Trabajo de Fin de Máster

RELEVANCIA DE LAS ARQUITECTURAS SOFTWARE EN LA MODERNIZACIÓN DE SISTEMAS HEREDADOS

Caso de estudio:

"Servicio de Migración Automática. Cobol (sic)-Java JEE"

Relevance of Software Architectures in the Legacy System Modernization

Mahrach Mahrach, Mohammed

La Laguna, 01 de julio de 2019

D. José Luis Roda García, con N.I.F. 43.356.123-L profesor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada *“Relevancia de las arquitecturas software en la modernización de sistemas heredados”, recoge* una descripción teórica con un enfoque arquitectónico de la modernización de sistemas heredados y que, además, ofrece una descripción concreta y detallada de un caso de estudio real del desarrollo de un proceso de modernización de un sistema de cobol (sic) a java JEE en un proyecto realizado en colaboración con una empresa líder en Canarias, OPEN CANARIAS, S.L. ha sido realizada bajo su dirección por D. **Mohammed Mahrach Mahrach**, con N.I.F. 43.844.985-Q.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firma la presente en La Laguna a 01 de julio de 2019.

Agradecimientos:

A mi tutor del trabajo, José Luis Roda García

A Antonio Estévez García y a todo el equipo de

OPEN CANARIAS.

Especial mención para Benito José Cuesta Viera, el Jefe del Proyecto en el que he participado, por su disponibilidad y ayuda durante todo el periodo de colaboración.

Gracias a todos.

Resumen

Cuando surgió la era de la información, las grandes organizaciones y corporaciones, intentaban cubrir sus necesidades de procesamiento y almacenamiento de datos apostando por la tecnología. Con el paso del tiempo, las organizaciones han ido dependiendo cada vez más de estos sistemas, al mismo tiempo que se estaban quedando obsoletos debido el impacto de los avances tecnológicos sobre sus plataformas tecnológicas. A estos sistemas se los denomina sistemas heredados (*legacy system*).

Actualmente, algunos de estos sistemas anticuados y poco competitivos, dejan un margen realmente escaso para la innovación y/o el desarrollo de nuevas líneas de negocio, además de que suponen grandes costes de mantenimiento y de adaptación a nuevos requisitos. Sin embargo, muchas organizaciones siguen manteniendo estos sistemas por la importancia que suponen para sus negocios, pero también por la incertidumbre sobre el coste y la efectividad de su reemplazo o modernización.

En el presente trabajo se analiza el origen del problema de los sistemas heredados, se describe la importancia de estos sistemas para las organizaciones frente a las tecnologías actuales y las nuevas tendencias, se analizan las herramientas disponibles para su modernización y se propone una metodología para la solución al problema, adoptando un enfoque arquitectónico basado principalmente en la modernización dirigida por la arquitectura (ADM) y combinando el Diseño Dirigido por el Dominio (Domain-Driven Design), con estrategias de modernización incremental para asegurar un valor continuo del negocio.

Este documento cubrirá el desarrollo de los componentes fundamentales de la modernización, con una propuesta conceptual basada principalmente en paradigmas arquitectónicos. Adicionalmente se describe un proceso de modernización de un sistema heredado mediante un caso de estudio en el que he colaborado personalmente, el cual consiste en la migración de un sistema heredado Cobol (sic) a una arquitectura moderna con Java JEE. Se ha realizado un estudio de viabilidad de migración, el análisis de la arquitectura de origen y de destino, análisis de brechas entre ambas arquitecturas, definición de planes de transición, descripción del proceso de migración y finalmente un resumen del ahorro obtenido en el proceso de migración por automatización.

El documento comenzará introduciendo al lector en el trabajo que se va a realizar y los objetivos planteados. Luego se llevará a cabo un análisis del estado del arte, seguidamente se describen los conceptos y los paradigmas adoptados para la modernización. Después de definir la metodología y las estrategias de modernización, se desarrollará el proceso conceptual de modernización. Por último, se describe el caso de estudio, finalizando con las conclusiones finales obtenidas.

Palabras clave: Arquitectura de sistemas, Sistemas Heredados, Modernización de Sistemas Heredados, Proceso de Modernización, Migración, Diseño Dirigido por el Dominio, Modernización dirigida por la arquitectura, Automatización de proceso de modernización.

Abstract

When the information age arose, large organizations and corporations tried to cover their data processing and storage needs by betting on technology. Over time, organizations have increasingly depended on these systems, while at the same time they were becoming obsolete due to the impact of technological advances on their technological platforms. These systems are called *Legacy Systems*.

Currently, some of these outdated and uncompetitive systems leave a really scarce margin for innovation and / or the development of new lines of business, in addition they involve large maintenance costs and adaptation to new requirements. However, many organizations continue to maintain these systems because of the importance for their businesses, but also because of the uncertainty about the cost and effectiveness of their replacement or modernization.

In the present work the origin of the problem of the inherited systems is analyzed, the importance of these systems for the organizations is described in front of the current technologies and the new tendencies, the tools available for their modernization are analyzed and a methodology for the solution to the problem, adopting an architectural approach based mainly on architecture-led modernization (ADM) and combining Domain-Driven Design, with incremental modernization strategies to ensure a continuous business value.

This document will cover the development of the fundamental components of modernization, with a conceptual proposal based mainly on architectural paradigms. Additionally, we describe a process of modernization of an inherited system through a case study in which I have personally collaborated, which consists in the migration of a Cobol (sic) Legacy System to a modern architecture with Java JEE. A migration feasibility study has been carried out, the analysis of the origin and destination architecture, analysis of gaps between both architectures, definition of transition plans, description of the migration process and finally a summary of the savings obtained in the process of migration by automation.

Keywords: Systems Architecture, Legacy Systems, Legacy System Modernization, Modernization Process, Migration, Domain-Driven Design, Architecture-Driven. Modernization, Automatic modernization process.

Índice General

Capítulo 1. Introducción 8
1.1 Objetivo General 9
1.2 Objetivos específicos 9

Capítulo 2. Estado del arte 10
2.1 Antecedentes 10
2.2 Sistemas heredados 11
2.3 Clasificación de Sistemas heredados 12
2.4 Evolución de los sistemas heredados 14
2.1 Problemática de los sistemas heredados 15
2.2 Análisis de las alternativas 16

Capítulo 3. Conceptos y Metodología de la modernización 19
3.1 Arquitectura del software 19
3.2 Domain Driven Design (DDD) 20
3.3 Desarrollo Dirigido por Modelos (MDD) 21
3.4 Arquitectura dirigida por modelos (MDA) 21
3.1 Modernización dirigida por la arquitectura (ADM) 22
3.2 Reingeniería 23
3.3 Modelo en herradura 23
3.4 Modernización incremental (StranglerApplication) 28

Capítulo 4. Proceso de Modernización 30
4.1 Actividades preparatorias 30
4.2 Inventario de patrones de origen (Idioms Precursores) 32
4.3 Inventario de patrones de destino (*Idioms* soluciones) 32
4.4 Análisis y resolución de brechas 32
4.5 Estrategia de migración de datos 33
4.6 Generación de código 33
4.7 Validación y Pruebas 33

Capítulo 5. Caso de estudio 34
5.1 Definición del caso de estudio 34
5.2 Actividades preparatorias 36
5.3 Patrones de origen (*Idioms* Precursores) 37
5.4 Patrones de destino (*Idioms* soluciones) 40
5.5 Análisis y resolución de brechas 43
5.6 Estrategia de migración de datos 46
5.7 Generación de código 47
5.8 Validación y Pruebas 48
5.9 Resultados 50

Capítulo 6. Conclusiones 54

Apéndice A. ScriptAhorro 56

Apéndice B. Salida del ScriptAhorro 57

Apéndice C. Resultados de la estimación del ahorro 58

Apéndice D. Pruebas 59

Índice de figuras

Figura 1: Ciclo de vida del sistema de software [13] 15
Figura 2 : Modelos MDA según nivel de abstracción.. 22
Figura 3: Modelo en herradura o HorseShoe Model [26] 24
Figura 4:(a) Modelo original y (b) sin transformaciones horizontales [27]............... 25
Figura 5: Modelo de Modernización en Herradura de ADM [31] 25
Figura 6: Arquitectura KDM 27
Figura 7: StranglerApplication[36] 29
Figura 8: Esquema del proceso de migración 30
Figura 9:CAKES 35
Figura 10: Servicio seleccionado para la migración "Recorridos" 36
Figura 11: Arquitectura de destino de aplicaciones 40
Figura 12: interacción de los componentes de la arquitectura 42
Figura 13: Patrones de estructura diferentes en el sistema origen (COBOL) 45
Figura 14 : Artefactos en plataforma DELTA/SIC 45
Figura 15: Interfaz JSON Parcer 49

Índice de tablas

Tabla 1:Evolución de lenguajes de programación de la tercera generación 14
Tabla 2: Correspondencias COBOL / Java 46
Tabla 3: Grado de implementación automática de patrones 47
Tabla 4: Código generado 48
Tabla 5: Artefactos del servicio... 51
Tabla 6: Código por herramientas Generadoras KEY y GNSIS 51

Capítulo 1. Introducción

Hoy en día la revolución tecnológica se ha caracterizado por su gran capacidad de penetración en nuestras vidas y ha modificado sustancialmente nuestros hábitos diarios. El avance imparable de la tecnología está teniendo un impacto transversal no solo en el ámbito social y en la vida cotidiana, sino en todos los ámbitos: economía, energía, comunicación, comercio, cultura, etc.

Durante las últimas décadas, las plataformas tecnológicas que ofrecen soporte a los sistemas de información han evolucionado continuamente, desde sistemas monolíticos y arquitecturas cerradas hacía entornos cada vez más distribuidos y con arquitecturas abiertas. Su objetivo ha sido alinear las infraestructuras tecnológicas con las estrategias y prácticas de negocio, pero también poder amortiguar el gran impacto de los avances tecnológicos sobre sus plataformas. No obstante, aún existe gran cantidad de sistemas que fueron desarrolladas en su momento para plataformas antiguas y que han sobrevivido a pesar del gran número de razones en contra de su continuidad, suponiendo no solo elevados costes de mantenimiento y operación, sino condicionando la capacidad de los negocios de beneficiarse de la evolución de la tecnología. Estos sistemas reciben el nombre de Sistemas *Heredados o Legados* (*legacy system*). [1]

Migrar un sistema heredado hacía un sistema moderno, supondría elevados costes de transición, además de los riesgos que implicaría un cambio de gran envergadura, sobre todo cuando se trata de sistemas con un factor crítico de éxito. Estos sistemas necesitan un tratamiento especializado que da a lugar al concepto de *Modernización de Sistemas Heredados* (*Legacy Systems Modernization*). Hoy en día la evolución de la arquitectura del software y los nuevos paradigmas de arquitectura del software ayudan a que estos sistemas puedan ser modernizados, consiguiendo así reducir los riesgos y los elevados costes de las alternativas clásicas de transformación.

Este documento contiene la descripción del Trabajo de Fin de Máster y se organiza como sigue: Primero se encuentra este capítulo de introducción que contiene los objetivos de este trabajo. Seguidamente, en el Capítulo 2 se realiza el análisis del estado del arte donde se definen los sistemas heredados, la problemática asociada a ellos, sus categorías y las alternativas de solución, mientras que en el Capítulo 3 se describe la metodología de modernización junto con los conceptos y paradigmas más relevantes en arquitectura del software. El Capítulo 4, describe el proceso de la propuesta de modernización en base de la metodología descrita en el capítulo anterior. En el Capítulo 5, se describe el caso de estudio, aplicado a la solución. Finalmente, el Capítulo 6 contiene las conclusiones obtenidas de este proceso.

1.1 Objetivo General

El objetivo de un proyecto de modernización de aplicaciones es crear un nuevo valor de negocio a partir de aplicaciones existentes. Esto se lleva a cabo, mediante la recuperación del conocimiento de un sistema heredado y a continuación, aplicando estrategias de modernización de arquitectura y modelo de datos a dicho sistema, consiguiendo así alinear la arquitectura y las infraestructuras tecnológicas de las organizaciones con las necesidades actuales del negocio.

El objetivo principal de este trabajo es proponer una solución conceptual al problema de los sistemas heredados en un marco arquitectónico. Se pretende estructurar un modelo de referencia para nuevos proyectos de modernización de sistemas heredados, lejos de soluciones poco eficientes.

En este trabajo se ha comprobado de forma práctica y detallada la solución aplicada a un caso real, mediante un caso de estudio de modernización de un sistema heredado en un entorno empresarial.

1.2 Objetivos específicos

El objetivo general descrito en el apartado anterior se materializa en los siguientes objetivos específicos:

- Evaluar el diseño, la implementación de arquitecturas y las tecnologías de soporte para la migración de sistemas heredados sin alterar el funcionamiento del sistema y los planes de negocio.
- Diseñar una solución que sea fácilmente escalable y que pueda adaptarse rápidamente a la mayoría de los escenarios de negocios. Además, esta debe permitir la migración incremental de las aplicaciones del cliente sin interrumpir su funcionamiento.
- Diseñar un proceso de modernización que brindará una solución bajo una arquitectura que soporta diferentes tecnologías y herramientas alternativas modernas de tal forma que sea flexible en términos de presupuesto.
- Refinar los conceptos de la metodología propuesta con su aplicación en un caso de estudio en entorno real que consiste en la migración incremental de un sistema heredado, que permita comprobar la factibilidad y retroalimentación de la solución planteada.
- Lograr el máximo grado de automatización posible y evitar los elevados costes de transición y los riesgos que implicaría la transición. Este objetivo solo es posible si se dispone de herramientas de migración automática.

Capítulo 2. Estado del arte

Los sistemas heredados son fuente de muchos problemas en sistemas de información. Entre el 60% y 85% del presupuesto de las direcciones de TI está destinado al mantenimiento de aplicaciones heredadas [1]. El mantenimiento puede implicar ajustes y adaptación de software, desde pequeños parches hasta cambios y modificaciones muy complejas. Además, el hardware antiguo puede requerir capas de compatibilidad adicionales para facilitar la funcionalidad del dispositivo en entornos incompatibles.

En el presente capítulo se brinda información sobre como surgieron estos sistemas, cuáles son sus categorías y características, cuál ha sido su evolución, qué problemas suponen para las organizaciones y cuáles son las posibles alternativas de solución.

2.1 Antecedentes

Los sistemas heredados tienden a expandirse con el tiempo, ya que los esfuerzos para eliminar el código no utilizado casi nunca se financian. "*La compañía Fortune 100, por ejemplo, mantiene 35 millones de líneas de código y agrega un 10 por ciento cada año solo en mejoras, actualizaciones y otro tipo de mantenimiento*". Como resultado, la cantidad de código mantenido por estas compañías duplica su tamaño cada siete años [2]. Más software significa más software para mantener. Por ejemplo, las empresas modifican el software en respuesta a las prácticas comerciales cambiantes, para corregir errores o para mejorar el rendimiento, la capacidad de mantenimiento u otros atributos de calidad. Todo esto está de acuerdo con la primera ley de Lehman: "*Un programa grande que se usa sufre un cambio continuo o se hace progresivamente menos útil*" [3].

Errickson-Connor [4] propuso los pasos de un proceso de modernización de software donde el código heredado se transforma a nuevos lenguajes y nuevos entornos. Propone que, en la primera etapa, el código heredado debe depurarse antes de que se pueda transformar. En la segunda etapa se debe realizar una reestructuración de software, como la identificación y aislamiento y extracción de reglas de negocio como servicios reutilizables. Cuando se extrae el código correspondiente a una regla empresarial, está listo para la transformación en componentes en la etapa tres. La cuarta y última etapa gestiona estos componentes reutilizables en un entorno de software.

Zhang Li, et al., [5] han proporcionado un proceso de modernización denominado Transformación del Modelo de Tollgate "*Modernization method based on tollgate*

1 An Executive Guide to Oracle Modernization. Enabling Strategic Business Transformation. Oracle.com

model" con el que el sistema heredado se puede adaptar a un sistema de arquitectura orientada a servicios (SOA)[2] cuya granularidad es cambiante.

Otros enfoques de modernización de sistemas heredados que han sido desarrollados por investigadores y profesionales del software incluyen Modelo en Herradura, Modernización Dirigida por la Arquitectura y otros paradigmas de desarrollo de software[1][6][7].

En los últimos años se han realizado multitud de proyectos de migración de sistemas heredados desde plataformas origen tales como mainframes IBM y UNISYS hacia sistemas abiertos en UNIX o Linux. En este contexto, también puede mencionarse la conversión de sistemas desarrollados bajo lenguajes propietarios (RPG[3], COBOL[4], etc.) hacia Java u otros sistemas de código moderno.

Cabe mencionar que el Grupo de trabajo de modernización dirigida por arquitectura (ADMTF)[5] del Grupo de Administración de objetos (OMG)[6] ha creado una serie de metamodelos para el intercambio de metadatos de aplicaciones entre plataformas, lenguajes y etapas del proceso de modernización. El ADMTF determinó la necesidad de identificar un conjunto de escenarios bajo los cuales se podría aplicar una o más combinaciones de estos estándares. Estos escenarios proporcionan plantillas para elaborar objetivos de proyectos, planes y entregables relacionados y son los siguientes: Gestión de un conjunto de aplicaciones, Mejora de aplicaciones, Conversión de lenguaje a lenguaje, Migración de plataforma, Integración de aplicaciones no invasiva, Transformación a SOA, Migración de arquitectura de datos, Consolidación de arquitectura de sistema y datos, Desarrollo de *Data Warehouse*, Selección y desarrollo de paquetes de aplicaciones de terceros, Reúso de componentes, Transformación a (MDA)[7] y Seguridad de software [8].

2.2 Sistemas heredados

Los sistemas heredados o legados en el contexto de los sistemas de información representan aquellos sistemas informáticos concebidos décadas anteriores a la actual. Estos sistemas siguen siendo el soporte de las operaciones centrales de muchas empresas, a pesar de sus limitaciones de compatibilidad, obsolescencia o falta de soporte de seguridad. El concepto de "*Sistemas Heredados*" (*Legacy Systems*) puede admitir muchas definiciones según el punto de vista considerado.

Así Ulrich (1994) hace referencia a "*sistemas independientes construidos en una era tecnológica anterior que disponen de precaria documentación*"[9].

---

2 Service-Oriented Architecture (SOA): www.omg.org/technology/readingroom/SOA.htm
3 RPG - IBM: https://www.ibm.com/support/knowledgecenter/es/ssw_ibm_i_71/rzahg/rzahgrpgcode.htm
4 COBOL- IBM: www.ibm.com/support/knowledgecenter/es/ssw_ibm_i_71/rzahg/rzahgcobol.htm
5 The Architecture-Driven Modernization Task Force (ADMTF): http://www.omgwiki.org/admtf/doku.php
6 The Object Management Group (OMG): www.omg.org/about/
7 Model Driven Architecture (MDA): www.omg.org/mda/

Brodie los define como "*todo sistema de información que se resista significativamente a su modificación y evolución, para cubrir cambios en sus requerimientos*" [10].

Bennett (1995) habla de "*grandes sistemas parcialmente desconocidos y vitales para las organizaciones*" [11].

Normalmente un sistema heredado, no es el mismo sistema que originalmente se empezó a utilizar, los sistemas se suelen adaptarse a las tendencias de los mercados, cambios legislativos y de administración entre otros. Esto conduce a que los negocios experimenten continuos cambios, por lo que estos sistemas van sufriendo cambios conforme cambian los negocios. Por otro lado, estos sistemas incorporan un gran número de cambios a lo largo de su vida y es difícil que una persona tenga un conocimiento completo del Sistema.

El legado del sistema heredado, puede estar asociado con plataformas, lenguajes de programación, software de aplicación o procesos que ya no son aplicables a contextos actuales. En teoría, es prioritario para las organizaciones el acceso a la tecnología más avanzada, pero en realidad, la mayoría de las organizaciones siguen teniendo sistemas heredados, de mayor o menor complejidad.

2.3 Clasificación de Sistemas heredados

La enorme diversidad de sistemas heredados, podría ser categorizada según diferentes criterios. Para los efectos de este trabajo, presentaremos dos clasificaciones diferentes, la primera teniendo en cuenta la función o el objetivo del sistema. La otra clasificación, según la generación de lenguaje con el que fue desarrollado.

2.3.1 Clasificación funcional

I.2.3.1.1 Sistemas de gestión de información

Son sistemas en los que se llevan a cabo tareas de gestión de información sobre un conjunto de datos (productos, recursos, ventas, entre otros), para satisfacer las necesidades y las expectativas de sus clientes. Los tipos de sistemas de la información más populares son: Sistemas de control de procesos de negocio, Sistemas de Información de Gestión, Sistemas de apoyo a la toma de decisiones y Sistemas de Información Ejecutiva, entre otros. Habría que Identifica las necesidades de la organización y opta por los sistemas de información más adecuados. La clave del éxito de una empresa no reside en poseer todas las herramientas disponibles, sino en contar con las que más se adaptan a sus características.

I.2.3.1.2 Sistemas de control y automatización

Los sistemas de procesos automatizados son implantados en equipos electrónicos o mecánicos, en su mayoría están hechos con lenguajes de bajo nivel como Assembler, C, Pascal, entre otros. Estos sistemas registran datos como medidas de tiempo, temperatura, niveles, etc.

2.3.2 Clasificación generacional de lenguaje de programación

Los lenguajes de programación se clasifican dentro de un marco llamado "Generaciones de lenguajes de programación". En este marco se encuentran cinco generaciones de lenguajes de programación[12]:

1. Primera generación: Lenguaje de máquina o de bajo nivel, son los primeros ordenadores basado en sistema binario, se programaban utilizando símbolos binarios para comunicar instrucciones al hardware.
2. Segunda generación: Lenguajes simbólicos, simplifican la escritura de las instrucciones con lenguajes ensambladores traduciendo los códigos simbólicos en instrucciones de la máquina.
3. Tercera generación: Lenguajes de alto nivel más especializados independientes de la máquina, inspirados en lenguaje humano, como COBOL, FORTRAN, BASIC, C, etc.
4. Cuarta generación: Programación orientada a objetos tales como Visual Basic, C ++ y lenguaje de consulta estructurado (SQL). Incluyen características nuevas como, generación de código y Gráficas.
5. Quinta generación: Generación de código basada en reglas e inteligencia artificial, para resolver un problema utilizan un enfoque basada en el conocimiento sin que tener especificar cómo resolverlo.

La mayoría de las aplicaciones heredadas son sistemas desarrollados con lenguaje de la tercera generación, por lo tanto, nos centraremos en esta categoría.

I.2.3.2.1 Evolución de lenguajes de programación de la tercera generación

La tercera generación de lenguajes de programación se conoce como lenguajes de alto nivel. Estos lenguajes tienen una sintaxis similar al lenguaje humano para que las personas escriban y entiendan los programas. Un programa escrito en lenguaje de alto nivel es independiente de la máquina y puede ser ejecutados en diferentes arquitecturas. Un ordenador se encarga de traducir el lenguaje de alto nivel a lenguaje ensamblador o código máquina para que pueda utilizar las instrucciones que contienen.

La Tabla 1 muestra una lista de lenguajes de programación más populares de la tercera generación:

Año	Nombre	Predecesor(es)	Desarrollador(es) principal(es)
1956	LISP	IPL	John McCarthy
1957	FORTRAN	A-0	John W. Backus en IBM
1958	ALGOL	FORTRAN	Esfuerzo internacional
1959	COBOL	FLOW-MATIC	El comité CODASYL
1961	COMIT	*	
1962	SNOBOL	FORTRAN II	Ralph Griswold, y otros
1964	BASIC	FORTRAN	John Kemeny y Thomas Kurtz
1967	SIMULA	ALGOL	Ole-Johan Dahl, Bjørn Myhrhaug y otros
1967	APL	*	Kenneth E. Iverson
1968	LOGO	LISP	Seymour Papert
1969	PL/1	ALGOL	IBM
1970	Pascal	ALGOL	Niklaus Wirth, Kathleen Jensen
1972	C	ALGOL	Dennis Ritchie
1972	Prolog	*	Alain Colmerauer

Tabla 1:Evolución de lenguajes de programación de la tercera generación

2.4 Evolución de los sistemas heredados

La evolución de un sistema heredado involucra diferentes aspectos, empezando por resolver incidencias o problemas presentados, el mantenimiento continuo, adaptación de software para las nuevas necesidades o agregar nuevas funcionalidades hasta reemplazar el sistema por otro totalmente nuevo.

En el caso del mantenimiento y resolución de incidencias o problemas en el sistema de información, es una situación que depende de factores de calidad del software y su arquitectura y la de la plataforma o el hardware que la soporta. En este caso habrá que implantar un sistema de mantenimiento y soporte para la resolución de incidencias y realización de actualizaciones periódicas. Sin embargo, el mantenimiento de un sistema no es una solución definitiva dentro de la evolución de los sistemas de información. Además, el mantenimiento suele ser el porcentaje más alto del costo de todo el ciclo de vida de un sistema de información y la mayoría de las empresas se enfrentan a este gasto las últimas décadas, sobre todo si la aplicación sigue resultando efectiva.

Una adaptación de software para nuevas necesidades transforma el sistema original de la aplicación agregando nuevas funcionalidades, supone un gran peligro. Colocar funcionalidad tras otras en un sistema, al final termina por perder el objetivo con el cual fue concebido, esto hace que el sistema se haga más difícil de controlar conforme pasa el tiempo, y su mantenimiento cada vez más difícil y con la falta de documentación se puede perder el control de la situación hasta acabar con un desorden total. En el siguiente apartado veamos cómo se origina el problema de los sistemas heredados.

2.1 Problemática de los sistemas heredados

La problemática de los sistemas heredados va más allá de la revolución tecnológica y la evolución de las tecnologías y plataformas. Los programas de software están sometidos a cambios continuos, las malas prácticas y la documentación desactualizada, no hacen más que empeorar la situación, lo que implica a que antes o después sea irremediable tomar una decisión a veces extrema acerca de su futuro.

El crecimiento del software sometido continuos cambios hasta que el sistema ya no se pueda modificar más y se le da en abandono y nuevamente se empieza por un nuevo desarrollo.

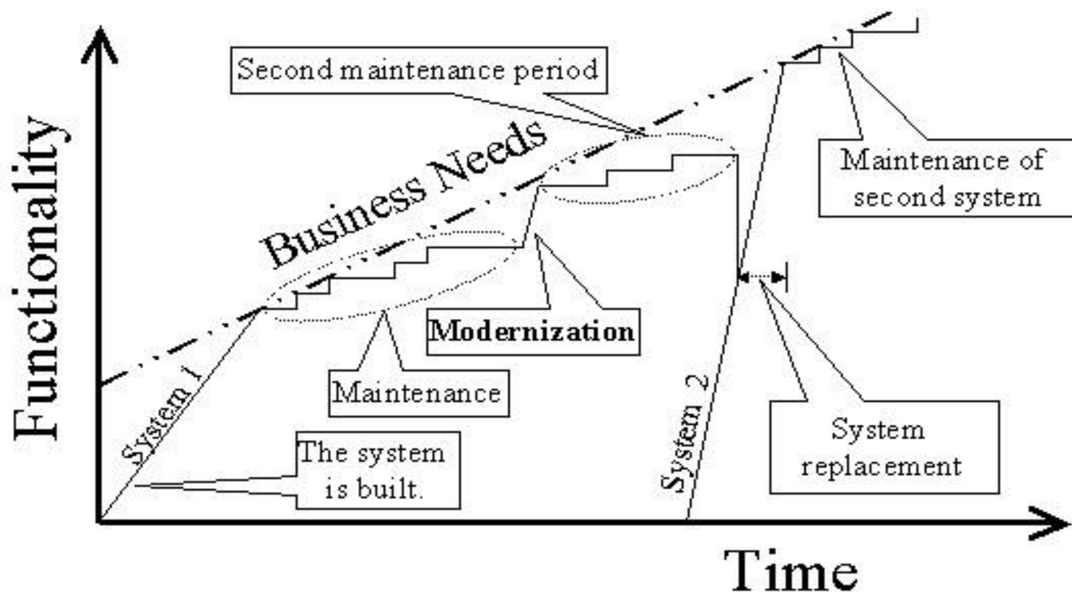

Figura 1: Ciclo de vida del sistema de software [13]

Un sistema heredado puede evolucionar de diferentes maneras, las actividades de evolución del sistema se pueden dividir en tres categorías: *mantenimiento*, *modernización* y *reemplazo* [13]. La evolución de sistemas heredados es un término amplio que abarca un proceso continuo desde la adición de un campo en una base de datos hasta la reimplementación completa de un sistema. La Figura 1 ilustra cómo se aplican diferentes actividades de evolución en diferentes fases del ciclo de vida del sistema de software. La línea punteada representa las crecientes necesidades comerciales, mientras que la línea continua representa la funcionalidad proporcionada por el sistema. El mantenimiento continuo del sistema soporta las necesidades del negocio, pero a medida que el sistema se vuelve cada vez más obsoleto, el mantenimiento deja atrás las necesidades del negocio. Se requiere un esfuerzo de modernización que represente un mayor esfuerzo, tanto en tiempo como en funcionalidad, que la actividad de mantenimiento. Finalmente, cuando el sistema anterior ya no puede evolucionar, debe ser reemplazado.

En la Figura 1 se muestra que en la primera fase se agregan nuevas funcionalidades al sistema para que pueda adaptarse a los nuevos requerimientos que van surgiendo. Esto

se va haciendo cada cierto tiempo, hasta que el sistema pierde el objetivo para el cual fue creado, llegando un momento en que se da al abandono al sistema y se empieza a construir uno totalmente nuevo.

Una vez se plantea el problema de sistema heredado, las opciones a las que se enfrenta un CIO[8] requieren de información suficiente que ayude a la tomar la decisión correcta acerca del futuro de dicho sistema ya que son de vital importancia y pueden tener un gran impacto para el futuro del negocio. Lo habitual es elaborar un informe de evaluación del sistema de información donde se describe la situación actual del sistema de información, los requisitos tanto técnicos como de negocio, cuál debe ser la situación final o deseada, así como las posibles alternativas existentes para lograr la transición a la nueva situación. Esta información facilita la determinación para la actuación necesaria acerca del sistema, que podría ser, reemplazarlo por otra solución o transformarlo mediante un proceso de modernización eficiente [14].

2.2 Análisis de las alternativas

Muchas organizaciones mantienen sistemas heredados los cuales guardan información valiosa y en crecimiento permanente. Esto hace que su acceso a nuevos modelos de negocio y aplicaciones empresariales de hoy en día sea cada vez más difícil. Es por ello surge la necesidad de modernizar sus sistemas heredados, apostando por tener presencia en el mercado actual y poder competir frente a otros nuevos con tecnología más reciente. Debido a la complejidad de estos sistemas y las particularidades de cada uno, aumenta la incertidumbre a la hora de tomar decisiones al respecto. El Arquitecto software o el CIO puede optar por mantener el sistema heredado buscando técnicas de ingeniería de software que prolonguen el tiempo de vida de los sistemas heredados o por alguna de las siguientes alternativas de transformación [15].

2.2.1 Mantenimiento

El mantenimiento es la modificación del software después de su lanzamiento sin afectan a su arquitectura, en el que los cambios tratan de resolver problemas o añadir mejoras funcionales. El mantenimiento es un proceso continuo e incremental, por lo que el costo de mantenimiento para el sistema aumenta. Cualquier sistema de software Se requiere mantenimiento para respaldar su evolución [16]. Una de las limitaciones es adoptar nuevas tecnologías. La ventaja competitiva derivada de la adopción de nuevas tecnologías se ve limitada porque las mejoras como la implementación de una arquitectura distribuida o una interfaz GUI no suelen considerarse actividades de mantenimiento. La otra limitación es modificar un sistema heredado para adaptarlo a los nuevos requisitos comerciales. Esto se vuelve cada vez más difícil, porque los pequeños cambios pueden tener un mayor impacto en los sistemas heredados en general.

---

8 Chief Information Officer (CIO): http://www.mintic.gov.co/gestionti/615/w3-propertyvalue-6205.html

El mantenimiento impacta en muchos atributos de calidad tales como disponibilidad, confiabilidad y facilidad de mantenimiento.

2.2.2 Reemplazo

Reemplazo significa construir o comprar por completo un nuevo sistema para reemplazar un sistema antiguo obsoleto y no extensible. El reemplazo es apropiado para sistemas heredados que no pueden mantener el ritmo de las necesidades del negocio y para los cuales la modernización no es posible o no es rentable [17]. La inversión en reemplazo es la mayor entre mantenimiento, modernización y reemplazo. Puede implicar el desarrollo de un sistema completamente nuevo y, por lo tanto, sigue las fases del ciclo de vida de desarrollo del sistema. Esto requiere la prueba completa de todos los módulos y unidades implementadas. El reemplazo también puede implicar la compra de un nuevo sistema que puede no cumplir con todos los requisitos comerciales de la organización y puede requerir personalización. Los sistemas heredados a menudo son sistemas hechos a medida y no es posible comprar un reemplazo, ya que es posible que el sistema no cumpla con todas las funcionalidades.

Una solución podría ser la aplicación de un sistema integrado y monolítico que reemplace los sistemas heredados, con procesos de negocio estándares para los diferentes requerimientos de la organización. Algunas de las soluciones monolíticas más famosas son la implementación de sistemas ERP[9].

2.2.3 Modernización

La modernización es la alternativa con más impacto en el sistema heredado. La modernización busca mantener las funcionalidades originales, sin alterar los procedimientos de negocio de la organización que los utiliza, de tal forma que se conserve la funcionalidad del sistema original a pesar de residir en una plataforma tecnológica diferente.

La modernización de sistemas heredados tiene como objetivo extender el valor de la inversión heredada a través de la migración a nuevas plataformas. El proceso de modernización significa mejorar de forma gradual y parcial el sistema heredado con alguna nueva técnica. La modernización implica cambios más extensos que el mantenimiento, pero conserva una porción significativa del sistema existente. Estos cambios a menudo incluyen la reestructuración del sistema, mejoras funcionales importantes o atributos de calidad mejorados, como la capacidad de mantenimiento. La modernización del software intenta evolucionar un sistema heredado, o elementos de un sistema, cuando las prácticas evolutivas convencionales, como el mantenimiento, ya no pueden lograr el resultado deseado [2].

---

9 ERP: sistemas de planificación de recursos empresariales (por sus siglas en inglés, enterprise resource planning) wikipedia.org

En muchas ocasiones los sistemas heredados se construyeron con métodos manuales, y es lo que las organizaciones de TI saben cómo hacer. Estos usualmente no salen bien, por varias razones, entre ellas las siguientes son las más importantes:[10].

- **Costo de realizar una migración manual**: El Grupo Gartner[11] observa que el costo de la conversión manual del código puede oscilar entre $ 6 y $ 26 por LOC (lines of code). Por lo que un millón de líneas de código cuesta $ 6 a $ 26 millones.
- **Duración del proyecto**: es muy difícil resistirse a agregar nuevas funciones y requisitos durante el proyecto. La tarea de migración se hace más larga.
- **Objetivos conflictivos**: Mientras el equipo de migración intenta construir un reemplazo, el sistema heredado aún debe continuar brindando soporte a la compañía. Debe evolucionar para satisfacer las necesidades corporativas y, por lo tanto, necesita cambios continuamente. Tales cambios son un problema para el equipo de migración.
- **Calidad de conversión desigual**: Una migración realizada manualmente depende de las habilidades individuales de los miembros del equipo. La calidad del código resultante variará en consecuencia, elevando los costos de mantenimiento para el sistema convertido.

Una tasa de conversión "semiautomática" entre 60%-80% de un sistema se considera buena, excepto proyecto de migración de bases de datos, una conversión automatizada al 100% es prácticamente imposible. El éxito es depende del código heredado y la metodología y herramientas de migración. Una migración automatizada o semiautomática resuelve los problemas anteriores de la siguiente manera:

- **Menor costo**: Una estimación de coste es de $ 1 y $ 5 por LOC, dependiendo de la complejidad y la cantidad de lenguajes involucrados.
- **Tiempo más corto**: estas migraciones ya que se evitan la mayoría de los problemas mencionado anteriormente.
- **No hay objetivos conflictivos**: El equipo de mantenimiento de software heredado puede agregar una nueva funcionalidad durante la migración, y la herramienta de migración la convertirá.
- **Limpieza y calidad de código**: las reglas de migración en efecto imponen un "estilo" consistente en todo el sistema. Se pueden ajustar para cambiar el estilo. Algunas reglas de migración se centran en convertir los conceptos del lenguaje con precisión. Otras se centran en la "limpieza" de construcciones incómodas, lo que garantiza un código limpio y legible.

Es conveniente que la migración se lleve a cabo utilizando estrategias de modernización incremental, como el caso de la *StranglerApplication*[12], donde la implementación de nuevos procesos de negocio del sistema original se lleva a cabo gradualmente en una nueva arquitectura y posteriormente se hace una transferencia gradual desde el sistema original hacia el nuevo sistema, estrangulando así el sistema heredado gradualmente.

---

10 Legacy Software Migration: http://www.semdesigns.com/Products/Services/LegacyMigration.html
11 Gartner Group study "Forecasting the Worldwide IT Services Industry: 1999"
12 StranglerFigApplication: https://www.martinfowler.com/bliki/StranglerApplication.html

Capítulo 3. Conceptos y Metodología de la modernización

La Modernización de sistemas se ha convertido en un área de investigación de interés debido al incremento de los costes de mantenimiento y las limitaciones respecto nuevas necesidades del negocio. Los métodos tradicionales de modernizar aplicaciones consistían en reescribir el código de aplicación existentes escrito en lenguajes de programación heredados a otros más modernos. En la actualidad, la modernización del software se puede llevar a cabo de manera más eficiente gracias a un conjunto de paradigmas de ingeniería del software. De acuerdo a [18], la *arquitectura dirigida por modelos* (MDA) permite modernizar los sistemas de información heredados, centrándose en todos los aspectos de su arquitectura mediante la definición y uso de metamodelos adecuados para obtener la arquitectura deseada partiendo de su arquitectura heredada y dadas las características de la modernización dirigida por la arquitectura (ADM)[13], cuyo enfoque es el de establecer un método automatizado de transformación frente a otras aproximaciones que resultan más invasivas para las organizaciones. La metodología propuesta en este trabajo combina este y otros paradigmas mencionados anteriormente con una estrategia de modernización incremental *StranglerApplication* que consiste en crear gradualmente un nuevo sistema al lado del sistema antiguo. Esto se lleva a cabo aplicando para cada submódulo del sistema un proceso de reingeniería "*El modelo de herradura* o *The HorseShoe Model*" que comprende una combinación de otros procesos tales como la ingeniería inversa, la reestructuración, la traducción y la ingeniería directa. Este proceso necesita una especificación independiente de la plataforma y lenguaje de programación, por lo que utilizaremos el metamodelo *Knowledge Discovery Metamodel* (KDM)[14], que define un formato de intercambio abstracto y una API sobre la que poder construir herramientas de nueva generación.

3.1 Arquitectura del software

Un proceso de modernización de un sistema heredado es similar a un proceso de desarrollo de software tradicional, con la peculiaridad de que opera bajo un marco de tiempo más largo, complejo y plagado de restricciones. Además de que utiliza la reingeniería como mecanismo de migración y mejora de un sistema heredado. Independientemente de la metodología que se utilice, es imprescindible que sea precedido por un desarrollo de la arquitectura de software.

---

13 Architecture-Driven Modernization (ADM): https://www.omg.org/adm/legacy/
14 The knowledge discovery metamodel specification: https://www.omg.org/spec/KDM/About-KDM/

De acuerdo con el *Software Engineering Institute* (SEI), la Arquitectura de Software se refiere a “las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos” [1, 18].

La Arquitectura de Software es el diseño de más alto nivel de la estructura de un sistema ocultando los detalles de implementación, este diseño es de especial importancia ya que la manera en que se estructura un sistema tiene un impacto directo sobre la capacidad de este para satisfacer los atributos de calidad de calidad del sistema. Debería mostrar una vista estructural con los componentes principales, el comportamiento de estos componentes, su interacción y coordinación definen la misión del sistema.

Mediante la Arquitectura de Software se define una solución estructurada que satisfaga los atributos de calidad del sistema, son requisitos no funcionales como es la disponibilidad, el rendimiento, la escalabilidad, usabilidad, la modificabilidad, etc. Es importante recordar que la arquitectura de software resuelve los requisitos no funcionales del sistema, mientras los requisitos funcionales se resuelven en la fase posterior de modelado y diseño del sistema.

3.2 Domain Driven Design (DDD)

El Diseño guiado por el dominio en inglés: domain-driven design (DDD)[20] , es un enfoque de desarrollo que provee una estructura de prácticas y reglas de implementación. Estas reglas ayudan la toma de decisiones para el diseño de arquitecturas software con necesidades complejas.

El *DDD* trata de mapear la idea de negocio “*Dominio*” en una capa de la arquitectura, esta capa será la responsable de representar la lógica del negocio. Se trata de generar una abstracción a través de un modelo que representa toda la lógica del negocio en el “core” de la aplicación, pero dividida por contextos.

Entre las ventajas de utilizar esta técnica:

- Comunicación efectiva entre expertos del dominio y desarrolladores.
- Desarrollo de un área dividida del dominio (subdominio)
- El software es más cercano al dominio, y por lo tanto es más cercano al cliente.
- Código bien organizado basado en subdominio, permitiendo el testing de los distintos módulos por separado.
- Lógica de negocio reside en un solo lugar.
- Mantenibilidad a largo plazo.

3.3 Desarrollo Dirigido por Modelos (MDD)

El Desarrollo de Dirigido por Modelos (Model Driven Development o MDD) ofrecer una representación de sistemas de información mediante la generación de modelos que son más abstractos que los sistemas procesados [21].

Un modelo es el objeto central del paradigma MDD, las siguientes definiciones nos pueden acercar más a su definición:

> *"un Modelo es el diseño estructural de un sistema complejo" [21].*
>
> *"Un modelo es un conjunto coherente de elementos que cubren ciertos aspectos de diseño, están restringidos a cierto nivel de abstracción, no se necesitan exponer todos los detalles ni necesita ser completo por sí mismo" [22].*

La importancia de los modelos nunca debería superaran en complejidad ni dimensión el sistema de información al que representan, con lo que facilitaran su manipulación y comprensión. Estos modelos serán el eje que permitirá las transformaciones necesarias para conseguir una implementación que implícitamente debe ser semiautomática en el Desarrollo de Software Dirigido por Modelos.

3.4 Arquitectura dirigida por modelos (MDA)

El Desarrollo de Software Dirigido por Modelos necesita de una serie de reglas. Estas están especificadas en la arquitectura dirigida por modelos, que las define como estándar el Object Management Group con lo que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos [23].

En esta especificación hay 3 tipos de modelos en función del nivel de abstracción cómo se puede ver en Figura 2:

- CIM (Computation Independent Model): Este modelo está al máximo nivel de abstracción donde se consideran detalles específicos de negocio, proporciona una visión del sistema a un alto nivel de abstracción y es independiente del punto de vista de la computación. Se ocupa del modelado de negocio y de los requisitos [30].
- PIM (Platform Independent Model): Este modelo ofrece una visión del sistema a un nivel de abstracción intermedio. Esconde los detalles de implementación del siguiente nivel más abajo. Se centra en el análisis y diseño, y es independiente de la plataforma tecnológica.
- PSM (Platform Specific Model): Este modelo muestra una visión del sistema desde un bajo nivel de abstracción. Trata el diseño dependiente de la plataforma tecnológica. El modelo será adaptado para una implementación tecnológica específica.

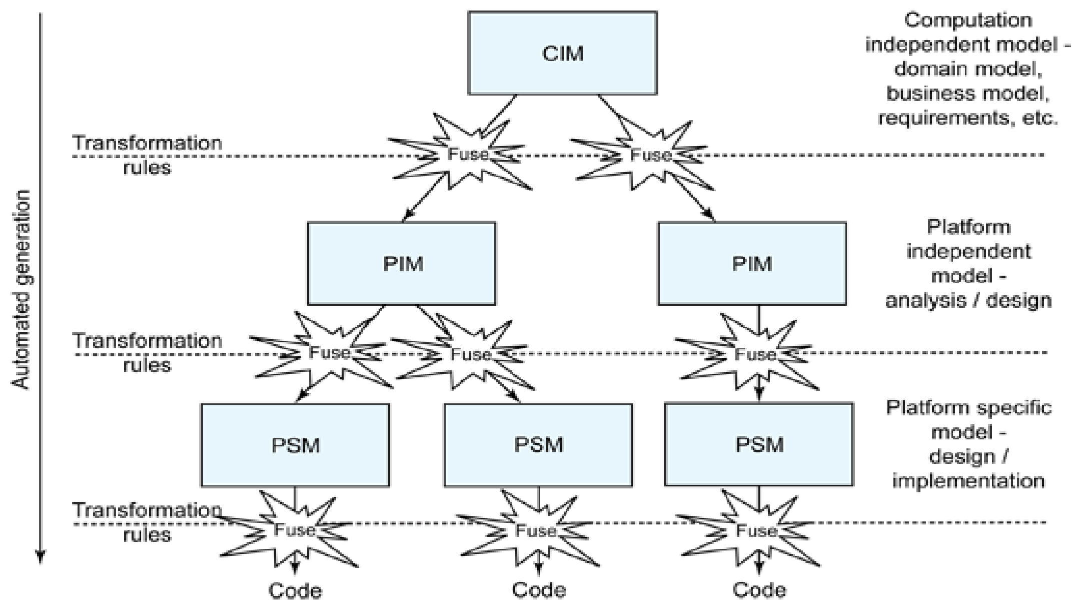

Figura 2 : Modelos MDA según nivel de abstracción[15]

En el contexto de modernización, aplicar este paradigma será clave para la aplicación de procesos de reingeniería mediante el modelo de herradura que veremos en los siguientes apartados.

3.1 Modernización dirigida por la arquitectura (ADM)

La Modernización Dirigida por la Arquitectura (Architecture-Driven Modernization, ADM) es una iniciativa de la OMG (Object Management Group) en relación con la construcción y la promoción de normas que pueden aplicarse para modernizar los sistemas heredado, basándose en el paradigma del Desarrollo Dirigido por Modelos (MDD), busca proponer y estandarizar técnicas, métodos y herramientas para la automatización de procesos de modernización. El objetivo de esta iniciativa es ofrecer un conjunto de estándares, que además de modernización de sistemas heredados ayudan y a definir tareas como el análisis de código y la comprensión, y la transformación de los modelos que representan los sistemas heredados (OMG, 2006).

Este modelo sigue el mismo enfoque mencionado previamente, más concretamente en el uso de transformaciones entre distintos modelos según el nivel de abstracción para obtener una arquitectura deseada en función del sistema de información heredado. Una combinación de este modelo con el modelo en herradura se puede conseguir que convivan las fases de ingeniería inversa, restructuración e ingeniería directa con los tres tipos de modelos de la arquitectura dirigida por modelos (PSM, PIM y CIM).

15 Opinions on Model-driven architecture: http://www.writeopinions.com/model-driven-architecture

3.2 Reingeniería

La reingeniería consiste en la revisión, análisis y modificación de un sistema de software existente para reconstituirlo de una nueva forma [24]. El proceso de típicamente comprende una combinación de otros procesos tales como la ingeniería inversa, la reestructuración, la traducción y la ingeniería directa. El objetivo es entender el software existente (especificación, diseño, implementación) y luego volver a ponerlo en práctica parta mejorar la funcionalidad, el rendimiento o la implementación del sistema. Además, trata de mantener la funcionalidad existente y dejar todo preparado para las funcionalidades que se puedan añadir en un futuro [25].

La reingeniería implica que un sistema heredado sea sometido a las etapas de ingeniería inversa, restructuración e ingeniería directa [24]. En los procesos de reingeniería de software se aplica generalmente el denominado modelo de herradura [26], un marco de trabajo para integrar los diferentes niveles de abstracción y las actividades que caracterizan este tipo de procesos. Mediante el modelo de herradura buscamos obtener una representación abstracta del sistema origen que nos facilite su comprensión y transformación para obtener un sistema destino. Aplicar el modelo de herradura se pueden identificar transformaciones verticales de diferente nivel de abstracción y transformaciones horizontales en el mismo nivel de abstracción.

3.3 Modelo en herradura

El Modelo de Herradura (*The HorseShoe Model*) [26], ofrece un marco de trabajo para la aplicación de procesos de reingeniería. A pesar de que el modelo de herradura cuenta con un importante bagaje dentro de la comunidad científica y el respaldo de numerosos proveedores de herramientas de modernización, no existen técnicas contrastadas para su automatización ni herramientas que den un soporte integral al modelo [27].

Este modelo refleja los 3 niveles de abstracción de una manera muy sencilla, ingeniería inversa (Parte I.3.3.3), restructuración (Parte I.3.3.2) e ingeniería directa (Parte I.3.3.3).

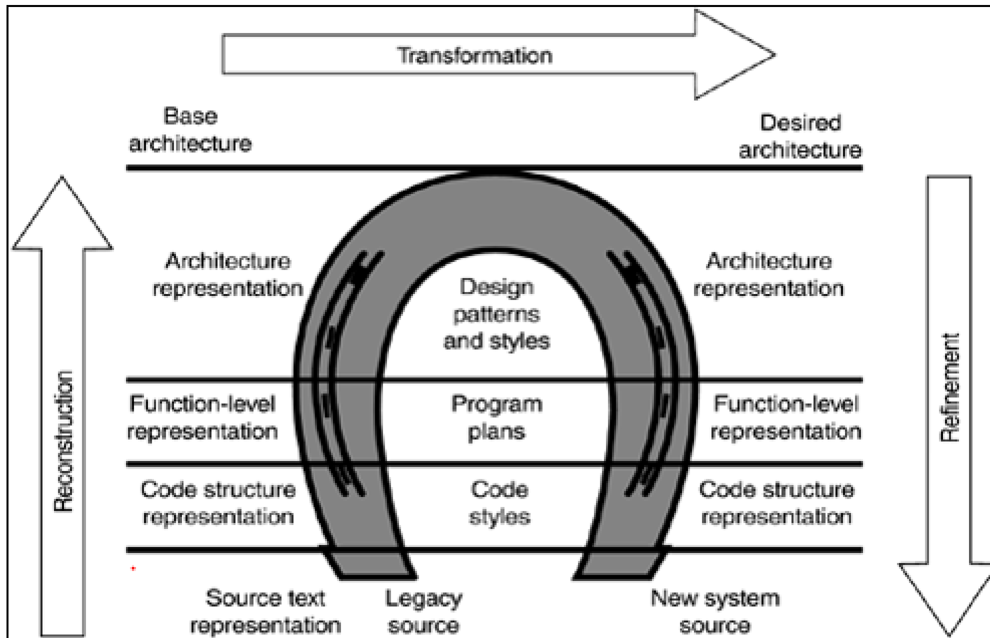

Figura 3: Modelo en herradura o HorseShoe Model [26]

La Figura 3, muestra una visión general del modelo en la que podemos observar cómo se alcanza el máximo nivel de abstracción en la primera fase de ingeniería inversa, donde se aplican transformaciones verticales que recuperan las decisiones de diseño tomadas en cada nivel de abstracción. En la segunda fase de reestructuración, se mantiene el nivel de abstracción y es donde se aplican transformaciones horizontales para adaptar o migrar los artefactos existentes al nuevo sistema y es en la tercera fase de ingeniería directa, donde normalmente se aplica un desarrollo basado en la arquitectura.

Es importante observar que la definición del nuevo sistema implica refinar los artefactos de más alto nivel (para añadir nuevas características), pero también integrar los artefactos existentes de bajo nivel a través de transformaciones horizontales, que de alguna forma deben ser informadas por los modelos de nivel superior. Sin embargo, el modelo de herradura puede no estar interpretado correctamente, como se muestra en la Figura 4.b. En este caso las transformaciones horizontales de la fase de reestructuración se aplican solamente a los artefactos de más alto nivel, y los artefactos resultantes son utilizados para generar el sistema mediante ingeniería directa. Esta interpretación es aplicada aplicado en trabajos como [17, 18]. Sin embargo. "el modelo de herradura directo" puede que en algunos casos de reingeniería no resultar factible. El problema principal es la pérdida de información que se produce en las transformaciones verticales

(al abstraer al nivel superior), que imposibilita "bajar por la derecho de la herradura" mediante transformaciones automáticas [27].

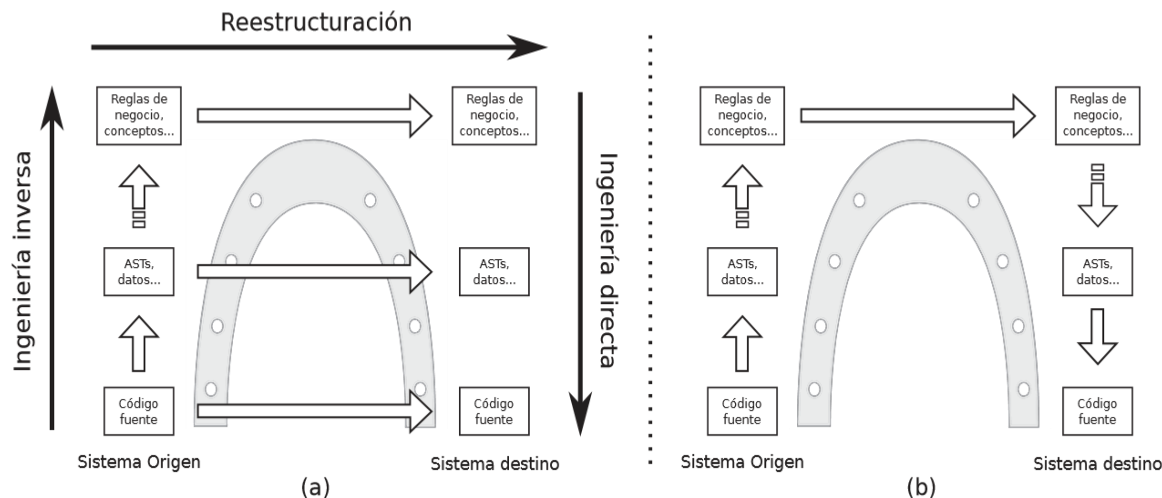

Figura 4:(a) Modelo original y (b) sin transformaciones horizontales [27]

Para aplicar el modelo de herradura en un contexto de Modenización dirigida por arquitectura que se muestra en la Figura 5, existen los tres tipos de modelos definidos anteriormente en el apartado(Parte I.3.4).

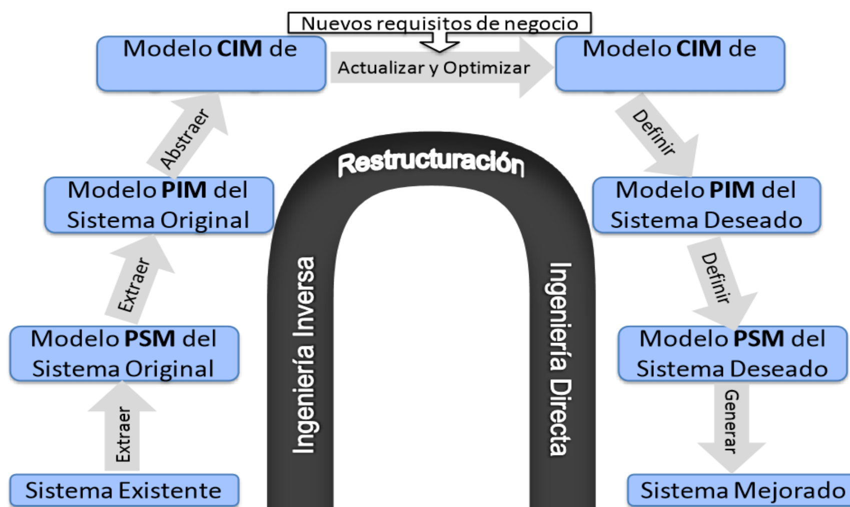

Figura 5: Modelo de Modernización en Herradura de ADM [31]

Este modelo incluye además *Knowledge Discovery Metamodel* (KDM), un metamodelo y estándar del OMG para modelar los artefactos de un *Legacy Information Systems* (LIS) [32], en un proceso de modernización. Al contrario que UML (*Unified Modeling Language*)[16] un metamodelo genera el código de forma descendente (Top-Down), KDM comienza desde el código heredado y construye un modelo abstracto [31].

A continuación, se detallan los 3 niveles de abstracción de este modelo.

16 About the unified modeling language specificationhttps://www.omg.org/spec/UML/About-UML/

3.3.1 Ingeniería Inversa

Está técnica se denomina así porque avanza en dirección opuesta a las tareas habituales de ingeniería, que consisten en utilizar datos técnicos para elaborar un producto determinado. Es la primera de las tres etapas anteriormente mencionadas y se basa en obtener un conjunto de especificaciones a partir del sistema heredado que formen la base para construir una nueva implementación del sistema software. El resultado de este método es una nueva representación del sistema a un nivel de abstracción mayor, reduce la complejidad del sistema a cambio de cierta pérdida de información [24].

3.3.2 Restructuración

En esta segunda etapa se toma como entrada las representaciones del sistema heredado obtenidas como resultado en la etapa anterior y se realizan transformaciones al mismo nivel de abstracción para obtener nuevas representaciones que mejoran de algún modo las propiedades del sistema. Estas transformaciones deben mantener el comportamiento externo del sistema de origen, con ello se consigue preservar la lógica y las reglas de negocio embebidas en el Sistema [24] .

3.3.3 Ingeniería directa

Esta es la tercera y última etapa, que al igual que la etapa anterior, toma como entrada el resultado de la etapa previa. En este caso, el objetivo es utilizar los modelos de representación de la etapa previa para alterar o reconstruir el sistema con la finalidad de mejorar su calidad global. En la mayoría de los casos el software sometido a reingeniería vuelve a implementar la función del sistema existente y también añade nuevas funciones o mejoras [24].

3.3.4 Metamodelo KDM

El metamodelo Knowledge Discovery Metamodel o KDM, nombrado anteriormente. Es una especificación del Object Management Group diseñada para ser la base de la iniciativa de la modernización dirigida por la arquitectura del OMG [33].

KDM provee una representación común intermedia para los sistemas software existentes y sus entornos operativos. Es una representación independiente de la plataforma y lenguaje de programación. KDM es un metamodelo *Meta Object Facility* (MOF) [33], que define un formato de intercambio de archivos en formato XMI (*XML Metadata Interchange*) entre herramientas que trabajan con software existente, y define una API sobre la que poder construir herramientas de nueva generación de modernización y aseguramiento del software (Force, 2007).

El propósito principal por el que apareció este metamodelo era proporcionar una amplia vista de la estructura de la aplicación y los datos, pero sin llegar a representar ningún tipo de software [34]. De forma que puede describir la estructura física y lógica de los sistemas heredados representando artefactos de software heredado como entidades, relaciones y atributos, y además soporta una amplia variedad de plataformas y lenguajes.

La Figura 6 muestra la arquitectura del metamodelo KDM, podemos ver que se divide en 12 paquetes cada uno encargado de modelar una vista de la arquitectura de un LIS. distribuidos en cuatro capas a diferente nivel de abstracción:

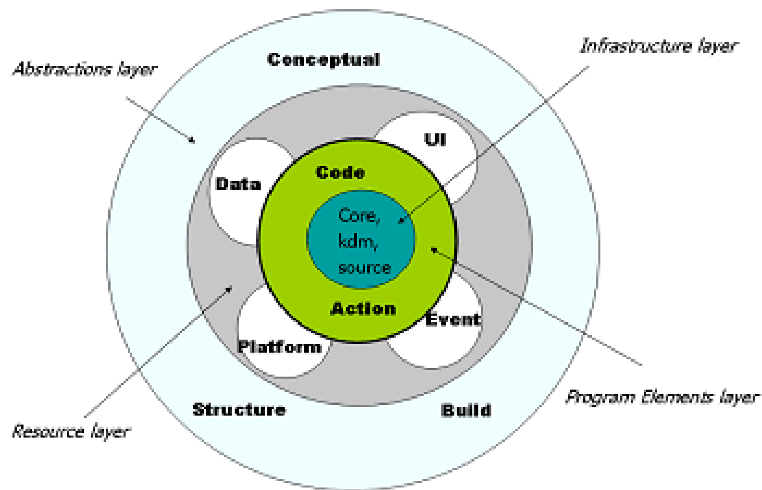

Figura 6: Arquitectura KDM[17]

A continuación, se describen los paquetes de cada capa [35]:

1. **Capa de Infraestructura**: define un pequeño conjunto de conceptos usados sistemáticamente a través de toda la especificación KDM. La forman los paquetes Core, **KDM** y **Source**, que proveen un pequeño núcleo común para los demás paquetes, el inventario de modelos de artefactos de los sistemas existentes entre los elementos del metamodelo en forma de enlaces al código fuente de los artefactos.
    - **Core:** Determina varios de los patrones que son utilizados por otros paquetes KDM.
    - **KDM:** Está alineado con otra especificación del OMG, llamada Common Warehouse Metamodel (CWM)[18]
    - **Source:** Proporciona el conjunto de artefactos físicos del sistema heredados y posibilita referenciar partes del código.
2. **Capa de Elementos de Programa**: provee una representación intermedia independiente del lenguaje para varios constructores comunes a varios lenguajes de programación. Dispone de 2 paquetes:
    - **Code**: representa los elementos de programación determinados por los lenguajes de programación, tipos de datos, procedimientos, clases, métodos, variables, etc. Este paquete provee un nivel menos abstracto.
    - **Action**: captura los elementos de comportamiento de bajo nivel de las aplicaciones, incluyendo los flujos de control.

17 Kdmanalytics: http://kdmanalytics.com/resources/standards/kdm/technical-overview/kdm-1-0-annotated-reference/specification-overview/
18 About the common warehouse metamodel specification https://www.omg.org/spec/CWM/About-CWM/

3. **Capa de Recursos:** permite representar conocimiento sobre sistemas heredados y su entorno operativo de los sistemas software existentes. Dispone de 4 paquetes:
   - **Platform**: representa el entorno operativo del software, relacionado con el sistema operante. incluyendo el flujo de control entre components.
   - **UI**: representa el conocimiento relacionado con las interfaces de usuario de los sistemas.
   - **Event**: representa el conocimiento relacionado con los eventos y comportamiento de las transiciones de estado de los sistemas software existentes.
   - **Data**: representa los artefactos relacionados con los datos persistentes, tales como bases de datos.
4. **Capa de Abstracción**: permite representar el conocimiento específico de dominio y aplicación a la vez que da una visión de negocio de los sistemas. La forman los paquetes:
   - **Conceptual**: representa las reglas de negocio, en la medida en que esta información pueda ser extraída de aplicaciones existentes. Estos paquetes están alineados con otra especificación del OMG, llamada SBVR (Semantics of Business Vocabulary and Business Rules).
   - **Structure:** describe los elementos del metamodelo para representar la organización lógica del sistema software en subsistemas, capas y componentes.
   - **Build:** representa los artefactos finales relativos al LIS. [35].

3.4 Modernización incremental (StranglerApplication)

En un artículo de 2004, publicado en su sitio web[19], Martin Fowler habla del patrón de la *StranglerApplication "crear gradualmente un nuevo sistema al lado del sistema antiguo, dejándolo crecer lentamente hasta que el viejo sistema sea estrangulado"*.

La idea de este patrón es usar la estructura de una aplicación web para dividir una aplicación en diferentes dominios funcionales y reemplazar esos dominios con una nueva implementación basada en SOA o microservicios de un dominio a la vez. Esto crea dos aplicaciones separadas que conviven una al lado de la otra en el mismo espacio de URI *"las aplicaciones web se construyen a partir de URI individuales que se mapean funcionalmente a diferentes aspectos de un dominio de negocio"*.

Con el tiempo, la nueva aplicación refactorizada "estrangula" o reemplaza la aplicación original hasta que finalmente se puede cerrar la aplicación monolítica. Por lo tanto, los pasos de la aplicación de estrangulamiento son transformar, coexistir y eliminar:

1. **Transformar**: crear un nuevo sistema paralelo.
2. **Coexistir**: Dejar el sistema existente donde está por un tiempo. Redirigir del sistema existente al nuevo para que la funcionalidad se implemente de forma incremental.
3. **Eliminar**: Eliminar o dejar de mantener la funcionalidad del sistema original a medida que el tráfico se redirecciona desde esa parte del sistema original.

19 StranglerFigApplication[Martin Fowler]: https://www.martinfowler.com/bliki/StranglerApplication.html

La aplicación de este patrón crea un valor incremental en un marco de tiempo mucho más rápido que si probara una migración en la que actualice todo el código de la aplicación "big bang". También brinda un enfoque gradual para la adopción de microservicios a arquitecturas basadas en SOA.

Los pasos para una migración de una aplicación en el marco de modernización de un sistema heredado [36]:

- **Construcción de una capa de abstracción:** lo primero que habría que hacer es crear una capa de abstracción. Expone una serie de API a través de las cuales accede a la funcionalidad disponible dentro de la aplicación del sistema heredado.
- **Preparación**: habría que detener cualquier desarrollo en el sistema heredado, se trata de aislar el monolito y solo acceder a él a través de las API. Cualquier nuevo desarrollo se realiza fuera del monolito e interactúa con él a través de la capa de abstracción. Se puede Construir nuevas funcionalidades usando los principios de SOA, o microservicios, las únicas interacciones con el mundo exterior son a través de API definidas.
- **Reemplazar la funcionalidad existente**: Desarrollar la funcionalidad que se necesita cambiar fuera del sistema heredados, se transformará en un componente, un servicio o microservicio. Se implementan nuevos requisitos del negocio si son necesarios. Es posible que se necesite cambiar algún código dentro del monolito para redirigir la llamada a esta parte de la funcionalidad [36].

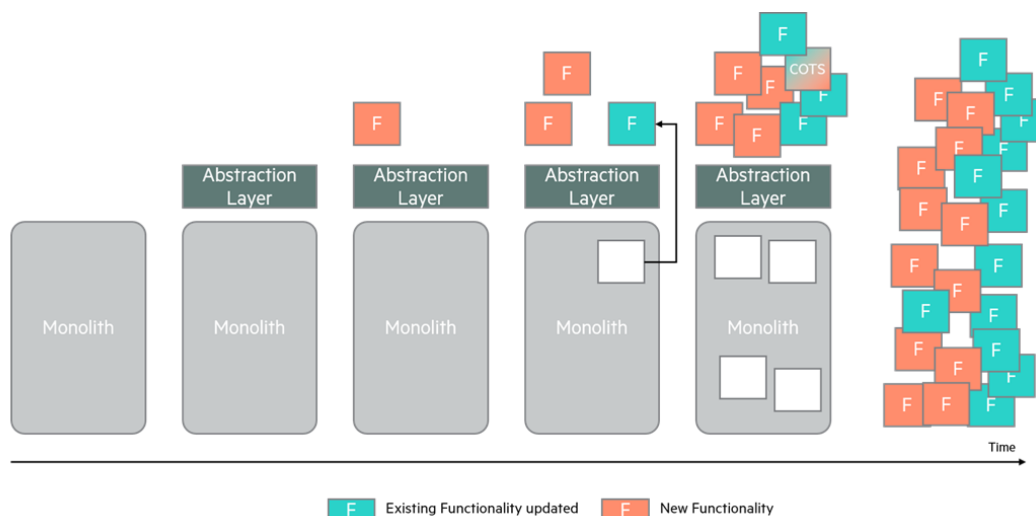

Figura 7: StranglerApplication[36]

De esta forma, se va reduciendo el monolito de alguna manera. Con el tiempo, se puede reemplazar los componentes uno a uno hasta una migración total del sistema.

Capítulo 4.
Proceso de Modernización

Teniendo en cuenta los conceptos y metodología de la modernización definidos en Capítulo 3, en el presente capítulo se detallan las actividades del plan de la migración de un módulo del sistema. En la Figura 8 se puede observar el diagrama de actividades del proceso.

Hay que tener en cuenta de que el plan de modernización se aplica al componente del sistema seleccionado para la migración y no al sistema completo. cada ciclo termina cuando se ha logrado la migración del componente hacia la nueva arquitectura. Se trata de una modernización incremental, donde el proceso de migración que se describe a continuación se repite para cada submódulo del sistema que migra desde la antigua arquitectura hacía la arquitectura de destino y esto se puede repetir tantas veces hasta migrar el sistema por completo de forma incremental.

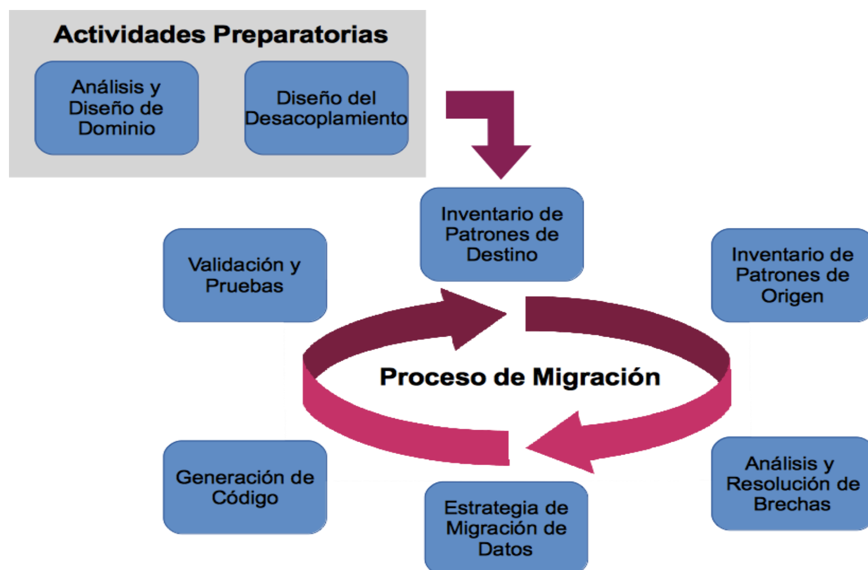

Figura 8: Esquema del proceso de migración[20]

4.1 Actividades preparatorias

Las actividades preparatorias no forman parte del modelo de herradura que se aplica en cada ciclo de la modernización incremental. Es una tarea que solo se realiza una vez al principio del proyecto para identificar los conceptos más relevantes del dominio, sus

[20] Plataforma de Soporte a la Modernización: https://www.opencanarias.com/productos/plataforma-de-soporte-a-la-modernizacion-de-sistemas-heredados-basados-en-cobol-cics-db2/

elementos claves, sus relaciones y también para detectar los puntos de acoplamiento del sistema. Estas actividades se describen a continuación.

4.1.1 Análisis y Diseño de Dominio

En este apartado se pretende dar una visión funcional del servicio a migrar, así como describir un modelo de dominio que sirva de elemento pivote entre el sistema origen y el sistema destino. Siguiendo las buenas prácticas de DDD (Domain Driven Design), el modelo de dominio debe surgir de las reglas y requisitos de negocio y ser abstracto de una implementación concreta. Por lo cual, para este análisis se ha de tener en cuenta únicamente las entrevistas a los expertos de negocio del cliente.

El Análisis y Diseño de Dominio integra un conjunto de tareas dirigidas a la obtención de un lenguaje que represente los conceptos del dominio de forma que pueda ser compartido y aceptado por todos los involucrados del dominio. Asimismo, permite dotar a esos conceptos de la granularidad adecuada para abordar su complejidad. En la modernización de sistemas heredados esta actividad resulta esencial, dado que a partir del simple análisis estático del código resulta imposible determinar su intencionalidad. En esta actividad se incluye también la selección del servicio a migrar.

Puede que en un primer momento el modelo de dominio sea incompleto, pero lo suficientemente ilustrativo para reflejar los conceptos más relevantes del dominio. Esto es debido a que el acceso a los expertos de negocio no suele ser tan fácil. Durante la ejecución del proyecto, y a medida que se vayan descubriendo nuevas entidades y reglas de negocio, se irá refinando dicho modelo.

4.1.2 Diseño del desacoplamiento

Con esta actividad establecemos en primer lugar un conjunto óptimo de programas a migrar, preservando la cohesión del servicio y minimizando el acoplamiento. Dependiendo de la dimensión del problema, se podrán aplicar técnicas de análisis estático de código. Estas mismas técnicas permiten lograr la detección de los puntos de acoplamiento y su tipología. Finalmente, con esta información se definirá la estrategia de integración del código extraído, así como la fachada o adaptador sobre la que se establecerá el desacoplamiento. Esta capa traduce las solicitudes que un subsistema hace al otro subsistema (patrón *AnticorruptionLayer*[21]). Este patrón asegura de que el diseño de la aplicación no se vea limitado por dependencias de subsistemas.

Al final de esta fase debemos tener el programa o el conjunto de programas a migrar y un inventario de bloques de código que accederán al servicio migrado.

[21] Patrón Anti-Corruption Layer: https://docs.microsoft.com/es-es/azure/architecture/patterns/anti-corruption-layer

4.2 Inventario de patrones de origen (Idioms Precursores)

Se trata de alinear estructuras de código origen con nuestro metamodelo de idioms, es decir identificar patrones en el código para establecer precursores de transformaciones automatizadas de código.

La calidad de los patrones detectados se verifica contra toda la base de código, pero también se mide el impacto de los patrones detectados en la base de código a migrar.

Los objetivos de esta fase son los siguientes:

- Entender la arquitectura de destino para maximizar la generación automática de código.
- Identificar patrones en el código (Idioms Precursores).
- Cuantificación del impacto de dichos patrones en el código a migrar.

4.3 Inventario de patrones de destino (*Idioms* soluciones)

Se trata de identificar en la arquitectura de destino la soluciones a los patrones identificados anteriormente. Los patrones se validarán contra toda la base de código de destino con el fin de garantizar la calidad y mantenibilidad de las soluciones aportadas.

Los objetivos de esta fase son los siguientes:

- Entender la arquitectura de destino para maximizar la generación automática de código
- Identificar patrones en el código (*Idioms* soluciones).
- Finalmente se diseña una plantilla del patrón. sobre la que se generará el código en actividades posteriores.

4.4 Análisis y resolución de brechas

Con esta actividad se resuelven las brechas entre los patrones origen (precursores) y los patrones destino (soluciones). Se parte de un mapa que relaciona dichos patrones y su correspondiente cardinalidad (un precursor puede tener más de una solución, y una solución puede aplicarse a más de un precursor), y se diseñan los algoritmos de transformación. Finalmente se codifican dichas transformaciones y se validan los resultados. Los objetivos de esta fase son los siguientes:

- Establecer el mapa entre los patrones origen (precursores) y los patrones destino (soluciones)
- Analizar las brechas entre los patrones y definir los algoritmos de transformación que las resuelvan
- Implantar las transformaciones de brechas.

4.5 Estrategia de migración de datos

Dependiendo de las diferencias de paradigma entre la persistencia origen y destino, con esta actividad se analizan las brechas entre ambas (tipos de datos, disparadores, procedimientos almacenados). Seguidamente se implementa la solución en la persistencia de destino, y se desarrollan y prueban los programas de carga de datos (ETLs) para la migración.

Los objetivos de esta fase son los siguientes:

- Definición del esquema de persistencia destino
- Análisis de brechas entre persistencia origen y destino

4.6 Generación de código

En este punto se lanzan los generadores de código para todas las brechas a resolver. Se generan los programas fuentes nuevos, y quedan modificados los programas a refactorizar en el contexto actual. En el caso de brechas que por sus características no se han podido generar, o que requieren de una modificación manual, se procede a su codificación. Finalmente se realiza la validación preliminar y se verifica que el código por lo menos no tiene errores de compilación. Los objetivos de esta fase son los siguientes:

- Obtener el código generado
- Realización de las modificaciones y/o adiciones al código generado
- Verificación preliminar del código

4.7 Validación y Pruebas

Durante esta actividad se procede a la incorporación del código generado en la infraestructura de integración continua del cliente, así como a la migración de datos. Paralelamente, se establecen los conjuntos de pruebas automatizados en el contexto de las políticas del cliente y los objetivos a alcanzar en el proyecto de modernización. Los objetivos de esta fase son los siguientes:

- Automatizar el proceso de validación y pruebas
- Cumplir con las políticas de calidad del cliente
- Migración de los datos
- Aseguramiento del producto de software

Capítulo 5.
Caso de estudio

Este caso de estudio es realizado en OPEN CANARIAS, S.L. Partner de IBM. Es una empresa canaria fundada en noviembre de 1996. Es una compañía de implantación nacional especializada en productos, servicios informáticos y proyectos tecnológicos que oferta tanto a empresas del sector privado como a organismos del sector público.[22]

5.1 Definición del caso de estudio

Se trata de una propuesta de la compaña a una compañía eléctrica para estudiar la viabilidad de la modernización semiautomática de su sistema, como piloto y marco de referencia para la migración del resto de servicios existentes en la plataforma.

5.1.1 Metodología

Dadas las características del proyecto, cuyo enfoque es el de establecer un método automatizado de transformación de un sistema heredado, partimos de la metodología de modernización descrita en el apartado anterior (ver Figura 8), con algunos matices relacionados con la naturaleza del proyecto y la utilización de aplicaciones de OPEN CANARIAS, S.L para la automatización de la modernización. Dicha metodología combina el Diseño Dirigido por el Dominio (Domain-Driven Design), o DDD, con estrategias de modernización incremental, que ha demostrado su idoneidad en la transformación de sistemas heredados críticos, y que ya ha venido aplicando el cliente en cierta medida. En la Figura 1 se muestra el esquema del proceso.

5.1.2 Objetivo del caso de estudio

El objetivo de este caso de estudio consiste inicialmente en la migración de un submódulo del sistema COBOL/CICS/DB2 a Java JEE, asegurando un valor continuo del negocio y reduciendo el coste de la transición en comparación con una migración manual.

Para la consecución del objetivo general se plantean los siguientes objetivos específicos:

- Establecer un proceso semiautomático que facilite la migración incremental de las aplicaciones del cliente en la plataforma COBOL/CICS/DB2 a Java JEE, en función a los objetivos y los recursos puestos a disposición por OPEN CANARIAS, S.L.

[22] La documentación utilizada para dar apoyo a este trabajo es propiedad de Open Canarias.

- Abordar la migración de un servicio a través de dicho proceso para validarlo. Para ello se desarrolla un conjunto de actividades preparatorias que permitan abordar la migración del servicio, y que incluyen la selección de dicho servicio.
- Conseguir una reducción de coste y tiempo por la automatización parcial de la migración frente a una migración manual.

Tener en cuenta que el principal objetivo de la migración incremental es realizar una modernización de aplicaciones del cliente sin interrumpir su funcionamiento, pero también nos permite comprobar la factibilidad y retroalimentación de la solución planteada.

5.1.3 Supuestos y restricciones

En la elaboración de la propuesta partimos de los siguientes supuestos:

1. Existe una arquitectura de destino validada y operativa.
2. Existen interfaces de acceso de la plataforma Java a Cobol.
3. No existen interfaces de acceso de la plataforma Cobol a Java.
4. Existe documentación tanto de la arquitectura de origen como de destino.
5. Se puede acceder de forma remota a los entornos de desarrollo y pruebas (pre) tanto de la plataforma Cobol como Java

En cuanto a las restricciones, se establecen las siguientes:

1. La base de datos de destino debe ser Oracle.
2. El código generado debe asimilarse a la arquitectura Java actual del Cliente.

5.1.4 Migración automatizada

OPEN CANARIAS, S.L. dispone de la herramienta "OC2-ADA" de migración automática. Esta herramienta ha sido desarrollada siguiendo la especificación *Architecture Driven Modernization (ADM)* del Object Management Group (OMG). Según el OMG, Architecture-Driven Modernization es el proceso de comprensión y evolución de activos de software existentes con el fin de mejorar el software, modificaciones, interoperabilidad, refactorización, reestructuración, reutilización, migración, traducción de idiomas, integración de aplicaciones empresariales, SOA y migración de MDA.

La herramienta "OC2-ADA" ha sido desarrollado un un stack de tecnologías (denominado CAKES) que incluye, entre otros el framework Akka, el lenguaje Scala y el sistema de ficheros HDFS, lo que garantiza la escalabilidad y rendimiento de la solución, siendo capaz de procesar de forma eficiente cualquier tamaño de sistema origen.

Figura 9:CAKES

5.2 Actividades preparatorias

5.2.1 Análisis y Diseño de Dominio

El dominio de aplicación del servicio que se requiere migrar “Recorridos" está asociado a lo que se denominan “rutas de lecturas”. Los lectores (personas que leen los contadores o equipos de lectura) deben pasar por una serie de ubicaciones donde están los contadores para realizar la lectura de los mismos.

Una ruta es lo que define el conjunto de suministros (contadores) que tienen que verse cada periodo, de manera que al final del periodo deben de haberse visto todos los contadores para volver a empezar con la ruta. Las rutas se dividen en fragmentos más pequeños, denominados recorridos, para definir el trabajo a realizar cada día, teniendo tablas diferentes para definir los recorridos y las fechas en las que hay que hacerlos.

Un itinerario define lo que hay que leer un determinado día. El itinerario podría definirse como el plan diario de trabajo (carga de trabajo), de una persona y un día, conteniendo un conjunto de recorridos consecutivos. Para definir el conjunto de recorridos de un itinerario se especifica el recorrido inicial y el recorrido final.

Por otro lado, un recorrido solo pertenece a un itinerario por cada ciclo de lectura de ese periodo. Por ejemplo, suponiendo un ciclo de lectura bimestral, el mismo recorrido se realizará seis veces en un año, y por lo tanto estará asociado con seis itinerarios diferentes.

El siguiente diagrama muestra gráficamente los conceptos de ruta, recorrido e itinerario descritos anteriormente.

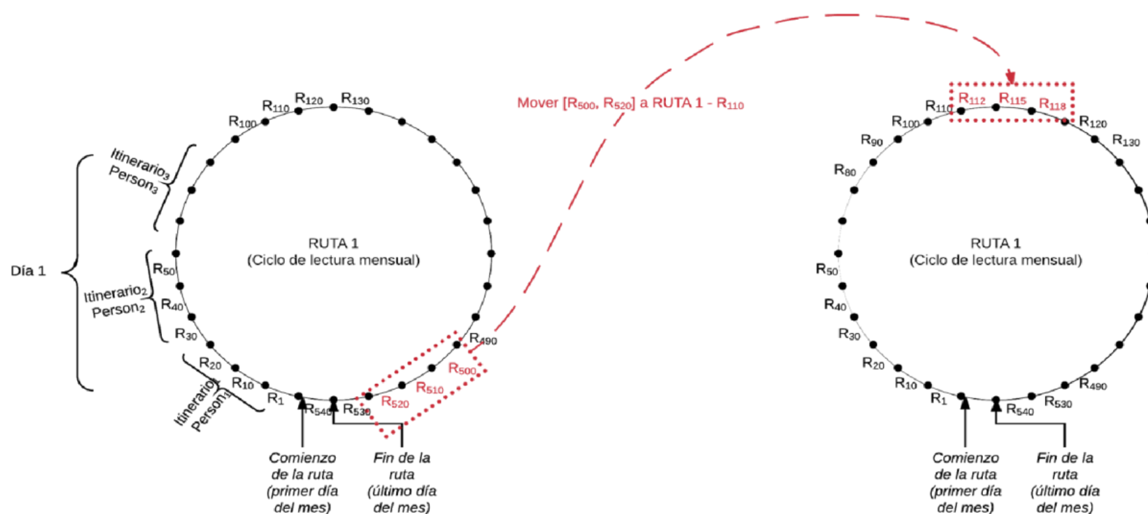

Figura 10: Servicio seleccionado para la migración “Recorridos”

5.2.2 Diseño del desacoplamiento

El submódulo (procedimiento de negocio) seleccionado para este estudio es un módulo autocontenido de máxima cohesión intramodular y de acoplamiento intermodular casi nulo, ya que es un servicio que no tiene dependencia del resto del sistema, por lo que

será más fácil entender su funcionamiento de manera aislada. Esto supone que no hay riesgos de que los cambios afecten el resto del sistema. Por otro lado, el servicio es lo suficientemente completo, y tiene una importante cantidad de lógica de negocio, por lo que su migración se considera un buen ejemplo para extrapolar en un hipotético proyecto de modernización global.

5.3 Patrones de origen (*Idioms* Precursores)

En base de la documentación, el código fuente y el modelo de dominio de la aplicación origen se analiza la arquitectura, plataforma y código, con el objetivo de definir de patrones que ayudan a establecer precursores de transformaciones.

El inventario de patrones de código es un documento muy clave para medir de impacto de los patrones en el código a migrar ya que esto nos puede dar alguna idea del grado de automatización que se podría lograr. En esta fase los pasos a seguir son:

1. Analizar la arquitectura, plataforma y código de la aplicación de origen
2. Definición de patrones en el código.
3. Validación de la calidad de los patrones.
4. Análisis de impacto en el código a migrar.

Al final de esta fase debemos tener el inventario de patrones de código origen y el informe de impacto de los patrones en el código a migrar.

5.3.1 Arquitectura de Origin (COBOL)

Se trata de analizar la arquitectura de aplicaciones utilizada por cliente en el contexto de la prueba de concepto de migración de Cobol a Java. Describiendo a nivel funcional el servicio a migrar a través de un modelo de dominio. Este análisis es utilizado como referencia a la hora de diseñar el proceso automatizado de migración de código.

Desde el punto de vista de la arquitectura de aplicación, en el mainframe se distinguen **transacciones CICS**[23] que implementan los procesos cliente, **rutinas** en las que se desarrolla la lógica de negocio y **rutinas generales** de soporte para, por ejemplo, la gestión de errores.

Desde un programa director, se pueden llamar a otros programas directores (con **EXEC CICS LINK**) o a rutinas (con **CALL**).

El intercambio de datos se realiza a través de **COMMAREAs**[24] definidas mediante **COPYS**[25]. En transacciones "nuevas", donde se ha superado la antigua restricción de

---

23 CICS es la interfaz del usuario a los programas, es un sistema donde residen una serie de transacciones de control del sistema y de usuario.

24 COMMAREAs son estructura de datos de entrada para comunicarse con CICS.

25 COBOL copybook: son archivos que contienen código fuente COBOL y se utilizan para sincronizar ese código entre varios programas COBOL. Pueden ser llamados desde más de un programa COBOL.

tamaño de las COMMAREA, la definición de las mismas es completa, por lo que con una única COPY se describe el mensaje al completo. Sin embargo, para transacciones "antiguas", cuando durante su desarrollo existía una limitación del tamaño de dichas COMMAREA, el mensaje de entrada no se componía con una única COPY. En su lugar, lo que se hacía es mandar un conjunto de mensajes de entrada, cada uno de un tamaño limitado (1500 bytes) y con COPYS independientes, que el programa COBOL se encargaba de recibir y de componer para su procesamiento. Cabe señalar que la composición de estos mensajes es posible porque tanto el emisor (cliente) como el receptor (servidor) conocen de antemano la especificación del mensaje. No obstante, es habitual que los mensajes se dividieran en una cabecera, en uno o varios mensajes de datos con estructuras repetitivas, y un mensaje de fin (pie). En los datos de cabecera se definen la cantidad de datos que vienen para que el receptor puede componer el mensaje correctamente (en el caso de las respuestas, este dato suele venir en el pie)

Así pues, de cara al proyecto, y en el caso de que el servicio a migrar tuviera esta manera de recibir la entrada (y de enviar la salida), y dado que en Java no existen limitaciones respecto al tamaño de mensajes, se hace necesario identificar mediante patrones las construcciones (boilerplate code) para la composición de mensajes de entrada y salida, ya que no forman parte de la lógica de negocio del programa y no es necesaria su migración.

Para hacerlo, en principio sería necesario la descripción por algún analista de cobol de dichas composiciones de mensajes por cada uno de los programas. No obstante, se analizará la posibilidad de utilizar la definición de dichos mensajes de entrada y salida en el código fuente GNSIS asociado a los programas, utilizando inicialmente como heurística el que, en el caso de composición de varios mensajes, el primero será la cabecera, el último será el pie, y el resto serán bloques repetitivos de datos. Además, se puede validar el buen funcionamiento de dicha heurística revisando el código cliente (NatStar) donde sí se encuentra detallada la relación entre los mensajes.

Para todo ello, se facilitarán las fuentes GNSIS a partir de los cuales se han generado los programas COBOL del servicio a migrar y el código NatStar del cliente que actualmente realiza las llamadas a las transacciones COBOL (usando DELTA/SIC como pasarela).

5.3.2 Ingenieria inversa

La herramienta "OC2-ADA" de Open Canarias, implementa un proceso de reingeniería utilizando el modelo de herradura que cubre el ciclo completo de MDA. Este ciclo que ya hemos detallado en el capítulo anterior Parte I.3.3), suele representarse en forma de herradura invertida, tal y como se muestra en la Figura 3.

Siguiendo este diagrama, el proceso de modernización comienza desde una solución existente (parte inferior izquierda del diagrama), a partir del cual se ejecuta un proceso de ingeniería inversa con el que, tras diferentes fases de ejecución, se obtiene un modelo independiente de la plataforma (PIM o Platform Independent Model). Este modelo

representa el 100% del programa original, pero no tiene ninguna dependencia con la plataforma origen (en nuestro caso COBOL/CICS). El lenguaje utilizado para este modelo es KDM. Como ya se ha comentado anteriormente, este lenguaje es también una especificación del OMG.

Sobre el modelo KDM se identifican una serie de patrones de código (denominados *Idioms*). Los idioms identifican estructuras específicas de la plataforma origen que, al migrarse a la nueva plataforma, han de resolverse de manera diferente. Por ejemplo, un idiom puede ser el acceso a datos (que en COBOL se resuelve mediante párrafos específicos que incluyen sentencias EXEC SQL, y en Java se resuelve a través de iBatis), o la gestión de errores (que en COBOL se resuelve mediante el retorno de códigos específicos de error, en Java a través del lanzamiento de excepciones).

5.3.3 Inventario de patrones de origen (*Idioms* Precursores)

Los patrones solucionan problemas que existen en muchos niveles de abstracción. desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

El objeto de este apartado es el de recoger el conjunto de patrones ***idioms*** COBOL identificados en el grupo de programas objeto del proyecto.

I.5.3.3.1 DomainDatabaseDefinition

Definición de las estructuras de datos incluidas en la WORKING-STORAGE SECTION vinculadas a entidades del dominio. En el contexto de un programa COBOL, este *idiom* nos permitirá identificar las Tablas / Vistas a las que accede el programa, y las estructuras de datos de utilizar en los accesos a dichas Tablas/Vistas.

I.5.3.3.2 DomainDatabaseAccess

El acceso a datos se organiza en secciones del programa bien establecidas y con una estructura de párrafos normalizada, al objeto de ser reutilizados en otros ámbitos del mismo programa.

I.5.3.3.3 TransactionParameters

Mapeo de los parámetros de entrada y salida a través del área de comunicaciones de la transacción.

I.5.3.3.4 ProgramParameters

Identificar los parámetros de entrada y salida de los programas para establecer los VO del contrato de los servicios de las entidades/acciones.

Por tanto, con este patrón se trata de identificar los diferentes ejemplos de intercambio de parámetros.

5.4 Patrones de destino (*Idioms* soluciones)

Partiendo del lenguaje de patrones de código (*idioms*) de OPEN CANARIAS, S.L, se trata de identificar en la arquitectura de destino y la soluciones a dichos patrones. Los pasos a seguir son:

1. Analizar la arquitectura, plataforma y código de la aplicación de destino.
2. Definición de patrones de código
3. Diseño de la plantilla de código.

5.4.1 Análisis de la Arquitectura de Destino (Java)

La Figura 11 muestra un diagrama, en el que se resume la arquitectura del sistema.

Figura 11: Arquitectura de destino de aplicaciones

Para la capa cliente se utiliza *NatStar*, que se comunica con la plataforma Java a través de un *servlet* de arquitectura que recibe la información utilizando el protocolo *hessian*. Este *servlet* se encarga de invocar al correspondiente *EJB* que a su vez lanza la ejecución de la lógica de negocio asociada. Los EJB's son EJB3 *Stateless Beans* (EJB de sesión sin estado).

Los EJB siguen la especificación EJB 3.0, aunque debido a que dichos componentes vienen de una versión EJB 2, muchos de ellos mantienen las definiciones de interfaces locales y remotas propias de dicha versión. Los nuevos EJB pueden usar la especificación EJB3 sin problema.

Cabe destacar que la capa de EJB's solo sirve como punto de entrada y aporta la transaccionalidad al sistema, siendo posible la ejecución de varios objetos de negocio (que se verán a continuación) y retroceder (rollback) en el caso de que necesidad para dejar las entidades sin modificar.

Por otro lado, los objetos de negocio son objetos planos (POJO's) que implementan la lógica de negocio. Desde un punto de vista lógico se denominan "Entidades", aunque no representan el concepto de Entidad desde el punto de vista del diseño orientado a objetos, ya que no tienen estado, sino que son la representación de las entidades del modelo de datos (tablas) y sirven fundamentalmente para gestionar las lecturas y cambios de estados en la base de datos.

Las entidades, aparte de acceder/modificar los datos, implementan comportamiento (lógica de negocio) como métodos públicos que pueden residir en la misma entidad (en el caso de que la lógica afecte a esa única entidad), o en nuevas clases denominadas "acciones" en el caso de que la lógica afecte a varias entidades. En algunos casos, como en la lógica de negocio del dominio de lecturas, es posible observar también objetos denominados "rutinas" que no son más que la lógica de las entidades, de manera que las entidades solo tienen código de acceso a datos. Las rutinas es una decisión particular del equipo de desarrollo del dominio de "Lecturas" y no sigue las directrices generales de la arquitectura.

Para el acceso a datos, las entidades (o acciones) acceden a base de datos usando el patrón DAO. Para el acceso a datos se utiliza iBatis, de forma que a través de ficheros de configuración se especifican las tablas, consultas y sentencias de actualización y el mapeo sobre los objetos que usará la aplicación. Se utiliza el propio framework de iBatis para la generación del código de los DAO a partir de la mencionada configuración.

Para el acceso a otros servicios se utilizan factorías de servicio (Services Factory), haciendo uso de una capacidad del propio framework de iBatis (diferente a la de acceso a base de datos), como otras formas de acceder a datos (aunque no sea directamente de la base de datos). En el caso de esta prueba de concepto, el servicio a migrar no consume datos de otros servicios, por lo que no será necesario hacer uso de la Service Factory de la arquitectura.

Por último, el intercambio de datos (dentro de la arquitectura DELTA/SIC) se realiza a través de otro tipo de objetos denominado "value objects" (VO). Nuevamente son objetos planos sin comportamiento, y su uso es obligatorio para el intercambio de datos entre las diferentes capas (EJB, objetos de negocio y DAO). A continuación, se presenta un diagrama donde se representa la interacción de los componentes comentados.

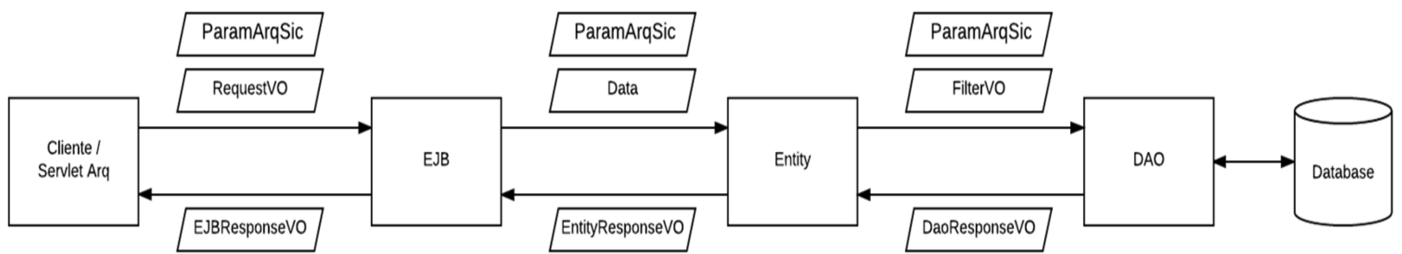

Figura 12: interacción de los componentes de la arquitectura

5.4.2 Ingeniería directa

Una vez identificados los *idioms* que según MDA estarían en el nivel más alto de la herradura se realizan una serie de transformaciones de modelos para conseguir representar (también en KDM) la solución a los mismos. Estas soluciones, a pesar de estar en el lenguaje KDM, están diseñadas para que representen elementos y estructuras similares o equivalentes a las que se necesitan en la plataforma destino. Por ejemplo, en el caso de los accesos a datos, la solución consistirá en definir elementos que representen los DAO, y otros que representen los ficheros XML de configuración de iBatis.

Una vez se disponga de las soluciones, el proceso de caída (ingeniería directa) consiste en la generación de código a partir de las soluciones modeladas. El código generado se corresponderá con los artefactos especificados para la arquitectura destino, es decir, incluirá el código de los EJB, los ENTITY de la capa de lógica de negocio, las clases DAO y ficheros de configuración XML de iBatis para el acceso a datos, y todos los POJO (Plain Old Java Objects) para los VO "value objects" que encapsulan los datos que maneja la aplicación.

5.4.3 Inventario de patrones de destino (*Idioms* soluciones)

I.5.4.3.1 DomainDatabaseDefinition

La definición de los datos en la arquitectura de destino se corresponde a objetos denominados "*Value Objects*" (VO).

Respecto a los tipos de datos, el ejemplo solo muestra tipos de datos básicos (numéricos y alfanuméricos). Puede darse el caso de datos estructurados, anidados o no, y repeticiones.

I.5.4.3.2 DomainDatabaseAccess

En la arquitectura J2EE utilizada por el cliente, utilizan el framework iBATIS 2 para la capa de acceso a la BBDD. Se han identificado los ficheros que son necesarios generar para realizar el acceso a la BBDD mediante el framework iBATIS 2:

- **VO** (Value Objects): Son POJOs utilizados para encapsular las estructuras de datos utilizadas en las operaciones CRUD con la BBDD.

- **DAO** (Data Access Object): Son clases Java en las que se definen las diferentes operaciones que se pueden realizar con la BBDD. Es necesario generar uno por cada tabla utilizada en el sistema.
- **SQLMap**: Son ficheros XML que utiliza el framework iBATIS 2 en los que se define el mapeo entre las sentencias SQL y su correspondiente VO. Es necesario generar uno por cada tabla utilizada en el sistema.
- **SqlMapConfig**: Es un fichero XML en el que se define el DataSource de la BBDD y se añaden todos los SQLMap utilizados para realizar el mapeo de sentencias SQL con sus correspondientes VO. Es necesario generar un fichero por cada fuente de datos diferente.

I.5.4.3.3 TransactionParameters

La estructura de datos de entrada / salida se corresponde con un VO a nivel JAR.

I.5.4.3.4 ProgramParameters

Como ya indicamos en el (apartado I.5.3.3.4) con este patrón se trata de identificar los diferentes ejemplos de intercambio (entrada y salida) de parámetros para establecer los VO del contrato de los servicios de las entidades/acciones.

5.5 Análisis y resolución de brechas

En base del inventario de patrones de código origen, el de destino y las plantillas diseñadas en la fase anterior para la generación de código, se establece en esta fase el mapa entre los patrones origen (precursores) y los patrones destino (soluciones) y analizar las brechas entre ambos para poder definir los algoritmos de transformación que las resuelvan. En esta fase las tareas a realizar son las siguientes:

1. Elaborar el mapa de relaciones entre patrones origen y destino
2. Analizar las brechas
3. Diseñar algoritmos de transformación
4. Implantar algoritmos de transformación

5.5.1 Diseño de migración entre arquitecturas

La brecha entre las arquitecturas COBOL y JAVA es baja, y en es sencillo establecer la correspondencia entre ambas:

- Un procedimiento cliente (conjunto de transacciones cobol) se corresponden con un EJB.
- Un programa director (transacción) en COBOL se corresponde con un método del EJB correspondiente.
- Las working áreas (copys) de entrada / salida de la transacción cobol se corresponde con value objects (VO) en Java.
- Una entidad de dominio (principalmente identificadas en el modelo de dominio) se corresponde con una clase java de lógica de negocio. Las entidades podrían identificarse inicialmente con los resultados de las consultas a base de datos. De esa forma, si se recuperan las entidades de la base de datos, se concluye que habrá una clase (entity) en la capa de negocio correspondiente.

- Una rutina en COBOL se corresponde con un método de la clase Entity de la lógica de negocio. Si la rutina maneja varias entidades, se establecería una clase “Acción” con dicha lógica.
- La comunicación de datos entre las transacciones y las rutinas se corresponden con value objects (VO).
- El acceso a datos en Cobol se hace fundamentalmente a través de EXEC SQL. Por cada entidad (tabla o vista) del modelo se corresponde con un objeto DAO de la capa de acceso a datos de Java. Estos objetos serán utilizados directamente por la lógica de negocio (clases Entity o Acciones)
- La comunicación de datos entre la capa de lógica de negocio y la de acceso a datos se realizará mediante objetos VO, cuando en COBOL se resuelve mediante la declaración de estructuras de datos en el propio programa.

Respecto a la gestión de errores:

- En Cobol los errores en tiempo de ejecución se traducen en ABEND (Abnormal End) y es gestionado por la arquitectura, mientras que en Java se traducen en Runtime Exceptions que no deben ser capturadas sino propagadas para que la arquitectura DELTA/SIC la gestione.
- Respecto a los errores de aplicación, en Cobol las rutinas de gestión de errores son generadas y repetitivas por todo el código, de forma que las transacciones devuelven un código de retorno y un texto descriptivo del error producido. En Java para estos casos se deben crear clases que extienden de ApplicationException y lanzar estas excepciones para que sean manejadas por los clientes que, al igual que en Cobol, previamente deben conocer los diferentes tipos de error de aplicación.

Respecto al entorno de desarrollo para la prueba de concepto, se acuerda utilizar la nube privada de IBM Bluemix que dispone la compaña contratada para desplegar el servidor de aplicaciones y la base de datos.

Respecto al servidor de aplicaciones se utilizará *IBM Websphere Liberty*, que no es el que se utiliza en los entornos corporativos, pero está probado y no debe dar problemas.

Para validar la prueba de concepto, se diseñará un conjunto de test junto con los expertos de negocio de la compaña contratada para verificar el correcto funcionamiento de los servicios migrados.

5.5.2 Proceso de transformación

Los patrones de estructura detectan los diferentes componentes del sistema origen y generan los esqueletos de los artefactos necesarios en el sistema destino. Sobre estos artefactos (en nuestro caso, principalmente clases Java), se incluirá el código generado por el resto de los patrones. A continuación, se presenta de forma diagramática el proceso de transformación, concretamente en lo relacionado a los patrones de estructura en la Figura 13.

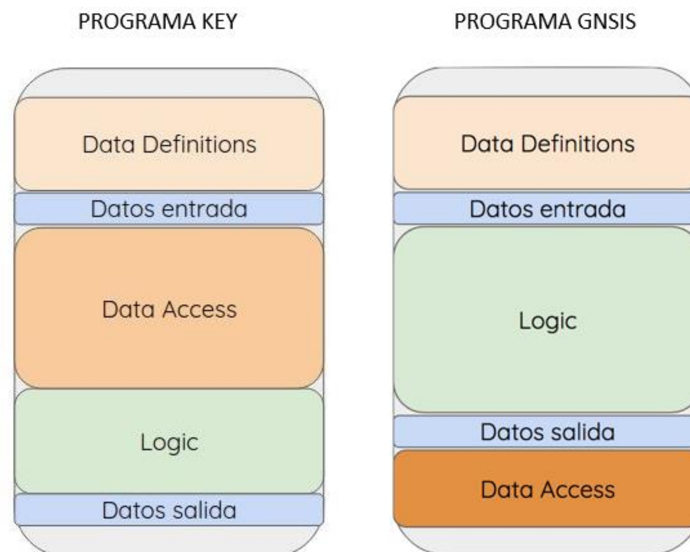

Figura 13: Patrones de estructura diferentes en el sistema origen (COBOL)

En lo que a estructura de un programa Cobol se refiere, hay que diferenciar entre el código generado por KEY del generado por GNSIS. Como se puede ver en el diagrama anterior, los accesos a datos se ubican en lugares diferentes, pero lo más importante a señalar es que también se hacen mediante patrones diferentes (hecho representado por la diferencia de tonalidades entre los mismos elementos en programas KEY y GNSIS). Es decir, el acceso a datos se realiza de manera diferente en los programas KEY y GNSIS, lo que obliga a disponer de patrones para cada caso.

Una vez identificados estos patrones de estructura, cada uno de los elementos asociados se corresponden con artefactos de la plataforma destino.

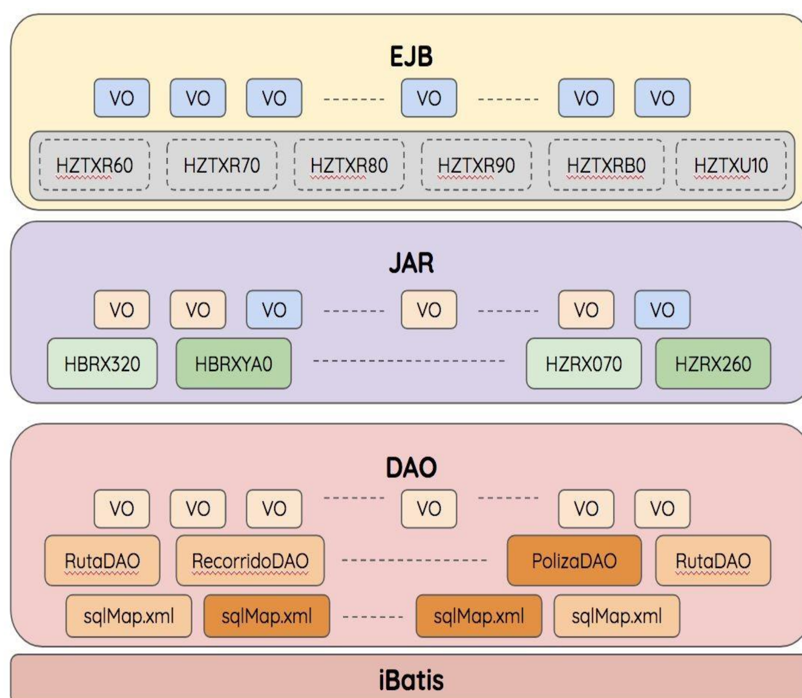

Figura 14 : Artefactos en plataforma DELTA/SIC

En la Figura 14, la correspondencia entre los elementos de los programas Cobol y los artefactos generados en las diferentes capas de la arquitectura Java.

A modo de ejemplo, en la siguiente tabla se clasifican algunas de estas correspondencias.

Patrón	COBOL	Capa	JAVA
DataDefinition	WORKING-STORAGE SECTION	DAO/ JAR	Clases VO
DataAccess	EXEC-SQL	DAO	Clase Interfaz Clase implement. iBatis XML SQLMap
Estructura	Transacción (programa director)	EJB	Método en EJB
TransactionParameters	MOVE TEX-DATOS-APLICACION	EJB	Clase VO
ProgramParameters	LINKAGE SECTION	JAR	Clase VO
Estructura	Rutina	JAR	Clase

Tabla 2: Correspondencias COBOL / Java

5.6 Estrategia de migración de datos

Respecto a la base de datos, se acordó utilizar en primera instancia PostgreSQL en lugar de DB2 por costes operativos, ya que las sentencias SQL usadas por los programas que usan DB2 deben funcionar sin mayores problemas. En caso de fuera necesario, se contemplaba la posibilidad de sustituir el servidor PostgreSQL por una instancia de DB2 en la nube, asumiendo la compaña contratada el coste. Sin embargo, una vez especificado el primer servicio a migrar, denominado "Reubicación", no hace prácticamente uso de datos de otros dominios (y si lo hace se ha acordado no tenerlo en cuenta en el proyecto) y no es llamado desde otros programas, al ser autocontenido. No obstante, las llamadas desde Java a servicios Cobol están resueltas en la arquitectura y se han probado en desarrollos hechos por el cliente, por lo que no suponen un gran riesgo al no tenerlos en cuenta en este proyecto.

5.7 Generación de código

La generación de código es un proceso manual a menos que se disponga de herramientas de automatización que identifiquen patrones de destino y origen y resuelven las brechas entre ambas. Aunque se automatice la generación de código, siempre habrá brechas sin resolver y requerirán modificaciones manuales. Finalmente se realiza la validación preliminar y se verifica que el código por lo menos no tiene errores de compilación. En esta fase realizaremos las siguientes tareas:

1. Generación de código automática o manual.
2. Revisión inicial del código generado.
3. Introducción de las adiciones y modificaciones manuales.
4. Verificación preliminar.

5.7.1 Grado de automatización por patrones

A continuación, en la Tabla 3, se relaciona el conjunto de patrones de arquitectura inventariados y el grado de implementación de los mismos en "OC2-ADA".

		KEY		GNSIS		OTROS	
Patrón	Descripción	Doc.	Impl.	Doc.	Impl.	Doc.	Impl.
Domain Database Definition	Definición de las estructuras de datos relacionadas con una entidad del dominio en el contexto de la base de datos.	S	S	S	S	S	S
Domain Database Access	Acceso a datos: consultas, inserciones, actualizaciones y borrados.	S	S(*)	S	S(*)	S	S(*)
Transaction Parameters	Mapeo de los parámetros de entrada y salida a través del área de comunicaciones de la transacción.	S	N	S(**)	N	S(**)	N
Program Parameters	Identificar los parámetros de entrada y salida de los programas para establecer los VO del contrato de los servicios de las entidades/acciones.	S	N	S	N	S	N

Tabla 3: Grado de implementación automática de patrones

(*) Implementado parcialmente, a falta de la generación de los xml de iBatis
(**) El patrón es el mismo independientemente del tipo de programa (KEY, GNSIS u OTROS)

5.7.2 Código generado

La Tabla 4, muestra con detalle los artefactos Java generados automáticamente.

Artefacto	#	Lines	LOC
VO's	142	54.320	19.768
Interfaces DAO	84	3.227	760
Clases DAO	84	6.156	2.513
Interfaz EJB	1	91	25
Clase EJB	1	212	98
TOTAL	**312**	**64.006**	**23.164**

Tabla 4: Código generado

Estos datos han sido obtenidos mediante la ejecución de un análisis estático de código Java utilizando la herramienta SonarQube.

5.8 Validación y Pruebas

Esta es una fase de validación y prueba de código como en cualquier proyecto software. A partir de la fase anterior, tendríamos el código fuente generado, el esquema de persistencia para la migración de datos. Además, necesitaríamos las políticas y prácticas de desarrollo en la organización e información sobre la infraestructura de integración continua del cliente. En este caso las tareas a realizar son las siguientes:

1. Análisis del código generado
2. Diseño de las pruebas
3. Codificación de las pruebas
4. Migración de datos
5. Lanzamiento de pruebas
6. Elaboración de informes (pruebas y calidad de producto).

Al final de esta fase tendremos los datos migrados junto al informes de pruebas y de calidad del producto.

Para la realización de las pruebas hemos utilizado JUnit, para facilitar el consumo de datos para las pruebas, serán cargados desde un objeto JSON, para ellos he creado una interfaz para la interacción entre JUnit y los ficheros JSon, el siguiente diagrama describe interfaz que me ayuda a "parsear" el objeto y manejarlo con pocas líneas desde JUnit.

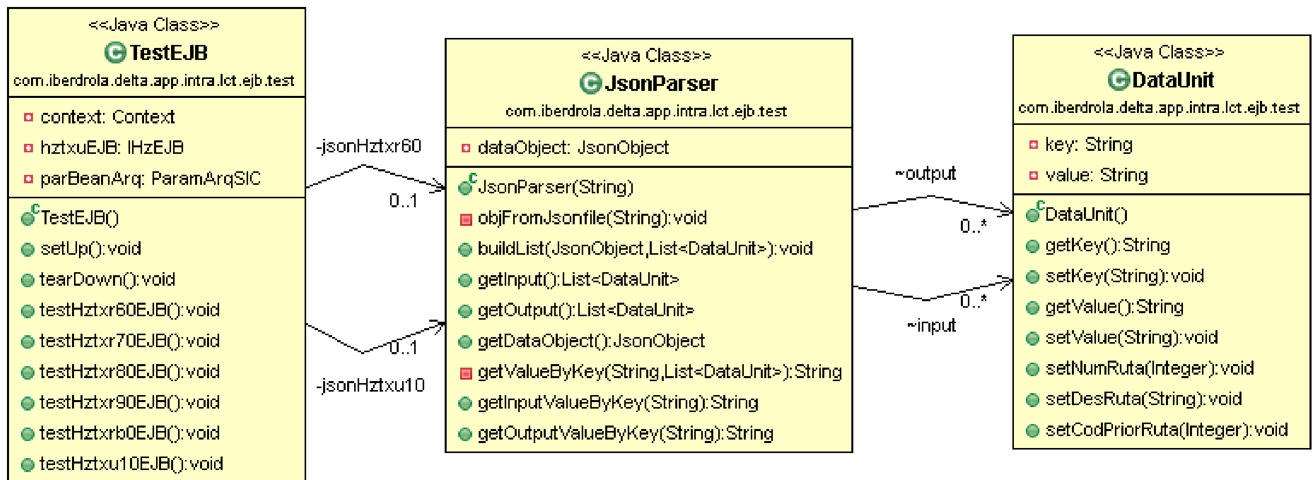

Figura 15: Interfaz JSON Parcer

Un ejemplo del objeto JSON desde fichero sería el siguiente, donde se distinguen los datos de entrada y los de salida:

```
{
 "input": {
                "IN-KEY-1":"aaa",
                "IN-KEY-2":111,
                "IN-KEY-3":222, },

 "output": [
              {  "OUT-KEY-7":1111,
                 "OUT-KEY-8":2222,
                 "OUT-KEY-9":3333    }
           ]
}
```

Desde JUnit solamente tengo que instanciar este objeto con el fichero que contiene los datos y luego manejarlo para extraer los datos, en el Apéndice D. se detalla un ejemplo completo.

```
test = new JsonParser("entrada.json" );
String tipOrgInterna = test.getInputValueByKey("KEY-1");
Integer codOrganInter = Integer.parseInt(jsonHztxr60.getInputValueByKey("KEY-2"));
```

Respecto la migración de datos, como ya se ha comentado en la definición del caso de estudio en el apartado (Parte I.5.1.2), se ha acordado con el cliente no tener en cuenta la migración de datos en la conversión de este módulo.

5.9 Resultados

Este apartado tiene como objeto mostrar de forma cuantitativa y metodológica, el ahorro obtenido por la automatización del proceso de migración de Cobol a Java en el contexto de la prueba de concepto denominada "Migración del código de un servicio en plataforma COBOL/CICS/DB2 a Java JEE".

Está estructurado como sigue. En primer lugar, se define la métrica utilizada para la medición del ahorro. A continuación, se presenta el método que utiliza la herramienta de automatización "OC2-ADA" para que se comprenda cómo se realiza el proceso de migración y generación de código. Por último, se presenta el resultado mediante datos cuantitativos del código generado y la estimación del ahorro obtenido utilizando la métrica presentada.

5.9.1 Métrica para la medición del ahorro

Para esta prueba de concepto se ha considerado utilizar como métrica para la medición del ahorro obtenido la "líneas de código", en adelante LOC (Lines Of Code). Entiéndase como línea de código cualquier línea del fichero de código fuente que no es un comentario, ni instrucciones de pre-procesamiento, como es el caso de las sentencias SKIP1, SKIP2, SKIP3, COPY, EJECT y REPLACE.

Esta métrica se puede utilizar para la estimación del coste de una migración manual. Tenemos como referencia el estudio realizado por el Grupo Gartner[11] donde se indica que la conversión manual de código puede oscilar entre $6 y $26 por LOC, con una tasa de 160 LOC por día.

Respecto al marco temporal, en el mismo estudio, Gartner estima que la migración de 26 MLOC (1 millón de líneas de código) llevaría un total de 28 años/hombre. Por una simple regla de tres, eso equivale a 3,2256 minutos por LOC. No obstante, la paralelización del trabajo con equipos muy grandes requeriría de más interacciones, ralentizando aún más el proceso. Gartner concluye que la migración manual de una aplicación de más de 10 MLOC en la práctica no tiene sentido debido a los plazos que se manejan.

En este informe incluso se asume que la migración se realiza entre dos lenguajes relativamente similares (lo que no aplica en nuestro caso, pues se intenta pasar de COBOL a un lenguaje orientado a objetos como es Java).

[26] Semantic Designs : Migration:http:/www.semdesigns.com/Products/Services/LegacyMigration.html

5.9.1 Artefactos del servicio

Desde el punto de vista técnico, el servicio se compone de los artefactos en la Tabla 5:

<table>
<tr><th colspan="2">Programas Cobol</th><th>Copybooks</th></tr>
<tr><td colspan="2">37</td><td rowspan="3">113</td></tr>
<tr><th>Programas directores</th><th>Rutinas</th></tr>
<tr><td>6</td><td>31</td></tr>
</table>

Tabla 5: Artefactos del servicio

El inventario de transacciones (programas directores) se ha facilitado por el equipo de proyecto del cliente, y las rutinas a las que se llama a través de análisis estático de código con las herramientas de Open Canarias. El catálogo de programas se ha contrastado con la revisión manual hecha por el equipo del cliente. La relación de copys se ha obtenido por ingeniería inversa del servicio utilizando las herramientas de Open Canarias.

A su vez, en la Tabla 6 el código de los programas cobol ha sido generados por dos herramientas diferentes (KEY y GNSIS). El servicio a migrar se compone de:

Programas KEY	Programas GNSIS	Programas OTROS
27	8	2

Tabla 6: Código por herramientas Generadoras KEY y GNSIS

Los programas clasificados como “OTROS” son principalmente programas GNSIS cuyo código cobol difiere del código GNSIS del resto de programas, posiblemente por haber sido modificado a mano después de la generación.

Respecto a los programas cobol, se han contabilizado un total de 165.746 líneas y 105.984 LOC.

Respecto a los copybooks, a la información anterior hay que sumarle 8.947 líneas y 7.810 LOC.

Como se puede observar, el servicio incluye un total de 174,682 líneas (en total) de los cuales de 113.794 LOC (resultado de sumar las líneas de código de los programas y de las copybooks, excluyendo comentarios e instrucciones de pre-procesamiento). Por lo tanto, el número de LOC que hay que migrar y que servirá de referencia para el proyecto será:

113.794 LOC

5.9.2 Estimación del ahorro

La primera entrega de este proyecto requería una estimación del ahorro del proceso de modernización y el grado de automatización del proceso. Esto se obtendría después de la generación completa del código, calculando cuánto del código COBOL se migra de forma automática frente al trabajo manual de migración. Sin embargo, para poder estimar dicho ahorro se me ha encargado realizar esta estimación de LOCs afectadas por los patrones definidos. Mediante un script programado para este fin, con la utilización de expresiones regulares se inspecciona todo el código COBOL. El programa recibe una lista con los nombres de los ficheros que componen todo el código COBOL del servicio, busca estos ficheros por su nombre en todo el directorio, luego recorre cada uno de estos ficheros y calcula las líneas de los fragmentos de código detectados por los patrones. Un ejemplo se muestra en el Apéndice A. ScriptAhorro.

El script tiene dos ficheros de salida, uno es un fichero con los detalles y fragmento de código detectado por cada patrón y cuantas líneas contiene y el otro es un resumen de la estimación de LOCs afectadas por los patrones definidos, se puede ver en el Apéndice B.
Salida del ScriptAhorro.

Se propone medir el ahorro del proceso de modernización, calculando el grado de automatización del proceso. Esto es, calculando el número de líneas del código COBOL que se migra de forma automática. El resto de líneas que no ha sido posible migrar de forma automática se migrará manualmente. Dicho ahorro lo calculamos como el cociente entre el número de líneas de código (LOC) afectadas por los patrones definidos (automatizado) y el número de líneas de código total:

$$Ahorro(\%) = \frac{LOC_{Patterns}}{LOC_{Total}} * 100$$

Para calcular el número de líneas de código afectadas por los patrones ($LOC_{Patterns}$) se han localizado las líneas de comienzo y final de los diferentes fragmentos de código COBOL identificados por cada uno de los patrones definidos. A continuación, se han eliminado los comentarios asociados a dichos fragmentos y se han contabilizado las líneas.

El resultado es el número de líneas de código de cada fragmento de código afectado por un patrón se obtiene sumando todas las cantidades para cada uno de los programas.

En el caso de los COPYBOOKS, por su propia definición, todo el código asociado encaja con el patrón de definición de datos (Data Definition), por lo que todos ellos son identificados como líneas de código afectadas por un patrón. Los copybooks tienen un total de 8.947 líneas de los cuales 7.810 LOC.

I.5.9.2.1 Cálculo de líneas afectadas por los patrones

1. $\boldsymbol{LOC_{Patterns}}$: Obtenemos el total de líneas de código afectadas por los patrones sumando las respectivas cantidades de los ***programas*** y de los ***copybooks***:

$$LOC_{Patterns} = LOC_{Patterns}(programs) + LOC(copybooks)$$
$$LOC_{Patterns} = 61.303 + 7.810 = 69.113\ LOC$$

2. $\boldsymbol{LOC_{Total}}$ **:** Calculando el número de ***líneas total*** de los programas (excluidos comentario e instrucciones de pre-procesamiento):

$$LOC_{Total} = LOC_{Total}(programs) + LOC(copybooks)$$
$$LOC_{Total} = 105.884 + 7.810 = 113.794\ LOC$$

Por lo tanto, calculamos el ahorro obtenido como:

$$Ahorro = \frac{69.113}{113.794} * 100 =$$

60,74 %

I.5.9.2.2 Valor económico del ahorro

Por último, podemos hacer una proyección a valor económico de este ahorro. Haciendo referencia al Grupo Gartner a través de las estimaciones facilitadas en dicho informe, en el caso más optimista, calcula el coste de migración de $6 / LOC. Esto supone que, en un proyecto como este, el coste total de la migración del servicio de Reubicación ascendería a:

$$Coste\ Migración_{Gartner} = 113.794\ LOC * \$6 = \$\mathbf{682.764}$$

En el caso del proceso de modernización, sólo habría que migrar manualmente las líneas que no se han podido migrar de forma automática, por lo que sale una estimación de:

$$Coste\ Migración_{Open\ C.} = (113.794\ LOC - 69.113\ LOC) * \$6 = \$268.086$$

Finalmente, esto supone un ahorro de:

$414.678

Capítulo 6. Conclusiones

Este trabajo ha consistido en la documentación de todo el proceso de modernización de sistemas heredados, mediante la propuesta de una metodología conceptual y la aplicación en un caso real, que consiste en la migración incremental de aplicaciones del cliente sin interrumpir su funcionamiento. La metodología se basa en especificaciones de *Modernización Dirigida por la Arquitectura* (ADM) y otros paradigmas de arquitectura del software definidos por Object Management Group (OMG), basándose en un proceso de reingeniería que emplea el modelo de herradura que comienza desde el código heredado y construye un modelo abstracto, facilitando su comprensión y transformación para obtener el sistema destino, utilizando como representación común el metamodelo (KDM).

El trabajo incluye un caso de estudio de un proyecto de gran envergadura que requiere de un alto nivel de ingeniería, en el cual he tenido el placer de participar y contribuir. El caso de estudio ha consistido en establecer un proceso semiautomático que facilita la migración incremental de las aplicaciones de una compañía del sector energía con presencia internacional. La migración ha sido de un submódulo de la plataforma COBOL/CICS/DB2 a Java JEE, como proyecto piloto y marco de referencia para la migración del resto de servicios existentes del sistema.

Mi participación tanto en las diferentes actividades del proceso de migración como en las reuniones de seguimiento con los diferentes grupos de trabajo (dirección, desarrollo, Investigación, Desarrollo e Innovación, etc.), me permitió tener una visión completa del proceso de modernización según las normas y los estándares internacionales. Durante el proceso se abordó la transición de un sistema heredado hacia una nueva arquitectura que responde a nuevas necesidades, con un enfoque integral y orientado a que dicha transición sea ordenada, controlada y adaptada a las especificidades del cliente en el futuro sistema. La migración ha sido un éxito, por lo que el cliente ha concedido el contrato de migración del resto del sistema.

Este trabajo se ha centrado en la relevancia de la arquitectura y sus paradigmas en la modernización de sistemas heredados y es una iniciativa que pretende establecer una base a mejorar de la metodología de la modernización de sistemas heredados donde se resalta la relevancia de la arquitectura para solucionar un problema común de muchas empresas, y que sirva de aporte a la sociedad y a las personas involucradas en tecnologías de la información y que pueda servir también como base para futuras investigaciones.

Un objetivo a largo plazo sería estudiar cómo mejorar la automatización de la modernización en general y al modelo de herradura en particular.

Conclusions

This work consisted in the documentation of the process of *Legacy System Modernization,* through the proposal of a conceptual methodology and the application in a real case of the incremental migration of client applications without interrupting its operation.

The methodology is based on *Architecture-Driven Modernization (ADM)* specifications and other software architecture paradigms defined by the *Object Management Group* (OMG) and using the "*horseshoe model*" that starts from the legacy code and builds an abstract model, making easy its understanding and transformation to obtain the target system, using the (KDM) metamodel as a common representation.

The work includes a case study of a large project that requires a high level of engineering, in which I have had the pleasure of participating and contributing. The case study has consisted in establishing a semi-automatic process that facilitates the incremental migration of the applications of a company from the energy sector with an international presence. The migration has been from a submodule of the platform COBOL / CICS / DB2 to Java JEE, as a pilot project and frame of reference for the migration of the rest of existing services of the system.

Thanks to the support of the professional team of OPEN CANARIAS, my participation in the different activities of the migration process as well as in the follow-up meetings with the different work groups (management, development, Research, Development and Innovation, etc.), allowed me to have a complete vision of the modernization process according to international norms and standards. During the process, the transition from an inherited system to a new architecture that responds to new needs was approached, with an integral approach oriented to ensure that this transition is ordered, controlled and adapted to the client's specificities in the future system. The migration has been successful, so the client has granted the migration contract from the rest of the system.

This work is an initiative that aims to establish a basis to improve the methodology of Legacy System Modernization which highlights the relevance of architecture to solve a common problem of many companies, and to serve as a contribution to society and the people involved in information technologies and that can also serve as a basis for future research.

This work has focused on the relevance of architecture and its paradigms in the Legacy System Modernization. A long-term goal would be to study how to improve the automation of modernization in general and the horseshoe model in particular.

Apéndice A. ScriptAhorro

```
#!/bin/bash
FILELIST_PROGRAMS_NAME=cliente_mod_cobol/file_list_names.txt
LINEAS=0
LOC=0
INDEX=0
DIR=cliente_mod_cobol/COBOL/
declare -A rfile
declare -A rloc

for line in $(cat $FILELIST_PROGRAMS_NAME)
do
 FILELINES=0
 FILELOC=0
 fichero=$DIR/$line
 if [ -f $fichero ]; then
 # ############### COBOL_NAME ###############################
 # ejemplo del patron :
             # Ini # 010000 COBOL_NAME-SECTION.
             # ...
             # Fin # 015000 COBOL_NAME-EXIT.
 let INDEX=INDEX+1
 PATTERNG_ACC=`awk '/[0-9]{6}\s*G-ACC-[A-Z0-9]*\s*SECTION./{ print $2 }' $fichero`
 echo "║ G-ACC DETECTED PATTERNS :"
 for i in $PATTERNG_ACC
 do
 RESTA_COUNT=0
 LOC_CONT=0
 INI='[0-9]{6}\s*'$i" SECTION."
 FIN='[0-9]{6}\s*'$i"-EXIT. EXIT."
 LOC_CONT=`awk "/$INI/{f=1}f" $fichero | awk "/$INI/{p=1} p; /$FIN/{exit}" | grep -v -E "[0-9]{6}\*" | wc -l`
 TEMP=`awk "/$INI/{f=1}f" $fichero | awk "/$INI/{p=1} p; /$FIN/{exit}" > $DIR/temporal$i.txt`
 CODE=`awk "/$INI/{f=1}f" $fichero | awk "/$INI/{p=1} p; /$FIN/{exit}" | grep -v -E "[0-9]{6}\*" `
 RESTA=`awk '/EXEC SQL/,/END-EXEC/' $DIR/temporal$i.txt`
 RESTA_COUNT=`awk '/EXEC SQL/,/END-EXEC/' $DIR/temporal$i.txt | wc -l`

 let LINE_CONT=LINE_CONT-RESTA_COUNT
 let rloc[$line]=rloc[$line]+LOC_CONT
 echo "║
 echo -e "║ ├PATTERNG_ACC [$i] \t LINES : $LINE_CONT LINEs"
 echo -e "║ │ \t\t\t\t\t\t\t\t LOC : $LOC_CONT LOCs"
 echo "║
 echo "║
 echo "║ ├CODIGO G-ACC INTERO :"
 echo "$CODE "
 echo "║ ├CODIGO EXCLUIDO DE G-ACC QUE CORRESPONDE A EXEC_SQL :"
 let FILELINES=FILELINES+LINE_CONT
 let FILELOC=FILELOC+LOC_CONT
 done
 else
 echo "╔════════════════════════════════════════════════════════════════════════════════"
 echo "║ : FILE NAME : $line : This file does not exist in "$DIR" "
 echo "╚════════════════════════════════════════════════════════════════════════════════"
 fi
done
 echo "╔════════════════════════════════════════════════════════════════════════════════"
 echo -e "║ : TOTAL \t\t\tLINES \t: $LINEAS lines"
 echo -e "║ : TOTAL \t\t\tLOCs \t: $LOC lines"
 echo "╚════════════════════════════════════════════════════════════════════════════════"
```

Apéndice B.
Salida del ScriptAhorro

RESUMEN POR FICHERO

NAME	LINES	LOC
1 PRGRCOBOL1	75	75
2 PRGRCOBOL2	429	389
3 PRGRCOBOL3	81	72
4 PRGRCOBOL4	80	71
5 PRGRCOBOL5	87	79
6 PRGRCOBOL6	35	35
7 PRGRCOBOL7	2188	1644
8 PRGRCOBOL8	3073	2697
10 PRGRCOBOL10	1137	1049
11 PRGRCOBOL11	113	104
12 PRGRCOBOL12	109	97
13 PRGRCOBOL13	56	56
14 PRGRCOBOL14	4659	4438
15 PRGRCOBOL15	3548	3126
16 PRGRCOBOL16	20497	15316
17 PRGRCOBOL17	14655	11840
19 PRGRCOBOL19	2510	1724
20 PRGRCOBOL20	1482	969
21 PRGRCOBOL11	4185	2715
22 PRGRCOBOL22	952	658
23 PRGRCOBOL23	513	360
24 PRGRCOBOL24	1075	722
25 PRGRCOBOL25	1330	902
26 PRGRCOBOL26	348	295
27 PRGRCOBOL27	570	457
28 PRGRCOBOL28	1810	1332
31 PRGRCOBOL31	1989	1420
32 PRGRCOBOL22	3152	2313
33 PRGRCOBOL23	479	395
34 PRGRCOBOL34	2149	1571
36 PRGRCOBOL36	1575	1127
37 PRGRCOBOL37	1078	759

: TOTAL	LINES	: 79184 lines
: TOTAL	LOCs	: 61303 lines

Apéndice C.
Resultados de la estimación del ahorro

Programa	Lines	Pattern Lines	Pattern Lines (%)	LOC	Pattern LOC	Pattern LOC (%)
COBOLPRGM1	31.988	20.497	64,08%	**20.888**	**15.316**	73,32%
COBOLPRGM2	25.470	14.655	57,54%	**17.193**	**11.840**	68,87%
COBOLPRGM3	7.403	4.659	62,93%	**6.335**	**4.438**	70,06%
COBOLPRGM4	6.686	3.073	45,96%	**4.979**	**2.697**	54,17%
COBOLPRGM5	6.426	3.548	55,21%	**4.758**	**3.126**	65,70%
COBOLPRGM6	7.742	3.152	40,71%	**4.526**	**2.313**	51,10%
COBOLPRGM7	7.082	4.185	59,09%	**3.896**	**2.715**	69,69%
COBOLPRGM8	5.054	75	1,48%	**3.173**	**75**	2,36%
COBOLPRGM9	4.903	2.149	43,83%	**3.023**	**1.571**	51,97%
COBOLPRGM10	4.421	2.188	49,49%	**2.930**	**1.644**	56,11%
COBOLPRGM11	4.609	1.989	43,15%	**2.800**	**1.420**	50,71%
COBOLPRGM12	4.751	2.510	52,83%	**2.774**	**1.724**	62,15%
COBOLPRGM13	3.650	1.810	49,59%	**2.282**	**1.332**	58,37%
COBOLPRGM14	3.587	1.575	43,91%	**2.263**	**1.127**	49,80%
COBOLPRGM15	3.813	1.137	29,82%	**2.087**	**1.049**	50,26%
COBOLPRGM16	2.500	521	20,84%	**1.739**	**370**	21,28%
COBOLPRGM17	2.648	1.330	50,23%	**1.538**	**902**	58,65%
COBOLPRGM18	2.712	1.482	54,65%	**1.494**	**969**	64,86%
COBOLPRGM19	1.786	113	6,33%	**1.335**	**104**	7,79%
COBOLPRGM20	2.281	1.078	47,26%	**1.332**	**759**	56,98%
COBOLPRGM21	2.287	1.075	47,00%	**1.269**	**722**	56,90%
COBOLPRGM22	1.670	80	4,79%	**1.238**	**71**	5,74%
COBOLPRGM23	1.648	87	5,28%	**1.228**	**79**	6,43%
COBOLPRGM24	2.015	804	39,90%	**1.131**	**578**	51,11%
COBOLPRGM25	1.930	952	49,33%	**1.122**	**658**	58,65%
COBOLPRGM26	2.109	659	31,25%	**1.023**	**561**	54,84%
COBOLPRGM28	1.975	570	28,86%	**949**	**457**	48,16%
COBOLPRGM29	1.482	429	28,95%	**828**	**389**	46,98%
COBOLPRGM31	1.525	513	33,64%	**816**	**360**	44,12%
COBOLPRGM33	1.303	348	26,71%	**627**	**295**	47,05%
COBOLPRGM34	1.011	56	5,54%	**487**	**56**	11,50%
COBOLPRGM35	830	81	9,76%	**481**	**72**	14,97%
COBOLPRGM36	999	35	3,50%	**473**	**35**	7,40%
COBOLPRGM37	822	109	13,26%	**349**	**97**	27,79%
TOTAL	**165.736**	**79.184**	**47,78%**	**105.984**	**61.303**	**57,84%**
COPYBOOKS	**8947**	**8947**	**100,00%**	**7810**	**7810**	**100,00%**
TOTAL	**174.683**	**88.131**	**50,45%**	**113.794**	**69.113**	**60,74%**

Apéndice D. Pruebas

```
/*@author ext-mmahrach*/
public class TestEJB {
 private JsonParser jsonHztxr60;
 private JsonParser jsonHztxu10;
 /**@throws java.lang.Exception */
 @Before
 public void setUp() throws Exception {
 jsonHztxr60 = new JsonParser("JsonEntry.json");
 }
 @After
 public void tearDown() throws Exception {
 JsonEntry = null;
 }
 @Test
 public void testHztxr60EJB() throws Exception {
 //objeto de entrada
 INVO IN = new INVO();
 //parametros del objeto de entrada
 String tipOrgInterna = jsonHztxr60.getInputValueByKey("IN-KEY-1");
 Integer codOrganInter = Integer.parseInt(jsonHztxr60.getInputValueByKey("IN-KEY-2"));
 Integer numCgl= Integer.parseInt(jsonHztxr60.getInputValueByKey("IN-KEY-3"));
 IN.setCodOrganInter(codOrganInter);
 IN.setNumCgl(numCgl);
 IN.setNumRuta(numRuta);
 System.out.println("Parámetros de entrada: ");
 //Objeto de salida
 OUTVO OUT = new OUTVO();
 //parametros del objeto de salida
 Integer codRetornoTzxr6 = Integer.parseInt(jsonHztxr60.getOutputValueByKey("OUT- KEY-7"));
 Integer codRetornoRvgir = Integer.parseInt(jsonHztxr60.getOutputValueByKey("OUT-KEY-8"));
 Integer codRetornoRtgrr = Integer.parseInt(jsonHztxr60.getOutputValueByKey("OUT-KEY-9"));
 OUT.setCodRetornoRtgrr(codRetornoRtgrr);
 OUT.setCodRetornoRvgir(codRetornoRvgir);
 OUT.setCodRetornoTzxr6(codRetornoTzxr6);
 OUTVO result = hztxuEJB.hztxr60(parBeanArq, IN);
 //Comparar resultado con el objeto de salida
 assertTrue(result.equals(OUT));
 }
 //Comparar resultado con el objeto de salida
 assertTrue(result.equals(wzgu2List));
 }
}
```

Bibliografía

[1] Open Canrias, "Modernización de Oracle Forms® / Oracle Reports®."

[2] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems : software technologies, engineering processes, and business practices*. Addison-Wesley, 2003.

[3] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press, 1985.

[4] P. F. Ericsson and R. Haswell, *Machine Scoring of Human Essays: Truth and Consequences*. Utah State University Press, 2006.

[5] Z. Li, X. Anming, Z. Naiyue, H. Jianbin, and C. Zhong, "A SOA Modernization Method Based on Tollgate Model," in *2009 International Symposium on Information Engineering and Electronic Commerce*, 2009, pp. 285–289.

[6] C. J. Fernández Candel, J. García-Molina, F. Javier Bermúdez Ruiz, and D. Sevilla Ruiz, "Una experiencia con transformaciones modelo-modelo en un proyecto de modernización."

[7] "Servicios de modernización transformación digital."

[8] OMG, "Architecture-Driven Modernization Scenarios," *OMG - Organ. Manag. Gr.*, pp. 1–12, 2006.

[9] W. M. Ulrich, *Legacy systems: transformation strategies*. Prentice Hall, 1994.

[10] M. L. Brodie and M. Stonebraker, *Migrating legacy systems: gateways, interfaces \& the incremental approach*. Morgan Kaufmann Publishers, 1995.

[11] K. Bennett, "Legacy systems: coping with success," *IEEE Softw.*, vol. 12, no. 1, pp. 19–23, Jan. 1995.

[12] Ileana María Salas Campos, / *Editorial UNED*. EUNED, 2006.

[13] C. S. & E. F. of E. U. Jha Meena, "Building a Systematic Legacy System Modernization Approach." Awarded by:University of New South Wales. Computer Science & Engineering, 2014.

[14] E. F. Muñoz Recuay, "Integrador de Sistemas Heredados, Una Solución para la Integración de Información," 2007.

[15] P. K. Rash and A. D. Miller, "A survey of practices of teachers of the gifted," *Roeper Rev.*, vol. 22, no. 3, pp. 192–194, 2000.

[16] M. Jha and L. O'Brien, "Comparison of Modernization Approaches: With and Without the Knowledge Based Software Reuse Process," *Proc. 2nd Int. Conf. Adv. Comput. Sci. Eng. (Cse 2013)*, vol. 42, no. Cse, pp. 68–71, 2013.

[17] J. Bisbal *et al.*, "An Overview of Legacy Information System Migration."

[18] Object Management Group, "Architecture-Driven Modernization Task Force Glossary of Definitions and Terms -- V3 24 JAN. 2006," pp. 1–34, 2006.

[19] John Klein, "Architecture Practices for Complex Contexts," *Software Engineering Institute*, 2017. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=506213. [Accessed: 30-May-2019].

[20] E. Evans, *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, 2004.

[21] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, Sep. 2003.

[22] S. J. Mellor, A. N. Clark, and T. Futagami, "Model-driven development - Guest editor's introduction," *IEEE Softw.*, vol. 20, no. 5, pp. 14–18, Sep. 2003.

[23] O. Pastor and J. C. Molina, *Model-driven architecture in practice : a software production environment based on conceptual modeling*. Springer, 2007.

[24] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, Jan. 1990.

[25] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel, "Model-Driven Engineering for Software Migration in a Large Industrial Context," in *Lecture Notes in Computer Science*, vol. 4735, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 482–497.

[26] R. Kazman, S. G. Woods, and S. J. Carriere, "Requirements for integrating software architecture and reengineering models: CORUM II," in *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, pp. 154–163.

[27] J. Sánchez Cuadrado, O. Avila García, J. Luis Cánovas Izquierdo, and A. Sánchez-Barbudo Herrera, "Parametrización de las transformaciones horizontales en el modelo de herradura," pp. 551–556, 2012.

[28] V. Khusidman, "SOA Enabled Workflow Modernization," *October*, no. October, pp. 1–10, 2006.

[29] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite, "Reverse Engineering Goal Models from Legacy Code," *RE '05 Proc. 13th IEEE Int. Conf. Requir. Eng.*, pp. 363–372, 2005.

[30] W. Richter and M. Conti, "The oligomerization state determines regulatory properties and inhibitor sensitivity of type 4 cAMP-specific phosphodiesterases," *J. Biol. Chem.*, vol. 279, no. 29, pp. 30338–30348, 2004.

[31] R. Pérez-Castillo, I. García-Rodríguez De Guzmán, O. Ávila-García, and M. Piattini, "MARBLE: Un enfoque ADM para la obtención de Procesos de Negocio," vol. 3, no. 2, 2009.

[32] J. Bisbal, D. Lawless, Bing Wu, and J. Grimson, "Legacy information systems:

issues and directions,” *IEEE Softw.*, vol. 16, no. 5, pp. 103–111, 1999.

[33] Object Management Group, “Knowledge Discovery Metamodel (KDM),” 2016. [Online]. Available: http://www.omg.org/spec/KDM/. [Accessed: 28-Jun-2018].

[34] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, “Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems,” *Comput. Stand. Interfaces*, vol. 33, no. 6, pp. 519–532, Nov. 2011.

[35] J. L. Montero Pozo, “Componente de migración de la lógica de sistemas heredados hacía un esquema funcional para plataforma móvil,” 2015.

[36] CHRISTIAN VERSTRAETE, “Go to Trunk, use the Strangler application approach. – cloudsourceblog,” 2017. [Online]. Available: https://cloudsourceblog.com/2017/02/14/go-to-trunk-use-the-strangler-application-approach/. [Accessed: 28-Jun-2018].