercambio de la mensajería, este bus de comunicaciones sirve de mecanismo para conseguir la ejecución asíncrona requerida.

- **Fácilmente escalable:** Al construir nuestra arquitectura utilizando mensajes, también cumpliremos esta premisa ya que cualquier nueva funcionalidad que se requiera desarrollar únicamente va a implicará dos pasos:

  - Construir las nuevas piezas. Ya que el sistema será desacoplado, agregar una nueva funcionalidad debería ser simplemente construir las diferentes piezas.

  - Conectarlas al bus de comunicaciones. El solo construir las diferentes piezas no es suficiente para que la nueva funcionalidad empiece a trabajar, esas piezas necesitan comunicarse entre ellas para coordinarse, por tanto, necesitan usar el bus de comunicanciones.

- **Independencia del sistema de entrada y salida**: Aunque no es una característica del patrón elegido, el sistema constará de piezas de conexión con los sistemas externos y con el bus de comunicación, de forma que, ante un cambio, únicamente cambiando la pieza que conecta con ellos y todo seguirá funcionando.

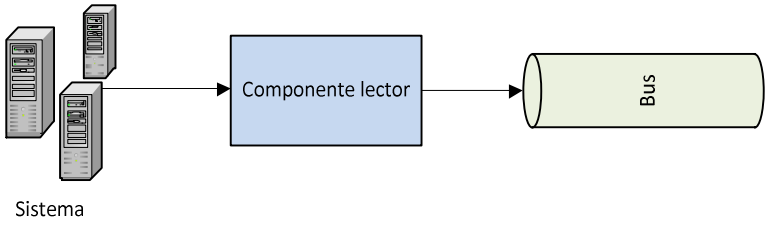



**Figura 63 - Componente lector**

**[fuente: propia]**

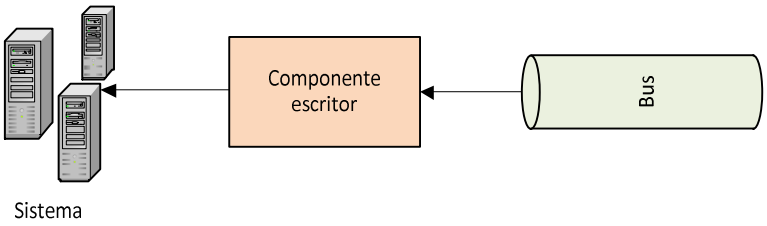



**Figura 64 - Componente escritor**

**[fuente: propia]**

Una vez decidido el patrón EAI a utilizar, se definirá en detalle cada una de las diferentes partes que lo componen.

- **Canal de comunicación:** Las diferentes piezas que van a formar el sistema deben comunicarse entre sí. Para ello se ha decidido utilizar un gestor de colas que actuará como bus de comunicaciones.

- **Mensajes:** Las piezas que conforman el sistema deben de alguna forma intercambiar información para conocer cómo va evolucionando el proceso. Ya que el patrón elegido es utilizar mensajería, la pieza de información básica que van a compartir será el mensaje.

  Aunque intercambien mensajes, cada una de ellas tiene un cometido específico, lectura desde el origen, validaciones, transformaciones, escrituras en disco… y por tanto, la información que intercambiarán será diferente. Debido a esto, se necesitará dotar al aplicativo de un mecanismo que las permita comunicarse.

  Para ello se optará por construir un **Envolvente del mensaje**, que poseerá dos partes bien diferenciadas:

  - Cabecera: Contendrá los datos necesarios para que las diferentes piezas puedan entenderse y realizar su trabajo.

  - Cuerpo: En el cuerpo de la envolvente irá ubicado el mensaje particular con el que trabajará cada una de las piezas. Este sub-mensaje incluido en el cuerpo será de tipo documento ya que internamente contendrá una estructura de datos.

  Como podemos por tanto observar, utilizando un envolvente se cumple el requisito principal, dotando al sistema de un mecanismo que permite a las diferentes piezas software interactuar entre ellas de forma común.

- **Enrutador de mensajes:** Todas las piezas software que vamos a construir estarán conectadas al mismo canal de comunicación, el cual, por sí solo no sabe cómo enrutar los mensajes que va recibiendo, es decir, no sabe cómo

comunicar origen y destino, ya que simplemente actúa como un mero sistema de intercambio de información.

Para solventar este problema se implementará una pieza adicional que estará únicamente encargada de realizar el enrutado de mensajes. Por tanto crearemos un enrutador siguiendo el patrón **Message Broker**, donde la pieza central recibirá los diferentes mensajes y se encargará de ir enrutándolos a la cola correspondiente.

- **Traductor de mensajes:** A nivel de las diferentes piezas que componen el sistema no se requiere establecer un mecanismo de transformación de mensajes, al trabajar con un envolvente común, ya tenemos un mecanismo a través del cual las diferentes piezas se van a entender y por tanto, no se necesita la traducción.

  Si se considera el sistema completo como un todo, es decir, una gran caja negra y los sistemas de BO y central como aplicaciones a comunicar, tendríamos la definición de **Traductor** propiamente dicha; dos aplicaciones necesitan comunicarse, pero los mensajes que intercambian entre ellas tienen formatos distintos y no se entienden.

  Para solventar este problema, necesitan un mecanismo que traduzca el formato de la aplicación emisora al formato de la receptora y viceversa.

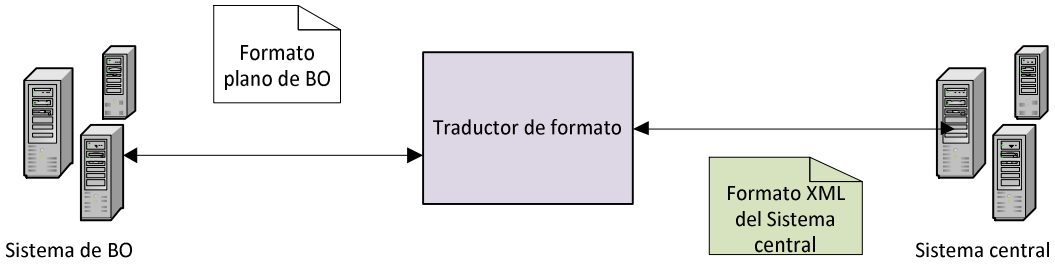



**Figura 65- Traductor de formatos**

**[fuente: propia]**

- **Endpoints:** Como se observa en la figura anterior, nuestro sistema traductor deberá poder comunicarse tanto con el sistema de BO como con el sistema central. Para cumplir esta misión se definirán piezas software que tendrán una doble funcionalidad.

  - Comunicarse con el sistema externo, bien sea el BO o el sistema central.

  - Comunicarse con el sistema de transformación emitiendo o recibiendo mensajes.

Estas piezas software se dividirán en dos tipos:

- **Lectores:** Serán los encargados de realizar la lectura del fichero que se desea procesar, fragmentarlo y enviar los fragmentos que contienen las operaciones a emitir al sistema de transformación para que realice su procesado.

  Existirán dos piezas lectoras, una para los ficheros emite BO y otra para los que recibimos del sistema central.

- **Escritores:** Estas piezas software recibirán los diferentes fragmentos del fichero del sistema de transformación, generará el fichero a enviar al sistema correspondiente y escribirá dicho fichero en el sistema de intercambio adecuado.

  Existirán al igual que en el caso anterior dos escritores, uno para los ficheros que debe recibir BO y otro para los que debe recibir el sistema central.

Una vez que hemos visto y comentado las diferentes partes que formarán el sistema de transformación que se va a construir, mostraremos una figura que resuma la arquitectura completa.

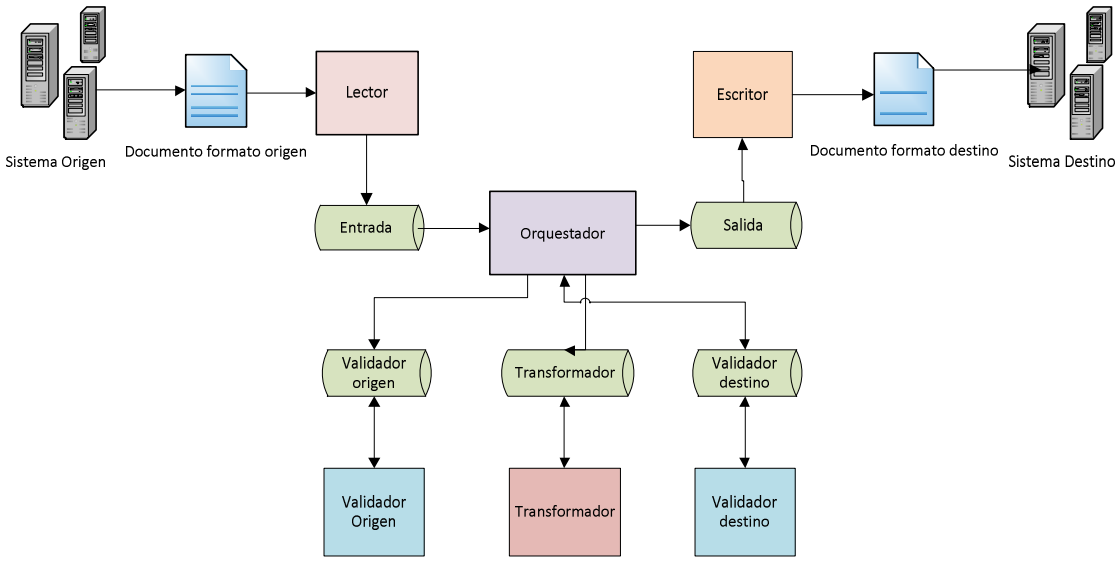



**Figura 66 – Arquitectura**

**[fuente: propia]**

Como podemos observar en la figura, la arquitectura propuesta cumple lo especificado en los puntos anteriores.

Tenemos diferentes piezas funcionales que compondrán el sistema. Estas piezas son:

- Componente lector, tal y como comentamos anteriormente, será la pieza encargada de leer del sistema origen el fichero con las operaciones, dividirlo en fragmentos y enviarlos al sistema de transformación a través de la cola Entrada.

- Componente escritor, el cual recibirá a través de la cola Salida los diferentes fragmentos con las operaciones transformadas, unificarlos en un único fichero y escribir dicho fichero en el sistema destino.

- Componente validador origen, esta pieza recibe un fragmento con operaciones y valida de cada una de ellas, marcando como inválidas aquellas que no cumplan las reglas en el definidas.

- Componente validador destino, tiene la misma funcionalidad que el componente validador origen pero sobre las operaciones ya transformadas.

- Componente transformador de formatos, transforma las diferentes operaciones recibidas en formato origen al formato destino esperado.

- Componente orquestador. Esta pieza forma el enrutador de mensajes. El orquestador actúa de la siguiente forma:

  - Recibe un mensaje procedente de la pieza que ha realizado el proceso.
  - Procesa el mensaje y decide cual es el siguiente paso a ejecutar.
  - Envía el mensaje al siguiente paso poniéndolo en la cola que corresponda.

A continuación podemos ver cómo está definido el **Message Broker** existente en la aplicación.

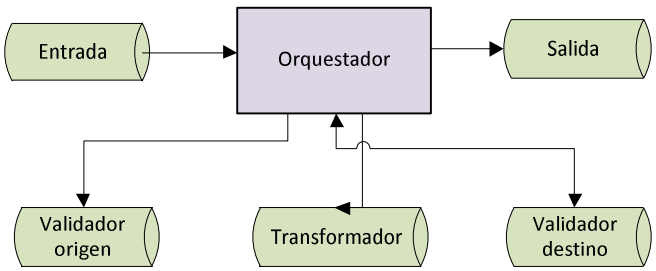



Figura 67 - Message Broker

[fuente: propia]

## 4.3. Herramientas que nos permitirán implementar la arquitectura propuesta y selección de las mismas

En este apartado se definirán las herramientas que nos van a permitir implementar la arquitectura propuesta en el apartado anterior.

Ya que la idea subyacente en el proyecto es construir el sistema propuesto tanto con software propietario como con software opensource y realizar una comparación entre ambos desarrollos, viendo si es justificada la preferencia de software propietario frente a software libre. En este apartado se definirán las herramientas que se utilizarán en cada caso.

### 4.3.1. Solución utilizando software propietario

Existen numerosas suites de integración en el mercado, cada una con sus ventajas y sus defectos, pero básicamente todas poseen el mismo objetivo, facilitar la forma de construir aplicaciones.

De entre las aplicaciones existentes en el mercado, se ha seleccionado **JavaCAPS versión 6 update 1**. Esta suite de desarrollo inicialmente perteneció a Sun Microsystems hasta que esta fue adquirida por Oracle, el cual continuó con el aplicativo.

El motivo principal por el cual se ha seleccionado esta suite es la posibilidad de acceder a ella y utilizarla para el desarrollo, otros productos son mucho más restringidos y no se encuentran accesibles, de todas formas, para los objetivos que se pretenden, el uso de JavaCAPS es más que suficiente.

La herramienta posee las siguientes características:

- **Netbeans 6** como IDE de desarrollo, al cual se le agregan plugins adicionales con la funcionalidad específica.
- **Glassfish 2.1** como servidor de aplicaciones.
- **OpenMQ 4.X** como gestor de colas y que será por tanto nuestro canal de comunicaciones.
- **Java 1.6.**

Mostraremos a continuación una figura explicativa sobre como estaría montada la aplicación con la arquitectura propuesta.

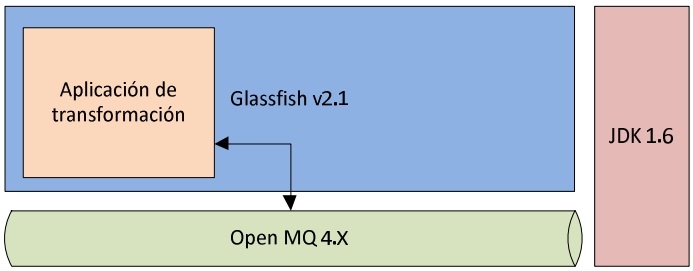



Figura 68 - Aplicación construida en JavaCAPS

[fuente: propia]

### 4.3.2. Solución utilizando software opensource

Si en el apartado anterior hemos definido la suite que utilizaremos para implementar la solución utilizando software propietario, en este apartado definiremos las herramientas opensource que utilizaremos para realizar el mismo cometido.

En primer lugar, debemos indicar que al igual que en el caso de software propietario existen numerosas suites de desarrollo, en el caso de utilizar software opensource el número de herramientas es algo más limitado, aunque si existen varias soluciones muy prometedoras.

Se ha optado por el uso conjunto de las siguientes herramientas:

- **Apache Camel** como framework de integración + **Spring**, el cual se integra perfectamente con Camel.

- En el caso de requerir algún tipo de acceso a base de datos se utilizará **Hibernate**.

- **Maven** como gestor de dependencias. Este es un punto muy importante ya que siempre que se trabaja con diferentes frameworks, a la hora de crear el entregable, siempre falta alguna dependencia y no funciona como se espera. Gracias a Maven nos evitaremos sorpresas inesperadas.

- **Eclipse Mars** como IDE de desarrollo.

- **Glassfish 4.1.0** como servidor de aplicaciones.

- **ActiveMq 5.13.2** como gestor de colas y por tanto como canal de comunicaciones.

- **Java 1.8**

Al igual que hicimos en el caso anterior, incluimos una figura sobre cómo estará montada la arquitectura y el aplicativo.

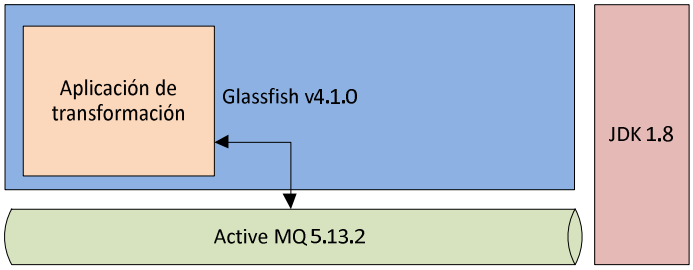



**Figura 69 - Aplicación construida con software opensource**

**[fuente: propia]**

# Capítulo 5: Desarrollo del aplicativo

En este capítulo se describirá paso a paso el proceso de construcción del software de transformación, comentando los aspectos más importantes a la hora de realizar el desarrollo.

Adicionalmente se mostrarán ejemplos de código que permitirán explicar en detalle cómo se realiza la codificación con cada una de las herramientas seleccionadas.

Antes de comenzar explicando el desarrollo del aplicativo utilizando software propietario, debemos definir algo que va a ser común a ambos desarrollos. Este elemento es el mensaje común, usado por las diferentes piezas software para intercambiar la información y lograr coordinarse.

## 5.1. Definición del mensaje común (Common Envelope)

El mensaje común va a ser la pieza de comunicación entre los diferentes módulos que componen el sistema de transformación.

El mensaje en formato común se divide en las siguientes secciones:

- **Header:** Cabecera del mensaje, contendrá los datos necesarios para orquestar de manera eficiente el flujo del mensaje recibido. Todos los datos irán almacenados en diferentes secciones de la cabecera.
- **Body:** Contiene el cuerpo del mensaje, poseerá diferentes apartados en los cuales irá ubicándose el mensaje según va pasando por las diferentes validaciones o transformaciones.

### 5.1.1. Header

La cabecera de nuestro formato común estará dividida en las siguientes secciones:

- **PhysicalData:** Almacenará los datos físicos del fichero o mensaje recibido.
- **FlowData:** Almacena los datos de transformación.
- **BulkData:** Almacena los datos del fragmento que estamos procesando.

A continuación indicaremos que datos irán en cada una de estas secciones.

#### 5.1.1.1. PhysicalData:

Como se ha indicado antes, almacena los datos físicos del elemento a tratar.

- ***Name (Opcional):*** En el caso de haber recibido un fichero, almacenará su nombre origen.
- ***ReceptionDate:*** Almacena la fecha en la que hemos recibido el mensaje.
- ***TotalOperations (Opcional):*** Si pudiera extraerse, almacenaría el número de operaciones que contiene el mensaje que vamos a procesar.
- ***TotalAmount (Opcional):*** Si aplica, importe total de las operaciones que contiene el mensaje a procesar.

#### 5.1.1.2. BulkData

Almacenará los datos del fragmento de mensaje que vamos a tratar. En algunas situaciones será un fragmento, en otras puede ser el mensaje completo.

- ***Order:*** Indica el orden del fragmento dentro del mensaje que hemos fraccionado.
- ***IsLastBulk:*** Booleano que marcará si el fragmento es el último.
- ***IsValidBulk:*** Booleano que marcará si el fragmento es válido.

#### 5.1.1.3. TransformData

Esta sección es realmente el Core del formato intermedio ya que almacena toda la información necesaria para realizar el proceso.

Los datos que deben ser cargados para poder realizar el proceso son:

- ***OrganizationData:*** Almacena los datos relativos a la organización con la que estamos trabajando.
    - **OrganizationId:** Identificador de la organización principal.
    - **RepresentedId:** Identificador del representado en nombre del cual actúa la organización. En el caso de que sea la propia organización, este identificador coincidirá con el valor del campo OrganizationId.

- ***PlatformData (Opcional):*** Datos relativos al sistema central con el que vamos a realizar la comunicación de operaciones.
    - **Platform:** Almacena el nombre del sistema central con el que realizamos la comunicación.
    - **Service (Opcional):** En ocasiones, debemos comunicar con un subsistema que pertenece al sistema central. En este campo indicamos dicho subsistema.

- ***EnvironmentData:*** Almacena los datos relativos al entorno en el que se ejecuta el flujo, los valores serán (T para Test) y (P para Producción).

- ***DocumentData:*** Almacena los datos relativos a los formatos fuente y destino.
    - **Reference:** Referencia del fichero dentro del sistema.
    - **SourceFormat:** Almacena el formato del fichero leido, los tipos permitidos serán PLANO o XMLv2.
    - **DestinationFormat:** Almacena el formato al que se debe realizar la transformación, los valores permitidos serán PLANO o XMLv2.

- ***RoutingData:*** Almacena los datos relativos a la transformación.
    - **Direction:** Almacena la dirección del flujo de proceso, TO_XXX o FROM_XXX
    - **Discriminator:** Almacena un valor que permitirá concretar el flujo de proceso a utilizar.
    - **NumMaxBulks:** Número máximo de fragmentos que puede contener un mensaje común.
    - **CompressInfo:** Almacena el tipo de compresión a utilizar (NONE|ZIP|GZIP). En nuestra prueba tomará el valor NONE y el código se adecuará a esta situación.

- ***Steps (0...n):*** Almacena los pasos que formarán el flujo de proceso. Si no existe esta estructura se entenderá que el proceso es la copia de entrada a salida.
    - **Step:** Estructura que almacena un paso del flujo.
        - *StepName:* Nombre del paso.

- *Order:* Orden de ejecución del paso.
- *IsActual:* Indica si el paso es el que se ha ejecutado actualmente.
- *IsLast:* Indica si es el último paso a ejecutar.
- *IsWithReply:* Indica si el envío a ejecutar el paso va a generar una replica.
- *Status:* Estado en el que se encuentra el paso.
    - PEND: Pendiente de ejecución.
    - OK: Ejecutado satisfactoriamente.
    - ERROR: Ejecutado con error
- *StatusMessage:* En el caso de que el campo status sea ERROR, en este campo debería incluirse el mensaje informativo del error.
- *QueueName:* Nombre de la cola a la cual se debe enviar el mensaje para ejecutar el paso.
- *Selector:* Selector que se deberá aplicar en el envío, de esta forma el receptor del mensaje sabe que es para él y puede procesarlo.
- *ValidationData:* Si el paso es una validación de tipo XML almacena la ubicación y el esquema que se debe aplicar.
    - SchemaPath (Opcional): En el caso de validación xml almacenará donde se encuentran los esquemas a usar.
    - Schemas (Opcional): En el caso de validación xml almacenará que esquemas debemos usar.

### 5.1.2. Body

Almacena el cuerpo de los diferentes mensajes con los que vamos a trabajar. Podemos dividirlo en tres secciones:

- **OriginalOperations:** Contiene las operaciones en formato original.
- **DestinationOperations:** Contiene las operaciones ya transformadas al formato destino.
-

#### 5.2.1.1. OriginalOperations

Contiene las operaciones que hemos leído de la fuente. Se compondrá de N registros de tipo **Operation** que tendrán la siguiente estructura definida.

- **Order:** Indica el orden de la operación dentro del fragmento.
- **Message:** Contiene el mensaje con la operación.
- **IsValid:** Indica si la operación es válida.
- **ErrorMessage:** Si existe error en cualquier tratamiento de una operación, en este punto se indica cual fue el error.

#### 5.2.1.2. DestinationOperations

Contiene las operaciones que hemos leído, pero transformadas al formato destino. Se compondrá de N registros de tipo **Operation** que tendrán la siguiente estructura definida.

- **Order:** Indica el orden de la operación dentro del fragmento.
- **Message:** Contiene el mensaje con la operación.
- **IsValid:** Indica si la operación es válida.
- **ErrorMessage:** Si existe error en cualquier tratamiento de una operación, en este punto se indica cual fue el error.

A continuación adjuntaremos el esquema que define el mensaje común, de esta forma queda disponible en el documento.

CommonEnvelope.xsd

**Adjunto 1 - Common Envelope**

**[fuente: propia]**

## 5.2. Implementación utilizando software propietario

La implementación utilizando software propietario constará de los siguientes pasos:

- Definición de los formatos que permitirán interpretar la información.
- Implementación de los endpoints, encargados de permitir la comunicación entre los sistemas externos y el sistema de transformación.
- Implementación de las piezas encargadas del proceso.
- Construcción del orquestador. Esta pieza será la encargada de organizar el flujo de transformación, controlando que paso se debe ejecutar cada vez.

### 5.2.1. Definición de formatos

Para comenzar la implementación del sistema, lo primero que se debe tener en cuenta es conocer cómo se va a poder interpretar la información contenida en los mensajes que se recibirán, es decir, ¿existirá algún mecanismo que de forma automática permite interpretar los datos recibidos?

En el caso particular de JavaCAPS, la respuesta a la pregunta anterior es afirmativa, la herramienta proporciona una utilidad para crear las estructuras de datos que permiten parsear los contenidos recibidos. Estas estructuras son denominadas OTDs (Object Type Definitions).

En el caso concreto del aplicativo que se está construyendo, se definirán tres tipos de OTDs, una definida por nosotros y otras dos, definidas mediante un esquema xsd.

En las siguientes imágenes podemos ver las capturas de pantalla correspondientes a esta definición de OTDs.

- Definición de la estructura de datos encargada de procesar las operaciones recibidas desde el BO.

  Para la definición de esta estructura no se utilizará ningún esquema predefinido ya que el formato de la operación es plano, con campos de longitud fija.

  Para estos casos, disponemos dentro del IDE de un editor mediante el cual podemos definir:

  - Fields: Son los elementos básicos de la estructura, poseen múltiples atributos como tamaño (en el caso de longitud fija), valor inicial, patrón que debe cumplir el contenido…

  - Elements: Un elemento es una agrupación de otros elementos o campos, permitiendo establecer la estructura del OTD.

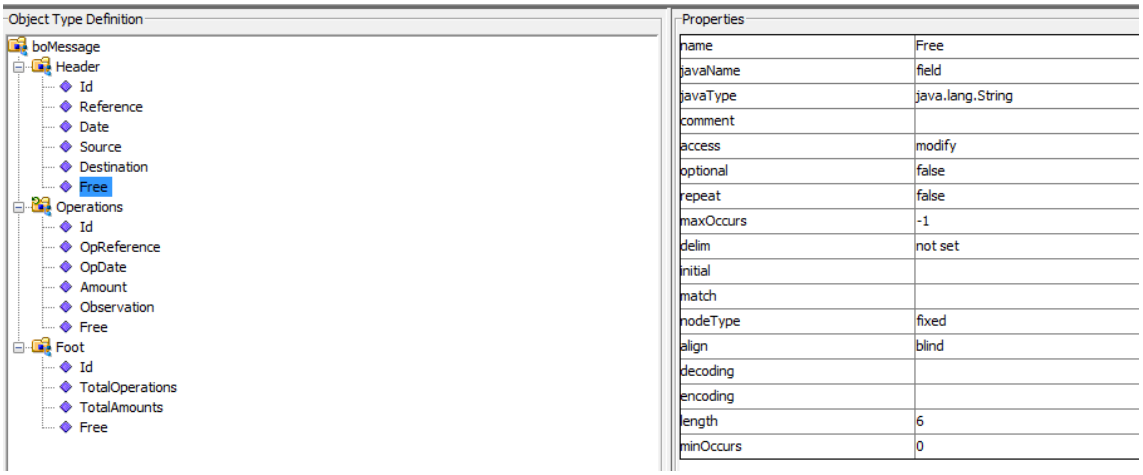



**Figura 70 - OTD Formato Plano**

**[fuente: propia]**

- Definición de las estructuras encargadas de procesar el mensaje común que compartirán las diferentes piezas software, así como la encargada de procesar las operaciones del sistema central.

  En este caso la construcción de los objetos es mucho más sencilla ya que se tienen esquemas que define la estructura que deben cumplir los OTDs.

  Para la definición de este tipo de estructuras, JavaCAPS nos proporciona un mecanismo automático, el cual a través del esquema xsd nos genera el correspondiente OTD.

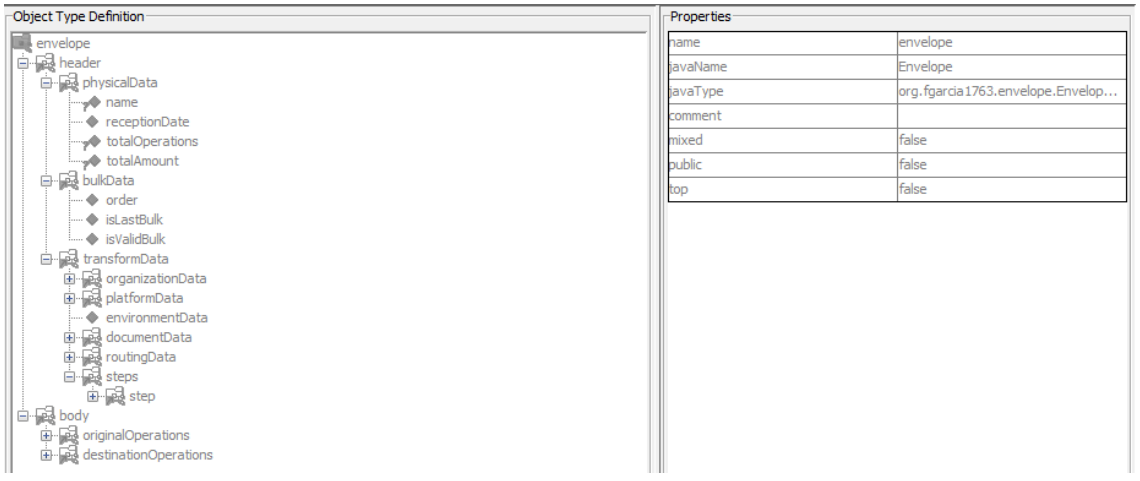



**Figura 71 - OTD Formato XSD**

**[fuente: propia]**

### 5.2.2. Introducción al desarrollo en JavaCAPS: Conceptos

Antes de comenzar a explicar cada una de las piezas que se han construido, es recomendable explicar, de forma resumida, algunos conceptos básicos sobre JavaCAPS.

Uno de los conceptos claves en JavaCAPS es el concepto **Colaboración**. Una colaboración no es más que una clase java que implementa una operación ya existente (por ejemplo, recibir un mensaje a través de una cola JMS), o bien, define una nueva operación, la cual puede ser invocada posteriormente desde otras colaboraciones, un webservice…

La creación de una colaboración se realiza siempre a través de un asistente y sus pasos básicos son los siguientes:

1. Se indica el nombre que tendrá la colaboración y si implementará una nueva operación o se debe crear desde una operación ya existente. En nuestro caso, se crearán desde una operación existente, excepto los dos lectores, el resto de colaboraciones deben lanzarse cuando se reciba un mensaje a través del correspondiente canal de comunicación.

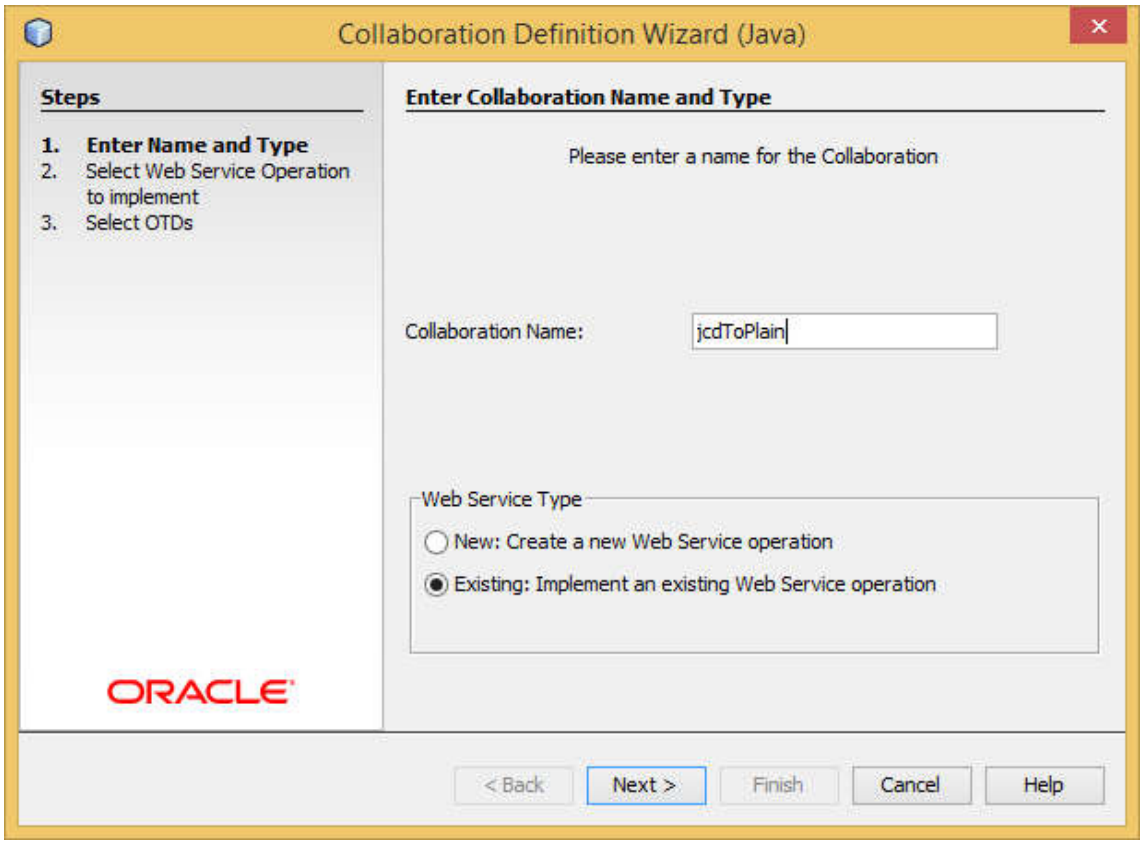



Figura 72 - Creación de Colaboración

[fuente: propia]

2. A continuación seleccionamos la operación ante la que responderá la colaboración que se va a crear.

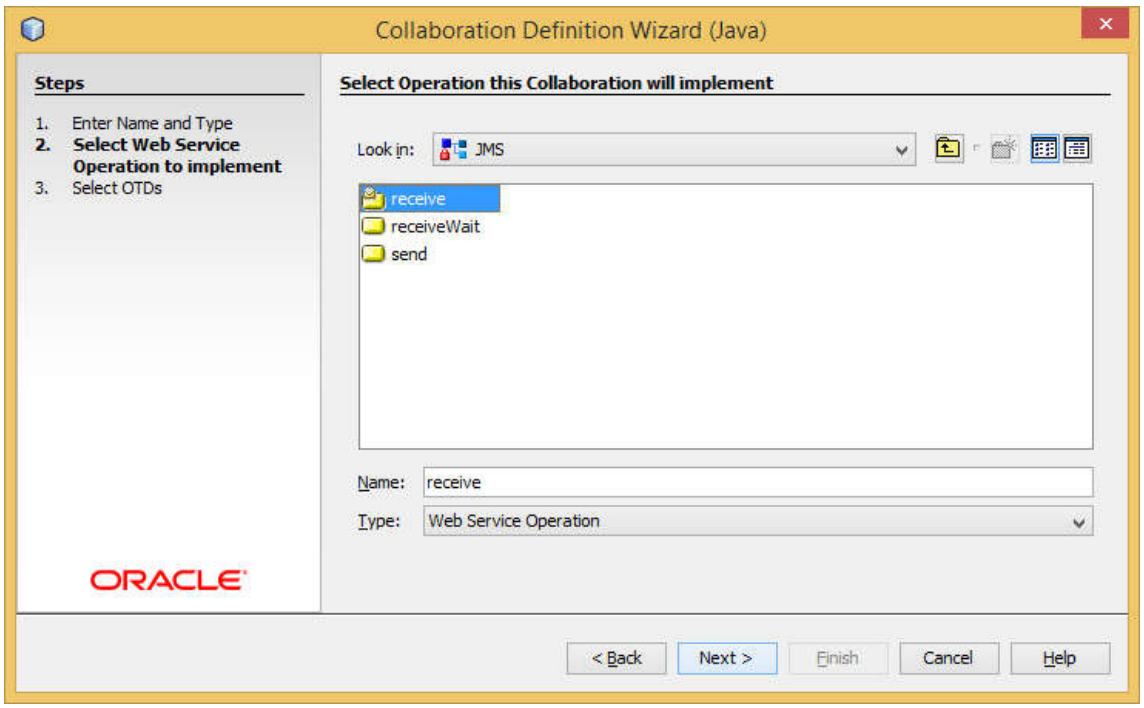



**Figura 73 - Selección de la Operación**

**[fuente: propia]**

3. En el último paso del proceso, se seleccionarán todos aquellos otds y eWays que necesitemos utilizar en nuestro proceso.

   Una aclaración sobre este paso, todos los otds y eWays que se declaren como utilizables en este punto, en tiempo de ejecución, una vez que se lance la colaboración, son instanciados. De esta forma se garantiza que podrán ser utilizados en nuestro código con la garantía de no incurrir en el uso de referencias nulas.

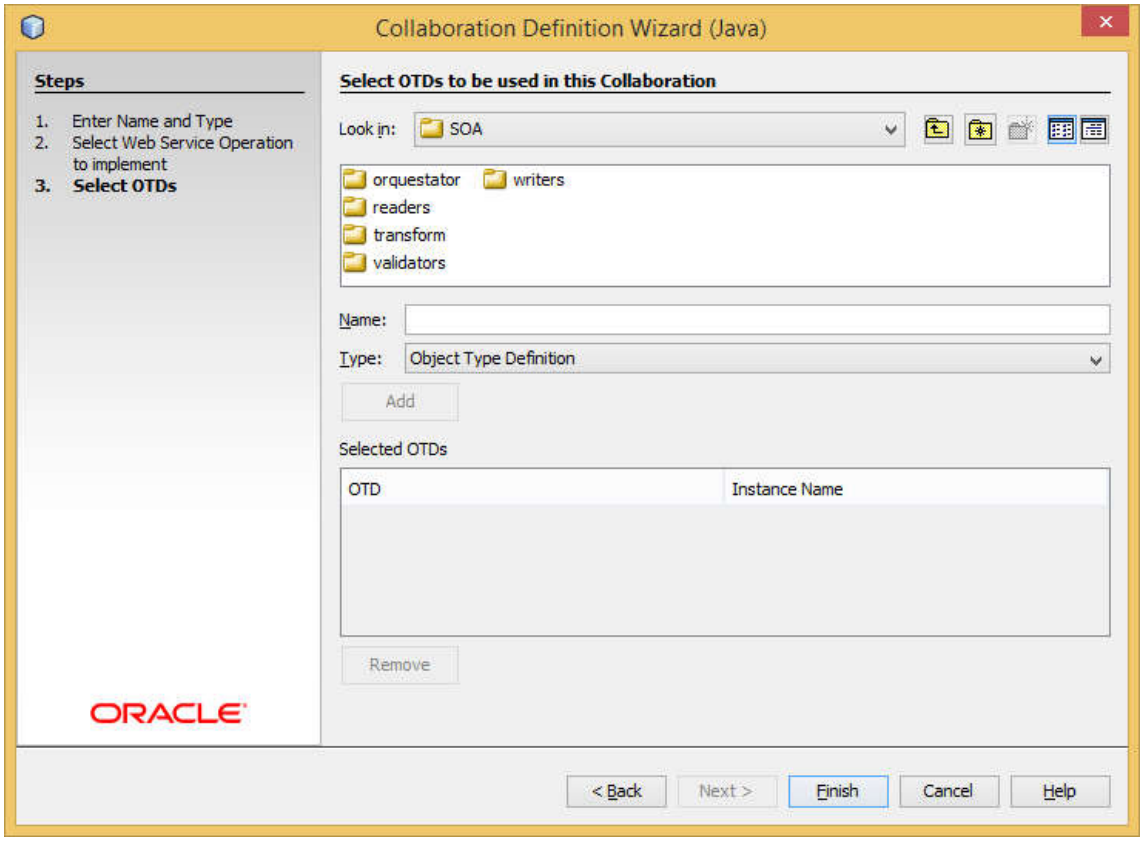



Figura 74 - Selección de otds y eWays adicionales

[fuente: propia]

4. Una vez finalizado el paso anterior, se abre una nueva ventana en el IDE de desarrollo con el esqueleto de la colaboración que se ha creado, de forma que se pueda comenzar la implementación.

```
package TransformSystemSOAwriters;


public class jcdToPlain
{

   public com.stc.codegen.logger.Logger logger;

   public com.stc.codegen.alerter.Alerter alerter;

   public com.stc.codegen.util.CollaborationContext collabContext;

   public com.stc.codegen.util.TypeConverter typeConverter;

   public void receive( com.stc.connectors.jms.Message input )
      throws Throwable
   {}

}
```

La definición de la colaboración es únicamente el primer paso de la definición del sistema. El siguiente paso es la definición del Mapa de Colaboración. Para JavaCAPS, un mapa de colaboraciones define la relación entre colaboración y eWays que utiliza la colaboración.

Otro aspecto a tener en cuenta es que una vez establecida la relación entre colaboración y eWay, en un mapa de conectividad se permite establecer una configuración personalizada para dicha relación.

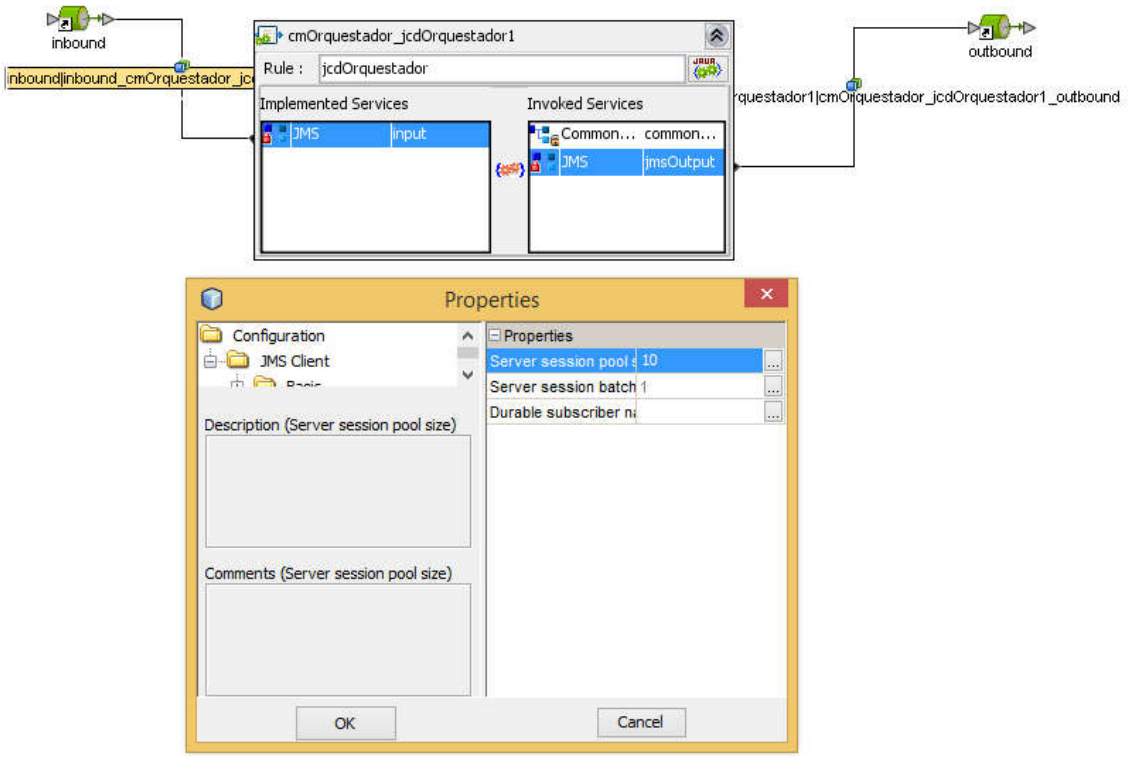



**Figura 75 - Conectivity Map**

**[fuente: propia]**

Con lo visto anteriormente se definiría completamente lo que podemos denominar **Capa Software**, es decir, el código del aplicativo y cómo va a interactuar entre sí.

Pero con esto no se ha finalizado el proceso, de creación, debe existir algún mecanismo que nos permita definir sobre que hardware trabajará el aplicativo. Para esto, JavaCAPS permiten definir **Environments**.

Dentro de un Environment el usuario definirá aquellos componentes que utiliza el aplicativo, así como la configuración de cada uno de ellos. Dentro de los componentes que se pueden definir están el servidor de aplicaciones a utilizar, el gestor de colas (bus de mensajería), configuración de los accesos a disco…

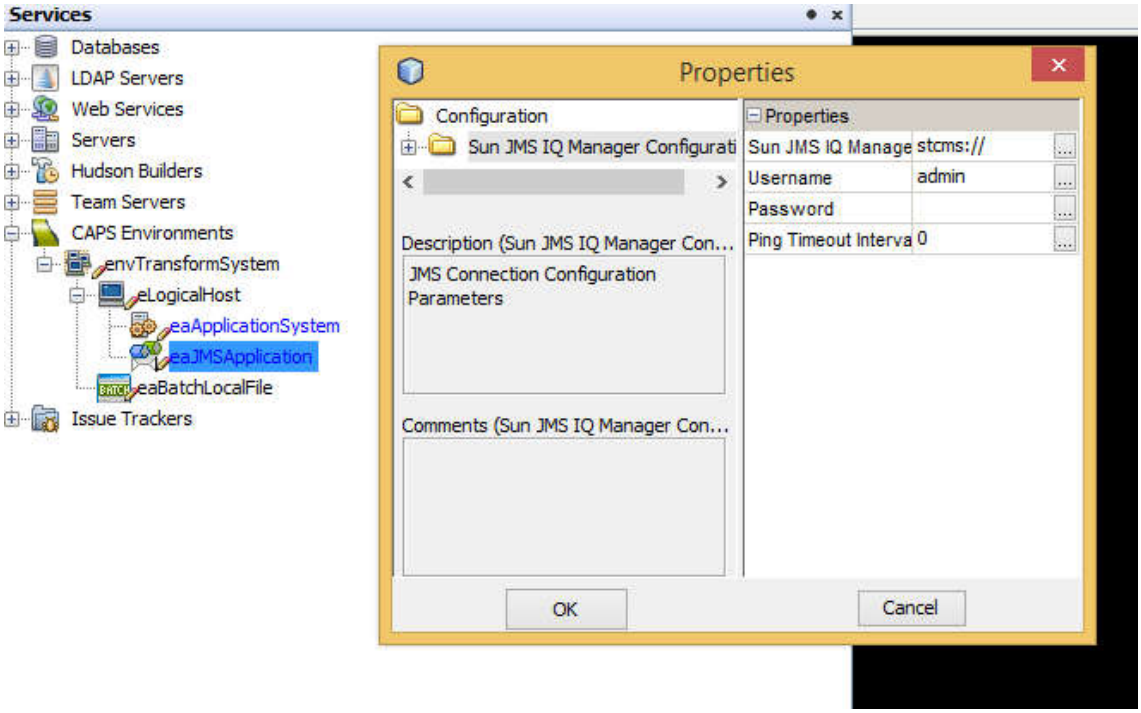



**Figura 76 - Definición de un Environment**

**[fuente: propia]**

Hasta este momento se ha explicado cómo se debe definir un proyecto utilizando JavaCAPS:

1. Creación e implementación de las colaboraciones que formarán el sistema.

2. Definición de cómo estás colaboraciones van a interactuar con los diferentes EndPoints (eWays) que nos proporciona JavaCAPS.

3. Establecer el marco hardware sobre el que se va a trabajar la solución.

Como se puede observar, existen dos partes bien diferenciadas, el marco software, representado por las colaboraciones y los mapas de conexión y el marco hardware, representado por los environments que se hayan definido.

Pero por si solas, estas dos partes no van a trabajar de forma conjunta, debe existir algún tipo de mecanismo que nos permita relacionarla, para ello, JavaCAPS permite crear diferentes perfiles de despliegue en los cuales relacionaremos el software con el hardware.

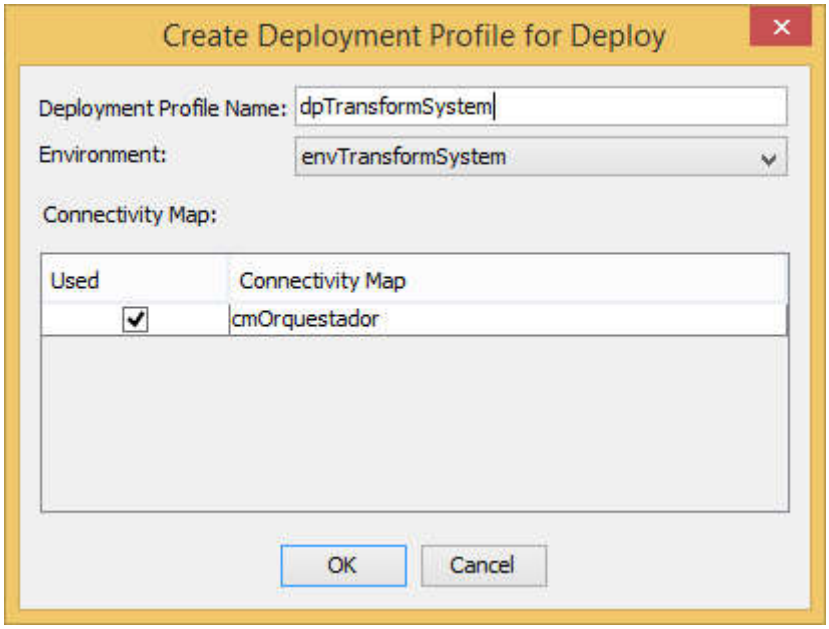



Figura 77 - Creación de un Deployment Profile

[fuente: propia]

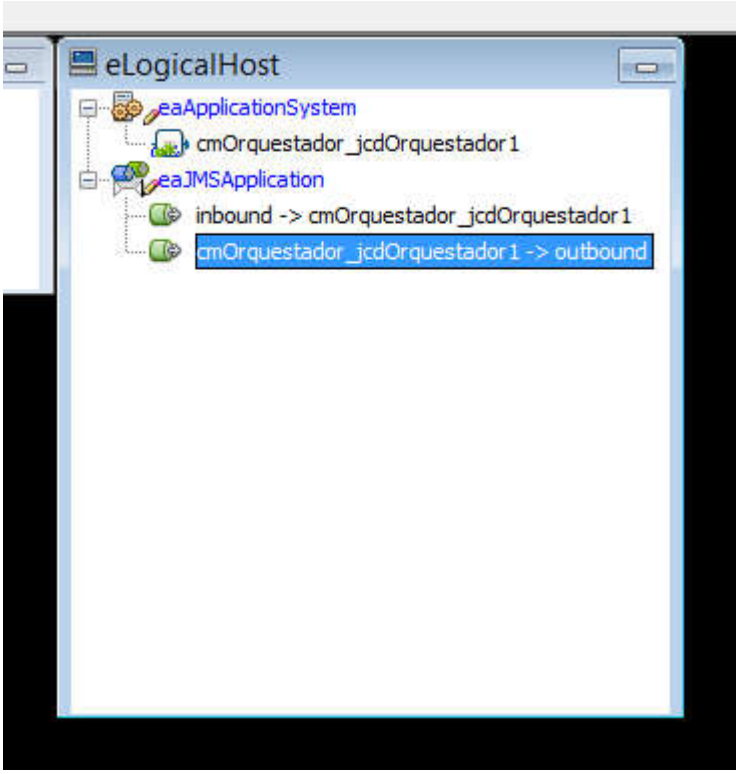



Figura 78 - Asociación Software-Hardware

[fuente: propia]

Finalmente, se realizará una compilación desde el propio perfil de despliegue para generar la aplicación.

### 5.2.3. Uso de Endpoints

Como se ha visto en anteriores definiciones, los Endpoints son piezas de código que:

- Conectan (o permiten comunicar) el sistema interno (nuestro transformador) con los diferentes sistemas externos. En nuestro caso concreto permiten comunicar el sistema de transformación, con los BO del Sistema Central y de la Entidad.

- Permiten acceder al sistema de mensajería, de forma que las diferentes piezas que componen el sistema puedan utilizarlo de forma transparente.

El software seleccionado, JavaCAPS, nos ofrece una amplia colección de Endpoints (denominados eWays), los cuales encapsulan la funcionalidad requerida, de forma que únicamente debemos utilizarlos. En nuestro ejemplo utilizaremos los siguientes:

- **JMS eWay:** Permite crear y enviar mensajes a las colas JMS utilizadas como sistema de mensajería.

  A continuación mostraremos un ejemplo de código sobre cómo se debe utilizar este eWay:

  1. Como hemos visto en el apartado previo, para disponer de un eWay totalmente operativo y funcional, únicamente se debe asignar a la colaboración en el momento de su creación y asociarlo a un componente en el mapa de conectividad.

     ```
     public void receive( com.stc.connectors.jms.Message input,
     stcgen.fcxotd.http___www_fgarcia1763_org_envelope.EnvelopeDocument
     commonEnvelope, com.stc.connectors.jms.JMS jmsOutput )
           throws Throwable  {}
     ```

  2. Una vez que se dispone del eWay operativo, para usarlo simplemente se deberá invocar el API que expone.

     ```
     Message jmsMessage = jmsOutput.createBytesMessage(
                                                   message.marshalToBytes() );

     if (destination.getSelector()!= null && destination.getSelector().length() > 0) {
     ```

```
            jmsMessage.storeUserProperty( "SELECT", destination.getSelector() );
        }
        jmsOutput.sendTo( jmsMessage, destination.getQueueName() );
```

- **BatchInbound eWay:** Encargado de monitorizar el sistema de ficheros, será utilizado como sistema de monitorización del directorio de entrada, facilitando por tanto conocer la existencia de un fichero a procesar.

  Por tanto, para nuestros objetivos, monitorizar un directorio buscando ficheros, no se usará el API que proporciona, sino que crearemos una colaboración a partir de la operación receive proporcionada por el eWay.

  El construir la colaboración desde la operación receive garantizará que la entrada al proceso de transformación únicamente será lanzada al depositar un fichero en el directorio que se encuentra monitorizado.

- **BatchLocal eWay:** Conector encargado de manejar ficheros. Es capaz de acceder al fichero, obtener su contenido, copiarlo, moverlo… en resumen, facilita la realización de operativas automatizables sobre el fichero.

  El uso que nuestro aplicativo realizará del este eWay será similar al uso que hacemos de un JMS eWay, es decir, a través de llamadas al API expuesto.

  El mecanismo es el mismo que para el caso de un JMS eWay. Al crear la colaboración se debe asignar y en el correspondiente Conectivity Map, asociar en la definición de la colaboración un componente de tipo BatchLocal.

```
public void receive(
com.stc.connector.batchadapter.appconn.BatchAppconnMessage input,
com.stc.eways.batchext.BatchLocal fileReader,
com.stc.eways.batchext.BatchRecord fileSplitter, com.stc.connectors.jms.JMS
inbound, ud1.boMessage1429383756.Bomessage boMessage,
stcgen.fcxotd.http___www_fgarcia1763_org_envelope.EnvelopeDocument
commonEnvelope )
        throws Throwable  {}
```

  Su uso, como ya se ha comentado será mediante la invocación del API que expone la clase.

```
/* Configuracion del eWay para realizar la lectura del fichero */
fileReader.reset();
fileReader.getConfiguration().setTargetDirectoryName( directorioFichero );
fileReader.getConfiguration().setTargetFileName( nombreFichero );
fileReader.getConfiguration().setTargetDirectoryNameIsPattern( false );
fileReader.getConfiguration().setTargetFileNameIsPattern( false );
```

```
/* Lectura del contenido del fichero */
fileReader.getClient().getInputStreamAdapter();

/* Liberacion de recursos */
if (fileReader.getClient() != null) {
        fileReader.getClient().getInputStreamAdapter().releaseInputStream( true );
}
```

- **BatchRecord eWay:** Recibe el contenido de un fichero y automatiza la extracción de fragmentos del mismo en base a un separador indicado. Será muy útil a la hora de extraer las diferentes operaciones que compondrán el fichero a procesar.

  El uso que nuestro aplicativo realizará del este eWay será similar al uso que hacemos en el caso de un JMS eWay o de un BatchLocal eWay, es decir, a través de llamadas al API expuesto.

  El mecanismo es el mismo que para el caso de los dos eWays anteriores. Al crear la colaboración se debe asignar y en el correspondiente Conectivity Map, asociar en la definición de la colaboración un componente de tipo BatchRecord.

```
public void receive(
com.stc.connector.batchadapter.appconn.BatchAppconnMessage input,
com.stc.eways.batchext.BatchLocal fileReader,
com.stc.eways.batchext.BatchRecord fileSplitter, com.stc.connectors.jms.JMS
inbound, ud1.boMessage1429383756.Bomessage boMessage,
stcgen.fcxotd.http___www_fgarcia1763_org_envelope.EnvelopeDocument
commonEnvelope )
        throws Throwable  {}
```

  Su uso, como ya se ha comentado será mediante la invocación del API que expone la clase.

```
/* Configuración del particionador */
fileSplitter.setInputStreamAdapter(
                fileReader.getClient().getInputStreamAdapter() );

/* Uso para la obtención de registros */
while (fileSplitter.get()) {
            linea = new String( fileSplitter.getRecord() );
            …
 }

/* Liberacion de recursos */
fileSplitter.reset();
```

- **Email eWay:** Las alertas que el sistema genere serán comunicadas a los usuarios a través de correo electrónico. Para realizar esta funcionalidad JavaCAPS nos proporciona un Endpoint que facilita la implementación de esta funcionalidad.

  El mecanismo de utilización es igual al de los eWays anteriores, así que no lo describiremos de nuevo, lo único indicar que aunque facilite el tratamiento de correo electrónico, su uso es muy básico, no permitiendo casi ninguna personalización.

### 5.2.4. Implementación de las piezas encargadas del proceso

Este apartado servirá para describir las diferentes piezas software utilizadas para construir nuestro sistema de transformación de formatos, así como la forma de comunicación entre ellas.

A continuación se enumeran a grandes rasgos los bloques que se deben construir. En esta enumeración no incluimos el orquestador ya que este módulo será comentado en un apartado propio.

- Entrada al sistema.
- Validación de contenidos.
- Transformación de formatos.
- Salida del sistema.
- Comunicación de alertas.

#### 5.2.4.1. Entrada al sistema

Como se ha especificado en apartados anteriores, la entrada al sistema de transformación y por tanto el desencadenante del proceso será la existencia de un fichero en un directorio predefinido.

Debido a que se pueden recibir en ambos sentidos, es decir, desde la entidad al sistema central y desde el sistema central a la entidad, se crearán dos lectores.

El primero de ellos se encargará de leer los ficheros que la entidad emite al sistema central y otro que realiza el proceso inverso, recibe los ficheros que el sistema central envía a la entidad.

Si observamos estos módulos como una caja negra, su funcionamiento es el siguiente:

1. Detecta la existencia de un fichero en el directorio de entrada utilizando el eWay BatchInbound.

2. Lee el fichero del sistema (utilizando un BatchLocal eWay).

3. Internamente, el módulo troceará el contenido del fichero en bloques, para realizar esta acción utilizará el eWay BatchRecord.

4. Por bloque, se genera un Common Envelope con los datos básicos para realizar la transformación, incluyendo el bloque propiamente dicho.

5. Cada Common Envelope será almacenado en una cache. De esta forma, se podrá controlar el proceso de transaformación.

6. Cada Common Envelope es enviado a la cola JMS de salida a través del conector apropiado (JMS eWay).

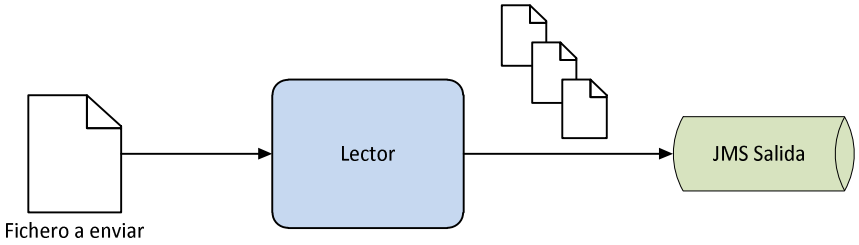



**Figura 79 - Funcionamiento básico del lector**

**[fuente: propia]**

A continuación vamos a mostrar el diagrama que describirá el funcionamiento de los módulos lectores.

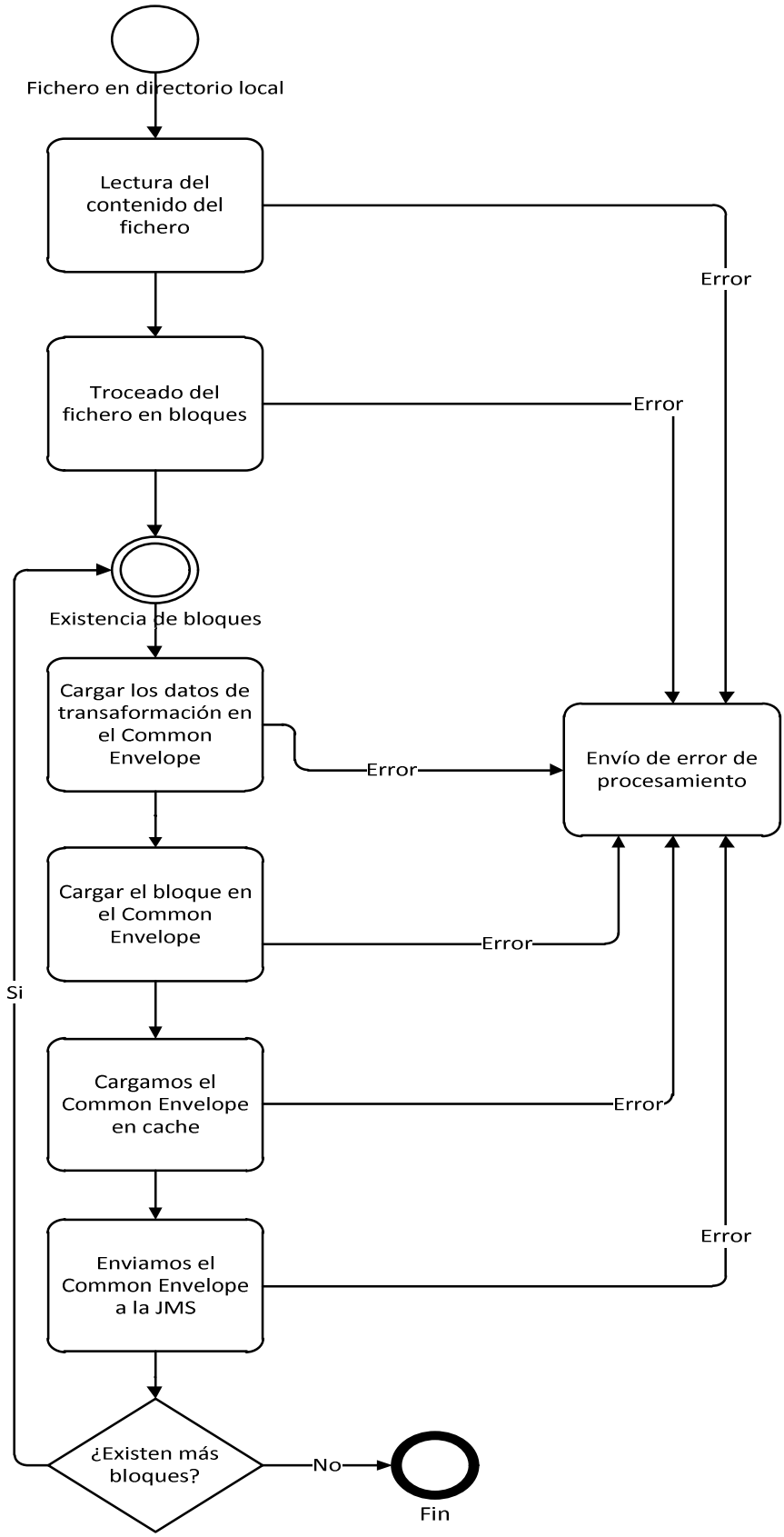



**Figura 80 - Proceso lector**

**[fuente: propia]**

#### 5.2.4.2. Validación de contenidos

Uno de los principales problemas que surgen cuando se intercambia información entre dos puntos es que el contenido de la información no es correcto, o el formato que se utiliza para realizarla no es el esperado por la otra parte.

Para garantizar que tanto el formato como el contenido son correctos necesitaremos realizar un proceso de validación, para ello, se deberán construir dos validadores por flujo de transformación, es decir, existirán:

- Un primer validador, que se ejecutará previo al proceso de transformación y será el encargado de comprobar el formato recibido.

- Un segundo validador, ejecutado después del proceso de transformación y que se encargará de comprobar si el formato que se va a entregar es el correcto.

A la vista de los formatos a transformar definidos en apartados anteriores, existen dos posibles tipos de validación:

- Validación del formato plano. Esta validación deberá realizarse campo a campo, comprobando:

    - Si el campo es obligatorio, se debe comprobar que está informado y que el formato es correcto.

    - Si el campo es opcional, se comprobará en el caso de que este informado si el formato del campo es correcto.

- Validación del formato XML: En este caso la validación es más sencilla ya que al poseer el esquema XSD únicamente se debe validar con él para saber si hay algún error.

En los párrafos anteriores se ha descrito que tipos de validadores existen y como se van a comportar. A continuación se describirá en detalle el funcionamiento de cada uno de ellos.

A alto nivel, el funcionamiento viendo el sistema como una caja negrá será el siguiente.

1. La validación será lanzada cuando recibamos un mensaje de tipo Common Envelope en la cola JMS de entrada utilizando el eWay JMS.

2. Una vez recibido el mensaje, se activa el módulo correspondiente, valida el contenido (cada operación), almacenando el resultado de la misma en el mismo mensaje.

3. Responde a través de una cola JMS de salida con el mismo Common Envelope recibido, que ahora contiene también el resultado de la validación invocando a los métodos proporcionados por el JMS eWay.

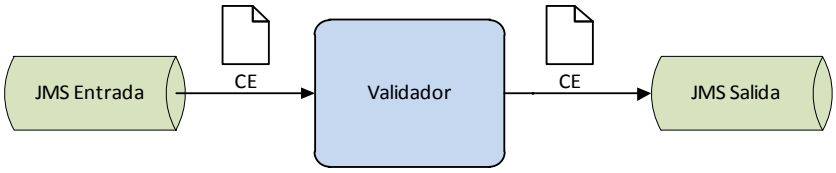



Figura 81 - Funcionamiento de un Validador

[fuente: propia]

A continuación se muestra un diagrama de flujo que intenta explicar el funcionamiento de los módulos de validación.

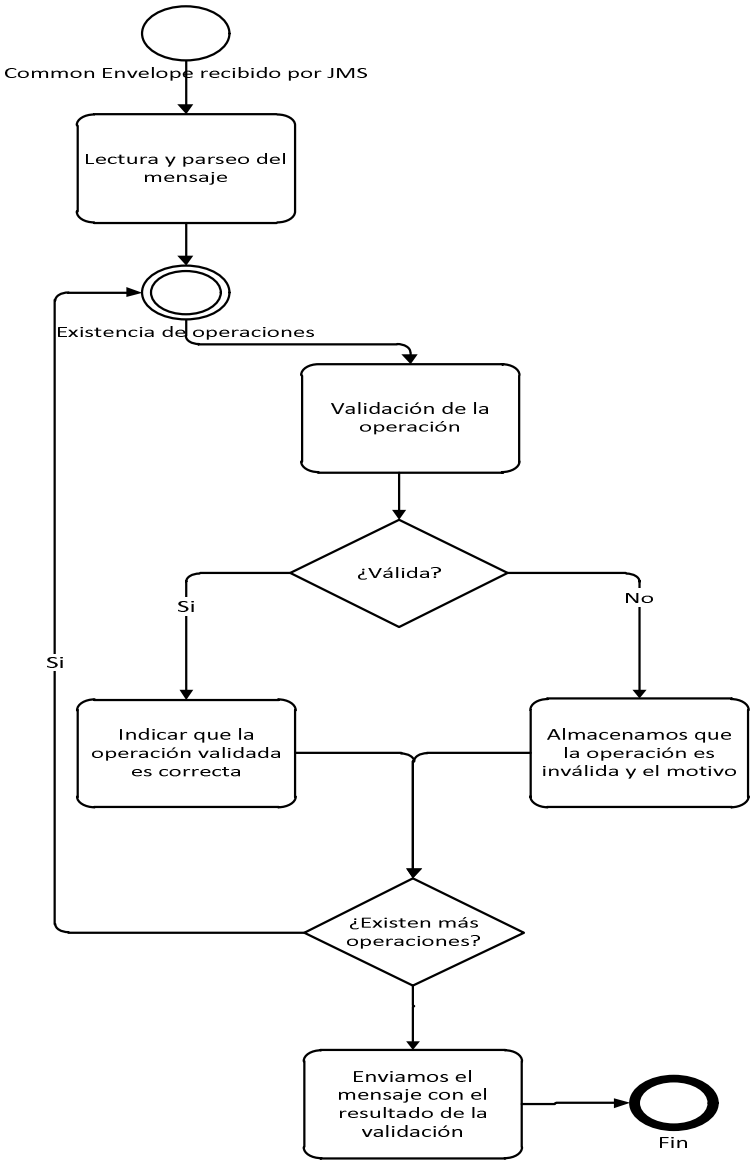



Figura 82 – Funcionamiento detallado de un Validador

[fuente: propia]

#### 5.2.4.3. Transformación de contenidos

Otro de los principales problemas que surgen cuando se intercambia información entre dos puntos es que el contenido de la información que espera el destinatario no es el mismo que el enviado por el emisor, es decir, el formato origen y destino no son los mismo.

Para garantizar que el emisor y el destinatario sean capaces de entenderse se necesita construir un traductor que se encargará de convertir el formato enviado en el formato esperado, es decir, existirán:

- Un transformador de formato plano a formato xml. Este transformador traducirá los mensajes enviados por la entidad al formato esperado por el sistema central.

- Un transformador de formato xml a formato plano. Este transformador será el encargado de traducir los mensajes enviados por el sistema central al formato que espera recibir la entidad.

En el párrafo anterior hemos descrito que dos transformadores existen y como se va a comportar. A continuación se describirá en detalle el funcionamiento.

A alto nivel, el funcionamiento viendo el sistema como una caja negrá será el siguiente.

1. La transformación será lanzada cuando recibamos un mensaje de tipo Common Envelope en la cola JMS de entrada utilizando el eWay de JMS.

2. Una vez recibido el mensaje, se activa el módulo correspondiente, transformando el contenido (cada operación), ubicado en OriginalOperations, almacenando el resultado de la misma en el correspondiente campo DestinationOperations.

3. Elimina del Common Envelope la operación original transformada.

4. Responde a través de una cola JMS de salida con el mismo Common Envelope recibido, que ahora contiene la transformación, para esto utiliza el API proporcionado por el JMS eWay.

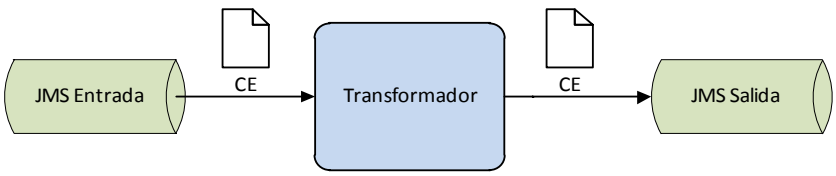



**Figura 83 - Funcionamiento de un Transformador**

**[fuente: propia]**

A continuación se muestra un diagrama de flujo que intenta explicar el funcionamiento de los módulos de transformación.

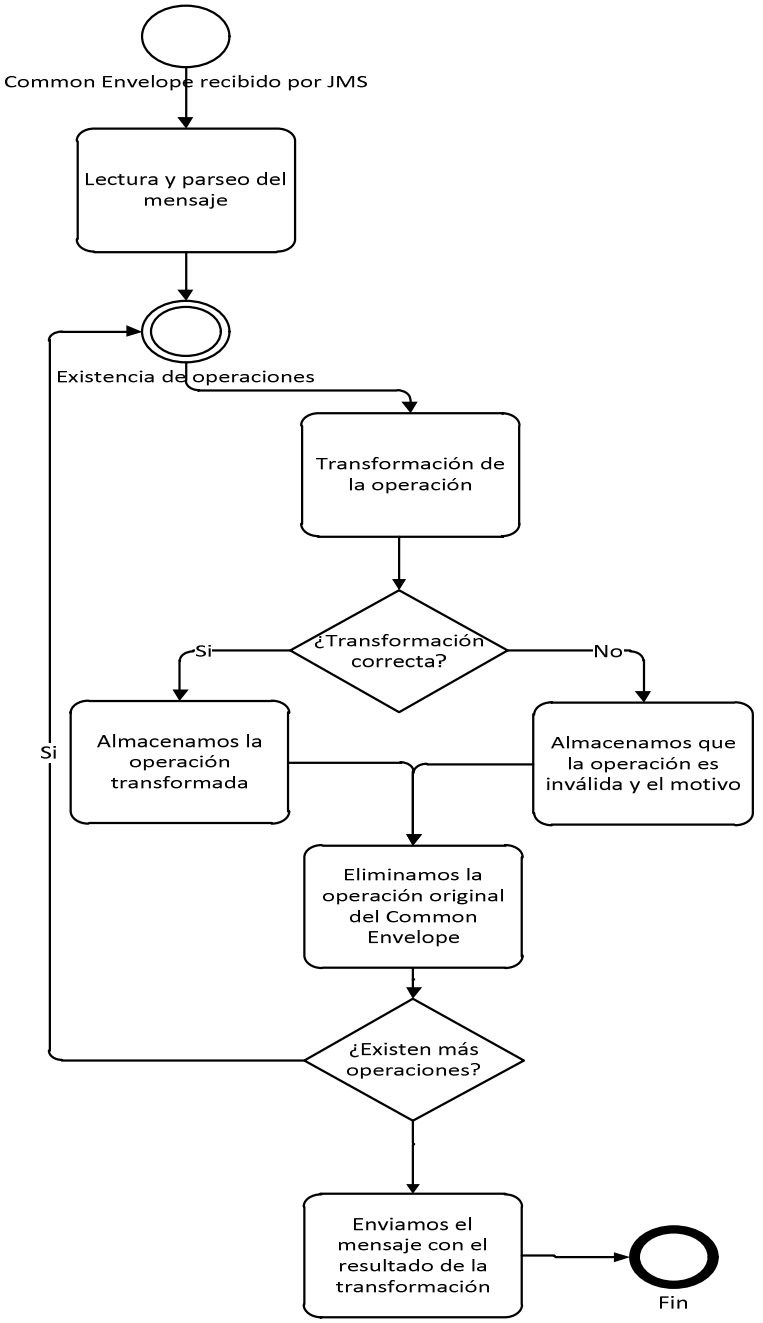



**Figura 84 - Funcionamiento detallado del Transformador**

**[fuente: propia]**

#### 5.2.4.4. Salida del sistema

El sistema de escritura será el encargado de generar el fichero final que se debe comunicar a la entidad o al sistema central, dependiendo del flujo de transformación que se haya ejecutado.

Al revés que en el caso de los módulos de entrada, en este caso, la activación del módulo de escritura se realizará con la recepción de un mensaje Common Envelope, generando un fichero con las operaciones transformadas en el correspondiente directorio de salida.

Debido a que se puede enviar un fichero en ambos sentidos, es decir, desde la entidad al sistema central y desde el sistema central a la entidad, se crearán dos escritores.

El primero de ellos se encargará de escribir el fichero con las operaciones transformadas que la entidad emite al sistema central y otro que realiza el proceso inverso, escribe los ficheros con las operaciones transformadas que el sistema central envía a la entidad.

Si observamos estos módulos como una caja negra, su funcionamiento es el siguiente:

1. Detecta la recepción de un Common Envelope en la cola de entrada al módulo, para esto utiliza el eWay JMS.

2. Parsea el Common Envelope recibido.

3. Marcamos el bloque como finalizado y escribimos un fichero temporal con la transformación recibida.

4. Si todos los bloques en que se descompuso el fichero original se encuentran procesados, listamos todos los ficheros temporales generados y los fusionamos.

5. Escribimos el fichero con todas las operaciones transformadas en el directorio donde lo recogerá el destinatario.

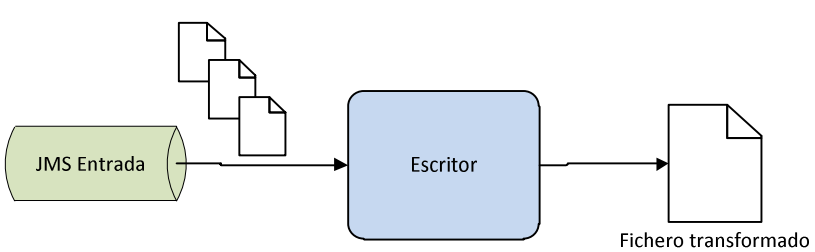



**Figura 85 - Funcionamiento básico del Escritor**

**[fuente: propia]**

A continuación vamos a mostrar el diagrama que describirá el funcionamiento de los módulos escritores.

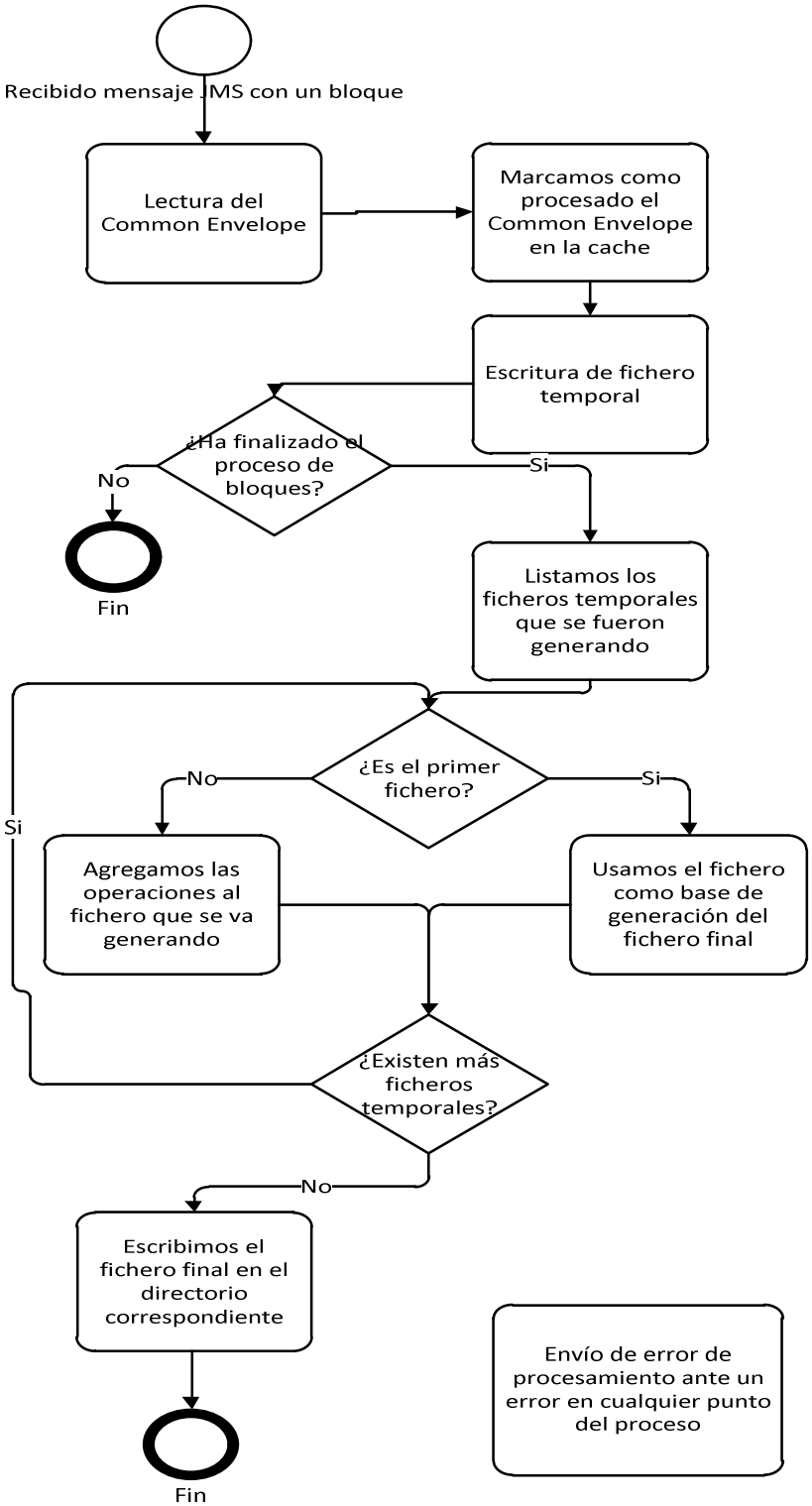



**Figura 86 - Funcionamiento del Escritor**

**[fuente: propia]**

#### 5.2.4.5. Comunicación de alertas

Ante la existencia de errores, el sistema necesita de alguna forma poder comunicar los errores de forma que puedan ser solucionados.

El módulo de comunicación de alertas será muy básico, únicamente recibirá un mensaje con la alerta a través de una cola JMS y lo enviará por eMail a los correspondientes destinatarios.

El funcionamiento por tanto será el siguiente:

1. Detecta la recepción de un mensaje con la alerta correspondiente en la cola de entrada al módulo, para esto utiliza el eWay JMS.

2. Se rellenará el cuerpo del eMail con los datos recibimos en el mensaje de alerta.

3. Se enviará el mail con la alerta. Para realizar esta acción utilizaremos el eMail eWay.

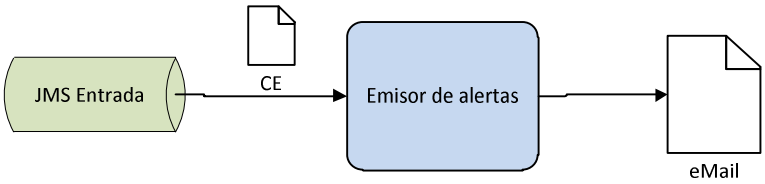



Figura 87 - Funcionamiento básico del Módulo de Alertas

[fuente: propia]

A continuación se detalla el proceso de envío de alertas.

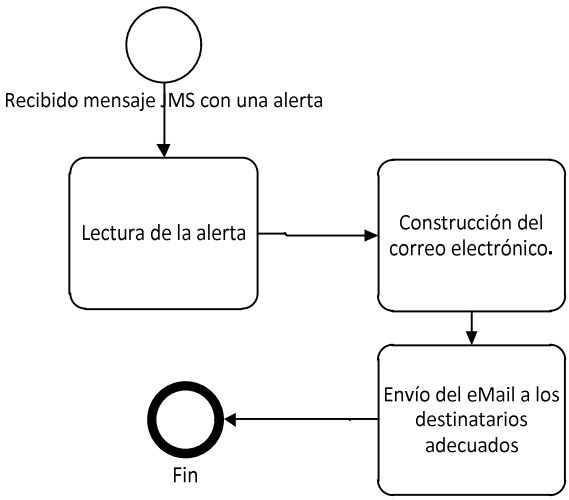



Figura 88 - Funcionamiento detallado del Módulo de Alertas

[fuente: propia]

### 5.2.5. Construcción del orquestador

En este apartado vamos a comentar como se implementó la pieza principal del sistema de transformación.

Aunque el resto de piezas comentadas definen la funcionalidad del sistema propiamente dicha, es decir, recepción, validación, transformación y emisión, la realización de la funcionalidad anteriormente descrita no sería posible sino incorporáramos un proceso orquestador.

Este tipo sistema, donde los diferentes módulos se comunicarán entre sí mediante un canal de comunicación (JMS) y que debe cumplir la premisa fundamental de que sea desacoplado, es decir, aunque los diferentes módulos deben colaborar entre ellos para realizar la transformación, no saben por si solos como organizarse entre ellos para colaborar ya que esto haría que se incrementara el acoplamiento al tener que incluir código de orquestación.

Para evitar acoplar los módulos incluyendo código de orquestación, construiremos una pieza de gobernanza, a la cual denominaremos orquestador. Este orquestador poseerá la siguiente funcionalidad:

1. Conoce desde donde se recibe un mensaje (Common Envelope) y hacía donde tiene que enviarlo para continuar el proceso.

2. Realiza el tratamiento de bloques u operaciones que han sido comunicadas como erróneas por otros módulos.

3. Gestiona la cache de bloques, comprobando donde ha sido enviado el mensaje con el bloque, si se ha recibido respuesta o no para un envío…

Como en casos anteriores, viendo el orquestador como una caja negra podemos describirlo como se muestra en la siguiente figura:

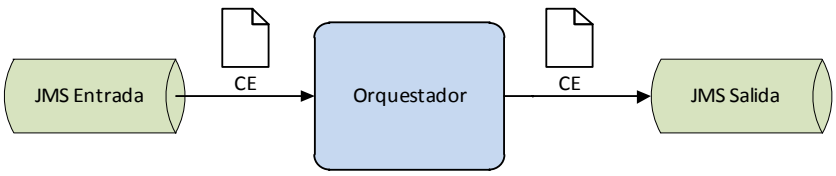



**Figura 89 - Orquestador**

**[fuente: propia]**

El orquestador recibirá mensajes procedentes del sistema de entrada y respuestas a invocaciones realizadas a los módulos de validación y transformación, con lo cual, como se ha indicado anteriormente, dispondrá de un mecanismo que le permite saber de dónde viene el mensaje recibido y hacía donde deberá ser encaminado.

Para realizar esto utilizará los **Steps** definidos en el Common Message.

El proceso se puede resumir de la siguiente forma:

1. El orquestador recibe un Common Envelope en la cola JMS de entrada utilizando el eWay de JMS. Este mensaje puede proceder tanto del sistema de entrada (es un bloque procedente del particionado del fichero recibido) o como respuesta a un envío previo, por ejemplo, una solicitud de validación.

2. El orquestador comprueba si el mensaje estaba ya siendo procesado, es decir, es una respuesta a una solicitud realizada previamente o bien, es el inicio del proceso de ese bloque.

3. Si el bloque no estaba siendo procesado, el orquestador lo incluirá en la cache de control de bloques procesados y lo enviará al primer paso del proceso.

4. Si el bloque estaba siendo procesado, el mensaje recibido es una respuesta a una solicitud de proceso previa. En este caso, lo primero que debemos comprobar es si se encuentra expirado, es decir, la respuesta llega tarde o no.

5. En el caso de que la respuesta llegue tarde, anulamos el fichero y comunicamos el error al sistema de salida para que trate el caso de fichero inválido.

6. Si la respuesta recibida se encuentra en tiempo, comprobaremos si el mensaje contiene operaciones erróneas.

7. En caso afirmativo, anulamos el fichero y comunicamos el mensaje al proceso de salida para que trate el caso de fichero inválido.

8. En caso negativo, buscamos el siguiente paso del procesado al que enviar el mensaje, lo marcamos como actual, actualizamos la cache de procesados y lo enviamos para que continúe el flujo de transformación. Para el envío utilizaremos el API expuesto por el eWay de JMS.

A continuación veremos un diagrama detallado del flujo de proceso que gobierna el orquestador.

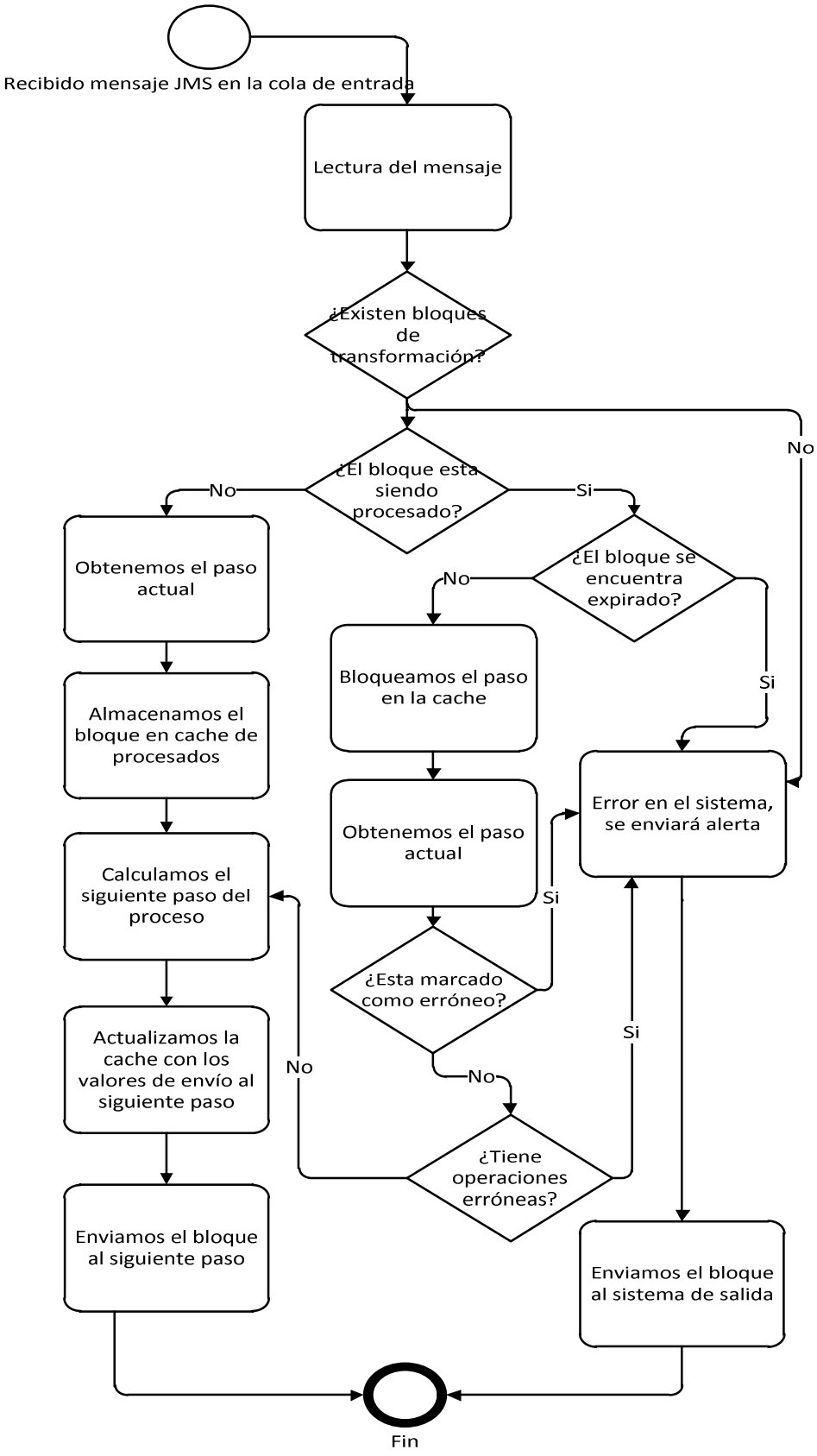



**Figura 90 - Funcionamiento detallado del Orquestador**

**[fuente: propia]**

## 5.3. Implementación utilizando software libre

La implementación utilizando software libre, al igual que en el caso de software propietario constará de los siguientes pasos:

- Definición de los formatos que permitirán interpretar la información.
- Implementación de los endpoints, encargados de permitir la comunicación entre los sistemas externos y el sistema de transformación.
- Implementación de las piezas encargadas del proceso.
- Construcción del orquestador. Esta pieza será la encargada de organizar el flujo de transformación, controlando que paso se debe ejecutar cada vez.

### 5.3.1. Definición de formatos

En este caso, a diferencia de la implementación anterior no disponemos de un entorno gráfico que nos permita la definición de los formatos que el sistema requiere para poder interpretar la información contenida en los mensajes que se recibirán.

En el caso de utilizar software libre para construir el sistema, no se dispone de un entorno para la definición de formatos como en el caso de JavaCAPS. En este caso únicamente se utilizarán herramientas libres para construirlos.

- Definición de la estructura de datos encargada de procesar las operaciones recibidas desde el BO.

  Para la definición de este tipo de formatos no existe una herramienta capaz de abstraer la estructura del documento, generando las clases necesarias para poder interpretarlo. Por tanto, para realizar el tratamiento del formato BO se propondrán dos opciones:

  - Utilización de extractores de línea. Al ser un formato plano, con líneas de longitud fija, no es muy complejo el tratamiento del contenido línea a línea.

  - Utilización de expresiones regulares, las cuales son capaces de validar que la línea que se trata sea correcta tanto en estructura como en contenido.

- Definición de las estructuras encargadas de procesar el mensaje común que compartirán las diferentes piezas software, así como la encargada de procesar las operaciones del sistema central.

  En este caso si existe una forma de definir el formato de documento en el caso de que este sea de tipo xml y dispongamos del correspondiente esquema xsd. Para la realización de esta tarea se utilizará Maven junto con el plugin xjc (el cual utiliza JAXB para la generación de clases a partir de un esquema xsd).

A continuación se muestra un ejemplo sobre cómo configurar y utilizar el plugin.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxb2-maven-plugin</artifactId>
    <configuration>
        <quiet>true</quiet>
        <verbose>false</verbose>
        <clearOutputDir>false</clearOutputDir>
        <readonly>true</readonly>
    </configuration>
    <executions>
         <execution>
         <id>xjc</id>
         <goals>
              <goal>xjc</goal>
         </goals>
         <configuration>
   <packageName><nombre paquete ubicar fuentes generados></packageName>
            <sources>
                 <source>${basedir}/src/xsd/<nombre esquema>.xsd</source>
            </sources>
            <outputDirectory>${basedir}/src/main/generated</outputDirectory>
   <staleFile>${project.build.directory}/generated/.jaxb-staleFlag-msg</staleFile>
       </configuration>
     </execution>
</plugin>
```

Como se puede ver en el fragmento de código maven, la configuración para el uso del plugin es muy simple, a nivel básico únicamente se deben configurar:

- El nombre del paquete donde se dejarán las clases generadas.
- La ruta, incluyendo el nombre del esquema utilizado para generar las clases.
- El directorio donde se generará la estructura.

### 5.3.2. Introducción al desarrollo en Camel: Conceptos

Antes de comenzar a explicar cada una de las piezas que se han construido, es recomendable explicar, de forma resumida, algunos conceptos básicos sobre Camel.

Antes de comenzar a explicar los conceptos básicos sobre cómo utilizar Camel para realizar un desarrollo, se deben mencionar las dos formas de posibles de implementación utilizándolo.

- **Uso de un lenguaje específico de dominio (DSL)**, el cual permite definir el sistema utilizando únicamente en código Java. Esta aproximación no será la utilizada para construir el sistema de transformación.

- **Camel + Spring.** Esta aproximación mezcla la potencia de Springframework y Camel. La definición del sistema se realizar utilizando los xmls de configuración utilizados para Spring, los cuales permitirán definir también los componentes que utilizará Camel.

Antes de definir cualquier otro componente, se definirá uno de los principales y menos visibles de Camel, el **Exchage**. A grandes rasgos, un **Exchange** no es más que un mensaje utilizado por Camel utilizado para intercambiar información entre los diferentes elementos que existentes en una ruta, es decir, es una especie mensaje común a utilizar entre elemento y se encuentra compuesto por dos secciones principalmente:

- **Headers:** Cabeceras que posee el mensaje.

- **Body:** Cuerpo del mensaje.

Los siguientes elementos los vamos a definir estableciendo un paralelismo entre Camel y JavaCAPS. Se podría decir, siempre con matices, que el concepto de **Ruta** en Camel, se asemeja al concepto **Colaboración** de JavaCAPS. A continuación intentaremos explicar la afirmación anterior.

Como se vio en el apartado 5.2.2, una colaboración en JavaCAPS consta de:

- Un punto de entrada, en nuestro caso, reimplementar una operación existente de un Enpoint.

- Un bloque de código, que realiza las acciones correspondientes sobre la entrada recibida.

- Una o más posibles salidas utilizando otros Endpoints recibidos como argumento.

A continuación se verán los componentes de una ruta Camel que definen los puntos anteriores.

- Un punto de entrada a la ruta, definido en Camel por la palabra clave **from** junto con una **uri** o una **referencia a bean**. Tanto la uri como la referencia a un bean definen un Enpoint de entrada.

```
<from uri="quartz2://listAccount?cron=0/120+*+*+*+*+?" />
```

- Un bloque de código que realiza las acciones que necesitemos sobre la entrada. En Camel, utilizaremos clases e invocaremos a sus métodos. Ya que utilizamos Spring, se delegará en el instanciar el objeto, utilizando dicha instancia desde la ruta Camel.

La ruta por tanto arrancará desde el Endpoint definido en la etiqueta **from**. A continuación se mostrará un ejemplo simple de definición de una ruta.

- El punto de bloque de código será cubierto con la clase y el método a utilizar en nuestra ruta, es decir, la ruta debe constar con un elemento de tipo **bean**. Por otro lado, si en el método se va a comunicar con otros componentes de la ruta deberá recibir el **Exchange** como parámetro, de forma que pueda acceder a él para consultar valores o para modificarlo.

  Para definir un bean debemos seguir los siguientes pasos:

  1. Crear la clase que incluye al menos el método que vamos a utilizar. Si el bean se debe comunicar con el resto de elementos, el método recibirá como parámetro el elemento **Exchange**.

     ```
     public class ReadBOService {
         …
         public void read(Exchange exchange) { … }
         …
     }
     ```

     Camel cuando vaya a invocar el método, observará que debe recibir el **Exchange** y se encargará de pasárselo de forma transparente para que pueda usarlo.

  2. Definir la creación de la instancia de la clase en el correspondiente fichero de configuración de Spring.

     ```
     <bean id="readBOService" class="x.y.z.ReadBOService"/>
     ```

  3. Utilizar el bean definido en la ruta, para ello se usará la tag **bean**, la cual hará referencia a la instancia especificada en el fichero de configuración de Spring.

     ```
     <bean ref="readBOService" method="read" />
     ```

- El último punto que se debe cubrir son las salidas de una ruta. Para ello tenemos dos opciones:

  - Definir en la ruta un elemento **to**, similar al elemento **from** que actuá como punto de entrada a la ruta. Su definición se realiza en el correspondiente fichero xml de configuración de Camel.

  - Utilizar dentro del **bean** una **producer template**, es decir, realizar el envío utilizando código java, es decir, esta aproximación es totalmente programatica.

  Mientras que la primera de las opciones únicamente nos permite definir una salida en la ruta, el uso del elemento **Producer Template** permite poder tener múltiples puntos de salida.

  Un ejemplo sería `<to uri="direct:envioElemento”/>`

A continuación vemos un ejemplo de una ruta completa:

```
   <route id="read-BO-file">
      <from uri="file:d:/temp/bo/input?move=d:/temp/bo/input
/backup/${file:onlyname}.${date:now:yyyyMMdd'T'HHmmssSSS}&amp;moveFail
ed=d:/temp/bo/input/error/${file:onlyname}.error.${date:now:yyyyMMdd'T
'HHmmssSSS}&amp;include=.*\.txt$&amp;charset=utf-8" />
      <bean ref="readBOService" method="read" />
      ...
      <to uri="direct:envioElemento”/>
   </route>
```

Ampliando un poco más las similitudes entre Camel y JavaCAPS, una ruta también puede ser similar al concepto de **Conectivity Map**, ya que nos permite encadenar diferentes acciones para conseguir un propósito, para ello, la salida de un paso será la entrada del siguiente, por ejemplo:

```
   <route id="read-BO-file">
      <from uri="file:d:/temp/bo/input?move=d:/temp/bo/input
/backup/${file:onlyname}.${date:now:yyyyMMdd'T'HHmmssSSS}&amp;moveFail
ed=d:/temp/bo/input/error/${file:onlyname}.error.${date:now:yyyyMMdd'T
'HHmmssSSS}&amp;include=.*\.txt$&amp;charset=utf-8" />
      <bean ref="readBOService"    method="read" />
      <bean ref="validateService"  method="validate" />
      <bean ref="transformService" method="transform" />
      ...
      <to uri="direct:envioElemento”/>
   </route>
```

A parte de poder definir una secuencia de acciones a realizar, utilizando Camel se podrá encadenar la ejecución de rutas. Para conseguir esto se deberá una ruta deberá de alguna forma invocar a la siguiente, esto en nuestro caso se conseguirá mediante el envío de un mensaje a una cola JMS. Veamos un ejemplo.

```
   <route id="read-BO-file">
      <from uri="file:d:/temp/bo/input?move=d:/temp/bo/input
/backup/${file:onlyname}.${date:now:yyyyMMdd'T'HHmmssSSS}&amp;moveFail
ed=d:/temp/bo/input/error/${file:onlyname}.error.${date:now:yyyyMMdd'T
'HHmmssSSS}&amp;include=.*\.txt$&amp;charset=utf-8" />
      <bean ref="readBOService" method="read" />
      <to uri="activemq:queue:inbound”/>
   </route>

   <route id="orquestador">
      <from
uri="activemq:queue:inbound?concurrentConsumers=10&amp;asyncConsumer=t
rue" />
      <bean ref="orquestator" method="process" />
   </route>
```

Como puede observarse en el ejemplo anterior, tenemos dos rutas, la primera lee de un directorio el correspondiente fichero y debe invocar a la segunda, nuestro orquestador.

Para ello, al finalizar la primera ruta, indicamos que el elemento **Exchange** debe ser enviado a la cola inbound, esto lo realizamos mediante la etiqueta **to**.

El punto de entrada de la segunda ruta, definido a través de la etiqueta **from** es también la cola inbount, esto significa que la segunda ruta está pendiente de recibir un mensaje en esa cola.

Cuando la primera ruta emite el mensaje a la cola inbound, la segunda ruta lo recibirá, activándose y procesando el mensaje recibido. Se puede decir por tanto, que la activación de la segunda ruta será mediante el evento de recepción de un mensaje en la cola inbound.

Mediante el procedimiento descrito en los párrafos anteriores se podrán encadenar múltiples rutas, todas activadas mediante la recepción del correspondiente mensaje en la cola definida en la etiqueta **from**. Este proceso también es similar a un **Conectivity Map** de JavaCAPS, encadenando la salida de una colaboración con la entrada de otra.

Con lo visto anteriormente se definiría completamente lo que podemos denominar **Capa Software**, es decir, el código del aplicativo y cómo va a interactuar entre sí.

Ahora deberemos definir de alguna forma el hardware sobre el que trabajará la aplicación. En JavaCAPS se disponía de un mecanismo adicional, el **Environment**, que nos permitía definir el hardware y relacionarlo con el software justo antes de compilarlo.

En Camel no existe ningún mecanismo de este tipo, es decir, el hardware se configurará en el mismo xml donde se define la ruta o en algún xml. Por ejemplo:

```
<from uri="file:d:/temp/bo/input?move=d:/temp/bo/input
/backup/${file:onlyname}.${date:now:yyyyMMdd'T'HHmmssSSS}&amp;moveF
ailed=d:/temp/bo/input/error/${file:onlyname}.error.${date:now:yyyy
MMdd'T'HHmmssSSS}&amp;include=.*\.txt$&amp;charset=utf-8" />
```

Aun así, se podrán utilizar properties para configurar de manera externa las propiedades del hardware a utilizar (hosts, puertos, usuarios, passwords, rutas, colas jms…).

### 5.3.3. Uso de Endpoints

Como se ha visto en anteriores definiciones, los EndPoints son piezas de código que:

- Conectan (o permiten comunicar) el sistema interno (nuestro transformador) con los diferentes sistemas externos. En nuestro caso concreto permiten comunicar el sistema de transformación, con los BO del Sistema Central y de la Entidad.

- Permiten acceder al sistema de mensajería, de forma que las diferentes piezas que componen el sistema puedan utilizarlo de forma transparente.

El software seleccionado, Camel, nos ofrece una amplia colección de EndPoints, los cuales encapsulan la funcionalidad requerida, de forma que únicamente debemos utilizarlos. En nuestro ejemplo utilizaremos los siguientes:

- **activemq:** Permite crear y enviar mensajes a las colas JMS sobre un servidor ActiveMQ.

  A continuación se verán varios ejemplos sobre cómo se utilizará este Endpoint:

  1. Como punto de entrada en una ruta, se define en la etiqueta **from** de la misma.

     ```
     <from
     uri="activemq:queue:inbound?concurrentConsumers=10&amp;asyncC
     onsumer=true" />
     ```

     En el bloque xml anterior se define como punto de entrada a la ruta la recepción de un mensaje en la cola inbound, para ello utilizamos la estructura <conector>:<tipo>:<nombre>, en nuestro caso:

     - Conector: activemq, es un bean que contiene la definición de conexión con el gestor de colas.
     - Tipo: Se indica si accedemos a una cola (queue) o a un topic.
     - Nombre: El nombre de la cola a la que queremos acceder.

     Por otro lado, los parámetros definen propiedades adicionales de configuración, como por ejemplo el número de consumidores concurrentes o que el consumo es de tipo asíncrono.

  2. Para emitir mensajes se disponen de dos opciones, tal y como se ha visto en el apartado anterior.

     1. Mediante el uso de la etiqueta xml **to**, como por ejemplo:

        ```
        <to uri="activemq:queue:inbound”/>
        ```

        A través de esta definición de acción, que será la última de las definidas en la ruta, se indica que el resultado del procesado debe ser enviado a la cola inbound. Igual que en el caso anterior, la estructura para definir esta acción es <conector>:<tipo>:<nombre>.

     2. Mediante el uso de una **Producer Template**. Este elemento Camel permite realizar múltiples envíos en cualquier punto del código java.

        Para poder utilizar una template se deberán seguir los siguientes pasos:

        2.1. Definir el elemento en el xml de configuración de Camel.

```
<template id="producerTemplate" />
```

2.2. En el bean que la va a utilizar, utilizando Spring realizamos la inyección del componente, para identificarlo utilizamos el id con el que la hemos definido, en el caso del ejemplo el id sería *producerTemplate*.

2.3. Utilizando el API expuesto por el elemento **Producer Template**, realizamos el envío.

```
String body = .... // Mensaje que vamos a enviar
Map<String, Object> headers = … // Cabeceras

producerTemplate.sendBodyAndHeaders("activemq:queue
:inbound", body, headers);
```

- **file:** Encargado de monitorizar el sistema de ficheros, será utilizado como sistema de monitorización del directorio de entrada, facilitando por tanto conocer la existencia de un elemento que debamos procesar.

  Por tanto, para nuestros objetivos, monitorizar un directorio buscando ficheros, definiremos este elemento como punto de entrada en el elemento ruta correspondiente.

```
<from uri="file:d:/temp/bo/input?move=d:/temp/bo/input
/backup/${file:onlyname}.${date:now:yyyyMMdd'T'HHmmssSSS}&amp;mo
veFailed=d:/temp/bo/input/error/${file:onlyname}.error.${date:no
w:yyyyMMdd'T'HHmmssSSS}&amp;include=.*\.txt$&amp;charset=utf-8"
/>
```

  El bean de procesado correspondiente deberá recibir un elemento de tipo Exchange, el cual contendrá:

  - Un elemento body con el contenido del fichero que se ha leído del sistema de ficheros.

  - Una sección headers con los datos del fichero como por ejemplo, ruta desde donde se ha leído, nombre original, tamaño…

- **mail:** Las alertas que el sistema genere serán comunicadas a los usuarios a través de correo electrónico. Para realizar esta funcionalidad Camel nos proporciona un EndPoint para enviar un email.

  Al ser un EndPoint de envío, como se ha visto existen dos opciones de uso, mediante la etiqueta xml **to** o mediante una **Producer Template**, pero ya que se debe poder enviar una alerta en cualquier momento, la opción elegida será realizar el envío mediante una **Producer Template**, ya que nos permite

configurar el EndPoint en código, lo cual ofrece una mayor versatilidad que la configuración en xml.

### 5.3.4. Implementación de las piezas encargadas del proceso

Este apartado servirá para describir las diferentes piezas software utilizadas para construir nuestro sistema de transformación de formatos, así como la forma de comunicación entre ellas.

A continuación se enumeran a grandes rasgos los bloques que se deben construir. En esta enumeración no incluimos el orquestador ya que este módulo será comentado en un apartado propio.

- Entrada al sistema.
- Validación de contenidos.
- Transformación de formatos.
- Salida del sistema.
- Comunicación de alertas.

#### 5.3.4.1. Entrada al sistema

Como se ha especificado en apartados anteriores, la entrada al sistema de transformación y por tanto el desencadenante del proceso será la existencia de un fichero en un directorio predefinido.

Debido a que se pueden recibir en ambos sentidos, es decir, desde la entidad al sistema central y desde el sistema central a la entidad, se crearán dos lectores.

El primero de ellos se encargará de leer los ficheros que la entidad emite al sistema central y otro que realiza el proceso inverso, recibe los ficheros que el sistema central envía a la entidad.

Si observamos estos módulos como una caja negra, su funcionamiento es el siguiente:

1. Detecta la existencia de un fichero en el directorio de entrada utilizando el EndPoint **file2** existente en Camel.

2. Lee el fichero del sistema, para ello utilizaremos código estándar java.

3. Se troceará el contenido del fichero en bloques, bien utilizando un lector de líneas, en el caso de un fichero plano o utilizando Stax en el caso de ficheros xml.

4. Por bloque, se genera un Common Envelope con los datos básicos para realizar la transformación, incluyendo el bloque propiamente dicho.

5. Cada Common Envelope será almacenado en una cache. De esta forma, se podrá controlar el proceso de transaformación.

6. Cada Common Envelope es enviado a la cola JMS de salida a través del conector apropiado (activemq).

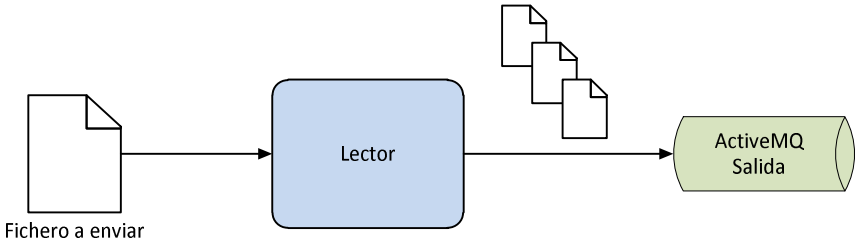



**Figura 91 - Funcionamiento básico del lector**

**[fuente: propia]**

A continuación vamos a mostrar el diagrama que describirá el funcionamiento de los módulos lectores.

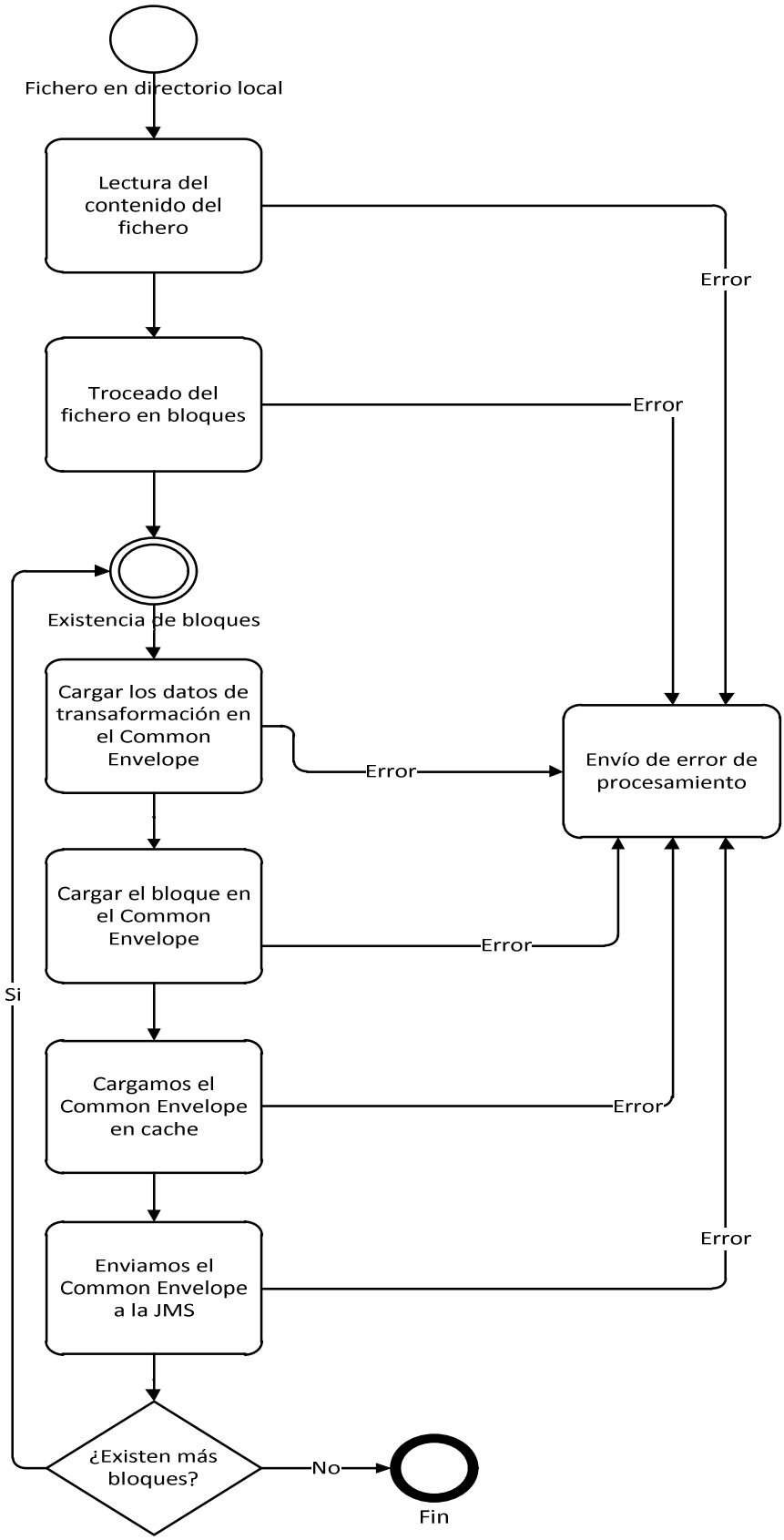



**Figura 92 - Proceso lector**

**[fuente: propia]**

#### 5.3.4.2. Validación de contenidos

Uno de los principales problemas que surgen cuando se intercambia información entre dos puntos es que el contenido de la información no es correcto, o el formato que se utiliza para realizarla no es el esperado por la otra parte.

Para garantizar que tanto el formato como el contenido son correctos necesitaremos realizar un proceso de validación, para ello, se deberán construir dos validadores por flujo de transformación, es decir, existirán:

- Un primer validador, que se ejecutará previo al proceso de transformación y será el encargado de comprobar el formato recibido.

- Un segundo validador, ejecutado después del proceso de transformación y que se encargará de comprobar si el formato que se va a entregar es el correcto.

A la vista de los formatos a transformar definidos en apartados anteriores, existen dos posibles tipos de validación:

- Validación del formato plano. Esta validación deberá realizarse campo a campo, comprobando:

    - Si el campo es obligatorio, se debe comprobar que está informado y que el formato es correcto.

    - Si el campo es opcional, se comprobará en el caso de que este informado si el formato del campo es correcto.

- Validación del formato XML: En este caso la validación es más sencilla ya que al poseer el esquema XSD únicamente se debe validar con él para saber si hay algún error.

En los párrafos anteriores se ha descrito que tipos de validadores existen y como se van a comportar. A continuación se describirá en detalle el funcionamiento de cada uno de ellos.

A alto nivel, el funcionamiento viendo el sistema como una caja negrá será el siguiente.

1. La validación será lanzada cuando recibamos un mensaje de tipo Common Envelope en la cola JMS de entrada definida mediante el EndPoint **ActiveMQ** en la etiqueta xml **from** de la ruta.

2. Una vez recibido el mensaje, se activa el módulo correspondiente, valida el contenido (cada operación), almacenando el resultado de la misma en el mismo mensaje.

3. Responde a través de una cola JMS de salida con el mismo Common Envelope recibido, que ahora contiene también el resultado de la validación. Para definir el EndPoint de salida se usará la etiqueta xml **to**.

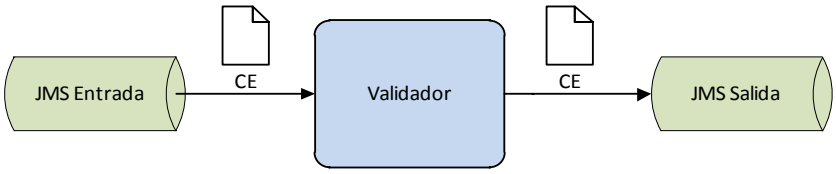



**Figura 93 - Funcionamiento de un Validador**

**[fuente: propia]**

A continuación se muestra un diagrama de flujo que intenta explicar el funcionamiento de los módulos de validación.

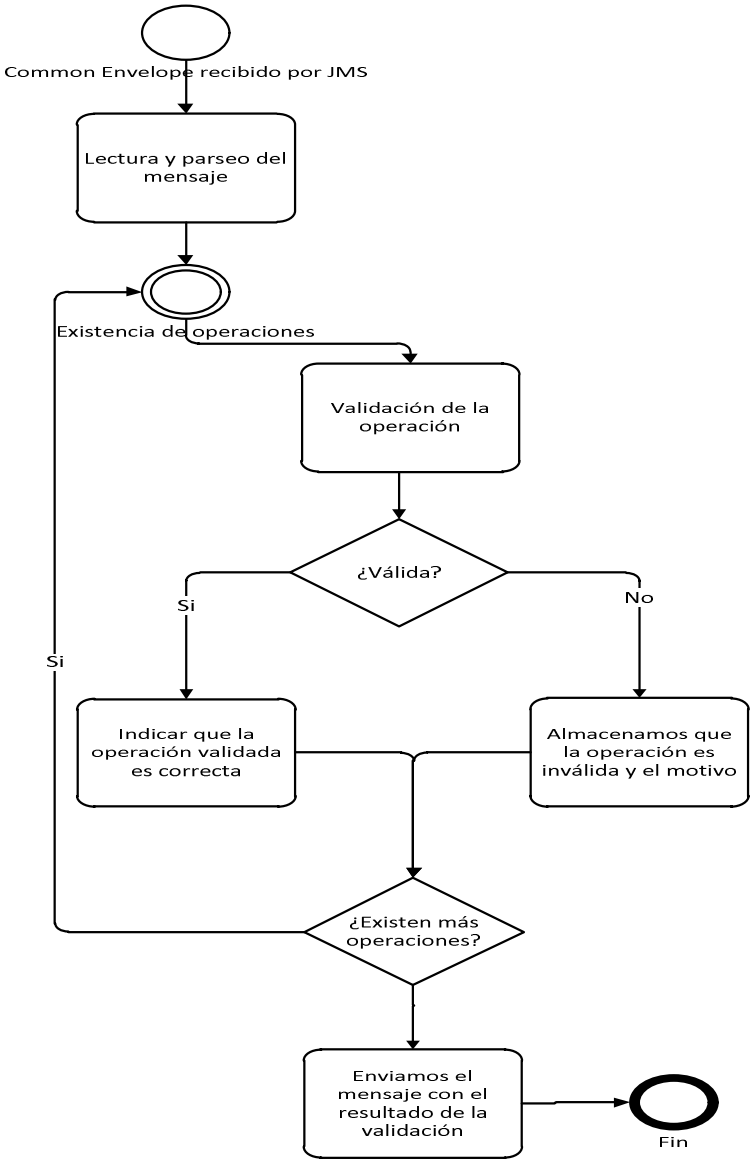



**Figura 94 – Funcionamiento detallado de un Validador**

**[fuente: propia]**

### 5.3.4.3. Transformación de contenidos

Otro de los principales problemas que surgen cuando se intercambia información entre dos puntos es que el contenido de la información que espera el destinatario no es el mismo que el enviado por el emisor, es decir, el formato origen y destino no son los mismo.

Para garantizar que el emisor y el destinatario sean capaces de entenderse se necesita construir un traductor que se encargará de convertir el formato enviado en el formato esperado, es decir, existirán:

- Un transformador de formato plano a formato xml. Este transformador traducirá los mensajes enviados por la entidad al formato esperado por el sistema central.

- Un transformador de formato xml a formato plano. Este transformador será el encargado de traducir los mensajes enviados por el sistema central al formato que espera recibir la entidad.

En el párrafo anterior hemos descrito que dos transformadores existen y como se va a comportar. A continuación se describirá en detalle el funcionamiento.

A alto nivel, el funcionamiento viendo el sistema como una caja negrá será el siguiente.

1. La transformación será lanzada cuando recibamos un mensaje de tipo Common Envelope en la cola JMS de entrada definida mediante el EndPoint **ActiveMQ** en la etiqueta xml **from** de la ruta.

2. Una vez recibido el mensaje, se activa el módulo correspondiente, transformando el contenido (cada operación), ubicado en OriginalOperations, almacenando el resultado de la misma en el correspondiente campo DestinationOperations.

3. Elimina del Common Envelope la operación original transformada.

4. Responde a través de una cola JMS de salida con el mismo Common Envelope recibido, que ahora contiene la transformación. . Para definir el EndPoint de salida se usará la etiqueta xml **to**.

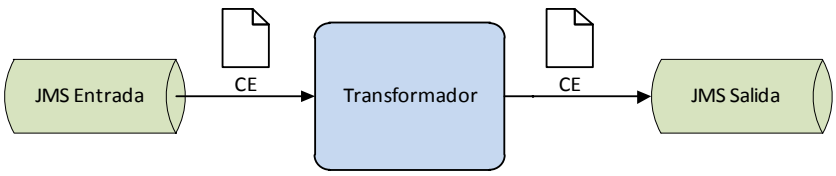



**Figura 95 - Funcionamiento de un Transformador**

**[fuente: propia]**

A continuación se muestra un diagrama de flujo que intenta explicar el funcionamiento de los módulos de transformación.

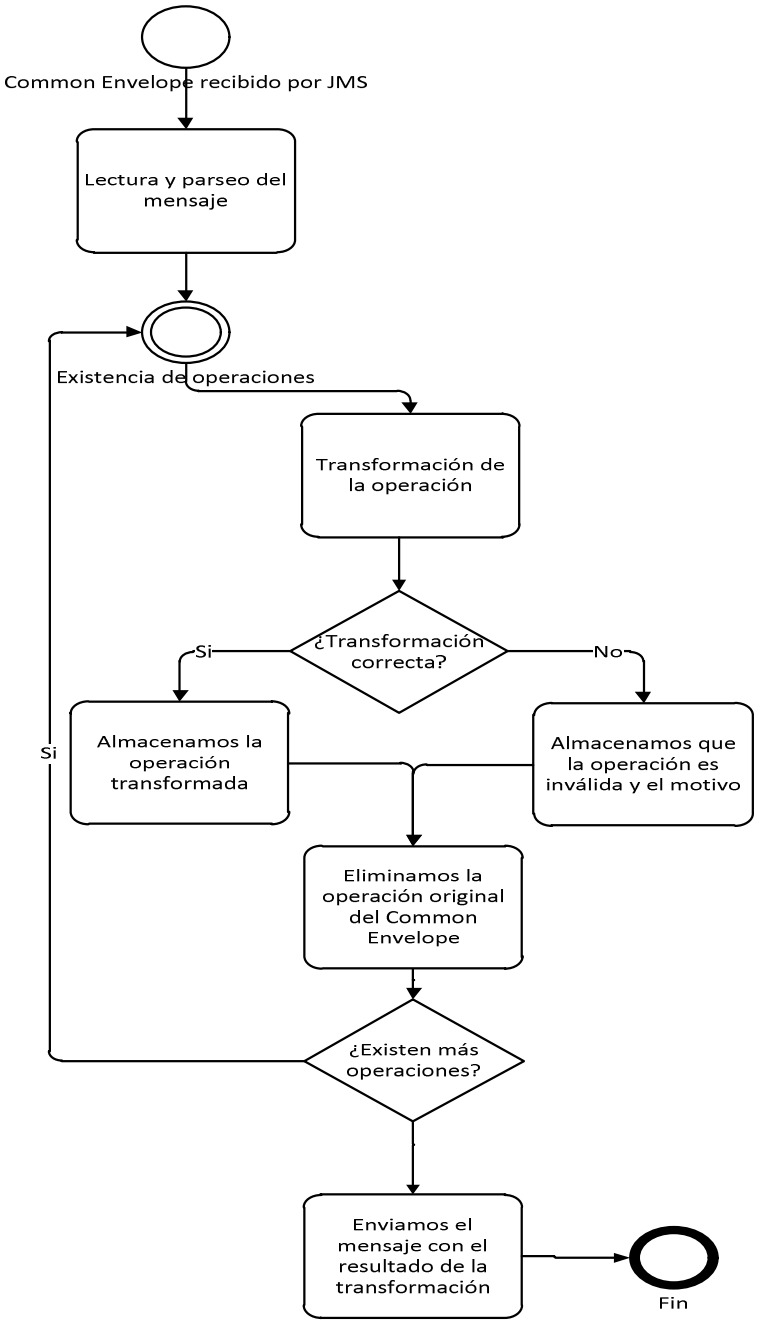



**Figura 96 - Funcionamiento detallado del Transformador**

**[fuente: propia]**

#### 5.3.4.4. Salida del sistema

El sistema de escritura será el encargado de generar el fichero final que se debe comunicar a la entidad o al sistema central, dependiendo del flujo de transformación que se haya ejecutado.

Al revés que en el caso de los módulos de entrada, en este caso, la activación del módulo de escritura se realizará con la recepción de un mensaje Common Envelope, generando un fichero con las operaciones transformadas en el correspondiente directorio de salida.

Debido a que se puede enviar un fichero en ambos sentidos, es decir, desde la entidad al sistema central y desde el sistema central a la entidad, se crearán dos escritores.

El primero de ellos se encargará de escribir el fichero con las operaciones transformadas que la entidad emite al sistema central y otro que realiza el proceso inverso, escribe los ficheros con las operaciones transformadas que el sistema central envía a la entidad.

Si observamos estos módulos como una caja negra, su funcionamiento es el siguiente:

1. Detecta la recepción de un Common Envelope en la cola JMS de entrada definida mediante el EndPoint **ActiveMQ** en la etiqueta xml **from** de la ruta.

2. Parsea el Common Envelope recibido.

3. Marcamos el bloque como finalizado y escribimos un fichero temporal con la transformación recibida.

4. Si todos los bloques en que se descompuso el fichero original se encuentran procesados, listamos todos los ficheros temporales generados y los fusionamos.

5. Escribimos el fichero con todas las operaciones transformadas en el directorio donde lo recogerá el destinatario.

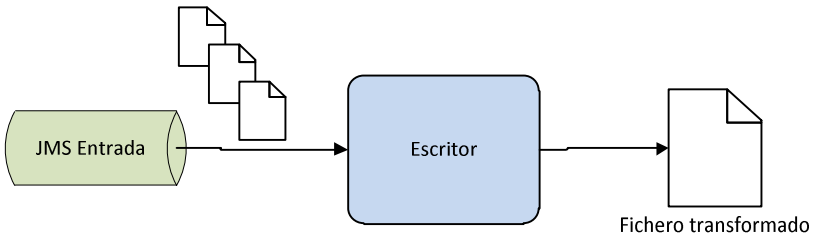



**Figura 97 - Funcionamiento básico del Escritor**

**[fuente: propia]**

A continuación vamos a mostrar el diagrama que describirá el funcionamiento de los módulos escritores.

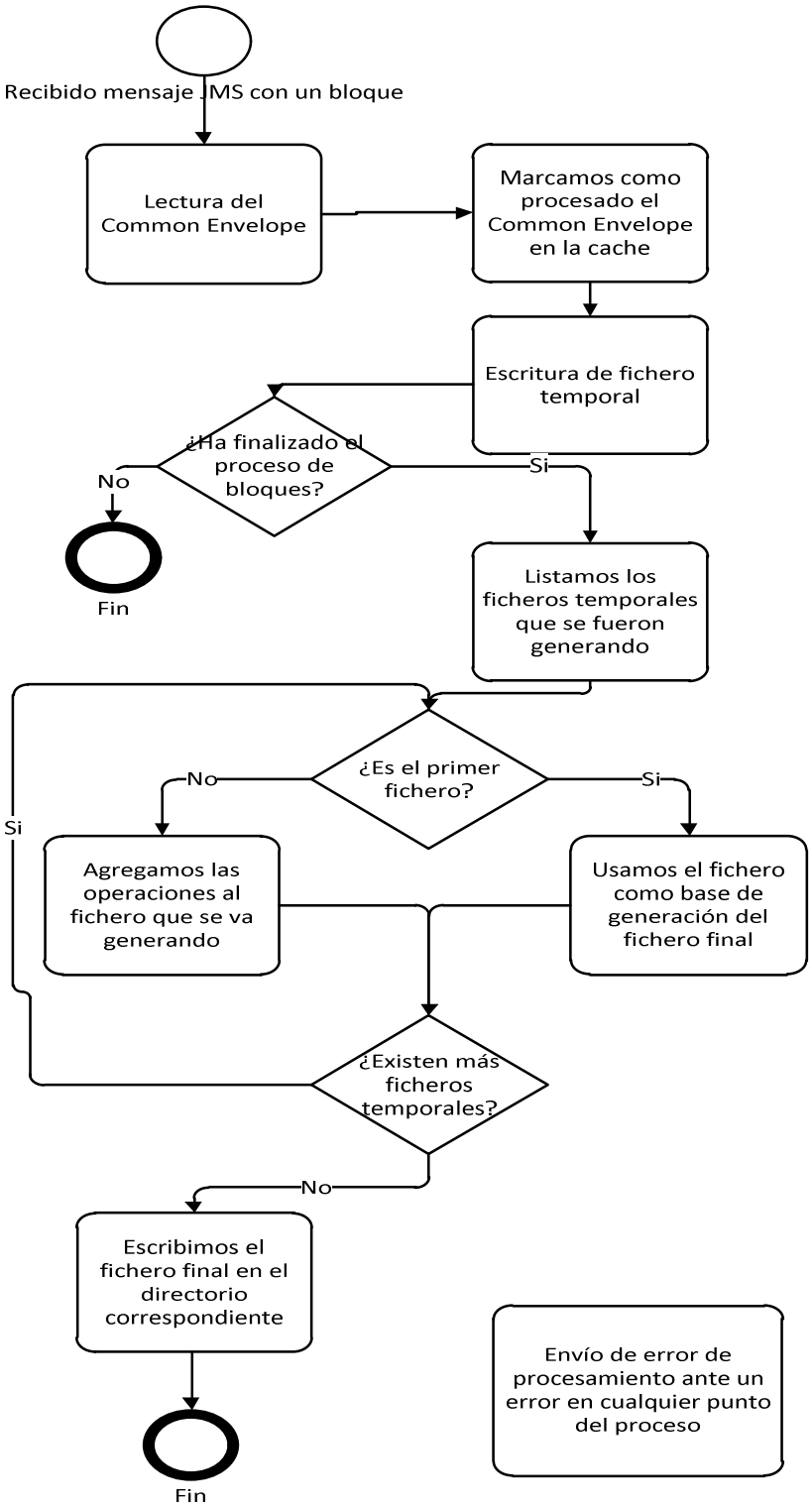



**Figura 98 - Funcionamiento del Escritor**

**[fuente: propia]**

#### 5.3.4.5. Comunicación de alertas

Ante la existencia de errores, el sistema necesita de alguna forma poder comunicar los errores de forma que puedan ser solucionados.

El módulo de comunicación de alertas será muy básico, únicamente recibirá un mensaje con la alerta a través de una cola JMS y lo enviará por eMail a los correspondientes destinatarios.

El funcionamiento por tanto será el siguiente:

1. Detecta la recepción de un mensaje con la alerta correspondiente en la cola JMS de entrada definida mediante el EndPoint **ActiveMQ** en la etiqueta xml **from** de la ruta.

2. Se rellenará el cuerpo del mail con los datos recibimos en el mensaje de alerta.

3. Se enviará el mail con la alerta. Para realizar esta acción utilizaremos el EndPoint mail, el cual configuraremos y usaremos mediante código.

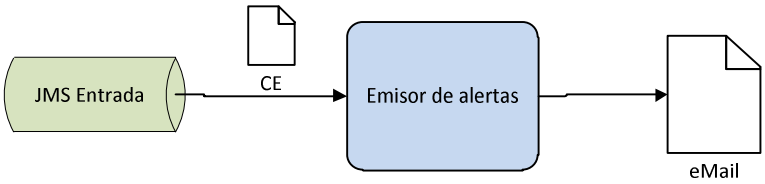



Figura 99 - Funcionamiento básico del Módulo de Alertas

[fuente: propia]

A continuación se detalla el proceso de envío de alertas.

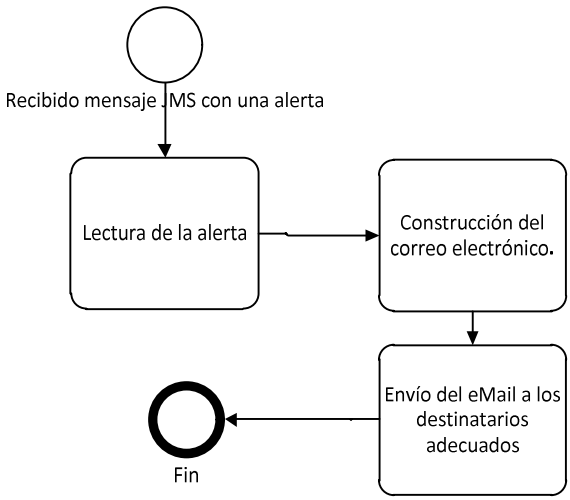



Figura 100 - Funcionamiento detallado del Módulo de Alertas

[fuente: propia]

### 5.3.5. Construcción del orquestador

En este apartado vamos a comentar como se implementó la pieza principal del sistema de transformación.

Aunque el resto de piezas comentadas definen la funcionalidad del sistema propiamente dicha, es decir, recepción, validación, transformación y emisión, la realización de la funcionalidad anteriormente descrita no sería posible sino incorporáramos un proceso orquestador.

Este tipo sistema, donde los diferentes módulos se comunicarán entre sí mediante un canal de comunicación (JMS) y que debe cumplir la premisa fundamental de que sea desacoplado, es decir, aunque los diferentes módulos deben colaborar entre ellos para realizar la transformación, no saben por si solos como organizarse entre ellos para colaborar ya que esto haría que se incrementara el acoplamiento al tener que incluir código de orquestación.

Para evitar acoplar los módulos incluyendo código de orquestación, construiremos una pieza de gobernanza, a la cual denominaremos orquestador. Este orquestador poseerá la siguiente funcionalidad:

1. Conoce desde donde se recibe un mensaje (Common Envelope) y hacía donde tiene que enviarlo para continuar el proceso.

2. Realiza el tratamiento de bloques u operaciones que han sido comunicadas como erróneas por otros módulos.

3. Gestiona la cache de bloques, comprobando donde ha sido enviado el mensaje con el bloque, si se ha recibido respuesta o no para un envío…

Como en casos anteriores, viendo el orquestador como una caja negra podemos describirlo como se muestra en la siguiente figura:

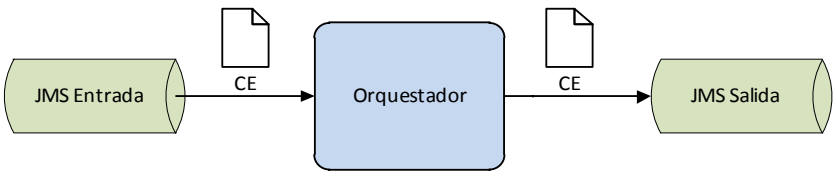



**Figura 101 - Orquestador**

**[fuente: propia]**

El orquestador recibirá mensajes procedentes del sistema de entrada y respuestas a invocaciones realizadas a los módulos de validación y transformación, con lo cual, como se ha indicado anteriormente, dispondrá de un mecanismo que le permite saber de dónde viene el mensaje recibido y hacía donde deberá ser encaminado.

Para realizar esto utilizará los **Steps** definidos en el Common Message.

El proceso se puede resumir de la siguiente forma:

1. El orquestador recibe un Common Envelope en la cola JMS de entrada definida mediante el EndPoint **ActiveMQ** en la etiqueta xml **from** de la ruta. Este mensaje puede proceder tanto del sistema de entrada (es un bloque procedente del particionado del fichero recibido) o como respuesta a un envío previo, por ejemplo, una solicitud de validación.

2. El orquestador comprueba si el mensaje estaba ya siendo procesado, es decir, es una respuesta a una solicitud realizada previamente o bien, es el inicio del proceso de ese bloque.

3. Si el bloque no estaba siendo procesado, el orquestador lo incluirá en la cache de control de bloques procesados y lo enviará al primer paso del proceso.

4. Si el bloque estaba siendo procesado, el mensaje recibido es una respuesta a una solicitud de proceso previa. En este caso, lo primero que debemos comprobar es si se encuentra expirado, es decir, la respuesta llega tarde o no.

5. En el caso de que la respuesta llegue tarde, anulamos el fichero y comunicamos el error al sistema de salida para que trate el caso de fichero inválido.

6. Si la respuesta recibida se encuentra en tiempo, comprobaremos si el mensaje contiene operaciones erróneas.

7. En caso afirmativo, anulamos el fichero y comunicamos el mensaje al proceso de salida para que trate el caso de fichero inválido.

8. En caso negativo, buscamos el siguiente paso del procesado al que enviar el mensaje, lo marcamos como actual, actualizamos la cache de procesados y lo enviamos para que continúe el flujo de transformación. Para realizar el envío utilizaremos una **Producer Template** ya que necesitamos configurar de forma dinámica la cola JMS a la que enviar el mensaje.

A continuación veremos un diagrama detallado del flujo de proceso que gobierna el orquestador.

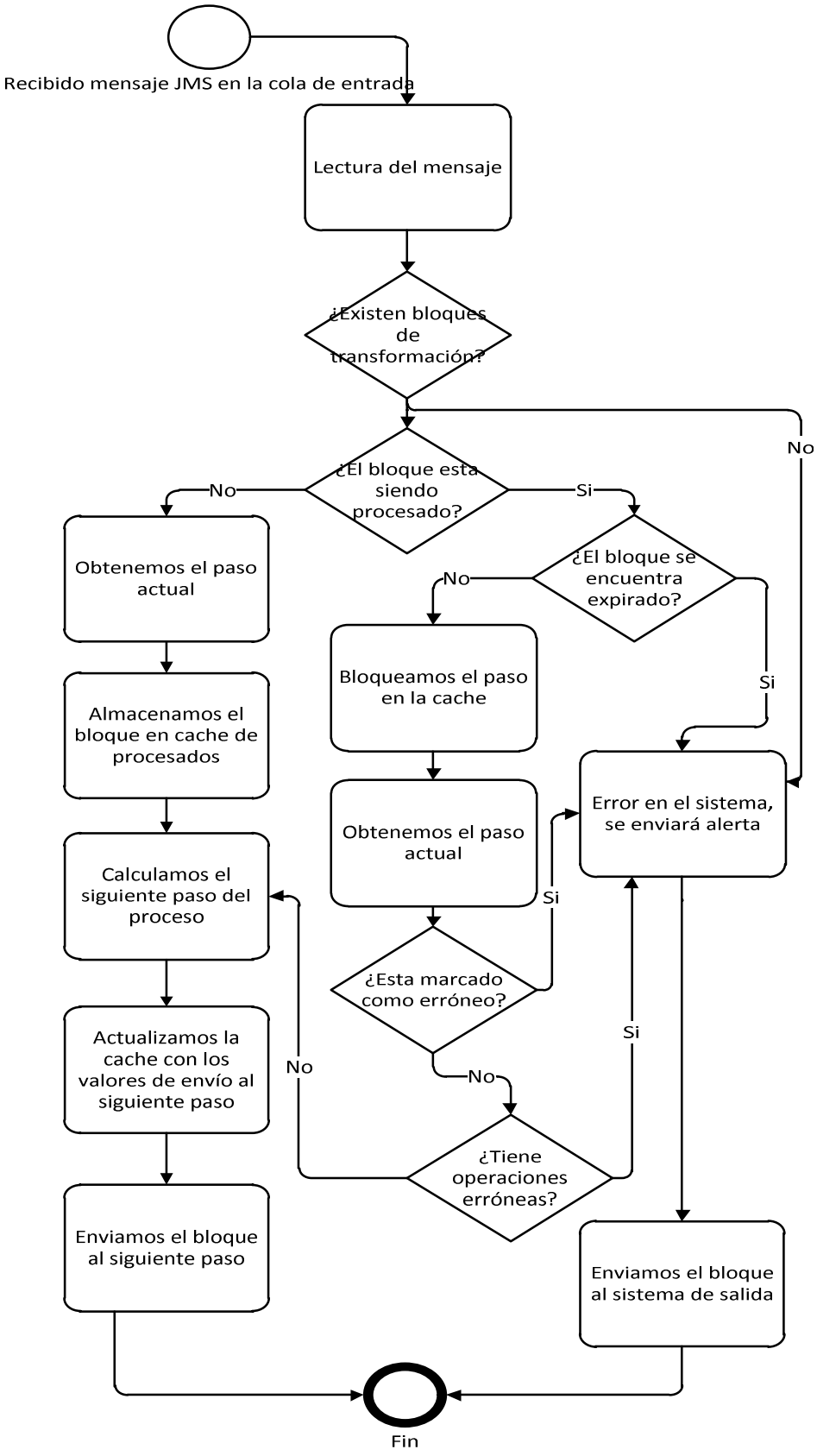



**Figura 102 - Funcionamiento detallado del Orquestador**

**[fuente: propia]**

# Capítulo 6: Comparación de los desarrollos

Una vez que se ha finalizado la implementación de ambos sistemas de transformación, el siguiente paso es compararlos, intentando utilizar medidas lo más objetivas posibles.

Antes de continuar se definirá el marco de ejecución para los dos sistemas en base a sus capacidades máximas, es decir, se usará para su ejecución las últimas tecnologías disponibles en cada uno de ellos.

## 6.1.Marco tecnológico

Se dispone de dos desarrollos diferentes así que existirán dos marcos de ejecución diferentes, uno para cada para uno de ellos.

Como plataforma de ejecución se utilizará Windows en su versión 8.1.

- **Marco tecnológico para la aplicación de transformación desarrollada sobre JCAPS63**

    - IDE de desarrollo Netbeans 6
    - Servidor de aplicaciones Sun GlassFish Enterprise Server 2.1.1.
    - Gestor de colas GlassFish Message Queue 4.4
    - Sistema de ficheros NTFS sobre máquina Windows.
    - Java versión 1.6

- **Marco tecnológico para la aplicación de transformación desarrollada sobre Software OpenSource**

    - IDE de desarrollo Eclipse Mars
    - Servidor de aplicaciones Sun GlassFish Enterprise Server 4.1.1.
    - Gestor de colas Apache ActiveMQ 5.14.0
    - Sistema de ficheros NTFS sobre máquina Windows.
    - Java versión 1.8

Una vez definido el marco tecnológico sobre el que se ejecutarán los desarrollos, se definirán, a grandes rasgos, los diferentes bloques de medición que servirán para compararlos.

## 6.2.Velocidad de procesado

Pasarán por el sistema 4 transformaciones (2 de BO a Sistema Central y 2 de Sistema Central a BO), tomando los tiempos que se tarda en realizarlas.

Las transformaciones a realizar serán exactamente iguales para ambos desarrollos, es decir, los ficheros que la desencadenan serán los mismos, obteniendo de esta forma un resultado no sesgado.

Además, debido a que el sistema debe poder tratar ficheros de diferentes tamaños los test a realizar serán monobulk (sin división de fichero fuente) y multibulk (con división de fichero fuente).

Ante lo comentado en los párrafos anteriores se definirán por tanto los siguientes Test:

- Test1: Transformación de BO a Sistema Central monobulk.


emision1.txt

- Test2: Transformación de Sistema Central a BO monobulk.


recepcion1.xml

- Test3: Transformación de BO a Sistema Central multibulk.


emision2.txt

- Test4: Transformación de Sistema Central a BO multibulk.


recepcion2.xml

Un factor a tener en cuenta antes de realizar los diferentes test es que el tiempo de procesamiento depende de numerosos factores, como por ejemplo, aplicaciones ejecutándose simultáneamente, carga del procesador, memoria ocupada…

Se intentará por tanto que las mediciones de tiempo se realicen bajo las mismas condiciones para ambos sistemas. Estas condiciones serán:

- Una consola de administración, un notepad++ y un administrador de tareas ejecutándose simultáneamente.
- Lanzaremos cada test 2 veces por aplicativo considerando como válidos los resultados de la segunda transformación.
- Los test únicamente serán lanzados cuando la carga del procesador sea baja (entre un 7% a un 12%).

Una vez transformado cada uno de los test se obtienen los siguientes resultados (expresados en milisegundos).

### 6.2.1. Test1. Transformación de BO a Sistema Central monobulk.

- *Tiempos para la transformación utilizando Software Propietario*
  - **Hora de inicio del procesado:** 01:49:17.627
  - **Hora de fin del procesado:** 01:49:17.955
  - **Tiempo:** 328 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector Plano** | 1 | 1471132157627 | 1471132157674 | 47 |
| **Orquestador** | 1 | 1471132157674 | 1471132157721 | 47 |
| **Val. Plano** | 1 | 1471132157721 | 1471132157752 | 31 |
| **Orquestador** | 1 | 1471132157752 | 1471132157783 | 31 |
| **Transformador** | 1 | 1471132157783 | 1471132157814 | 31 |
| **Orquestador** | 1 | 1471132157830 | 1471132157846 | 16 |
| **Val. XML** | 1 | 1471132157846 | 1471132157877 | 31 |
| **Orquestador** | 1 | 1471132157877 | 1471132157908 | 31 |
| **Escritor XML** | 1 | 1471132157924 | 1471132157955 | 31 |

**Tabla 23 - Tiempos Test1 con Software Propietario**

**[fuente: propia]**

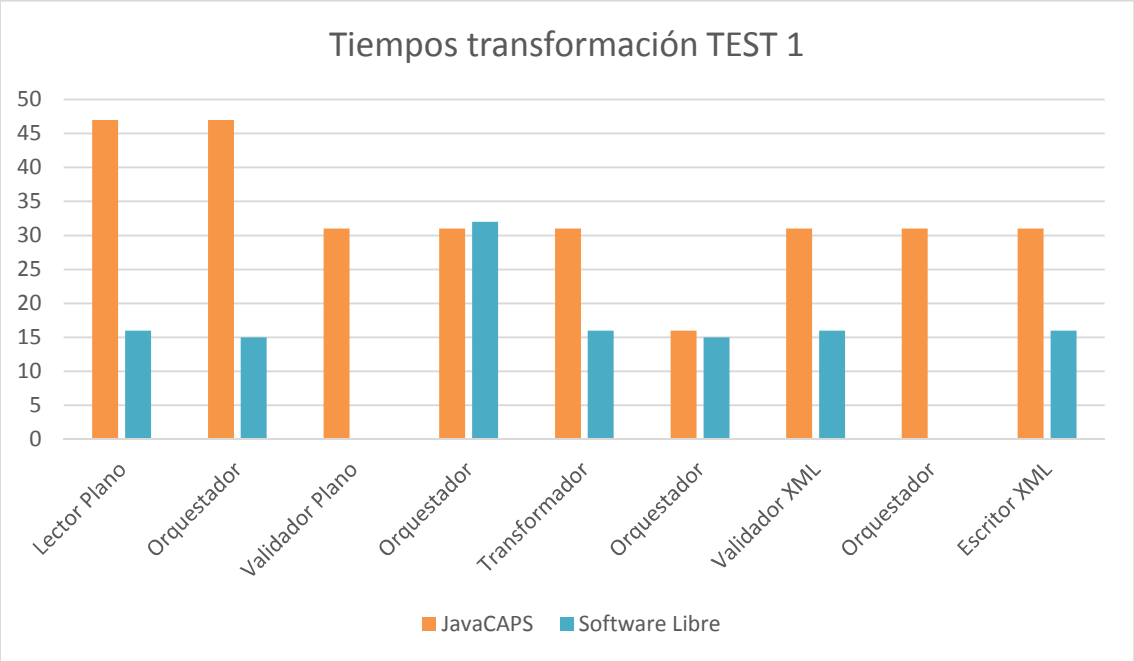
- *Tiempos para la transformación utilizando Software Libre*
  - **Hora de inicio del procesado:** 14:10:21.653
  - **Hora de fin del procesado:** 14:10:21.872
  - **Tiempo:** 219 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector Plano** | 1 | 1471176621653 | 1471176621669 | 16 |
| **Orquestador** | 1 | 1471176621669 | 1471176621684 | 15 |
| **Val. Plano** | 1 | 1471176621700 | 1471176621700 | 0 |

| | | | | |
|---|---|---|---|---|
| **Orquestador** | 1 | 1471176621715 | 1471176621747 | 32 |
| **Transformador** | 1 | 1471176621762 | 1471176621778 | 16 |
| **Orquestador** | 1 | 1471176621794 | 1471176621809 | 15 |
| **Val. XML** | 1 | 1471176621809 | 1471176621825 | 16 |
| **Orquestador** | 1 | 1471176621840 | 1471176621840 | 0 |
| **Escritor XML** | 1 | 1471176621856 | 1471176621872 | 16 |

**Tabla 24 - Tiempos Test1 con Software Libre**

**[fuente: propia]**




A la vista del gráfico de tiempos se puede llegar a pensar que la solución implementada utilizando Software Libre es mejor, en términos de rendimiento, que la solución implementada utilizando Software Propietario, ya que en el gráfico se puede ver que los tiempos de procesado por módulo por lo general son menores en el caso de Software Libre.

La realidad no es la expresada en el párrafo anterior, ya que debemos ver el conjunto y no analizar los módulos independientemente. Viendo los tiempos totales de procesamiento, (hora de finalización - hora de inicio), que incluyen también los tiempos de espera en la comunicación, cuanto tiempo pasa el mensaje en la cola desde que lo envía el emisor hasta que el receptor lo comienza a procesar, se puede concluir que el tiempo de procesado total es muy similar en las dos aproximaciones.

### 6.2.2. Test2. Transformación de Sistema Central a BO monobulk.

- *Tiempos para la transformación utilizando Software Propietario*
  - **Hora de inicio del procesado:** 02:31:30.578
  - **Hora de fin del procesado:** 02:31:30.797
  - **Tiempo:** 219 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector XML** | 1 | 1471134690578 | 1471134690610 | 32 |
| **Orquestador** | 1 | 1471134690610 | 1471134690625 | 15 |
| **Val. XML** | 1 | 1471134690625 | 1471134690656 | 31 |
| **Orquestador** | 1 | 1471134690656 | 1471134690672 | 16 |
| **Transformador** | 1 | 1471134690672 | 1471134690688 | 16 |
| **Orquestador** | 1 | 1471134690703 | 1471134690703 | 0 |
| **Val. Plano** | 1 | 1471134690719 | 1471134690735 | 16 |
| **Orquestador** | 1 | 1471134690735 | 1471134690750 | 15 |
| **Escritor Plano** | 1 | 1471134690766 | 1471134690797 | 31 |

**Tabla 25 - Tiempos Test2 con Software Propietario**

**[fuente: propia]**

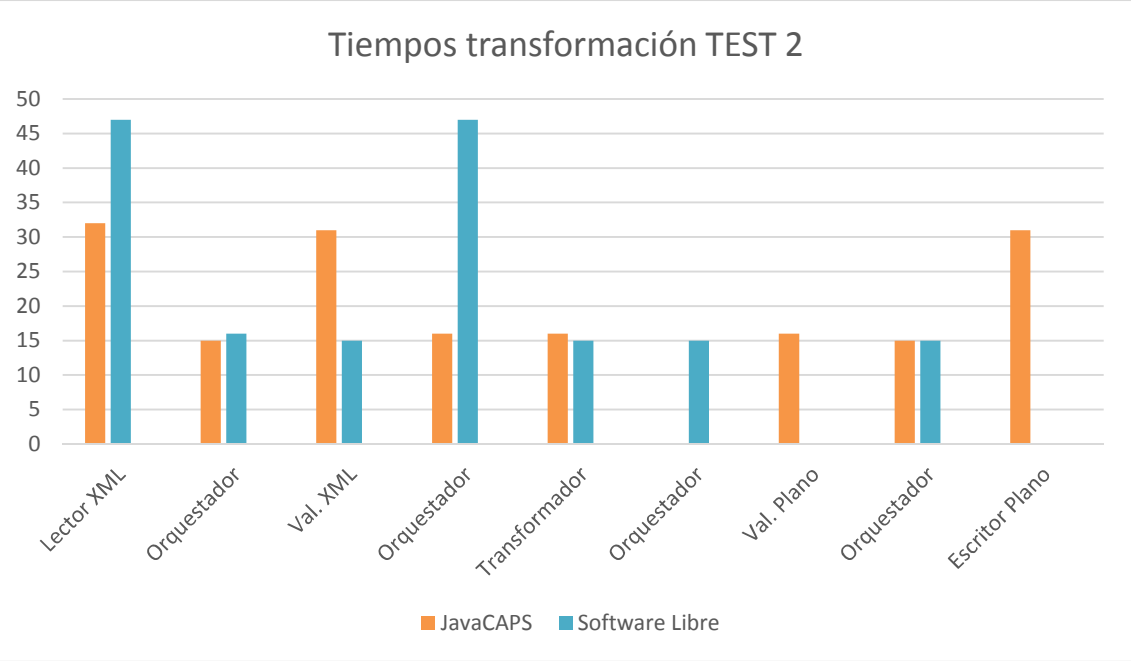
- *Tiempos para la transformación utilizando Software Libre*
  - **Hora de inicio del procesado:** 13:59:54.349
  - **Hora de fin del procesado:** 13:59:54.552
  - **Tiempo:** 203 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector XML** | 1 | 1471175994349 | 1471175994396 | 47 |
| **Orquestador** | 1 | 1471175994380 | 1471175994396 | 16 |
| **Val. XML** | 1 | 1471175994412 | 1471175994427 | 15 |
| **Orquestador** | 1 | 1471175994427 | 1471175994474 | 47 |
| **Transformador** | 1 | 1471175994443 | 1471175994458 | 15 |

| | | | | |
|---|---|---|---|---|
| **Orquestador** | 1 | 1471175994490 | 1471175994505 | 15 |
| **Val. Plano** | 1 | 1471175994505 | 1471175994505 | 0 |
| **Orquestador** | 1 | 1471175994537 | 1471175994552 | 15 |
| **Escritor Plano** | 1 | 1471175994552 | 1471175994552 | 0 |

Tabla 26 - Tiempos Test2 con Software Libre

[fuente: propia]




Observando en el gráfico los resultados del test realizado y evaluando las tablas de tiempo obtenidas, concluiremos que las dos aplicaciones se comportan, siempre en términos de tiempo de procesado, de forma similar, sin destacar una sobre otra.

Además, en el gráfico se puede observar que los tiempos tienden a compensarse, es decir, donde una aplicación mejora la otra empeora y viceversa, llegando a igualarse el tiempo empleado por cada aplicación en realizar la transformación.

Si además consideramos los tiempos totales (hora finalización - hora de inicio) se puede observar que son idénticos, lo cual reafirma el párrafo anterior.

### 6.2.3. Test3. Transformación de BO a Sistema Central multibulk.

En esta prueba, al ser un proceso multibulk, se tendrán diferentes hilos ejecutándose simultáneamente compitiendo por los recursos de la máquina.

Igual que en los dos test anteriores se extraerán los tiempos de procesado por módulo y bulk (paquete), pero en esta situación, los tiempos obtenidos no serán tan significativos como los de los dos test anteriores ya que al ser multibulk existirán esperas para poder usar algunos recursos.

Se tendrá en cuenta el tiempo total de procesado, es decir, cuánto tarda el sistema desde que se inicia el lector hasta que finaliza el último de los escritores, genrando el fichero final.

- *Tiempos para la transformación utilizando Software Propietario*

  - **Hora de inicio del procesado:** 15:58:03.237
  - **Hora de fin del procesado:** 15:58:03.674
  - **Tiempo:** 437 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector Plano** | 1,2,3 | 1471183083237 | 1471183083331 | 94 |
| | | | | |
| **Orquestador** | 1 | 1471183083284 | 1471183083331 | 47 |
| **Val. Plano** | 1 | 1471183083331 | 1471183083346 | 15 |
| **Orquestador** | 1 | 1471183083362 | 1471183083393 | 31 |
| **Transformador** | 1 | 1471183083393 | 1471183083424 | 31 |
| **Orquestador** | 1 | 1471183083440 | 1471183083518 | 78 |
| **Val. XML** | 1 | 1471183083503 | 1471183083534 | 31 |
| **Orquestador** | 1 | 1471183083549 | 1471183083581 | 32 |
| **Escritor XML** | 1 | 1471183083596 | 1471183083628 | 32 |
| | | | | |
| **Orquestador** | 2 | 1471183083284 | 1471183083346 | 62 |
| **Val. Plano** | 2 | 1471183083346 | 1471183083362 | 16 |
| **Orquestador** | 2 | 1471183083393 | 1471183083409 | 16 |
| **Transformador** | 2 | 1471183083424 | 1471183083456 | 32 |
| **Orquestador** | 2 | 1471183083456 | 1471183083518 | 62 |
| **Val. XML** | 2 | 1471183083534 | 1471183083549 | 15 |
| **Orquestador** | 2 | 1471183083565 | 1471183083612 | 47 |
| **Escritor XML** | 2 | 1471183083628 | 1471183083643 | 15 |
| | | | | |
| **Orquestador** | 3 | 1471183083331 | 1471183083346 | 15 |

| | | | | |
|---|---|---|---|---|
| **Val. Plano** | 3 | 1471183083378 | 1471183083409 | 31 |
| **Orquestador** | 3 | 1471183083409 | 1471183083440 | 31 |
| **Transformador** | 3 | 1471183083456 | 1471183083503 | 47 |
| **Orquestador** | 3 | 1471183083518 | 1471183083549 | 31 |
| **Val. XML** | 3 | 1471183083565 | 1471183083581 | 16 |
| **Orquestador** | 3 | 1471183083581 | 1471183083612 | 31 |
| **Escritor XML** | 3 | 1471183083659 | 1471183083674 | 15 |

**Tabla 27 - Tiempos Test3 con Software Propietario**

**[fuente: propia]**

- *Tiempos para la transformación utilizando Software Libre*
  - **Hora de inicio del procesado:** 17:59:14.721
  - **Hora de fin del procesado:** 17:59:15.297
  - **Tiempo:** 576 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector Plano** | 1,2,3 | 1471190354721 | 1471190354905 | 184 |
| | | | | |
| **Orquestador** | 1 | 1471190354747 | 1471190354787 | 40 |
| **Val. Plano** | 1 | 1471190354780 | 1471190354782 | 2 |
| **Orquestador** | 1 | 1471190354802 | 1471190354875 | 73 |
| **Transformador** | 1 | 1471190354879 | 1471190354909 | 30 |
| **Orquestador** | 1 | 1471190354945 | 1471190355003 | 58 |
| **Val. XML** | 1 | 1471190355009 | 1471190355020 | 11 |
| **Orquestador** | 1 | 1471190355044 | 1471190355067 | 23 |
| **Escritor XML** | 1 | 1471190355076 | 1471190355110 | 34 |
| | | | | |
| **Orquestador** | 2 | 1471190354781 | 1471190354832 | 51 |
| **Val. Plano** | 2 | 1471190354841 | 1471190354847 | 6 |
| **Orquestador** | 2 | 1471190354874 | 1471190354904 | 30 |
| **Transformador** | 2 | 1471190354930 | 1471190354951 | 21 |
| **Orquestador** | 2 | 1471190355045 | 1471190355076 | 31 |

| | | | | |
|---|---|---|---|---|
| **Val. XML** | 2 | 1471190355080 | 1471190355087 | 7 |
| **Orquestador** | 2 | 1471190355118 | 1471190355242 | 124 |
| **Escritor XML** | 2 | 1471190355213 | 1471190355224 | 11 |
| | | | | |
| **Orquestador** | 3 | 1471190354891 | 1471190354926 | 35 |
| **Val. Plano** | 3 | 1471190354939 | 1471190354941 | 2 |
| **Orquestador** | 3 | 1471190354959 | 1471190355060 | 101 |
| **Transformador** | 3 | 1471190355047 | 1471190355062 | 15 |
| **Orquestador** | 3 | 1471190355076 | 1471190355112 | 36 |
| **Val. XML** | 3 | 1471190355128 | 1471190355204 | 76 |
| **Orquestador** | 3 | 1471190355230 | 1471190355288 | 58 |
| **Escritor XML** | 3 | 1471190355251 | 1471190355297 | 46 |

**Tabla 28 - Tiempos Test3 con Software Libre**

**[fuente: propia]**

Se puede observar, vistos los tiempos de procesamiento total calculados que la diferencia entre las dos aplicaciones es prácticamente insignificante, 139 milésimas a favor de la construcción realizada utilizando Software Propietario.

Evaluando la relación de tiempos por módulo obtenidos en el test y de forma anecdótica, se observa que los resultados utilizando Software Propietario, son mucho más uniformes que los resultados obtenidos por el sistema implementado utilizando Software Libre.

### 6.2.4. Test4. Transformación de Sistema Central a BO multibulk.

En esta prueba, al ser un proceso multibulk, se tendrán diferentes hilos ejecutándose simultáneamente compitiendo por los recursos de la máquina.

Igual que en los tres test anteriores se extraerán los tiempos de procesado por módulo y bulk (paquete), pero en esta situación, los tiempos obtenidos no serán tan significativos como los de los dos test monobulk, ya que al ser multibulk, existirán esperas para poder usar algunos recursos.

Se tendrá en cuenta el tiempo total de procesado, es decir, cuánto tarda el sistema desde que se inicia el lector hasta que finaliza el último de los escritores, generando el fichero final.

- *Tiempos para la transformación utilizando Software Propietario*

  - **Hora de inicio del procesado:** 16:01:23.096
  - **Hora de fin del procesado:** 16:01:23.549
  - **Tiempo:** 453 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector XML** | 1,2,3 | 1471183283096 | 1471183283237 | 141 |
| | | | | |
| **Orquestador** | 1 | 1471183283174 | 1471183283221 | 47 |
| **Val. XML** | 1 | 1471183283237 | 1471183283252 | 15 |
| **Orquestador** | 1 | 1471183283268 | 1471183283284 | 16 |
| **Transformador** | 1 | 1471183283299 | 1471183283331 | 32 |
| **Orquestador** | 1 | 1471183283331 | 1471183283362 | 31 |
| **Val. Plano** | 1 | 1471183283393 | 1471183283409 | 16 |
| **Orquestador** | 1 | 1471183283424 | 1471183283456 | 32 |
| **Escritor Plano** | 1 | 1471183283471 | 1471183283487 | 16 |
| | | | | |
| **Orquestador** | 2 | 1471183283206 | 1471183283237 | 31 |
| **Val. XML** | 2 | 1471183283252 | 1471183283268 | 16 |
| **Orquestador** | 2 | 1471183283284 | 1471183283299 | 15 |
| **Transformador** | 2 | 1471183283315 | 1471183283346 | 31 |
| **Orquestador** | 2 | 1471183283377 | 1471183283409 | 32 |
| **Val. Plano** | 2 | 1471183283409 | 1471183283424 | 15 |
| **Orquestador** | 2 | 1471183283440 | 1471183283487 | 47 |
| **Escritor Plano** | 2 | 1471183283502 | 1471183283518 | 16 |
| | | | | |
| **Orquestador** | 3 | 1471183283237 | 1471183283268 | 31 |
| **Val. XM** | 3 | 1471183283268 | 1471183283299 | 31 |
| **Orquestador** | 3 | 1471183283299 | 1471183283346 | 47 |
| **Transformador** | 3 | 1471183283377 | 1471183283424 | 47 |
| **Orquestador** | 3 | 1471183283424 | 1471183283456 | 32 |
| **Val. Plano** | 3 | 1471183283456 | 1471183283487 | 31 |
| **Orquestador** | 3 | 1471183283487 | 1471183283518 | 31 |

| | | | | |
|---|---|---|---|---|
| **Escritor Plano** | 3 | 1471183283534 | 1471183283549 | 15 |

**Tabla 29 - Tiempos Test4 con Software Propietario**

**[fuente: propia]**

- *Tiempos para la transformación utilizando Software Libre*

  - **Hora de inicio del procesado:** 19:39:55.988
  - **Hora de fin del procesado:** 19:39:56.441
  - **Tiempo:** 453 milésimas

| Módulo | Paquete | Inicio Proceso | Fin Proceso | Tiempo ejecución |
|---|---|---|---|---|
| **Lector XML** | 1,2,3 | 1471196395988 | 1471196396207 | 219 |
| | | | | |
| **Orquestador** | 1 | 1471196396035 | 1471196396035 | 0 |
| **Val. XML** | 1 | 1471196396066 | 1471196396082 | 16 |
| **Orquestador** | 1 | 1471196396082 | 1471196396097 | 15 |
| **Transformador** | 1 | 1471196396113 | 1471196396113 | 0 |
| **Orquestador** | 1 | 1471196396144 | 1471196396175 | 31 |
| **Val. Plano** | 1 | 1471196396175 | 1471196396175 | 0 |
| **Orquestador** | 1 | 1471196396222 | 1471196396363 | 141 |
| **Escritor Plano** | 1 | 1471196396347 | 1471196396347 | 0 |
| | | | | |
| **Orquestador** | 2 | 1471196396128 | 1471196396144 | 16 |
| **Val. XML** | 2 | 1471196396144 | 1471196396160 | 16 |
| **Orquestador** | 2 | 1471196396160 | 1471196396285 | 125 |
| **Transformador** | 2 | 1471196396253 | 1471196396269 | 16 |
| **Orquestador** | 2 | 1471196396316 | 1471196396394 | 78 |
| **Val. Plano** | 2 | 1471196396394 | 1471196396394 | 0 |
| **Orquestador** | 2 | 1471196396425 | 1471196396441 | 16 |
| **Escritor Plano** | 2 | 1471196396441 | 1471196396441 | 0 |
| | | | | |
| **Orquestador** | 3 | 1471196396191 | 1471196396238 | 47 |
| **Val. XML** | 3 | 1471196396222 | 1471196396238 | 16 |

| | | | | |
|---|---|---|---|---|
| **Orquestador** | 3 | 1471196396253 | 1471196396316 | 63 |
| **Transformador** | 3 | 1471196396332 | 1471196396347 | 15 |
| **Orquestador** | 3 | 1471196396363 | 1471196396378 | 15 |
| **Val. Plano** | 3 | 1471196396378 | 1471196396378 | 0 |
| **Orquestador** | 3 | 1471196396410 | 1471196396425 | 15 |
| **Escritor Plano** | 3 | 1471196396441 | 1471196396441 | 0 |

**Tabla 30 - Tiempos Test4 con Software Libre**

**[fuente: propia]**

Se puede observar, vistos los tiempos de procesamiento total que no existe diferencia entre las dos aplicaciones, ambos sistemas han tardado exactamente lo mismo.

Evaluando la relación de tiempos por módulo obtenidos en el test y de forma anecdótica, se observa que los resultados utilizando Software Propietario, son mucho más uniformes que los resultados obtenidos por el sistema implementado utilizando Software Libre.

## 6.3.Consumo de recursos

Uno de los puntos más importantes a la hora de evaluar una aplicación es observar como consume recursos, cuanta memoria ocupa, cuál es su máximo de uso de memoria, cómo de eficiente es la liberación, cuanta CPU es capaz de utilizar...

Para tomar estas medidas se utilizará la herramienta **JConsole**, la cual permite visualizar el uso de CPU, la cantidad de memoria ocupada, el número de clases cargadas en el sistema o el número de hilos vivos existentes en el servidor.

La toma de medidas se realizará por tanto siguiendo el siguiente patrón:

- Captura de los datos una vez que el servidor ha sido arrancado. Estos datos serán tomados como base para el resto de mediciones.
- Captura de datos pasado el test1.
- Captura de datos tras pasar el test2.
- Captura de datos pasado el test3.
- Captura de datos tras pasar el test4.

### 6.3.1. Software Propietario.

- *Captura de datos una vez arrancado el servidor.*

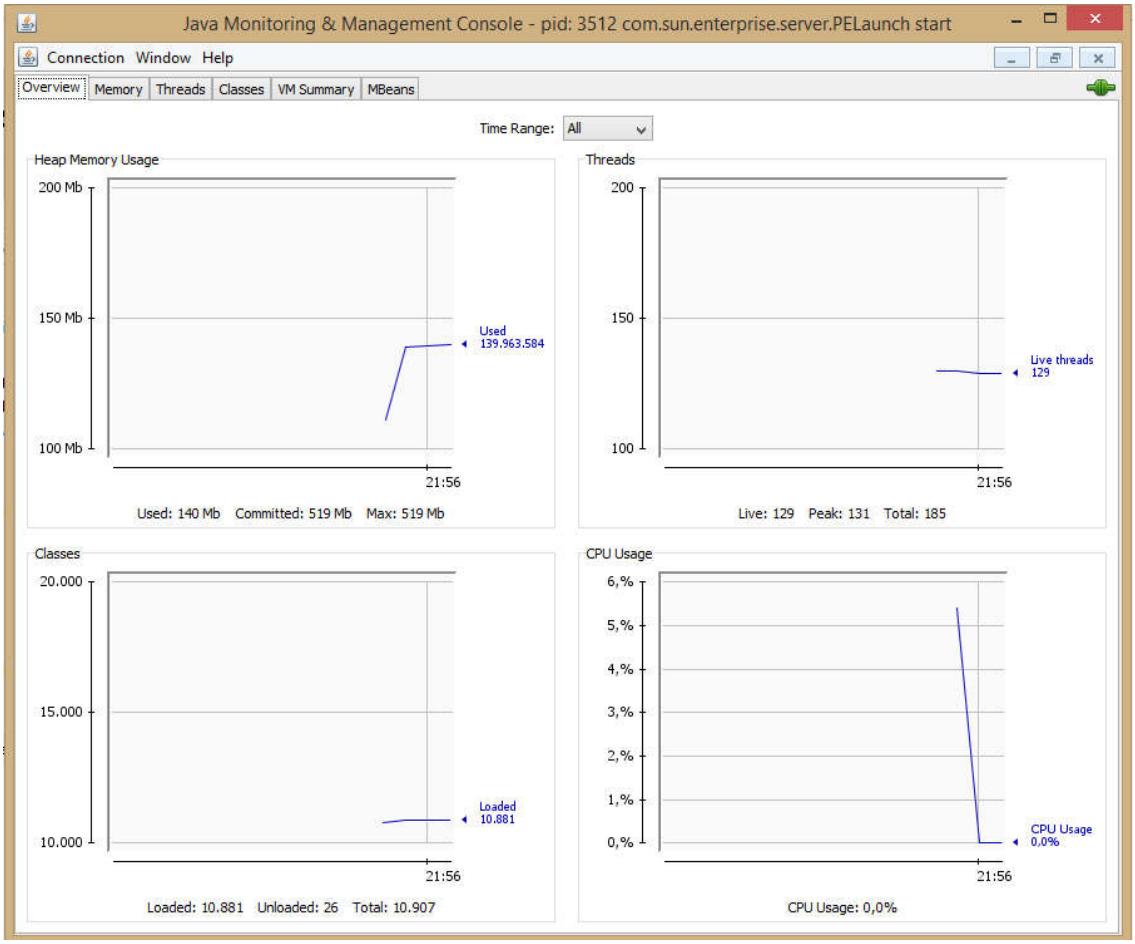



**Figura 103 - Datos iniciales de arranque del servidor Glassfish**

**[fuente: propia]**

Una vez arrancado el servidor podemos observar que el consumo de recursos no es excesivo, tenemos los siguientes parámetros:

- **Memoria en uso:** 140 Mb
- **Uso de CPU:** 0 %
- **Clases cargadas:** 10881
- **Número de hilos vivos en el servidor:** 129

- *Captura de datos después de test1.*

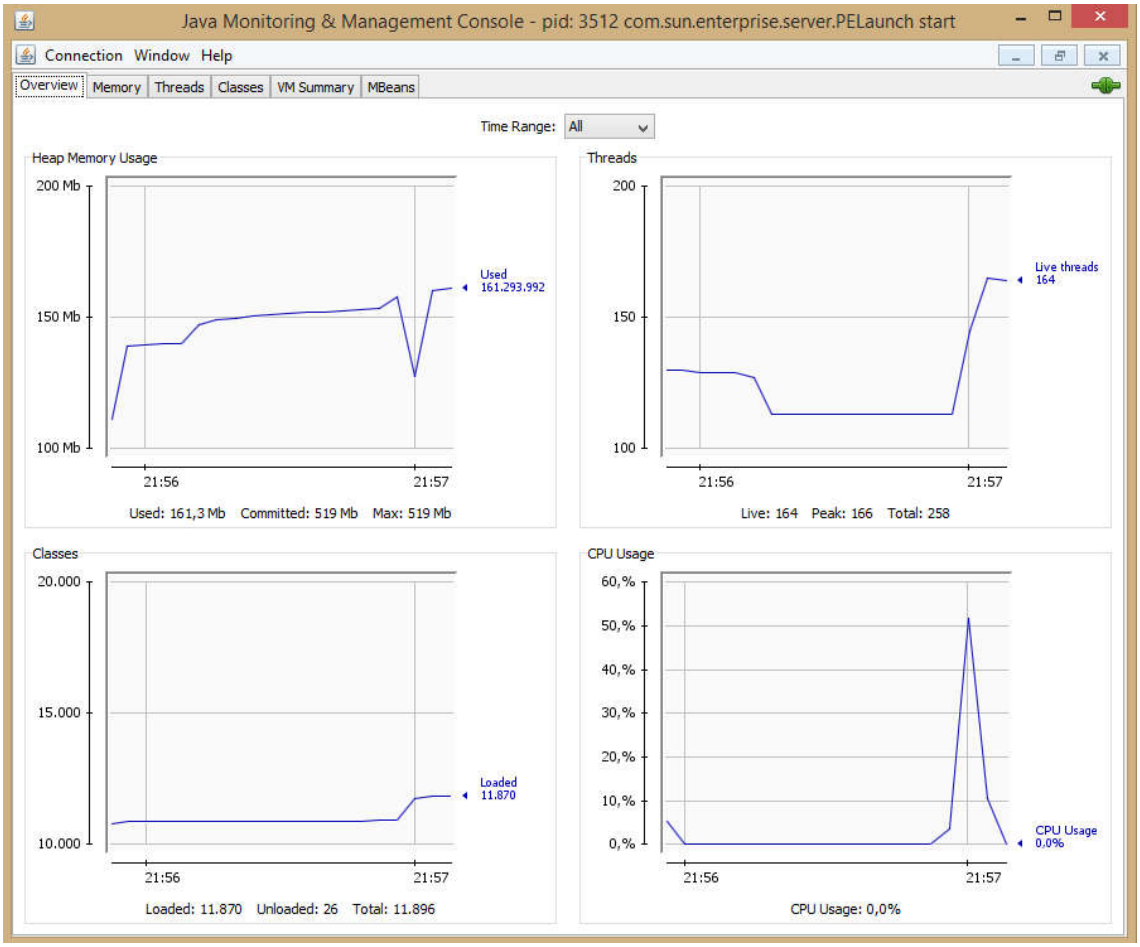



**Figura 104 - Datos servidor Glassfish tras pasar Test1**

**[fuente: propia]**

Tras pasar el primero de los test, transformación de BO a Sistema Central con el sistema sin cargar (no se han creado EJBs, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 21:57:00, obteniendo los siguientes resultados.

- **Memoria en uso:** 159 Mb, posteriormente se ejecuta un GC.
- **Uso de CPU:** Aproximadamente 50 %. Es alto debido a que el sistema realiza su primera carga.
- **Clases cargadas:** 11870
- **Número de hilos vivos en el servidor:** Se incrementa hasta los 167 y comienza a bajar.

- *Captura de datos después de test2.*

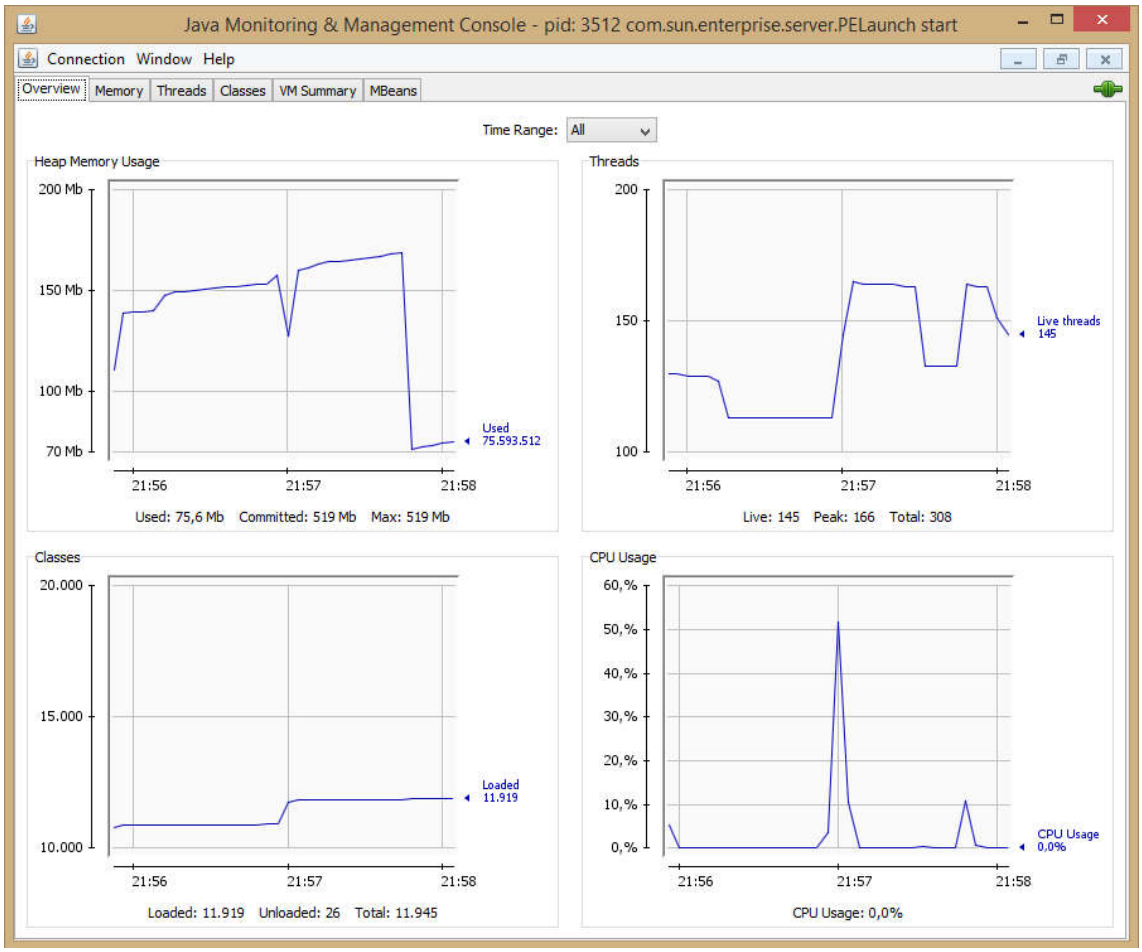



**Figura 105 - Datos servidor Glassfish tras pasar Test2**

**[fuente: propia]**

Tras pasar el segundo de los test, transformación de Sistema Central a BO con el sistema ya cargado (EJB's creados, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 21:56:50, obteniendo los siguientes resultados.

- **Memoria en uso:** Aproximadamente 170 Mb, posteriormente se ejecuta un GC, liberando casi 100 Mb de memoria en uso.
- **Uso de CPU:** Aproximadamente 10 %.
- **Clases cargadas:** 11919
- **Número de hilos vivos en el servidor:** Aproximadamente 164 y comienza a bajar.

- *Captura de datos después de test3.*

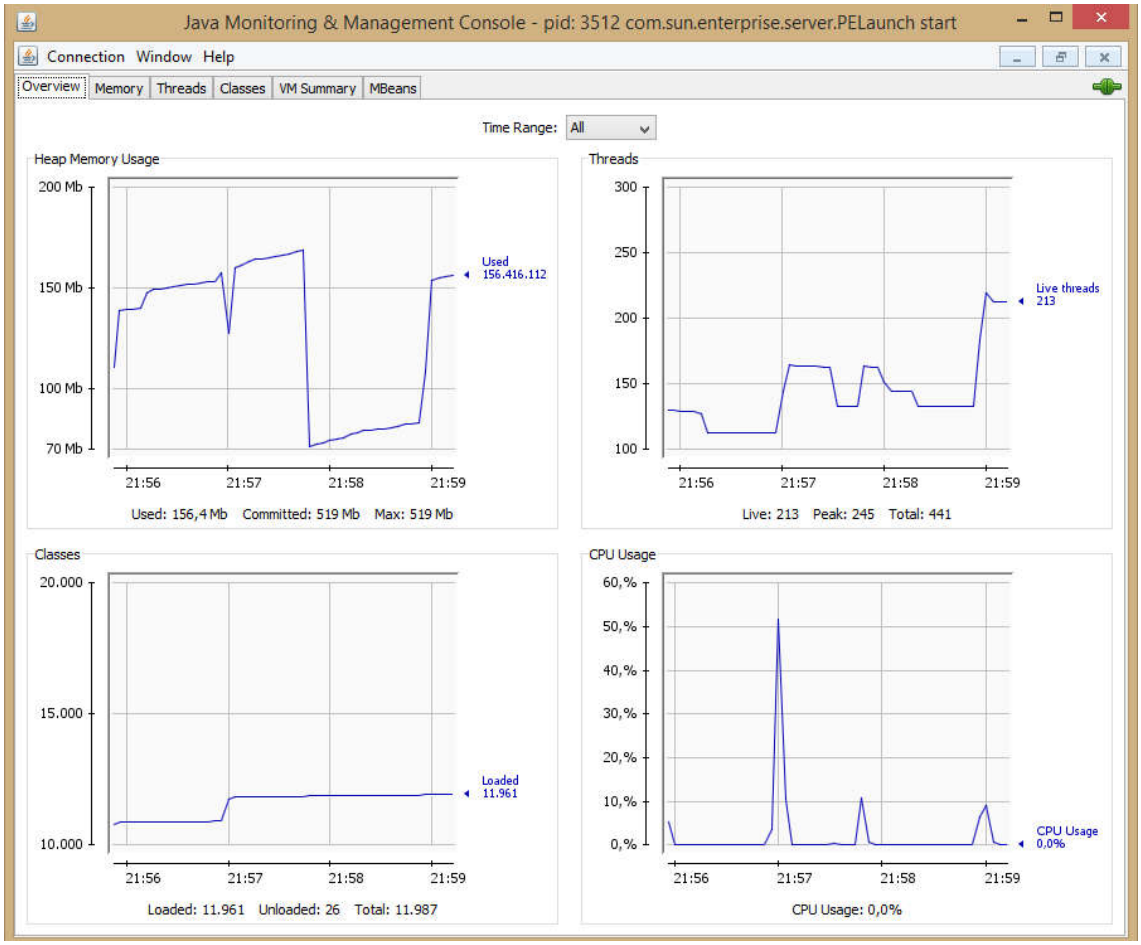



**Figura 106 - Datos servidor Glassfish tras pasar Test3**

**[fuente: propia]**

Tras pasar el tercero de los test, transformación de BO a Sistema Central con el sistema ya cargado (EJB's creados, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 21:59:00, obteniendo los siguientes resultados.

- **Memoria en uso:** 156 Mb.
- **Uso de CPU:** Aproximadamente 10 %.
- **Clases cargadas:** 11961
- **Número de hilos vivos en el servidor:** Se incrementa hasta los 215 y comienza a bajar.

- *Captura de datos después de test4.*

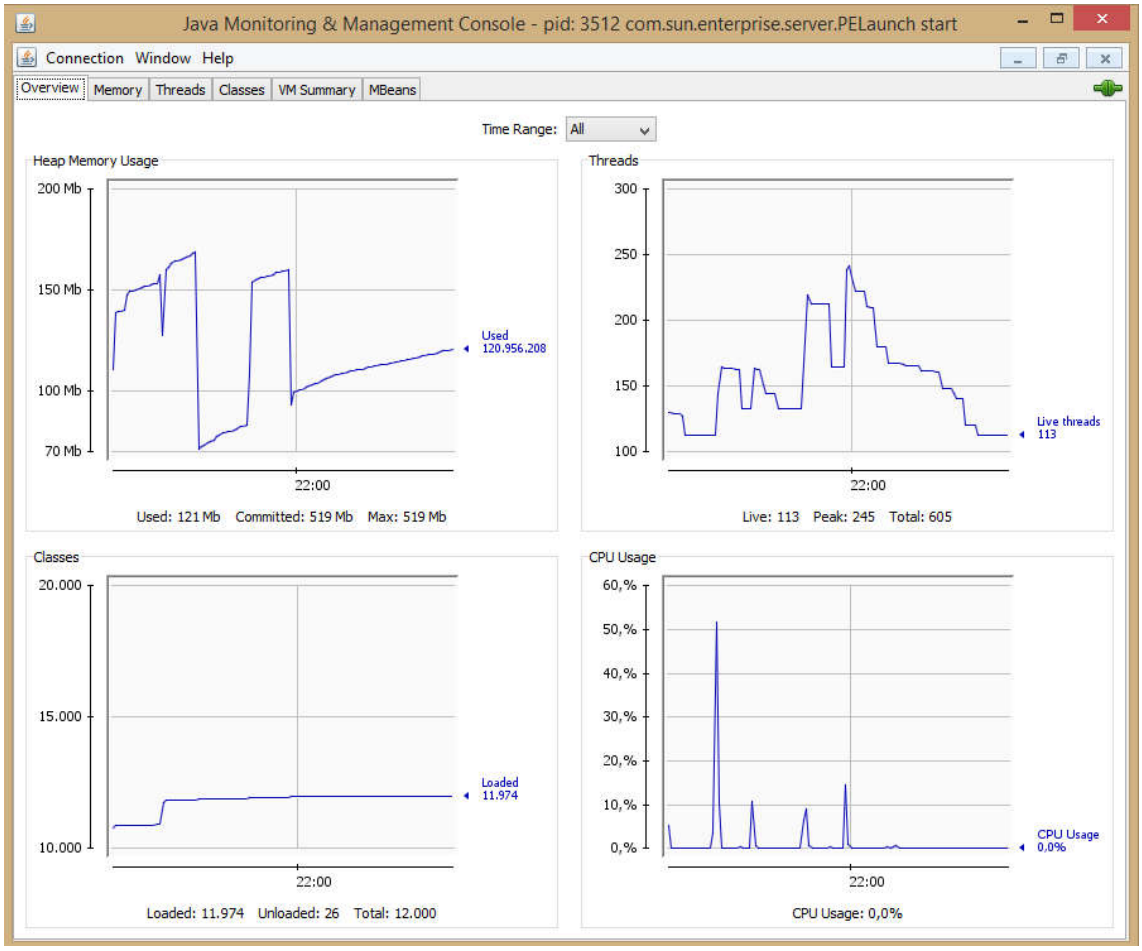



**Figura 107 - Datos servidor Glassfish tras pasar Test4**

**[fuente: propia]**

Tras pasar el cuarto de los test, transformación de Sistema Central a BO con el sistema ya cargado (EJB's creados, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 21:59:55, obteniendo los siguientes resultados.

- **Memoria en uso:** Se incrementa a aproximadamente 100 Mb.
- **Uso de CPU:** Aproximadamente 15 %.
- **Clases cargadas:** 11974
- **Número de hilos vivos en el servidor:** Se incrementa hasta los 240 y comienza a bajar.

### 6.3.2. Software Libre.

- *Captura de datos una vez arrancado el servidor.*

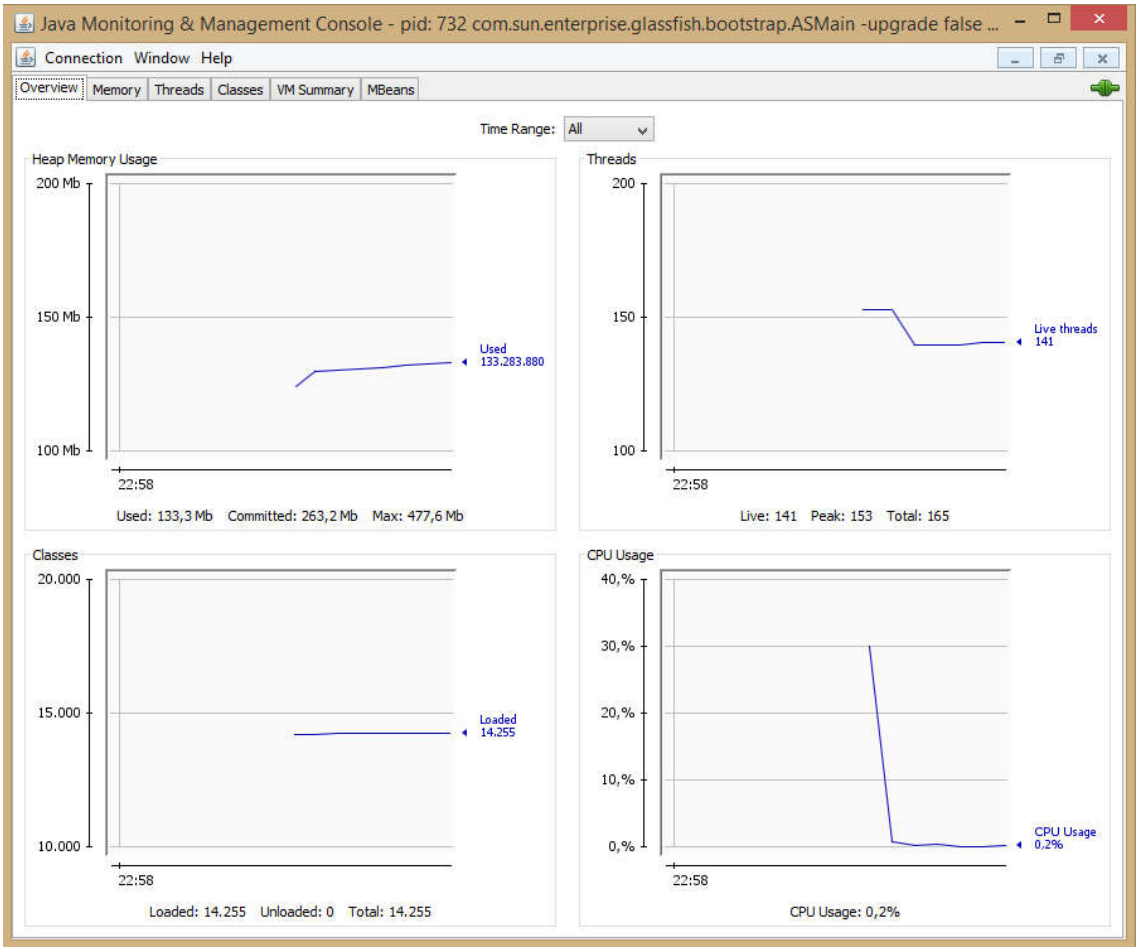



**Figura 108 - Datos iniciales de arranque del servidor Glassfish4**

**[fuente: propia]**

Una vez arrancado el servidor podemos observar que el consumo de recursos no es excesivo, tenemos los siguientes parámetros:

- **Memoria en uso:** 133,3 Mb
- **Uso de CPU:** 0,2 %
- **Clases cargadas:** 14255
- **Número de hilos vivos en el servidor:** 141

- *Captura de datos después de test1.*

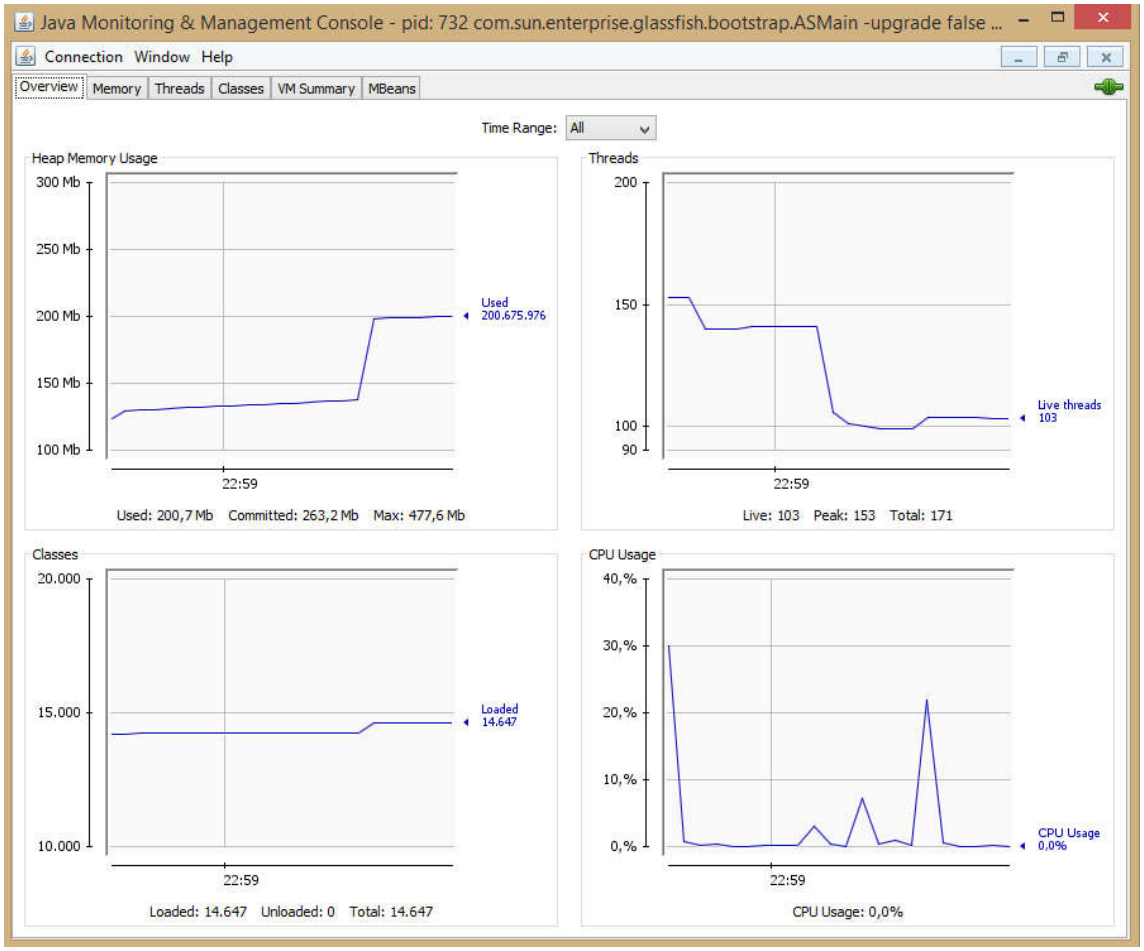



**Figura 109 - Datos servidor Glassfish4 tras pasar Test1**

**[fuente: propia]**

Tras pasar el primero de los test, transformación de BO a Sistema Central con el sistema sin cargar (no se han creado EJBs, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 21:59:40, obteniendo los siguientes resultados.

- **Memoria en uso:** Realiza una subida bastante brusca, hasta los 200 Mb.
- **Uso de CPU:** Aproximadamente 22 %. El sistema realiza su primera carga.
- **Clases cargadas:** 14647
- **Número de hilos vivos en el servidor:** 103

- *Captura de datos después de test2.*

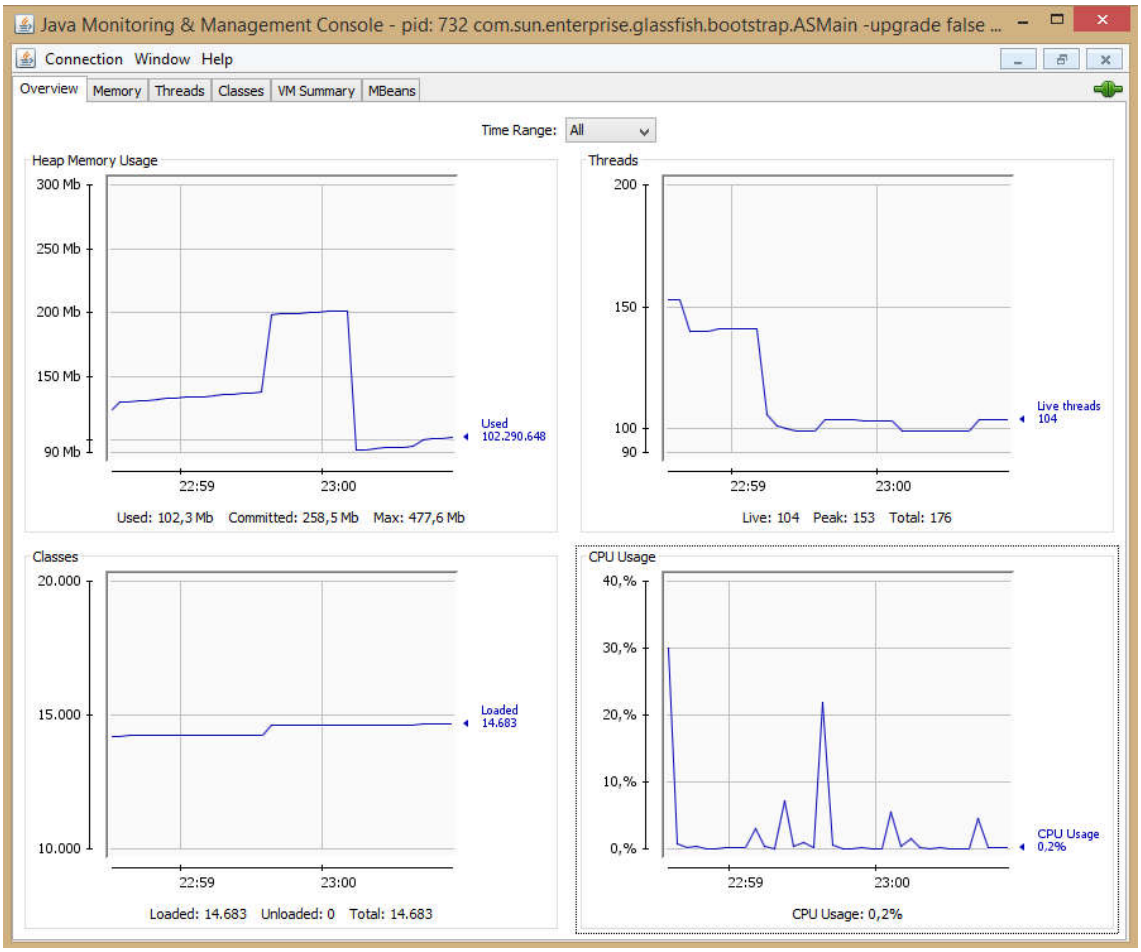



**Figura 110 - Datos servidor Glassfish4 tras pasar Test2**

**[fuente: propia]**

Tras pasar el segundo de los test, transformación de Sistema Central a BO con el sistema ya cargado (EJB's creados, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 23:00:40, obteniendo los siguientes resultados.

- **Memoria en uso:** Aproximadamente 102 Mb, posteriormente se ejecuta un GC, liberando casi 100 Mb de memoria en uso.
- **Uso de CPU:** Aproximadamente 5,3 %.
- **Clases cargadas:** 14683.
- **Número de hilos vivos en el servidor:** 104.

- *Captura de datos después de test3.*

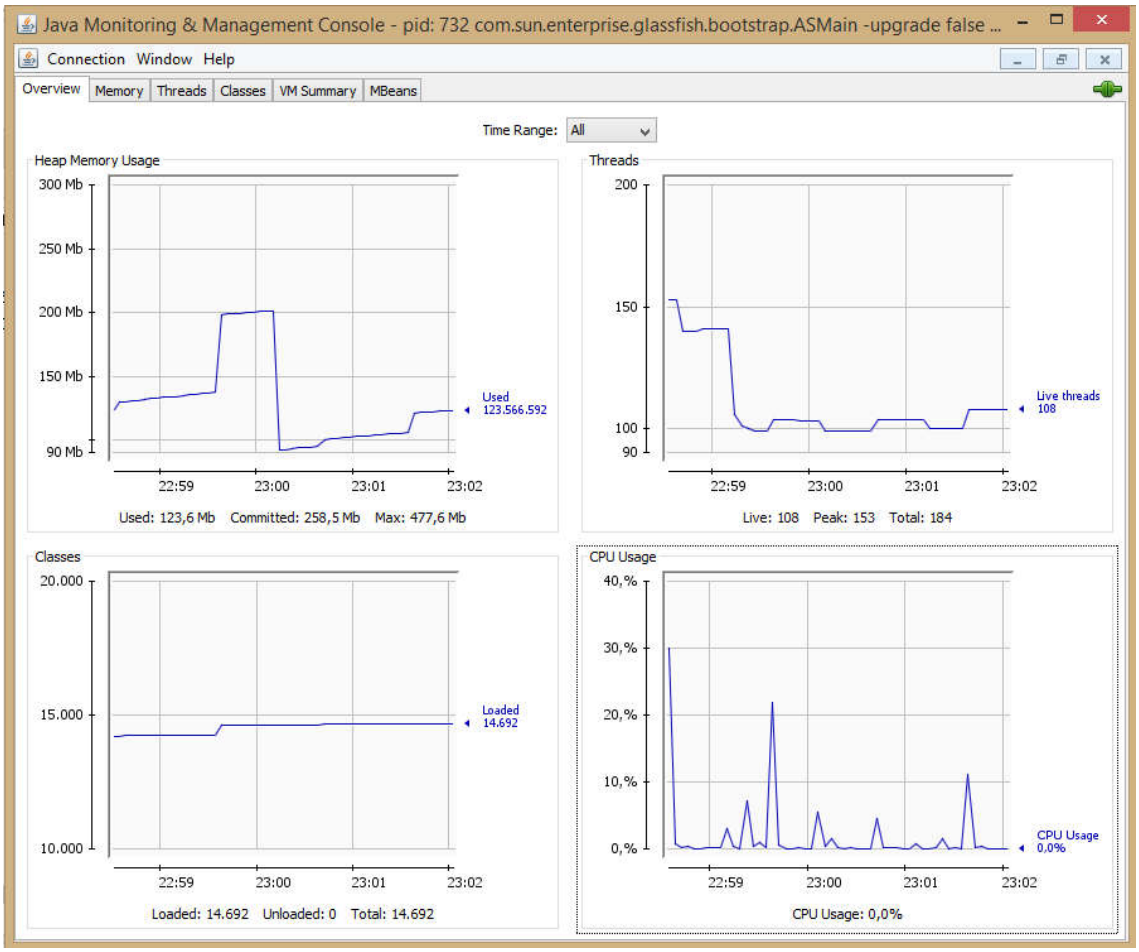



**Figura 111 - Datos servidor Glassfish4 tras pasar Test3**

**[fuente: propia]**

Tras pasar el tercero de los test, transformación de BO a Sistema Central con el sistema ya cargado (EJB's creados, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 23:01:35, obteniendo los siguientes resultados.

- **Memoria en uso:** 123 Mb.
- **Uso de CPU:** Aproximadamente 12 %.
- **Clases cargadas:** 14692.
- **Número de hilos vivos en el servidor:** 108.

- *Captura de datos después de test4.*

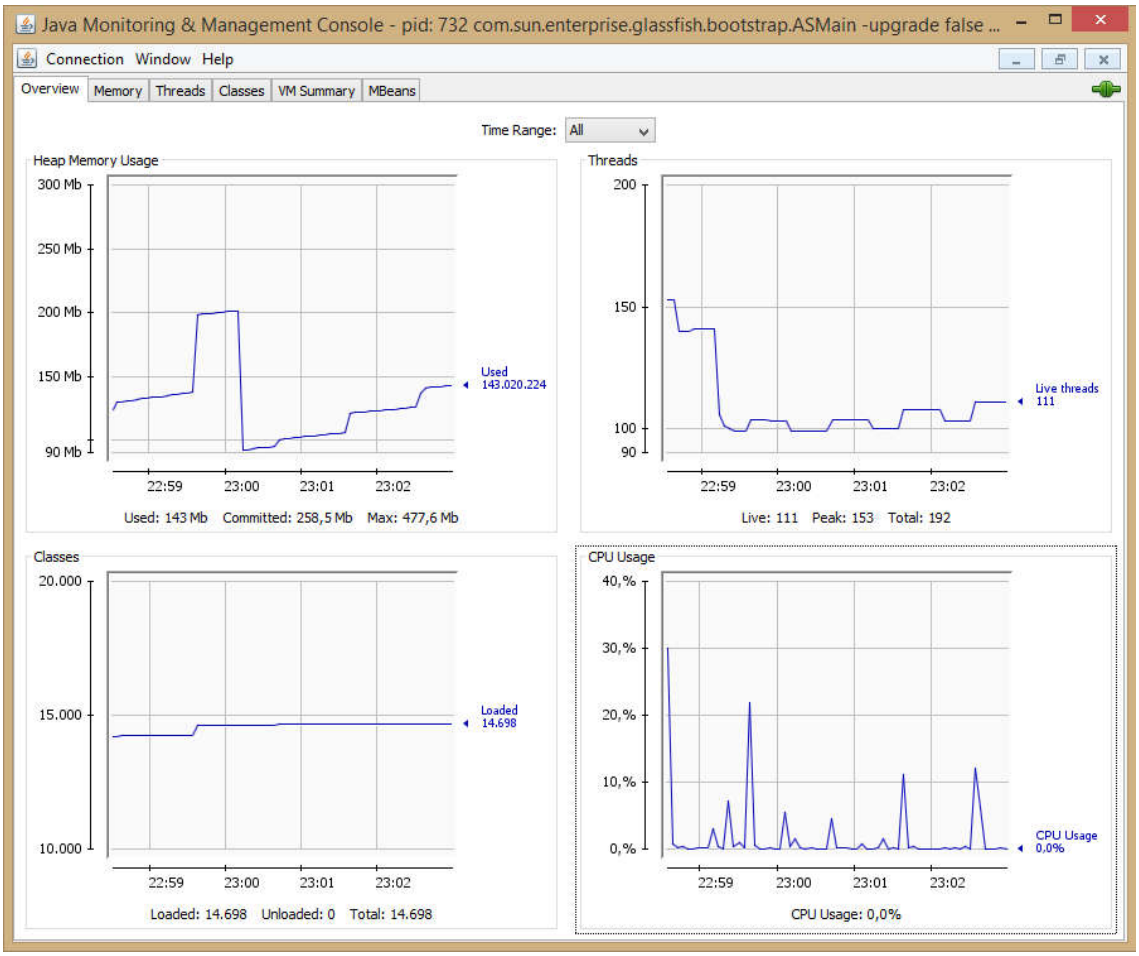



**Figura 112 - Datos servidor Glassfish4 tras pasar Test4**

**[fuente: propia]**

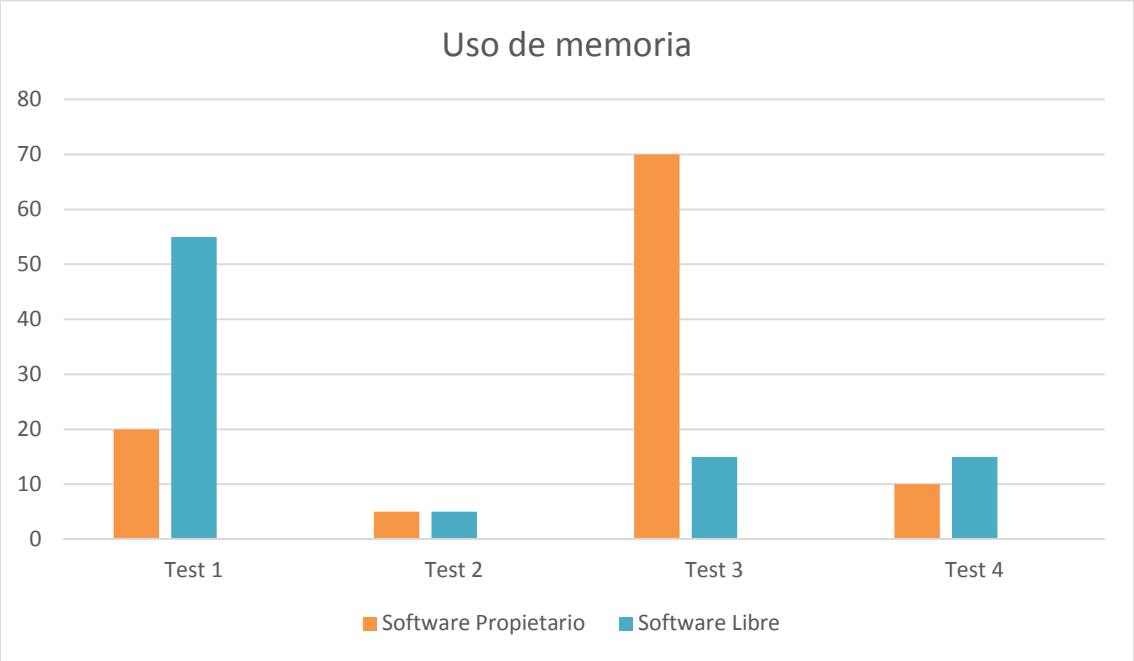
Tras pasar el cuarto de los test, transformación de Sistema Central a BO con el sistema ya cargado (EJB's creados, MDBs…) obtenemos los siguientes resultados.

La ejecución se realiza a las 23:03:40, obteniendo los siguientes resultados.

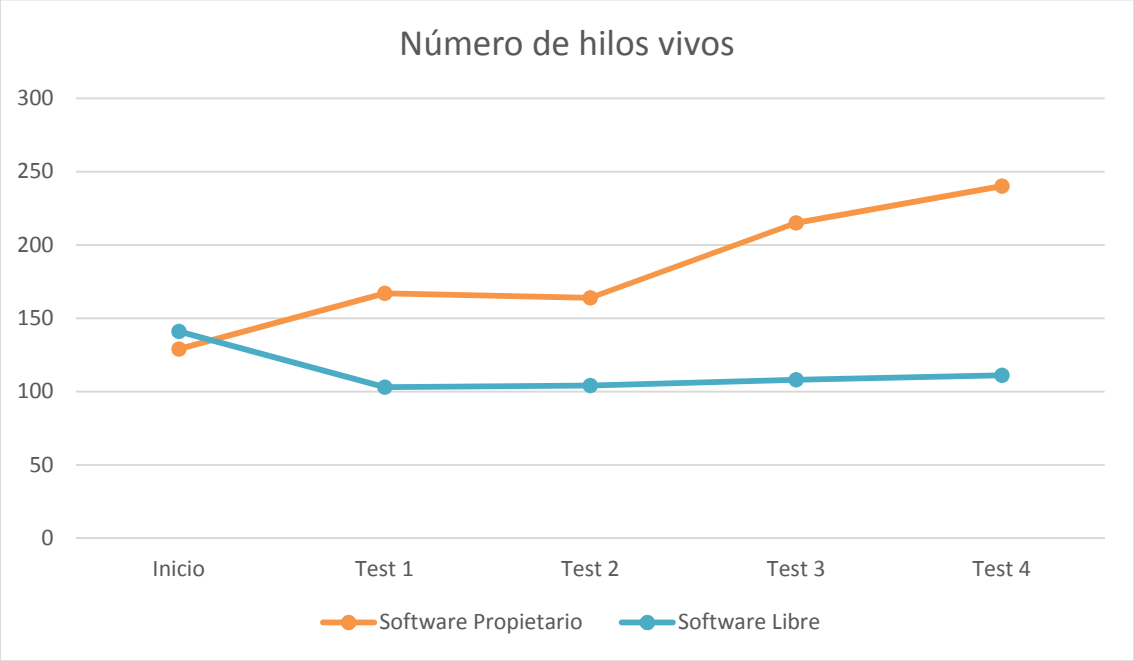
- **Memoria en uso:** Se incrementa a aproximadamente 143 Mb.
- **Uso de CPU:** Aproximadamente 12,5 %.
- **Clases cargadas:** 14698
- **Número de hilos vivos en el servidor:** 111

Con los datos obtenidos se crearán cuatro gráficos, uno para cada uno de los parámetros medidos, de forma que podamos comparar el comportamiento de los dos servidores ante la ejecución de los test.

Para construir el gráfico de evolución de la memoria revisaremos los datos obtenidos, extrayendo de forma aproximada la variación desde que arranca el test hasta que termina.




A la vista de los valores representados en el gráfico, y la evolución que se muestra en las capturas realizadas a la aplicación JConsole, el uso de memoria no va a ser un factor significativo en nuestra comparativa, ya que se encuentra muy influenciado por los ciclos de GC del servidor.

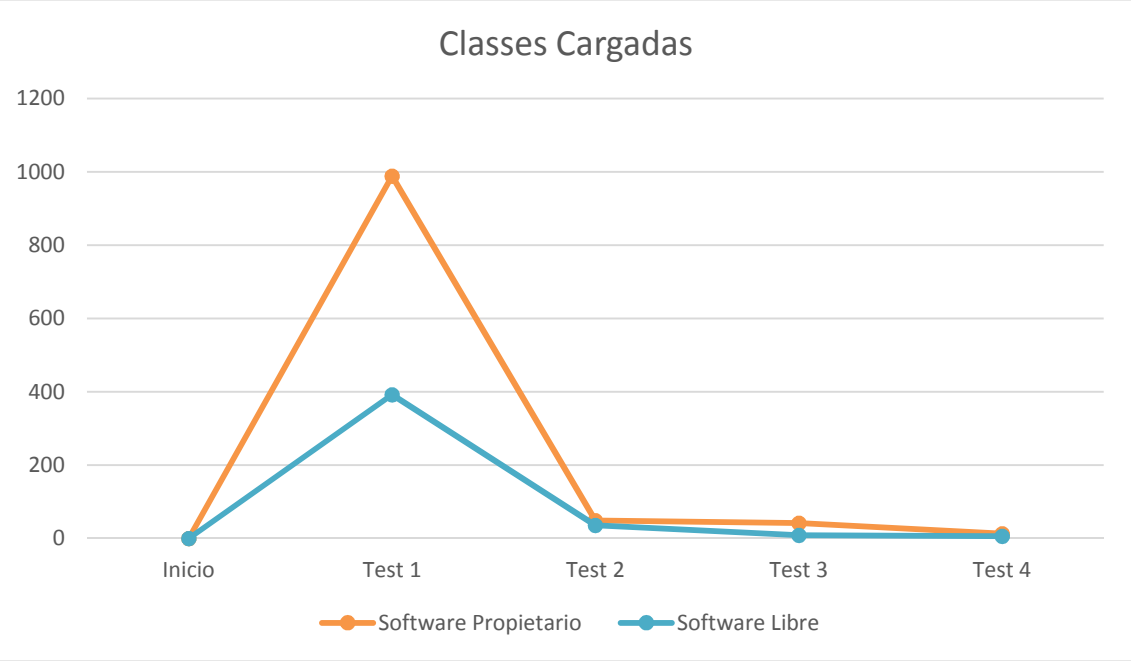


El número de hilos es un parámetro que depende del servidor de aplicaciones en el que ejecutamos cada aplicación, por tanto, es un parámetro que debemos tener en cuenta a la hora de evaluar las diferentes soluciones.

Como podemos ver en el gráfico, el número de hilos al arrancar el servidor es muy similar (129 frente a 141), pese a esto, la evolución no es la misma.

En el caso del aplicativo construido sobre Software Propietario, se observa que el número de hilos va creciendo en cada ejecución, descendiendo cuando el sistema se encuentra en reposo.

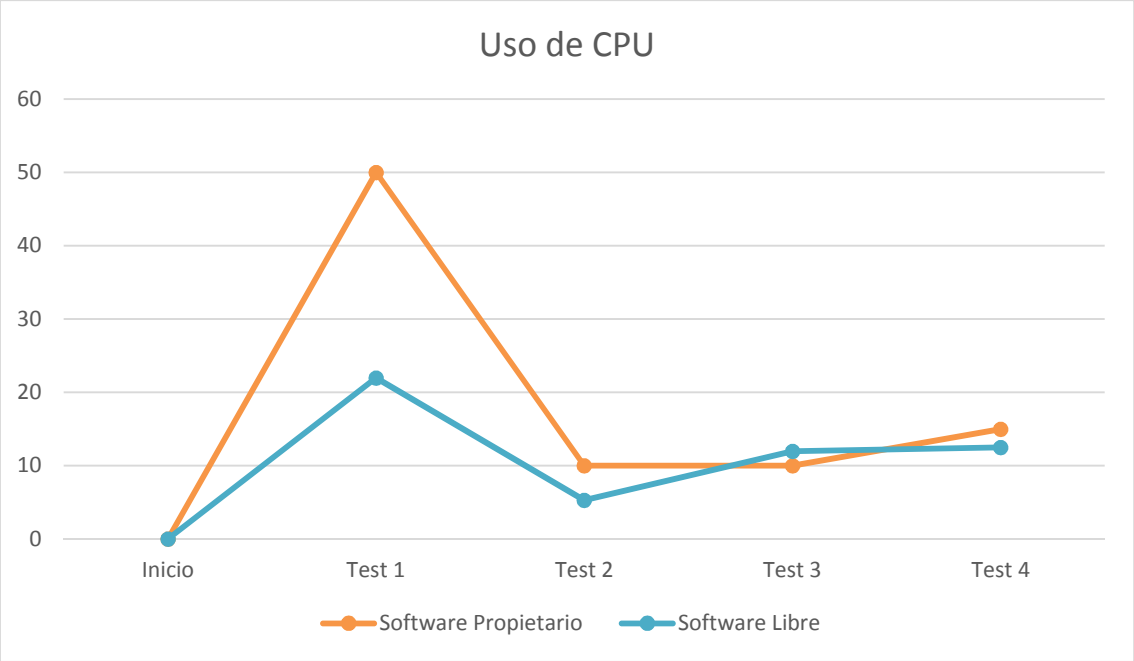
En el caso del aplicativo construido con Software Libre también se observa que el número de hilos crece, pero en este caso el crecimiento no es tan pronunciado como el anterior, sino más suave.




Una vez que el servidor de aplicaciones levanta, lo hace con el número de clases necesario para funcionar y poder levantar las aplicaciones desplegadas en él.

A la vista de los valores representados en el gráfico podemos concluir que las clases cargadas en la ejecución de cada uno de los Test no son un parámetro diferenciador entre aplicaciones, excepto en la ejecución del primero de ellos (Test1).

Esta diferencia puede ser explicable basándose en la carga de clases que realiza el servidor inicialmente. En el caso del servidor Glassfish4, el número de clases cargadas en el arranque es superior a las del servidor Glassfish2, por tanto, la aplicación desplegada en él requerirá cargar menos clases en su primer arranque.




Viendo los datos representados en el gráfico y teniendo en cuenta el gráfico previo sobre clases cargadas, intentaremos explicar la diferencia de uso de CPU existente entre aplicaciones ante la ejecución de Test1.

La aplicación implementada utilizando Software Propietario requiere un mayor uso de CPU en su primera ejecución debido en parte, a que carga un mayor número de clases que la desarrollada utilizando Software Libre.

## 6.4.Implementación

En este apartado se efectuará una revisión de los detalles técnicos relativos a las dos implementaciones, se comparará:

- La arquitectura de clases utilizada para implementar cada una de las soluciones.
- El número de clases que componen cada una de las soluciones.
- Las líneas de código utilizadas para construir cada uno de los módulos funcionales.
- Componentes que simplifican el desarrollo. Conectores y clases de utilidad.
- Configuración.
- Uso del IDE de desarrollo.
- Evolución.
- Facilidad en el desarrollo.
- Depuración y pruebas del sistema.
- Aprendizaje y Soporte.

### 6.4.1. Arquitectura de clases utilizada

Los dos desarrollos compartirán una base común, el sistema de gestión de paquetes, encargado de almacenar y gestionar las diferentes partes en los que se divide el fichero de entrada, permitiéndonos conocer:

- Cuántos fragmentos componen el fichero de entrada.
- Cuáles de estos fragmentos se encuentran procesándose actualmente.
- Cuantos han finalizado y se han escrito a disco.
- Sí ha finalizado completamente el proceso del fichero de entrada.

La arquitectura de clases que proporciona la funcionalidad anterior es la siguiente:

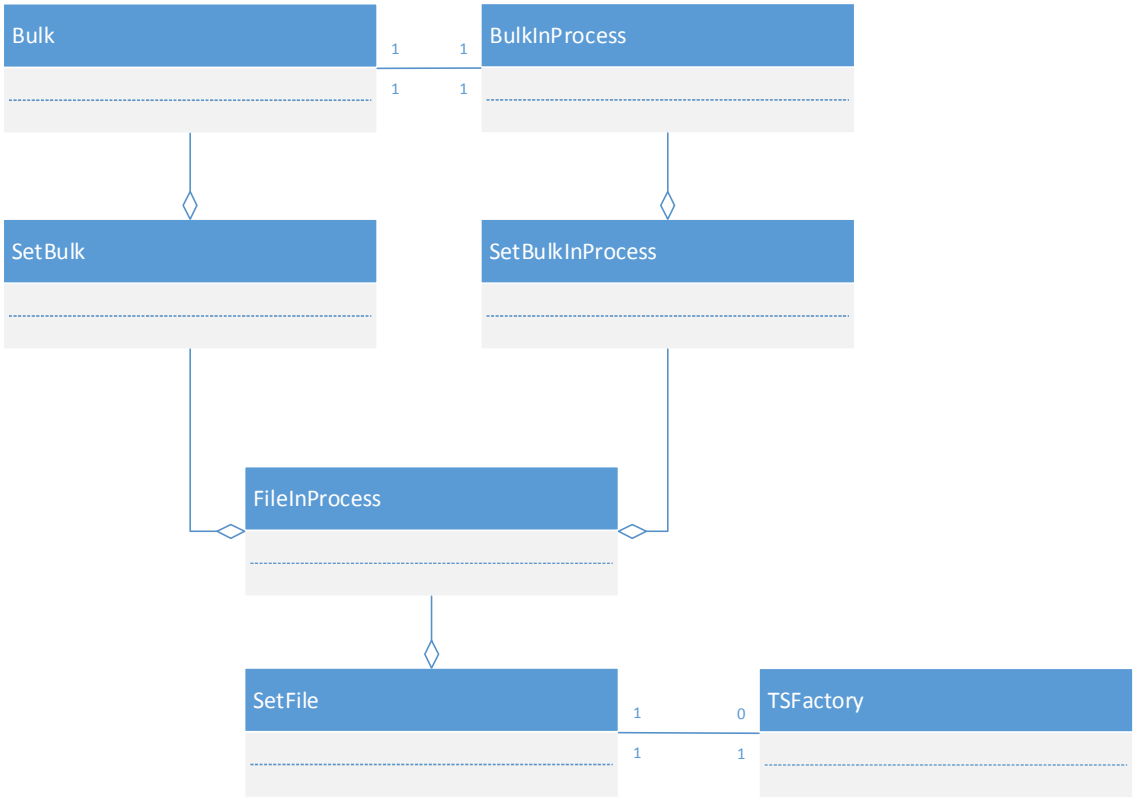



**Figura 113 - Arquitectura usada para la gestión de paquetes**

**[fuente: propia]**

Como ya se ha comentado, el anterior diagrama representa la arquitectura de gestión de paquetes que comparten los dos desarrollos realizados. A continuación, mostraremos la arquitectura usada en la construcción de cada una de las soluciones de transformación.

- *Arquitectura utilizada en la implementación realizada con Software Propietario*

  En este caso no existe una arquitectura de clases como tal, sino que se implementa la funcionalidad en clases (colaboraciones) auto contenidas, es decir, cada una contiene toda la funcionalidad necesaria para realizar la tarea que tiene encomendada.

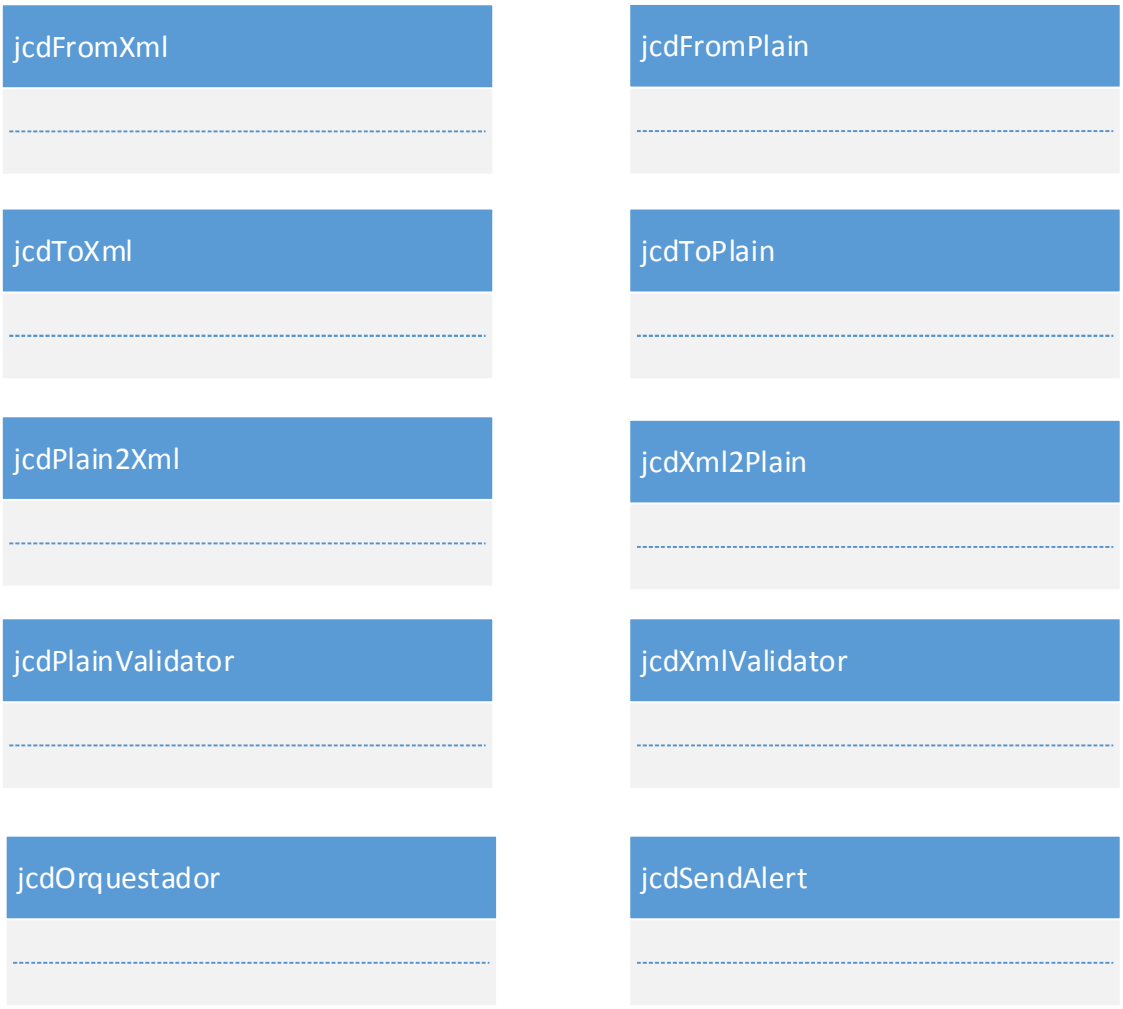



**Figura 114 - Arquitectura del aplicativo utilizando Software Propietario**

**[fuente: propia]**

Ya que la implementación ha sido realizada ciñéndose lo máximo posible a las posibilidades básicas ofrecidas por el IDE, el cual, no se encuentra diseñado para implementar de forma directa una jerarquía de clases y paquetes, sino que se debe construir una librería externa para poder tener dicha jerarquía.

- *Arquitectura utilizada en la implementación realizada con Software Libre*

  El caso de la implementación utilizando software libre es diferente, para la realización de este desarrollo no tenemos ningún IDE que restringe la forma de desarrollar, por tanto podemos de manera sencilla aplicar técnicas de herencia.

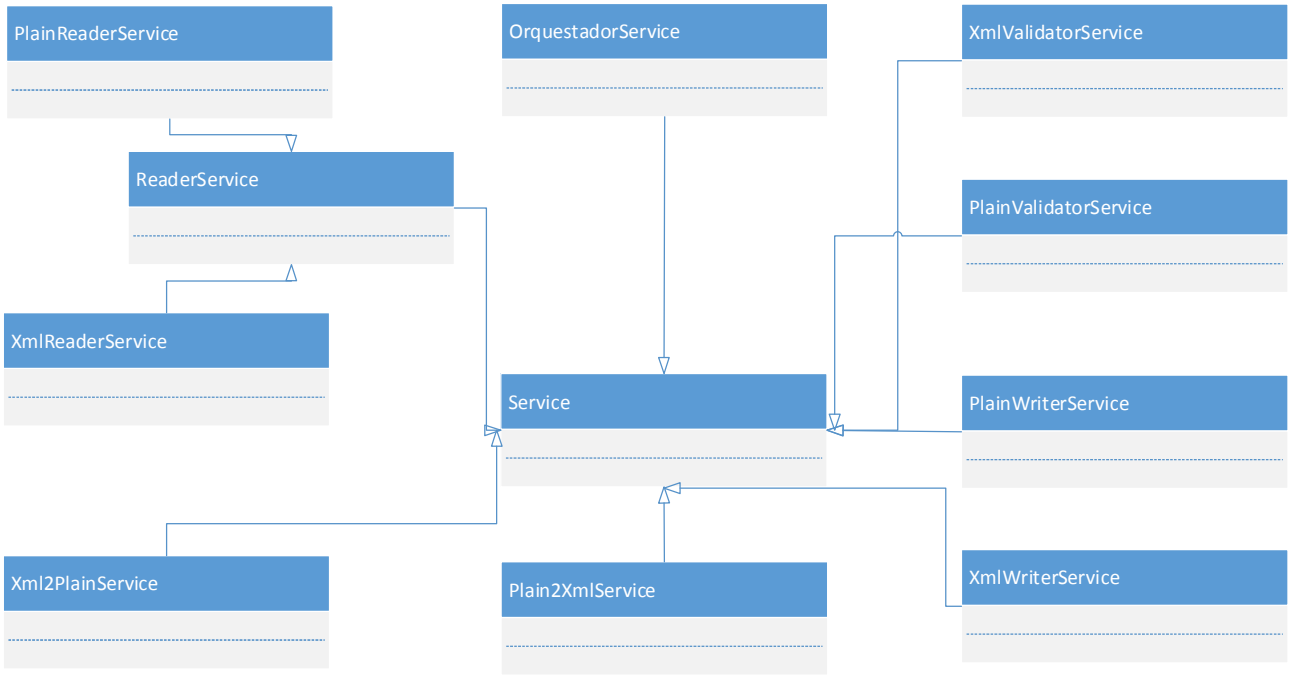



**Figura 115 - Arquitectura utilizada en la implementación usando Software Libre**

**[fuente: propia]**

A la vista de las arquitecturas presentadas en las figuras 114 y 115 podemos concluir que la solución basada en Software Libre nos ofrece una mayor versatilidad ya que:

- Favorece el uso de herencia, que se puede implementar de forma directa y no a través de librerías externas.
- Favorece la reutilización de software. El uso de herencia permite crear clases con funcionalidad común disponible para los hijos.
- Favorece el mantenimiento del software. Reutilizar software facilita en gran medida el mantenimiento del aplicativo. Ante una situación de fallo en un método, únicamente se debe corregir el método, quedando dicha corrección disponible para todas las clases que lo usan.
- Reduce costes. Por ejemplo, en el caso de no tener herencia y encontrar un fallo en un método implementado en varias clases. Si se detectara un error en una de las implementaciones, deberíamos

corregirlas todas, lo cual incrementa el coste y los posibles puntos de fallo. Sin embargo, existiendo reutilización, únicamente habría que corregir en un solo sitio.

**Conclusión:**

Aunque las dos implementaciones permiten el uso de herencia, es un punto a favor del uso de Software Libre, poder implementarla de forma directa, no teniendo que recurrir a su construcción utilizando librerías externas, como sería en el caso de JavaCAPS.

### 6.4.2. Número de clases que componen cada una de las soluciones.

Realmente no existe una diferencia notable en cuanto al número de clases que componen una u otra solución.

- *Solución desarrollada utilizando JavaCAPS consta de:*
    - 10 clases, una por cada uno de los módulos funcionales. Cada una de las clases contiene las constantes necesarias para su funcionamiento.
    - 1 clase de utilidad para la fusión de documentos XML.
    - Clases autogeneradas para el tratamiento de mensajes (BO Message, Message (XML Sistema Central), Common Envelope y AlertMenssage).

- *Solución desarrollada utilizando Software Libre consta de:*
    - 10 clases, una por cada uno de los módulos funcionales.
    - 1 clase de utilidad para la fusión de documentos XML.

  Debido a que se utiliza herencia aparecen dos clases nuevas.
    - 1 clase lectora común que actua como padre de los lectores particulares. Contiene funcionalidad común para el resto de lectores.
    - 1 clase padre del resto de Servicios.

  Dos nuevas clases de utilidad.
    - 1 clase de constantes, la cual agrupa las constantes utilizadas por el aplicativo.

- 1 clase de utilidades para el tratamiento de fechas.

También aparecen las clases necesarias para el tratamiento de la mensajería XML mediante JAXB (Message (XML Sistema Central), Common Envelope y AlertMenssage).

**Conclusión:**

Comparando el número de clases que componen cada uno de los sistemas, se puede concluir que este no será un parámetro determinante a la hora de decidir cuál es mejor implementación, ya que en ambos casos el número de clases es prácticamente el mismo.

### 6.4.3. Número de líneas de código

Una métrica objetiva será ver el número de líneas de código (sin contar espacios en blanco) necesarias en cada implementación para hacer lo mismo, por ejemplo, el número de líneas de código escritas para que un lector funcione igual en JavaCAPS y en la implementación Libre.

| **Lector Formato Plano** | |
|---|---|
| Software Propietario | Software Libre |
| 172 | 249 |
| **Lector Formato XML** | |
| Software Propietario | Software Libre |
| 166 | 375 |
| **Orquestador** | |
| Software Propietario | Software Libre |
| 412 | 382 |
| **Validador Formato Plano** | |
| Software Propietario | Software Libre |
| 179 | 195 |
| **Validador Formato XML** | |
| Software Propietario | Software Libre |
| 92 | 101 |

| Transformador Formato Plano a XML | |
|---|---|
| Software Propietario | Software Libre |
| 100 | 148 |
| **Transformador Formato XML a Plano** | |
| Software Propietario | Software Libre |
| 100 | 122 |
| **Escritor Formato Plano** | |
| Software Propietario | Software Libre |
| 175 | 189 |
| **Escritor Formato XML** | |
| Software Propietario | Software Libre |
| 197 | 203 |

**Tabla 31 - Número de líneas de código**

**[fuente: propia]**

**Conclusión:**

Comparando el número de líneas de código que componen cada uno de los módulos, salvo en el caso de los lectores, no se aprecian diferencias significativas, es decir, usar un Software Propietario no simplifica en gran medida el código fuente que se debe generar.

Donde si vemos diferencias es en el número de líneas que componen cada uno de los lectores. Esta diferencia se debe a que en JavaCAPS si existe un mecanismo de particionado que en el caso de usar Software Libre no tenemos y debemos construir.

### 6.4.4. Componentes y utilidades disponibles

Cuando se decide utilizar un Software Propietario para implementar un aplicativo, en numerosas ocasiones, una de las justificaciones escuchada es que incluye gran cantidad de componentes, que en el caso de usar Software Libre debemos de implementar, ya que no existen.

En este apartado se intentará comprobar si el párrafo anterior es cierto, es decir, ante una funcionalidad existente en JavaCAPS, veremos si existe su alternativa en Software Libre o si existe una forma sencilla de implementarla.

- *BatchInbound*: Componente encargado de realizar la monitorización de un directorio, lanzando el sistema cuando detecta un fichero en él.

  Debido a que nuestro sistema está basado en Camel, este componente puede ser sustituido por el componente *file2*.

- *BatchLocalFile*: Componente que simplifica el acceso (lectura o escritura) a un fichero. Proporciona acceso al contenido del mismo, bien como bloque, retornando todo el contenido o a través de streams (en el caso de ficheros grandes).

  En este caso también existen alternativas, pero en vez de un único componente usaríamos varios:

  - La clase *File*, propia del API de java e incluida en el paquete java.io nos proporcionará acceso al fichero.
  - La clase *IOUtils* de Apache Commons IO, proporciona métodos que permiten el acceso al contenido del fichero como bloque.
  - Para obtener acceso a través de streams tenemos varias alternativas:
    - Clases *FileInputStream* o *FileOutputStream* propios del API de java.
    - Clases *FileReader* o *FileWriter* existentes también en el API de java.
    - Clase *IOUtils* de Apache Commons IO.

- *BatchRecord*: Componente encargado de obtener registros de un determinado fichero en base a un separador. En nuestro sistema los separadores son:

  - **"\r\n"** en el caso del formato emitido por el backoffice de la entidad.
  - **<operacion>** en el caso del formato recibido desde el sistema central.

  En este caso, el componente *BatchRecord* puede ser sustituido de la siguiente manera:

  - La obtención de registros utilizando **"\r\n"** como separador deberá realizarse mediante la clase *LineIterator* existente en Apache Commons IO.
  - La obtención de registros utilizando la etiqueta **<operacion>** se realizará mediante *Stax* o *JDOM*.

- *JMS*: Componente encargado de interactuar con el canal de comunicación, en nuestro caso, OpenMQ.

La aplicación realizada sobre Software Libre utiliza ActiveMQ, perteneciente a Apache, como canal de comunicación, el cual se integra con Apache Camel, a través del componente *activemq*.

- Definición de OTD's para XMLs.

  JavaCAPS posee una utilidad que permite generar la estructura de clases que representa un documento XML desde su esquema. Esta estructura permite parsear o serializar contenidos de forma directa.

  En el caso de Software Libre también se dispone de esta funcionalidad, para ello se deben combinar varias tecnologías:

  - JAXB que permite construir la estructura de clases desde el esquema XSD.
  - Spring oxm. Mediante el cual se crearán los parseadores y serializadores de contenido.
  - Definiciones de tipos propias de Apache Camel.

- Definición de OTD's para el formato plano.

  Al igual que en el caso anterior, JavaCAPS posee una utilidad que permite construir la estructura que representa el documento plano, generando la estructura de clases representativa en el momento de la compilación. Esta estructura permite parsear o serializar contenidos de forma directa.

  En el caso de Software Libre no disponemos de nada que se parezca o que nos proporcione una funcionalidad similar de manera directa. Aun así, la funcionalidad puede ser construida utilizando *Substrings*, *Expresiones Regulares* o *StringBuffers*.

- Email: Componente que facilita el envío de correos electrónicos.

  Como alternativa libre a este componente podemos utilizar JavaMail, que realmente nos proporciona la misma funcionalidad.

**Conclusión:**

Como se ha visto a lo largo del capítulo, los componentes/utilidades que nos proporciona JavaCAPS, excepto para el caso de tener que construir OTD's de parseo para formatos planos, si disponen de una alternativa en software libre.

Por tanto, no se justifica adquirir un software propietario de desarrollo basándose únicamente en las utilidades que proporciona, ya que, para la mayoría de ellas van a existir alternativas libres, lo cual permite ahorrar el coste de adquirir la licencia.

### 6.4.5. Configuración

En este apartado evaluaremos la facilidad a la hora de configurar ambos sistemas.

Ambos sistemas permiten tener la configuración separada del código, es decir, en ningún caso existirá en el código ningún parámetro de configuración.

El párrafo anterior marca uno de los requisitos más importantes a la hora de construir cualquier sistema y las dos opciones lo cumplen, pero con matices.

- La configuración del aplicativo en JavaCAPS se encuentra repartida entre la definición del Environment y los valores de configuración establecidos en los diferentes Conectivity Maps, esto implica, que si se quiere realizar un cambio en alguno de los parámetros de configuración del sistema, la modificación no es inmediata sino que se debe:

  1. Abrir el IDE.
  2. Modificar el parámetro en el componente, enviroment o conectivity.
  3. Compilar la aplicación con el cambio realizado. Se regenerará el ear con la nueva configuración.
  4. Desplegar.

  Aunque realmente cumple la premisa de separar código fuente de parámetros de configuración, a efectos prácticos esto no es así, ya que para realizar la modificación de un parámetro debemos igualmente recompilar y redesplegar el aplicativo.

- En el caso del aplicativo desarrollado utilizando Software Libre, la configuración del sistema se encuentra en ficheros de propiedades que son cargados utilizando la clase *PropertiesPlaceHolder* de Spring.

  La ubicación de estos ficheros de propiedades puede ser:

  - Incluidos en el desplegable. Este caso es el mismo que el anterior, realizar una modificación implica cambiar el properties, recompilar y redesplegar.

  - Fuera del desplegable. En este caso, los ficheros de propiedades se deben ubicar en un directorio accesible desde el classpath del servidor, de esta forma la aplicación será capaz de leerlos.

    En esta segunda situación, para realizar una modificación de un parámetro de configuración, únicamente se modifica/n la/las propiedades y se reiniciará el aplicativo.

**Conclusión:**

Los dos tipos de implementación permiten cumplir la premisa principal, que es permitir separar código fuente de configuración del sistema.

Dicho lo anterior, poder ubicar los ficheros de propiedades en un directorio en el classpath del servidor proporciona una gran ventaja ya que, reconfigurar uno o varios parámetros no implica tener que recompilar ni redesplegar el aplicativo, lo cual es un coste en tiempo y un foco desde donde se pueden incluir errores.

### 6.4.6. Uso de un IDE de desarrollo

El siguiente punto a tratar será el IDE de desarrollo utilizado para realizar la implementación de cada una de las soluciones.

Para JavaCAPS utilizaremos NetBeans 6 + Conjunto de Plugins.

Para la implementación con Sofware Libre utilizaremos Eclipse Mars.

Aunque Eclipse Mars es más nuevo que NetBeans 6, ambos IDEs cumplen perfectamente su cometido a la hora de realizar la implementación del sistema.

**Conclusión:**

Ya que los dos IDEs cumplen perfectamente el cometido para el que van a ser usados, no hay en este punto aspectos que hagan destacar una opción sobre otra.

### 6.4.7. Actualizaciones de Software

Un aspecto importante a la hora de considerar utilizar un software es que ofrezca posibilidades para evolucionar, es decir, que sea capaz de actualizarse a nuevas versiones.

Ejemplos del párrafo anterior serían la actualización de la versión de la JVM, el servidor de aplicaciones, librerías…

Por tanto, ser capaces de actualizar los componentes del core es crucial ya que nuevas releases corrigen bugs o introducen mejoras técnicas y/o de rendimiento que merece la pena aprovechar.

JavaCAPS, es un Software Propietario que actualmente se encuentra en su versión 6.3 liberada en Abril 2011 y que finaliza soporte en Abril de 2017 y uno de los principales problemas es, que depende de la empresa creadora para su

actualización.  Mientras esta no actualice su producto no existe la opción de poder usar las mejoras implementadas en nuevas versiones del software base.

La solución basada en Software Libre no depende de ninguna empresa, aunque el mantenimiento principal es realizado por Apache ya que nuestra base es Apache Camel.

Pese a esto, los componentes core, como la versión de máquina virtual, el servidor de aplicaciones… pueden ser actualizados a sus últimas versiones sin ningún problema y garantizando que el sistema seguirá trabajando. Esto es posible porque no dependemos de una versión de producto concreta, sino de un conjunto de componentes (librerías) independientes del core seleccionado para trabajar.

**Conclusión:**

En este apartado podemos observar uno de los principales problemas del software propietario, depender de una empresa. Si la empresa decide unilateralmente no actualizar o no evolucionar el software, será imposible acceder a mejoras o correcciones que existirían con nuevas versiones.

### 6.4.8. Facilidad para realizar el Desarrollo

El sistema de desarrollo que utiliza JavaCAPS consta de los siguientes componentes:

- Un repositorio para control de versiones.
- Un IDE de desarrollo (NetBeans 6).

Para realizar cualquier tipo de desarrollo, es obligatorio primero tener activo el repositorio, si éste no estuviera activo, no se permite crear ningún proyecto nuevo o continuar con el desarrollo de alguno ya existente.

El IDE nada más arrancar conecta con el repositorio, obteniendo información sobre el estado de los diferentes proyectos, no permitiendo realizar ninguna otra acción si éste no se encuentra activo.

Una vez que el IDE conecta con el repositorio se puede comenzar a desarrollar o continuar con un desarrollo ya existente.

Un aspecto negativo en el uso de JavaCAPS es el tiempo que se tarda en abrir un recurso, desde que se indica la intención de abrirlo hasta que realmente se abre pasa bastante tiempo.

También existen fallos cuando se intentan agregar componentes nuevos a una colaboración, renombrado de elementos…

El sistema para realizar el desarrollo utilizando Software Libre varía bastante respecto del proporcionado por JavaCAPS, en este caso no existen tantas restricciones como en el caso anterior.

Se puede o no usar un control de versiones, como por ejemplo CVS, Subversion…, pero no será necesario tener que conectar obligatoriamente con él para poder crear un nuevo proyecto o continuar con uno ya existente.

Además, se podrá utilizar cualquier IDE, no teniendo que utilizar uno en concreto como en el caso anterior. Otro aspecto importante es que la apertura de un recurso es inmediata.

**Conclusión:**

Realmente JavaCAPS no facilita, hablando en términos de tiempo, el desarrollo de un aplicativo, haciendo que éste se incremente.

Otro punto negativo es requerir que el repositorio esté levantado para poder desarrollar, si estuviéramos en una situación donde fuera imposible contactar con el respositorio, se tendría parado el desarrollo hasta que se restableciera la comunicación.

### 6.4.9. Depuración y Pruebas del Sistema

Una de las funcionalidades más importantes que debe proporcionar cualquier sistema de desarrollo es la capacidad de poder depurar el código ante cualquier error, de forma que la identificación del problema sea sencilla.

De igual forma se necesita tener un mecanismo que permita testear de forma sencilla el aplicativo, de forma que se pueda garantizar su correcto funcionamiento.

Además, cuando se trabaja con sistemas de integración se necesita poder realizar tanto pruebas unitarias como pruebas de integración del sistema.

JavaCAPS cumple con todas las peticiones realizadas en los párrafos anteriores, es decir:

- Permite realizar test unitarios sobre los OTDs. De esta forma garantizamos que el objeto creado parsea/serializa de forma correcta.

- Permite también realizar test unitarios de las diferentes colaboraciones implementadas, permitiendo comprobar que el código escrito es correcto.

- Si habilitamos la depuración en el servidor de aplicaciones, el código implementado puede ser enlazado con la aplicación pudiendo de esta forma depurarlo.

De igual forma, la aplicación desarrollada utilizando Camel + Spring (Software Libre), ofrece las mismas posibilidades:

- Utilizando JUnit se podrán realizar test unitarios sobre cualquier pieza de código.

- Los test unitarios, si son lanzados en modo depuración pueden utilizar las herramientas de depuración proporcionadas por el propio IDE.

- El IDE seleccionado permite configurar el servidor de aplicaciones, enlazando de forma automática el código con la aplicación desplegada, permitiendo por tanto la depuración.

- También se puede lanzar el aplicativo en modo depuración, esto proporciona la capacidad de realizar pruebas de integración sin tener que desplegar el aplicativo en un servidor de aplicaciones.

**Conclusión:**

En ambos desarrollos existe la posibilidad de realizar pruebas unitarias o del sistema completo, con lo cual, independientemente de los detalles particulares sobre cómo se realizarán cada una de estas pruebas, éste es un aspecto en el que ninguna de las dos aproximaciones destaca mucho sobre la otra, es decir, son similares.

Sobre la posibilidad de depurar el código, aunque las dos aplicaciones lo permiten, la forma que tiene JavaCAPS es mucho más compleja que la proporcionada por Eclipse que simplifica mucho más el proceso.

### 6.4.10. Aprendizaje y Soporte

En este apartado se hablará de cómo es la curva de aprendizaje estimado para cada una de las tecnologías.

También se comentará cómo se proporciona Soporte Técnico para dudas o errores.
En ambas situaciones la curva de aprendizaje es alta, debido a esto, este parámetro no será característico.

- En el caso de JavaCAPS, se debe buscar mucho ya que casi no existe documentación que explique cómo debe ser usado, además, casi toda

la documentación existente es privada y hay que entender muy bien cómo funciona para poder trabajar con él.

- En el caso de las tecnologías de Software Libre, como Camel, Spring…, también tienen una curva de aprendizaje alta, la ventaja este caso es que existen numerosos tutoriales que pueden ser consultados, así como una gran cantidad de ejemplos.

La forma de proporcionar Soporte varía bastante para ambos:

- En el caso de JavaCAPS, el soporte lo proporciona la empresa dueña del producto, actualmente Oracle.

- En la solución utilizando Software Libre lo proporciona la comunidad. Esto realmente no es una desventaja ya que los componentes usados para desarrollar la solución tienen detrás una comunidad muy activa y foros oficiales donde realizar las consultas oportunas.

**Conclusión:**

Teniendo en cuenta los factores explicados ninguna de las dos alternativas destaca sobre la otra, ya que, en ambos casos la curva de aprendizaje será alta y se ofrece un buen soporte.

# Capítulo 7: Conclusiones

Como ya se ha comentado, muchas entidades poseen sistemas antiguos, los cuales deben comunicarse entre sí o con un sistema central que actúa como receptor de mensajería.

Estas entidades, como ya se ha explicado, tratan sus datos mediante formatos muy diferentes y deben comunicarse entre ellas, por esto, deben construir sistemas de integración capaces de actuar como traductores de formato.

Muchas grandes empresas sacaron al mercado suites SOA que permiten construir estos sistemas de traducción (integración) como, Sterling B2B Integrator de IBM, JavaCAPS de Sun Microsystems, Oracle SOA Suite de Oracle… que permiten desarrollar el sistema pero con un coste por licencia muy alto.

Al mismo tiempo, comenzó a surgir software libre capaz también de desarrollar sistemas SOA completos, entre estas soluciones se encuentran Camel o Mule, que junto con otras tecnologías también libres se pueden equiparar incluso con los sistemas propietarios.

La mayor parte de las entidades, a la hora de construir sus sistemas de traducción seleccionan software de pago, pero, existiendo la posibilidad de usar software libre, merece la pena pagar el importe de una licencia para construirlo o se puede utilizar una solución libre, siendo en este punto donde surge la pregunta:

**¿Es preferible utilizar software propietario para desarrollar un sistema de integración?**

Según los comentarios escuchados en diferentes reuniones de trabajo la opinión generalizada es que sí por los siguientes motivos:

- El software generado es más fiable y robusto.
- La solución será más rápida que la construida utilizando software libre.
- Se dispondrá de un mejor soporte técnico.
- Ya que dispone de funcionalidades predefinidas, el coste de desarrollo será menor.
- ….

Para intentar responder a la pregunta anterior, se ha desarrollado este proyecto.

La idea ha sido construir el mismo sistema utilizando software propietario y software libre, comparando los resultados para justificar lo más objetivamente posible si está justificada la preferencia anterior. Para ello se han seguido los siguientes pasos:

- Definición del problema, toma de requisitos y diseño técnico. Realizado en los Capítulos 3 y 4.
- Desarrollo de los aplicativos (Software Propietario y Software Libre). Capítulo 5.
- Paso de test y comparación de resultados. Capítulo 6.

Tal y como se ha ido enumerando en el Capítulo 6, podemos concluir que **no existe una justificación real para preferir utilizar Software Propietario frente a Software Libre salvo que sea por preferencias personales.**

Como resumen de los puntos más importantes que nos hacen llegar a la conclusión anterior tenemos:

- Un sistema desarrollado con software propietario debería ser más rápido, que uno desarrollado con software libre. En las pruebas realizadas podemos ver que esto no es cierto siendo ambas implementaciones similares en tiempos.

- El consumo de recursos del sistema (memoria, procesador, estado del servidor de aplicaciones) en ambos casos es similar, no obteniendo ganancia por el uso de componentes propietarios que en teoría deberían ser más óptimos.

- Un aplicativo implementado utilizando software propietario, debido a que este posee muchos componentes ya implementados, debería tener un código fuente más sencillo y ocupar menos líneas de código. Comparando los ficheros fuente de ambos proyectos vemos que el número de líneas es similar para todos los módulos, con lo cual la afirmación anterior tampoco es correcta.

- Como si se ha observado, y se ha comentado en los anteriores puntos, una posible ventaja que pudiera tener un software propietario es la existencia de estos componentes ya implementados. Realmente no hay tal ventaja ya que por cada componente propietario, existe un componente en software libre con una funcionalidad igual o similar.

- Ya hemos comentado en un punto anterior que se tiene la idea de que el desarrollo es más sencillo, lo cual implica un menor tiempo de construcción y por tanto un menor coste. Esta premisa también es errónea ya que:

    - La curva de aprendizaje es similar en ambos tipos de desarrollo.
    - El número de líneas de código es similar en los dos casos, por tanto, el tiempo de desarrollo será similar también.
    - El funcionamiento del IDE en el caso de JavaCAPS es desastroso, tardando mucho en abrir cada módulo, lo cual incrementa el tiempo de desarrollo.

- El soporte técnico del producto tampoco es diferenciador, ya que en los dos casos va a existir; para el Software Propietario lo va a proporcionar la empresa creadora, en el caso del Software Libre lo proporciona la comunidad.

Además, existen otros parámetros comparativos que hacen que el sistema implementado utilizando Software Libre, tome cierta ventaja sobre el implementado utilizando Software Propietario:

- La posibilidad de realizar un mantenimiento de forma sencilla, por ejemplo, el cambio de un parámetro de configuración es mucho más complicado en el

caso de JavaCAPS que usar ficheros de propiedades, ya que en este último caso únicamente se debe cambiar la propiedad y listo.

- El Software Propietario únicamente es actualizado por la empresa que lo creó, con lo cual, si esta empresa no vuelve a actualizarlo, el usuario perderá todas las posibles mejoras de tecnología que vayan apareciendo. Además, el poder descatalogar el software, o dejar de darle soporte, es una decisión unilateral de la empresa propietaria mientras que el software libre, existirá una comunidad que podrá proporcionar este soporte.

# Capítulo 8: Mejoras

En este capítulo comentaremos brevemente que mejoras son aplicables a cada una de las implementaciones:

- **Migración del monitor de paquetes en proceso a base de datos.**

  Actualmente los diferentes paquetes que se encuentran en vuelo se están almacenando en memoria, lo cual es muy peligroso ya que si el sistema cae, no existe posibilidad de recuperación debido a que los datos de la memoria se habrán borrado.

  Si realizáramos la migración del monitor a base de datos, los datos de los diferentes paquetes que se van procesando serían persistidos en la base de datos.

  Ahora ante una caía del sistema podríamos construir un mecanismo de recuperación, el cual cargaría la foto persistida y permitirá continuar el procesado desde ese punto.

- **Inserción de los datos del fichero en una base de datos.**

  El almacenar información relativa al fichero en una base de datos, permitirá poder crear servicios que proporcionen valor añadido explotando los datos almacenados, por ejemplo, crear un sistema de monitorización de la transformación a través de un portal web.

- **Monitor de peticiones expiradas.**

  El sistema implementado está basado en un sistema asíncrono petición/respuesta.

  Una vez que se realiza una petición, el mensaje es almacenado en el monitor y cuando se recibe la respuesta es eliminado del mismo y así sucesivamente hasta que se termina con él.

  ¿Qué ocurriría en el caso de que no se reciba nunca una respuesta?, la respuesta es sencilla, no se eliminaría nunca del monitor, quedando indefinidamente como pendiente. Para solucionar esto una mejora sería crear un monitor que analice las peticiones enviadas, de forma que cuando detecte una que lleva enviada mucho tiempo la trate como errónea eliminándola del monitor.

## Capítulo 9: Ámbito de negocio

Aunque ya lo hemos explicado en la introducción y en el apartado de conclusiones, el sistema desarrollado (bien en Software Propietario, bien en Software Libre) es ideal para situaciones en las que se requiera realizar transformación de formatos.

Estos sistemas serán claramente utilizables en caso de entidades bancarias, que comunican diferentes tipos de operaciones a las cámaras de compensación donde se realizan las oportunas liquidaciones.

Los mainframes de una entidad bancaria suelen estar realizados en Cobol, con lo cual, el formato del fichero de operaciones que se genera será una Copy, el cual no es más que un fichero plano con los datos de las operaciones.

Las cámaras de compensación son de creación más reciente y utilizan sistemas basados en lenguajes más modernos y formatos de intercambio de información actuales como XML.

Estos sistemas de traducción son necesarios debido a que las entidades bancarias deben comunicar sus operaciones en formato Copy a la cámara que entiende XML y recibir las liquidaciones procedentes de cámara en formato XML, traducidas a formato Copy para que la entidad lo procese.

En un espectro más amplio, estos sistemas son aplicables siempre que se requiera realizar una traducción de formatos entre sistemas.

## Bibliografía Utilizada


- WIKIPEDIA. System Integration
  https://en.wikipedia.org/wiki/System_integration

- WIKIPEDIA. Enterprise Application Integration
  https://es.wikipedia.org/wiki/Enterprise_application_integration

- NGECACIT. Application Integration – A short tutorial
  http://ngecacit.com/AIM/AIM-explain/Tutorial-Application%20Integration.htm

- TECNOPEDIA. EAI
  https://www.techopedia.com/definition/1506/enterprise-application-integration-eai

- INVESTOPEDIA. Enterprise Application Integration Definition
  http://www.investopedia.com/terms/e/enterprise-application-integration.asp

- TECHTARGET. What is EAI?
  http://searchsoa.techtarget.com/definition/EAI

- MULESOFT. Understanding EAI
  https://www.mulesoft.com/resources/esb/enterprise-application-integration-eai-and-esb

- Michael Czapski, Saurabh Sahai, Andrew Walker, Brendan Marry, Sebastian Kreuger, Peter Vaneris (2008): **JAVA CAPS Basics: Implementing Common EAI Patterns**

- Gregor Hohpe and Bobby Woolf: **ENTERPRISE APPLICATION PATTERNS**
  http://www.enterpriseintegrationpatterns.com/

- SOADEVELOPERS. Basic Concepts: Hub/Spoke vs ESB
  https://soadevelopers.wordpress.com

- EAITOOLS. EAI Topologies
  http://eaitools.blogspot.com.es/

- elvex.ugr.es. Especificación de requerimientos
  http://elvex.ugr.es/idbis/db/docs/design/2-requirements.pdf

- Universidad de Valencia. Plantillas de casos de uso
  http://users.dsic.upv.es/asignaturas/facultad/lsi/trabajos/032000(2).doc