Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

ITINERARIO DE INGENIERÍA DE SOFTWARE
Cód. 31105151 – Trabajo Fin de Máster

MEJORA DEL COMPONENTE GESTIÓN DE CONTRATOS Y PROYECTOS (GCP), MEDIANTE SU REDISEÑO UTILIZANDO ADD Y MDA EN UN PROCESO ÁGIL

Estudiante: Abraham Gimeno García - Consuegra
Director: José Félix Estívariz López
Curso 2016/2017. Convocatoria de defensa: Ordinaria de junio.

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

ITINERARIO DE INGENIERÍA DE SOFTWARE
Cód. 31105151 – Trabajo Fin de Máster

MEJORA DEL COMPONENTE GESTIÓN DE CONTRATOS Y PROYECTOS (GCP), MEDIANTE SU REDISEÑO UTILIZANDO ADD Y MDA EN UN PROCESO ÁGIL

Estudiante: Abraham Gimeno García - Consuegra
Director: José Félix Estívariz López
Curso 2016/2017. Convocatoria de defensa: Ordinaria de junio.

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 21 / 06 / 2017

Quién suscribe:

Autor(a): **ABRAHAM GIMENO GARCIA-CONSUEGRA**
D.N.I/N.I.E/Pasaporte.: **53220171B**

Hace constar que es la autor(a) del trabajo:

Título completo del trabajo.
"**MEJORA DEL COMPONENTE GESTIÓN DE CONTRATOS Y PROYECTOS (GCP), MEDIANTE SU REDISEÑO UTILIZANDO ADD Y MDA EN UN PROCESO ÁGIL**"

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo. **ABRAHAM GIMENO GARCIA-CONSUEGRA**

Impreso TFdM05_Autor. Autorización de publicación

y difusión del TFdM para fines académicos

Autorización

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

Firma del/los Autor/es

ABRAHAM GIMENO GARCIA-CONSUEGRA

Juan del Rosal, 16
28040, Madrid

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

www.issi.uned.es

ÍNDICE

Parte 1. Introducción y objetivos

Este trabajo de fin de máster tiene como objetivo justificar la mejora de los atributos de interoperabilidad y modificabilidad del componente GdPC, parte de una plataforma de licitación electrónica, mediante las modificaciones en su arquitectura. Este rediseño de la arquitectura se realizará tomando como base los conocimientos adquiridos durante el máster y principalmente en la literatura recomendada para el trabajo, el libro "*Software Architecture in Practice*" ([1]). También se utilizará literatura más actual, que refina los métodos introducidos en [1]. El documento está dividido en los siguientes bloques:

- La "**Parte 1**" del trabajo es esta introducción, en la que se describen los objetivos del trabajo y la estructura del documento.
- La "**Parte 2**" que documenta el planteamiento del problema.
- La "**Parte 3**", con el análisis y resolución de la tarea de mejora de la plataforma.
- La "**Parte 4**" en las que se dejará constancia de las conclusiones derivadas de la realización del trabajo.
- La "**Parte 6**" que incluye los anexos con información adicional que se ha utilizado a lo largo del trabajo y que se referencia en este documento.
- La "**Parte 7**" con la lista de referencias utilizadas.

1.1.1 Objetivos

Los objetivos de este trabajo son los siguientes:

1. Rediseño de la arquitectura del componente GPC para la mejora de los atributos de interoperabilidad y modificabilidad.
2. Reducción de los recursos empleados para el rediseño del componente GPC y adecuación a los objetivos de negocio de la empresa de desarrollo, mediante el uso de un proceso compatible con las metodologías ágiles.

Parte 2. Mejora de un Sistema de Licitación Electrónica - Planteamiento del problema

En esta parte del trabajo se planteará la mejora de un componente de una plataforma de licitación, a través del rediseño de su arquitectura. Se ha querido seguir un proceso que sea repetible en un entorno de desarrollo de software industrial. Dice nuestro compañero José Antonio Pérez Campanero en su excelente trabajo de fin de máster [2] que la falta de penetración de las técnicas de arquitectura de software en el mundo industrial se debe al desconocimiento de estas y no a una supuesta desconexión entre la Universidad y la Industria. El autor de este trabajo está en desacuerdo con esta afirmación. En la opinión del que escribe existe, y debe existir, un espacio de desconexión entre la Universidad y la Industria. Simplemente que en el momento adecuado se deben tender los puentes necesarios para que ambos se beneficien de los desarrollos del otro. Es decir, que si existe un desconocimiento de las técnicas de arquitectura software en el sector industrial es porque estas no están diseñadas para sus necesidades, y por lo tanto no se consideran prácticas, o se consideran lo que suele llamar en terminología inglesa un *overhead,* es decir, un coste que no genera un beneficio, al menos inmediato o que se pueda apreciar directamente. Se podría buscar algún informe que indicara la penetración de estas técnicas del diseño de arquitecturas software en el ámbito industrial, pero es suficiente con citar el principio 11 del manifiesto ágil: "*Las mejores arquitecturas, requisitos y diseños* ***emergen*** *de equipos auto-organizados.*" Es toda una declaración de intenciones. Según este principio, las arquitecturas no se diseñan intencionalmente, sino que emergen espontáneamente del trabajo del equipo. Si el principio 11 del manifiesto se combina con el informe "*Capitalize on current IT trends*" [3], donde se muestra que el 91% de las organizaciones o son completamente ágiles o están moviendo hacia la adopción de estas metodologías (con diferentes grados de adopción), vemos claramente cuál es la tendencia de la industria. Aligerar el proceso software eliminando las actividades que se consideran menos importantes, en línea con los postulados de las metodologías ágiles. Por lo tanto, si el diseño intencional de la arquitectura mediante un proceso definido se entiende como redundante, lo normal es que se elimine del proceso software (ya que como dice el principio 11, la mejor arquitectura es la que emerge de forma espontánea).

Por lo tanto, más bien que ser una cuestión de Universidad vs. Industria, parecería que es más bien una cuestión de beneficio percibido. Probablemente, si el Manifiesto Ágil se hubiese originado en el departamento de investigación de una universidad, hubiera encontrado la misma aceptación y rápida expansión en la industria del software, simplemente porque daba respuesta a una necesidad existente en el mundo del desarrollo del software, que no era otra que la de agilizar el proceso software, al menos en algunas circunstancias.

Con esto en mente, en este trabajo se intenta, humildemente, "*tender un puente*" entre la teoría de la arquitectura software que se ha estudiado en el máster y en este curso, para relacionarlo con el conocimiento adquirido de la experiencia profesional en el desarrollo de sistemas de software complejos del que realiza el trabajo. Es decir, se han intentado usar las herramientas y los métodos que se han estudiado para establecer un proceso el cual se considera apto para implementarlo como parte del proceso software de una empresa de la industria del software.

2.1.1 ADD Ágil + MDA

En el informe "*Capitalize on current IT trends*" [3] que se menciona en la sección anterior, publicado por Hewlett Packard, se preguntó a las organizaciones encuestadas que definieran sus metodologías de desarrollo como: completamente ágiles, híbridas, moviendo hacia metodologías ágiles, moviendo hacia metodologías en cascada, completamente en cascada.

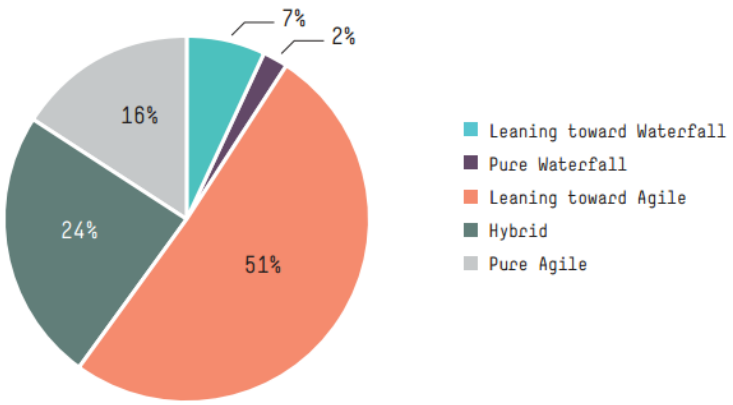

FIGURA 1: *LAS METODOLOGÍAS ÁGILES EN LA INDUSTRIA DEL SOFTWARE* [3]

Como se ve en la figura, el 40% de las organizaciones utilizan en cierto grado las metodologías ágiles (el 16% son completamente ágiles y el 24% las utiliza junto con otras metodologías no ágiles). Del 60% restante, el 51% dice estar moviendo hacia la adopción de metodologías ágiles. Esta fuerte tendencia se explica ya que la implementación de un proyecto utilizando metodologías ágiles es en la mayoría es de 4 a 5 veces más corta, en comparación con otras metodologías tradicionales. Además, la eficiencia del equipo se incrementa en aproximadamente 200-300%, y por lo tanto también aumenta a la par de la eficiencia, la probabilidad de éxito de un proyecto [4]. El famoso informe "Chaos Report" publicado por Standish Group también presenta unos resultados en la misma línea, donde el uso de las metodologías ágiles está asociado a un porcentaje superior de éxito del proyecto, comparado con las metodologías en cascada [5].

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

FIGURA 2: *RESOLUCIÓN DE PROYECTOS AGILE VS. WATERFALL* [5]

Así, aunque el desarrollo de software basado en los principios ágiles no es apropiado para todos los casos, sí parece claro que, viendo el éxito de estas metodologías, se esperaría que cualquier técnica o método que se perciba como lejos de las **metodologías de desarrollo de software ágiles** encontrase resistencia, ya que las organizaciones están embarcadas en la adopción de estas metodologías. Por lo tanto, se esperaría una tímida

adopción de métodos costosos cómo ATAM[1] o Attribute Driven Design, medido en términos de penetración industrial. Los mismos autores de Attribute Driven Desing en [6] dan por válida esta hipótesis cuando reconocen la necesidad de adaptar el método para resolver sus deficiencias, ya que la industria del software ha cambiado dramáticamente desde que se publicó la primera versión. Se reconoce explícitamente que la omisión del diseño de arquitecturas en un entorno ágil era una deficiencia, que es lo que se defiende también en este trabajo y el motivo por el que se ha seleccionado la versión 3.0 de Attribute Driven Design (llamado en este trabajo ADD Ágil) para realizar el diseño de la arquitectura. ADD Ágil permite el diseño de la arquitectura de forma incremental y por lo tanto compatible con las metodologías ágiles.

Es posible que, como en el caso de los autores de [1], haya proyectos se desarrollen en dominios y organizaciones que requieran procesos como ATAM o el clásico ADD, ya que se necesita una planificación detallada de antemano, documentar con exhaustivamente todo el desarrollo del sistema, incluyendo el proceso de decisión. Es decir, hay organizaciones que requieren estas metodologías que en otras circunstancias se considerarían pesadas. Este será probablemente el caso en la NASA (donde los autores firman varios trabajos de investigación) o en los contratos con el Departamento de Defensa de los EEUU, donde parece que nacen estas técnicas (hay que recordar que el Software Engineering Institute de la universidad de Carnegie Mellon está fundado por el Departamento de Defensa de los EEUU).

Pero esta no es la realidad a la que se enfrentan la mayoría de las organizaciones, como ya se ha mostrado. Ni siquiera los proyectos realizados en ámbitos públicos, bien sean de carácter nacional o internacional, son ya tan estrictos en la documentación y el proceso de especificación y diseño de la arquitectura, como el descrito en [1]. Al contrario, en los últimos años hay una tendencia clara a reducir costes lo máximo posible [7] . Esta reducción de costes hace que muchos organismos públicos se decanten por paquetes de soluciones (en un modelo SaaS), evitando desarrollar un sistema adaptado a sus necesidades, para así reducir el coste. Un ejemplo de esto es el Acuerdo Marco en el Reino Unido, G-Cloud, que ofrece a los organismos públicos británicos una colección de soluciones en la nube. En estas circunstancias, ni siquiera hay un proceso de diseño de la arquitectura ya que la solución ya está desarrollada y simplemente se realizan modificaciones menores.

Así, por un lado, tenemos la irrupción de las metodologías que agilizan el proceso software y por el otro la presión de reducir costes por parte, no solo de la industria software como sería de esperar, sino también de sectores, como el de la administración pública, que tradicionalmente exigía procesos más estandarizados y evaluables. ¿Hay entonces espacio para el diseño de la arquitectura en el entorno de desarrollo actual que se mueve hacia la agilidad? Mucha gente lo considera incompatible. En este trabajo se defiende que sí, que es posible, en línea con la teoría estudiada en el máster. La idea en la que se basa en este trabajo es que se pueden utilizar métodos de diseño de la arquitectura software, pero en lugar de diseñar toda la arquitectura del sistema de antemano, realizar un diseño incremental. Además, si se realiza este diseño utilizando herramientas de modelado MDA, se consigue que el diseño de la arquitectura pase a formar parte del proceso de desarrollo y no sea relegado a, meramente, un ejercicio de documentación. Esto es así porque a partir de los modelos se genera el código fuente del sistema. Este modelo no es necesario que describa el sistema finalizado. Es suficiente con modelar un esqueleto de la arquitectura seleccionada, sobre el cual los desarrolladores comenzarían a desarrollar sistema. En iteraciones subsiguientes se puede realizar un diseño más detallado, de acuerdo con el progreso del proyecto. Esta es la estrategia, propuesta por Alistair Cockburn, una de las firmantes del manifiesto ágil en [8], que sugiere realizar un proceso de diseño de la arquitectura que sea iterativo e incremental, y por ende más ágil.

[1] ATAM según [1] requiere esfuerzos de 1 pmonth, además de la participación de evaluadores externos a la empresa/ equipo, etc.

2.1.2 El proceso definido

Para realizar el rediseño de la arquitectura de GPC se van a seguir los siguientes pasos:

1. ***Lightweight ATAM:*** Presentación de los motivos de negocio
2. ***Lightweight ATAM:*** Presentación de la arquitectura
3. ***Lightweight ATAM:*** Análisis de las estrategias de arquitectura existentes
4. ***Lightweight ATAM:*** Requisitos arquitectónicamente significativos
5. ***Lightweight ATAM:*** Evaluación de la arquitectura existente usando Como salida se identificará una lista de cambios de arquitectura
6. ***Lightweight ATAM:*** Priorización de cambios deseados
7. ***ADD:*** Análisis de las estrategias de arquitectura
8. ***ADD:*** Evaluación de las estrategias usando CBAM
9. Diseño incremental de la solución con una herramienta MDA

Parte 3. Mejora de un Sistema de Licitación Electrónica - Análisis de las soluciones y resolución

Capítulo 1 Arquitectura software de GPC

Hay conceptos que no resultan demasiado naturales y que para tener una idea se necesita experimentar con ellos y trabajarlos. Por ejemplo, espacios de 5 o 6 dimensiones no es algo que la mayoría de personas tengamos una intuición de forma natural. Por el contrario, si preguntamos a alguien que pasa por la calle, ¿qué es la arquitectura?, seguro que tiene una intuición natural y probablemente darían respuestas en la dirección correcta, apuntando a la planificación, diseño y construcción de edificios u otras estructuras físicas.

Así que podría pensarse que es un ejercicio relativamente sencillo definir el concepto de arquitectura del software. La realidad es que, debido a las características del software, la arquitectura software no es tan intuitiva como, por ejemplo, la arquitectura tradicional de construcción edificios. Aunque es cierto que las dos modelan construcciones complejas descomponiéndolas en sus estructuras básicas y definiendo sus relaciones y atributos, la arquitectura tradicional se compone de elementos tangibles gobernados por las leyes de la física mientras que la arquitectura del software trata elementos intangibles, lo que hace su representación y estudio más complicado.

La definición del libro "Software Architecture in Practice" ([1]) es: "*la arquitectura de software es la estructura o estructuras del sistema necesarias para razonar sobre el sistema, que están compuestas por los elementos software, sus relaciones y las propiedades de dichos elementos y sus relaciones*.". De esta definición sacamos 4 ideas importantes:

1. Todo sistema de software tiene una arquitectura. Otra cosa es si ésta es conocida o no, si se ha diseñado intencionalmente o por el contrario ha aparecido por casualidad.
2. La arquitectura software permite razonar de forma efectiva sobre las sobre las **propiedades visibles externamente** de un sistema y, por ende, habilita el análisis y mejora de los sistemas.
3. Los conjuntos de estructuras de software del sistema constituyen la arquitectura software. Para poderla cualificar como estructura de software, ésta tiene que servir para ayudar al razonamiento del sistema[2].
4. La arquitectura es una **abstracción**. Los elementos del sistema que no ayudan al razonamiento del sistema, se omiten de forma intencionada.

3.1.1 Arquitectura software: la disciplina

La arquitectura del software como disciplina se centra en el diseño, análisis, selección y documentación de estas arquitecturas. Como en muchas otras disciplinas, no existe una definición unánime sobre lo que es la arquitectura software. Según [1] la Arquitectura Software está compuesta por las vistas del sistema que incluye los componentes principales, el comportamiento de esos componentes según las formas en que los componentes interactúan y se coordinan para alcanzar el objetivo para el cual el sistema fue desarrollado. Las vistas de la arquitectura software son abstractas, lo que ayuda a comprender ciertos elementos/ interacciones de forma más efectiva, omitiendo la información que no es relevante. Es por ello que se

[2] Por ejemplo, *la lista de las líneas de código que incluyen la letra z* no proporciona información alguna que permita razonar sobre el sistema mientras por el contrario un *diagrama de clases* sí que lo haría.

necesita una colección de vistas diferentes para describir el sistema totalmente. Además, la arquitectura software nos proporciona un catálogo de patrones reusables de arquitectura.

3.1.1.1 La importancia de la arquitectura software

Como ya hemos comentado en la sección anterior, la arquitectura software está presente en cualquier sistema de software, aunque no esté documentada o no se haya seguido un proceso estandarizado para su diseño y documentación. Con el auge de las metodologías ágiles, quizás las palabras "*proceso estandarizado*" y "*documentación*" puedan poner a alguien en alerta. Sea alguien uno de los llamados "evangelizadores" de las metodologías ágiles o no, lo cierto es que siempre es saludable preguntarse: ¿Realmente necesito realizar esta actividad? Aplicado al tema de este trabajo, ¿es realmente necesario hacer un diseño intencional y documentar la arquitectura software? ¿Es algo importante? ¿Añade valor?

Como parte de este trabajo, no solo intentaremos explicar la importancia que tiene el diseñar y documentar intencionalmente la arquitectura, sino que también se mostrará un caso práctico que justificará la mejora de una aplicación después de modificar su arquitectura usando las técnicas y herramientas que nos proporciona la disciplina de la arquitectura del software.

En [1] se listan las siguientes 13 de las razones técnicas[3] más importantes que justificarían que el diseño de la arquitectura deba considerarse un elemento importante del desarrollo de un sistema de software.

1. La arquitectura inhibirá o potenciará los atributos de calidad de un sistema.
2. El análisis de la arquitectura permite la predicción temprana de las cualidades de un sistema.
3. Las decisiones tomadas en una arquitectura permiten razonar acerca y gestionar el cambio a la vez que el sistema evoluciona.
4. La arquitectura puede ser la base para crear prototipos evolutivos.
5. Una arquitectura se puede crear como un modelo transferible, reutilizable que forma el corazón de una línea de productos.
6. El desarrollo basado en la arquitectura centra la atención en cómo se ensamblan los diferentes componentes, en lugar de en su creación.
7. Al restringir las alternativas de diseño, potencia la creatividad de los desarrolladores, lo que reduce la complejidad del sistema.
8. La arquitectura define un conjunto de restricciones que tendrán vigencia en implementaciones subsecuentes.
9. Una arquitectura correctamente documentada mejora la comunicación entre las partes interesadas.
10. La arquitectura documenta las decisiones de diseño críticas del sistema. Con lo que es portadora de una información fundamental.
11. La arquitectura dicta la estructura de una organización, o viceversa.
12. Una arquitectura es el artefacto clave que permite al arquitecto y al director del proyecto razonar acerca de los gastos y el calendario del proyecto.
13. La arquitectura puede ser la base para la formación de un nuevo miembro del equipo.

De ellas se pueden extrapolar 2 categorías principales y decir que la arquitectura software es importante porque:

1. Predictibilidad: Que un sistema sea predecible en sus atributos ayuda a elegir la arquitectura adecuada para nuestros requisitos de calidad y por lo tanto produce mejores sistemas (1-8).

[3] Las 13 razones se presentan en un orden diferente a como se presentan en el libro

2. Comunicación: La documentación de la arquitectura facilita las labores de comunicación y razonamiento sobre el sistema (9-13).

3.1.1.2 Adecuabilidad

En las categorías que hemos mencionado arriba se ha introducido implícitamente el concepto de buena, y por ende también el de mala, arquitectura. Hemos dicho que el diseño intencional de la arquitectura software "*produce mejores sistemas*" (o al menos debería). Lo cierto que es que no hay arquitecturas inherentemente buenas o malas, simplemente hay mejores y peores, dependiendo primero, de lo adecuada que es para su objetivo.

Además de la relatividad de las arquitecturas, tampoco simplifica las decisiones el hecho de que no suele haber una única buena arquitectura. Lo normal es encontrarse con un conjunto de arquitecturas que son más o menos buenas. ¿Cómo decidimos entonces cuál utilizar? Aquí es donde entran lo que podíamos denominar como criterios económicos. Siempre teniendo en cuenta que se cumplen requisitos funcionales y no funcionales del sistema, en teoría, habría que escoger la que confiera al producto final unas características cualitativas que otorguen el máximo número de ventajas comerciales, estratégicas o competitivas. Estas ventajas constituyen, de forma directa o indirecta, un beneficio económico, sea por la reducción del tiempo para lanzar al mercado el sistema/ funcionalidad (time-to-market) y/ o en una reducción de costes de desarrollo/ mantenimiento.

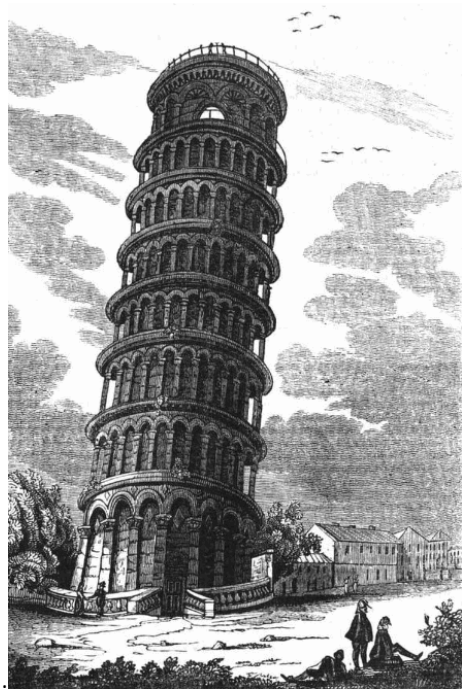

FIGURA 3: *LA TORRE INCLINADA DE PISA*

Para ilustrar esto podemos pensar en la torre inclinada de Pisa. ¿Es el resultado de una buena arquitectura? Si el objetivo era crear un campanario que aguantara erguido unos cuantos meses, probablemente sí[4]. Por

[4] Según el mito, Galileo Galilei pareció encontrarle la utilidad a la inclinación de la torre (se dice que dejó caer dos balas de cañón de diferente masa desde la torre para demostrar que la velocidad de descenso era independiente de la masa).

el contrario, si lo que se pretendía es crear un campanario erecto que permaneciera impertérrito al paso del tiempo, la arquitectura fue realmente mala.

3.1.1.3 La importancia económica de la arquitectura software

Intuitivamente, con analogías como la de la torre de Pisa u otros fracasos debidos a arquitecturas deficientes se ve muy claro la importancia de la arquitectura. ¿Por qué entonces se necesita invertir tanto tiempo en tratar de justificar la importancia de la arquitectura software? La respuesta es sencilla. Cuando se completaron los 3 primeros pisos de la torre de Pisa, enseguida se dieron cuenta que la torre estaba inclinada (se dice que para entonces ya se había hundido en el suelo apreciablemente, entre 30 y 40 centímetros, y que tenía ya una inclinación de unos de 5 centímetros [9]) y su construcción se detuvo. Aunque la construcción se reanudó, la inclinación de la torre de Pisa era (y todavía es) evidente a cualquier observador. Esto no sucede con el software, que se construye de elementos intangibles, por lo que no se ve tan claramente si la "torre" está inclinada, no se suele detectar ningún problema (por parte del usuario) siempre y cuando suenen las campanas (es decir mientras el software cumpla su cometido). Por lo tanto, la tentación de eliminar o minimizar la importancia de la arquitectura software en el proceso de desarrollo software es mucho más grande y el cortoplacismo entra en juego con fuerza.

Al fin y al cabo, diseñar y documentar una arquitectura tiene un coste asociado. ¿Por qué invertir en una arquitectura que no va a ser visible? ¿Por qué ocuparse de algo que parece auxiliar o tangente? La respuesta es sencilla y a la vez compleja. Decimos sencilla porque basta con decir que diseñar la arquitectura de forma correcta reducirá el coste del proyecto. O al menos así debería ser para poder justificar la selección de una arquitectura particular. Y es a la vez compleja porque no es tan trivial el trazar una relación que muestre como estas características cualitativas se traducen al final en una ventaja comercial, estratégica o competitiva. Es decir, no es fácil identificar el beneficio proporcional al esfuerzo, lo que en el idioma inglés se conoce como Return On Investment (ROI).

Para entender esto con un ejemplo, vamos a volver a la torre inclinada de Pisa y asumir que el objetivo del proyecto era disponer de un campanario que diese las horas. Suponemos que la inversión en diseñar la arquitectura de forma correcta hubiese sido 100 mil euros. El diseño defectuoso actual costó 10 mil euros y mantener la torre en pie y funcionando hasta el día de hoy ha costado un total de 5 mil euros. Con estos parámetros podríamos argumentar que no estaba justificado invertir más tiempo en diseñar la arquitectura, ya que la torre lleva en pie y funcionando cientos de años con un mínimo gasto (15 mil euros). Diseñar una arquitectura que mantuviese el campanario erecto y sin generar ningún gasto adicional hubiera costado 100 mil euros. Por lo tanto, se hubieran gastado innecesariamente, tomando los requisitos de partida, 85 mil euros. ¿Por qué invertir tanto dinero en mantenerla erecta, cuando torcida sigue dando las horas y es mucho más barata?

Es evidente que el ejemplo que hemos usado dista mucho de ser real, pero ilustra el dilema que se tiene ante cualquier inversión. La realidad, incluyendo el ejemplo de la torre de Pisa, es que los fallos en fases de diseño son más baratos y fáciles de corregir que en fases posteriores. Como ya se ha comentado, el problema de la producción de software es que cuenta con la dificultad de cuantificar el ROI dadas diferentes arquitecturas. No obstante, se han publicado diferentes estudios que confirman la intuición que exportamos de otras disciplinas de la ingeniería, y es que: diferentes arquitecturas tienen diferentes ROI. O lo que es lo mismo, que hay arquitecturas malas, regulares, buenas y buenísimas. En [10] por ejemplo, se cuantifica el ROI para diferentes estrategias de arquitectura (AS) en un desarrollo software en el proyecto ECS, realizado en la NASA.

AS	Cost	Total AS Benefit	AS ROI	AS Rank
1	1200	950	0.79	1
4	200	100	0.5	3
5	400	120	0.3	7
6	200	50	0.25	8
7	200	70	0.35	6

TABLA 1: *ROI EN LAS DIFERENTES ESTRATEGIAS ARQUITECTÓNICAS PARA EL PROYECTO ECS* [10]

Es interesante para observar el impacto de diferentes estrategias de arquitectura. Por ejemplo, vemos que AS4 y AS6 tienen el mismo coste de inversión, pero el ROI AS4 dobla al de AS6. Por otro lado, la arquitectura con mayor ROI es AS1, la que mayor coste de inversión tiene asociado. En un entorno empresarial, sin una forma metodológica de seleccionar la arquitectura software, se tendería a elegir entre las de menor coste (AS4, AS6 y AS7) y a dejar de lado AS1.

3.1.1.4 Diseñando "buenas" arquitecturas

Aunque no hay arquitecturas que sean inherentemente buenas o malas, sino que más bien depende del grado en el que satisfacen los requisitos definidos, en este capítulo se proporcionan las recomendaciones del libro, que son de aplicación general (rule-of-thumb) que se necesitan para crear buenas arquitecturas. No observar alguna de estas reglas no significa que la arquitectura resultante sea en todos casos incorrecta o tenga algún defecto, pero sí debería verse como una señal de advertencia.

Recomendaciones de proceso:

1. La arquitectura debería ser el producto de uno o más arquitectos y el director técnico. Esto le dará integridad conceptual y consistencia técnica.
2. El arquitecto debería, de forma regular, basar la arquitectura en unos atributos de calidad especificados claramente. La funcionalidad no es tan importante como estos atributos.
3. La arquitectura debería ser documentada usando vistas, que incluirían las preocupaciones de los stakeholders.
4. Se debería evaluar la arquitectura para ver si puede cumplir con los atributos de calidad definidos.
5. Debe ser posible especificar la arquitectura de forma incremental. Se puede crear un esqueleto con la comunicación necesaria pero que tiene mínima funcionalidad.

Recomendaciones estructurales:

1. Hay que definir correctamente los módulos en la arquitectura, teniendo en cuenta las responsabilidades funcionales, basándonos en los principios de encapsulación de la información (se encapsulan cosas que es probable que cambien) y separación de responsabilidades. Esto también permite que equipos trabajen de forma independiente.
2. Lo más normal es que los atributos de calidad puedan describirse con un patrón de arquitectura ya definido.
3. La arquitectura nunca debería depender de una versión del producto.
4. Módulos que producen datos deben separarse de los módulos que consumen datos.
5. No hay que esperar relaciones uno a uno entre los componentes. Sistemas donde se utiliza concurrencia, etc.
6. Los procesos deben ser independientes del procesado
7. La arquitectura debe especificar un número pequeño de maneras de interactuación entre componentes.
8. La arquitectura debe incluir un conjunto específico (y pequeño) de áreas de contención de recursos.

3.1.1.5 Requisitos con impacto en la arquitectura (ASR)

Los requisitos arquitectónicamente significativos son obviamente importantes por su impacto en la arquitectura a elegir. Por este motivo es importante identificarlos previamente, para así incluirlos en el proceso de decisión. Un requisito con impacto en la arquitectura tiene las siguientes características:

- Un profundo impacto en la arquitectura. Es decir, que probablemente se elegiría otro tipo de arquitectura si no fuese por la existencia de ese requisito.
- Un valor importante desde el punto de vista de los objetivos de negocio. Si se va a diseñar la arquitectura para que cumpla el requisito, quiere decir que, en un principio, se estaría dispuesto a renunciar a otros requisitos si fuese necesario. Por lo tanto, este es un requisito importante para los stakeholders.

3.1.2 GPC - Motivaciones de negocio y requisitos arquitectónicamente significativos (ASRs)

Como ya se ha visto, es importante a la hora de realizar el diseño de la arquitectura conocer las motivaciones de negocio ya que de ellas se derivan los requisitos arquitectónicamente significativos que influencian el proceso de decisión. El componente GPC (Gestión de Proyectos y Contratos) sobre el cual se realiza este trabajo, forma parte de una plataforma de licitación electrónica eLP (Licitación Pública electrónica). Esta plataforma proporciona los servicios necesarios para el proceso de compras público completo, desde la gestión de proveedores y sus perfiles hasta la gestión del contrato y su cierre. GPC se centra en los servicios necesarios para la gestión del contrato, una vez concluye el proceso de licitación, eligiendo al proveedor que se le adjudica el contrato. La misión de este sistema es asistir al usuario en el proceso de finalización y firma del contrato a través del sistema, así como la posterior gestión y monitorización. Estos procesos incluyen la gestión de documentos, gestión los niveles de servicio, hitos, gestión de incidencias, entregables del proyecto, tareas, información financiera, motor para la generación de informes, etc. Dependiendo de las necesidades, las organizaciones que procuran este tipo de plataformas puede que simplemente requieran un sistema de gestión de contratos o por el contrario un sistema que cubra todo el proceso de contratación.

Por lo tanto, se pueden identificar los principales objetivos de negocio que condujeron la implementación de GPC:

- Disponer de una plataforma que soporte el ciclo completo de contratación pública.
- Disponer de flexibilidad para ofrecer este componente de forma independiente.
- Minimizar el coste de desarrollo (Ej. preferencia por reutilizar sistemas existentes).

3.1.2.1 GPC - Requisitos arquitectónicamente significativos (ASRs)

De los objetivos de negocio se derivan los siguientes requisitos arquitectónicos significativos para el componente GPC:

1. Alta modificabilidad necesaria ya que las metodologías de gestión de contrato no están legisladas (al contrario que el proceso de licitación) y por lo tanto cambian ostensiblemente de organización en organización.
2. Alta interoperabilidad en los módulos de Gestión de Eventos, Gestión de Incidencias y la parte financiera de Gestión del Contrato, ya que esta información con frecuencia se genera en otros sistemas (e.g. JIRA, SAP, etc.).
3. EL sistema existente está construido con estas tecnologías J2EE.
4. Los módulos deben utilizar tecnologías y arquitecturas que sean clusterizables, ya que la plataforma y sus componentes deben desplegarse en esos entornos de producción.

3.1.3 GPC - Presentación de la arquitectura

Teniendo en cuenta los objetivos de negocio, se decidió que GPC se desarrollara como resultado de adaptar una herramienta de colaboración en la creación de publicaciones, que ya existía en la organización. Este sistema disponía de un gestor documental y de roles, y alrededor de estos se desarrolló la funcionalidad mínima para proporcionar un gestor de proyectos y contratos, sin realmente modificar la arquitectura existente, que fue diseñada para otros objetivos de negocio diferentes.

GPC está integrado en la plataforma a través de un componente de SSO, que permite que se autentifique solo una vez. El esquema de componentes que compone la plataforma de licitación electrónica se muestra en la siguiente imagen

- *SSO (Single Sign On)*: Componente que gestiona las sesiones de usuarios en los otros componentes de la plataforma.
- *eLP (e-Licitación Pública)*: Componente que proporciona la funcionalidad asociada con el proceso de licitación.
- *FFM (Gestión de Formularios)*: Componente que gestiona los diferentes documentos que se han de utilizar para los anuncios de licitación.
- *GPC (Gestión de proyectos y contratos)*: Componente que proporciona la funcionalidad para gestionar los contratos y proyectos.
- *IdN (Inteligencia de Negocio)*: Componente que opera sobre un almacén de datos y proporciona la funcionalidad necesaria para realizar análisis de datos, informes e inteligencia de negocio.

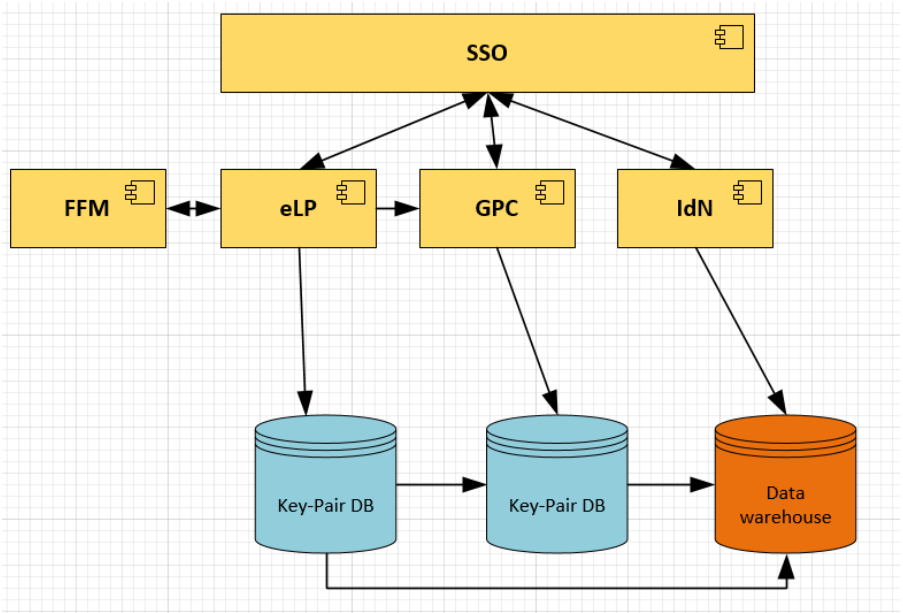

FIGURA 4: *ARQUITECTURA DE LA PLATAFORMA DE LICITACIÓN*

3.1.3.1 El componente de Gestión de Proyectos y Contratos

El componente de Gestión de Proyectos y Contratos está formado por los siguientes módulos:

- Administración: este módulo se encarga de la administración de los espacios de trabajo y de los roles de usuario.
- Gestor de Documentos: se encarga de gestionar los documentos que se almacenan en el espacio de trabajo del contrato. Incluye funcionalidad para control de versiones de los documentos y para la revisión de documentos.

- Gestión de Eventos: proporciona una funcionalidad que permite la gestión de los eventos del contrato (hitos, reuniones y eventos).
- Gestión de Incidencias: permite registrar y gestionar las incidencias del contrato.
- Gestión y Seguimiento del Contrato: proporciona funcionalidad para gestionar las fechas claves (Ej. fecha que expira, renovaciones, etc.) entregables, pagos y presupuesto del contrato. Además, proporciona funcionalidad para el registro y gestión de los indicadores de calidad e indicadores de desempeño.
- Informes: este módulo permite generar informes basados en los datos de los contratos existentes.
- Auditoría: registro con las acciones de los usuarios en el sistema

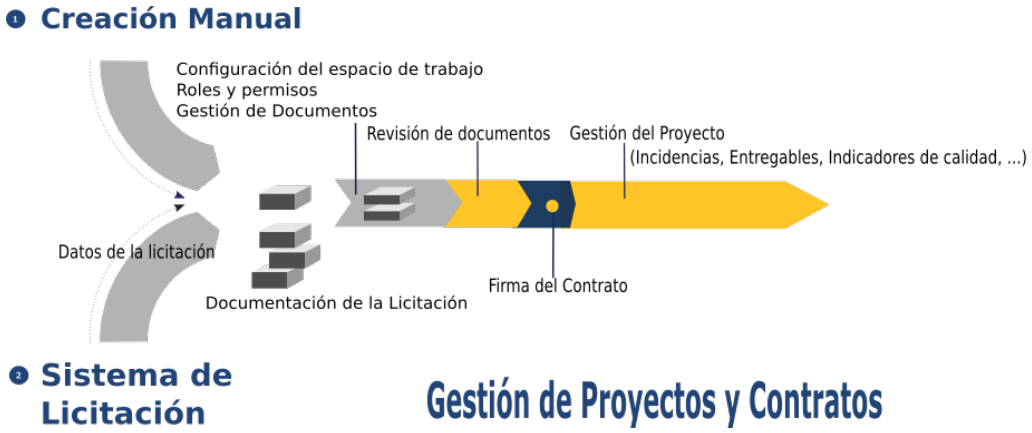

FIGURA 5: *MÓDULOS GPYC EN EL CICLO DE VIDA DE UN CONTRATO*

En el anexo a este documento se pueden consultar los casos de uso asociados a la funcionalidad existente.

3.1.4 Patrones arquitectónicos

Un patrón de arquitectura es un conjunto de tácticas operando juntas para solucionar un problema conocido. Los patrones aparecen para ayudar a navegar la complejidad del espacio de posibilidades a las tácticas, ya que en teoría no hay un límite particular para en número de estrategias que se pueden seguir para influenciar/ determinar un atributo de calidad. Es como los números primos. Sabemos que hay un número infinito de números primos, pero solo conocemos una parte de ellos. Cualquier número que se demuestre que tiene los atributos de un número primo, es un número primo. Lo mismo ocurre con las tácticas, teóricamente hay un número infinito de ellas, cualquier táctica que influya en el atributo de calidad se puede considerar como válida de acuerdo a la definición, pero en la práctica solo conocemos algunas y solo algunas de las conocidas serán útiles para el contexto. Por lo tanto, este espacio multidimensional de posibilidades es complejo y los patrones ayudar a navegar esta complejidad. Esto ha hecho que en la práctica se haya remplazado el diseño de la arquitectura con la selección, composición personalización y comprensión de los patrones existentes.

3.1.4.1 Los puentes de Bailey como analogía

Antes de entrar en los patrones de arquitectura software específicos vamos a introducir un ejemplo análogo de otra disciplina, para facilitar la comprensión. Este es el Puente Bailey, un tipo de puente portátil, modular, prefabricado. Fue desarrollado por los británicos durante la Segunda Guerra Mundial para uso exclusivamente militar y fue utilizado ampliamente por los Cuerpos de Ingenieros ingleses, canadienses y americanos en las operaciones de guerra. Fue calificado por el General Eisenhower como uno de los tres avances técnicos más importantes durante la Segunda Guerra Mundial, junto con el radar o los bombarderos pesados [11]. Es un buen ejemplo para ilustrar la relación entre las tácticas y los patrones de arquitectura.

Podemos considerar el puente de Bailey como un patrón de arquitectura. Es una composición de diferentes elementos que debía cumplir los siguientes requisitos:

1. Flexibilidad para crear puentes de diferentes longitudes, con diferentes niveles de suspensión y que pudiesen ser reforzados in-situ.
2. Todas las partes debían estar hechas de materiales fácilmente accesibles, sin requerir soldaduras y sin partes de aluminio.
3. Todas las partes debían poder fabricarse usando prácticas de ingeniería estándar excluyendo prácticas que resultaran en una tolerancia demasiado exigente. Esto permitiría la interoperabilidad (ensamblar piezas fabricadas por diferentes fabricantes).
4. Todas las piezas debían caber en un camión militar de 3 toneladas y no exceder 272 kg. (600 libras) o no poder ser levantadas por 6 hombres.
5. Debía ser fácil y rápido de montar.

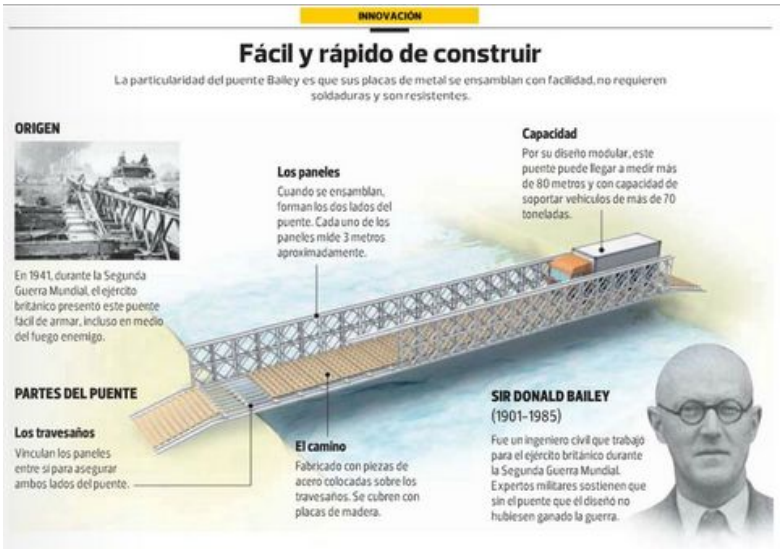

FIGURA 6: *EL PUENTE DE BAILEY* [12]

Aunque en la imagen de arriba se simplifica un poco las partes que componen el puente, en [11] se identifican al menos 28 componentes estándar, es suficiente para trazar una analogía con la arquitectura software. Los bloques básicos de construcción del puente, entre los cuales estarían los paneles o los travesaños que se identifican en la imagen de arriba, constituirían lo que denominamos en arquitectura software **tácticas**.

El puente de Bailey, como **patrón de arquitectura**, especifica como las diferentes partes del puente han de ensamblarse (lo que llamamos interacciones en la arquitectura software) y ya tiene asociada una lógica u objetivo particular, ya que el puente se diseñó para satisfacer los requisitos que hemos mencionado arriba.

En las siguientes subsecciones se introducen algunos de los patrones arquitectónicos relevantes para el rediseño de la arquitectura que se realiza en este trabajo de fin de máster.

3.1.5 GPC - Análisis de las estrategias utilizadas

En esta sección analizaremos las estrategias de arquitectura utilizadas actualmente en el componente GPC.

3.1.5.1 Cliente-Servidor

El componente es una aplicación web, por lo que utiliza una arquitectura Cliente-Servidor. Se despliega en servidores Java EE como JBoss o WebLogic.

3.1.5.2 Layered architecture

El patrón Layered es una forma de organización lógica de los módulos de un sistema en diferentes capas, que se comunican entre sí. La principal característica es que una capa solo se suele comunicar con la capa adyacente. Estas capas pueden estar compuestas por varios módulos. Un ejemplo de una arquitectura organizada en capas es J2EE.

Como se define en [13], J2EE es una plataforma de programación—parte de la Plataforma Java—para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. Permite utilizar arquitecturas de N capas distribuidas y se apoya ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones.

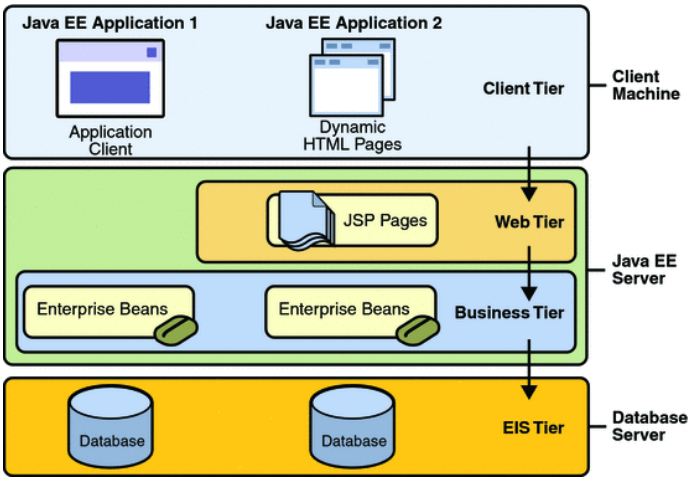

FIGURA 7: *ARQUITECTURA EN CAPAS DE J2EE*

Este patrón de arquitectura hace uso de la encapsulación para proporcionar niveles adicionales de abstracción (cada capa es un nivel de abstracción). Las responsabilidades se agrupan de forma cohesiva detrás de cada capa. La comunicación entre capas se realiza de forma ordenada a través de interfaces públicas con lo que se minimiza el impacto de cambios realizados en una capa en particular. En términos de atributos de calidad, el patrón permite desarrollar sistemas con altos niveles de modificabilidad y usabilidad. Por el contrario, cuando se utiliza esta arquitectura se obtiene un menor desempeño y mayor latencia.

Aunque el componente utiliza J2EE, que proporciona una arquitectura en capas, que son las capas que se muestran en la Figura 6.

3.1.5.3 Key-value database

El componente utiliza un sistema de almacenamiento key-value, es decir que no utiliza una base de datos relacional. Este tipo de bases de datos se utiliza en arquitecturas que necesitan una baja latencia, ya que la simplicidad de la base de datos hace más fácil la sincronización y actualización de las diferentes copias utilizadas en soluciones desplegadas en clústeres de servidores.

3.1.5.4 MVC (Modelo Vista Controlador)

En GPC se utiliza el patrón de arquitectura Modelo-Vista-Controlador para eliminar la dependencia entre el modelo de datos y la vista de los datos. El Modelo se implementa usando Java Persistence, Enterprise Java Beans se usan para la implementación del Controlador. La Vista se implementa utilizando Java Server Pages y Java Tags, junto con estilos en cascada (CSS), Javascript y otras tecnologías web.

En MVC se dividen los objetos de dominio que modelan nuestra percepción del mundo real (el Modelo), y los objetos que forman parte de la interfaz gráfica con los que interactúa el usuario (Vista). El Controlador

actúa de intermediario entre esos dos. Los objetos de la interfaz gráfica generan ciertos eventos que son gestionados por el controlador para actualizar el modelo, la vista, o ambos.

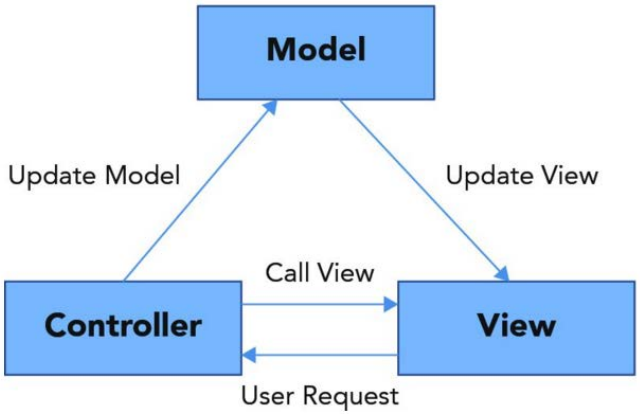

FIGURA 8: *REPRESENTACIÓN DEL MVC A UN ALTO NIVEL DE ABSTRACCIÓN*

Es interesante resaltar que:

- Tanto la vista como el controlador dependen del modelo.
- El modelo no depende ni de la vista ni del controlador.
- El modelo actualiza la vista

3.1.6 Atributos de calidad de un sistema

Unas de las principales hipótesis[5] que presenta [1] es que la mayor causa de rediseño de sistemas no tiene que ver con la funcionalidad sino para solucionar problemas de calidad (son demasiado lentos, demasiado costosos de mantener, etc.). De hecho, se ilustra esto diciendo que un sistema con los mismos requisitos funcionales puede ser diseñado con arquitecturas completamente diferentes. Por lo tanto, diseñar sistemas alrededor de los requisitos funcionales como elemento principal es cortoplacista. Lo que se debe hacer es diseñar la arquitectura en base a los atributos de calidad requeridos en el sistema.

¿Qué es entonces un atributo de calidad? Un **atributo de calidad** es una propiedad no ambigua, medible o verificable (es decir observable) que se usa para indicar hasta qué grado se satisfacen los requisitos de los stakeholders. Aunque diferentes disciplinas definen sus propios atributos, y con frecuencia se solapan, el libro nos proporciona estas 3 categorías para clasificar los requisitos de un sistema:

1. **Funcionales**: especifican que tiene que hacer el sistema y cómo reacciona ante acciones en tiempo de ejecución.
2. **Calidad**: propiedad medible o verificable (Ej. la velocidad a la que debe ejecutarse una determinada operación, etc.)
3. **Limitaciones**: Decisiones con 0 grados de libertad. (Ej. el lenguaje de programación a utilizar, reutilizar cierto módulo, etc.)

La "respuesta" de la arquitectura a esos tipos de requisitos es:

1. **Funcionales**: se satisfacen asignando la secuencia apropiada de responsabilidades en el diseño.

[5] Lo denominamos premisa porque no hay en el libro de esta asignatura ninguna referencia a algún estudio que verifique esto de forma empírica.

2. **Calidad**: se satisfacen a través de las estructuras de software diseñadas.
3. **Limitaciones**: se satisfacen a través de las decisiones de diseño.

3.1.6.1 Funcionalidad y atributos de calidad

Aunque se ha argumentado que los requisitos funcionales pueden ser completamente independientes de los requisitos de calidad, a nivel práctico no es así la gran mayoría de las veces. En la práctica, la funcionalidad se organiza alrededor de las responsabilidades (por ejemplo, para permitir que diferentes equipos puedan trabajar en paralelo, etc.). Así que el diseño de la arquitectura implica crear y asignar responsabilidades a diferentes estructuras software (módulos, etc.). Si no fuese así, si no hubiese esta organización estaríamos implementando lo que se conoce en la jerga de la ingeniería del software como un monolito, o lo que es lo mismo, un sistema sin estructura modular.

Además, los atributos de calidad no son independientes entre sí, sino ciertos atributos están interrelacionados. No es que sus niveles sean inversamente proporcionales, pero sí ocurre que, al aumentar el nivel de un atributo de calidad, disminuye el nivel de otro. Así que para conseguir un nivel aceptable en ciertos atributos sin duda habrá que hacer concesiones en otros.

3.1.6.2 Escenario de Atributos de Calidad - Especificando los atributos de calidad requeridos

Para estandarizar el proceso de diseño y captura de los requisitos de calidad, el libro sugiere crear escenarios de atributos de calidad donde se especifiquen los atributos de calidad. Para especificarlo el libro sugiere un crear Escenario de Atributos de Calidad (QAS) que deberían estar compuesto por los siguientes elementos:

1. **Origen del estímulo**: La entidad que genera el estímulo (Ej. un humano, un computador, un sistema, etc.).
2. **Estímulo**: La condición que requiere una respuesta por parte del sistema.
3. **Entorno**: Las condiciones bajo las cuáles se produce el estímulo. Modo normal, saturado, etc.
4. **Artefacto**: el artefacto que se estimula. La mayoría de las veces será el sistema en sí mismo, pero se pueden especificar aquí partes específicas.
5. **Respuesta**: la actividad que hay que realizar como respuesta al estímulo.
6. **Métrica de la respuesta**: la respuesta debe ser medible para que se pueda verificar el requisito.

Habría que definir por un lado los "*escenarios generales*", que serían independientes del sistema y por otro lado los "*escenarios concretos*" que serían específicos para el sistema que se está diseñando.

Escenario de Atributo de Calidad	
Origen	*Quién o qué ...*
Estímulo	*hace algo ...*
Artefacto	*al sistema del que es parte ...*
Entorno	*bajo unas condiciones ...*
Respuesta	*El sistema reacciona con las siguientes acciones ...*
Metrica	*las cuales pueden ser medidas con las siguientes métricas ...*

TABLA 2: *PLANTILLA GENÉRICA PARA ESPECIFICACIÓN DE ESCENARIOS DE ATRIBUTO DE CALIDAD*

3.1.6.3 La relación entre casos de uso y QAS

Como hemos comentado con anterioridad, los sistemas se suelen diseñar alrededor de los requisitos funcionales. Por lo tanto, es útil en este punto trazar una relación entre casos de uso y los QAS ya que en la práctica es más habitual encontrarse con casos de uso. ¿Cómo podemos relacionarlos? Una opción es añadir la información de calidad al caso de uso. Por ejemplo, en un caso de uso para hacer Login en el sistema, se podrían añadir los siguientes elementos:

⇒ especificar el artefacto (Ej. un sistema SSO)
⇒ especificar en nivel de disponibilidad de la funcionalidad (Ej. 99.9%)
⇒ especificar tiempos aceptables de respuesta (Ej. <1 segundo)

Otra opción sería convertir los casos de uso en QAS, relacionando la información como se muestra en la siguiente imagen.

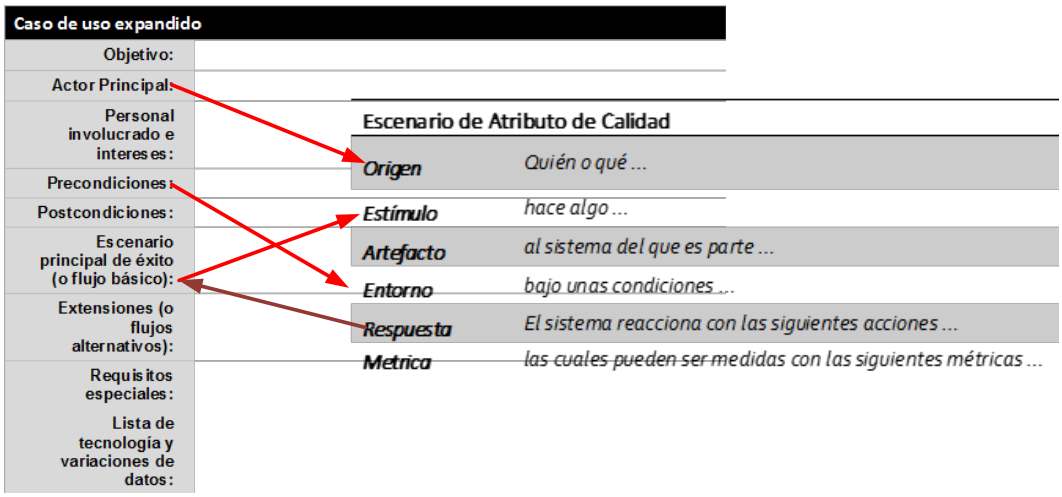

FIGURA 9: *RELACIÓN ENTRE CASOS DE USO Y QAS*

3.1.6.4 Especificación de los principales atributos de calidad

Además de estandarizar el diseño y documentación de los atributos de calidad de un sistema, también hay que proporcionar un estándar que describa los atributos de un sistema. En [1] se proporciona una lista de atributos de calidad de los sistemas software. También se muestra una forma de especificar estos atributos de calidad, mediante escenarios de atributos de calidad. La lista que se presenta en el libro no es exhaustiva ni la única. Primero, no hay consenso universal en la lista de atributos de un sistema. Diferentes colectivos utilizan sus propias definiciones que en muchos casos se solapan (por ejemplo, los estándares IEEE o ISO tienen listas predefinidas de atributos de calidad). Se entiende entonces que hay multitud de atributos válidos a tener en cuenta cuando se diseña la arquitectura de un sistema.

Como se ha comentado en secciones anteriores, para conseguir ciertos niveles de calidad en un atributo suele requerir realizar concesiones en el nivel de calidad que podremos conseguir para otro. No es el propósito de este trabajo dar una lista exhaustiva de definiciones de atributos y de cada una de las tácticas asociadas, que equivaldría prácticamente a traducirla literalmente de [1] sino más bien razonar sobre ellas de modo que evidencie una comprensión del material.

En este trabajo en un principio se utilizará como base la lista que se provee en el libro, pero si durante la parte que trata la mejora del sistema se entiende que se necesitan nuevos atributos, se definirán utilizando Escenarios de Atributos de Calidad.

3.1.6.5 Realizando los atributos de calidad mediante tácticas

Un sistema se puede considerar, como se considera en el libro, como una colección de decisiones de diseño. Una parte de esas decisiones se encarga de controlar la respuesta de los atributos de calidad ante un estímulo mientras que la otra se asegura que se proporciona la funcionalidad requerida.

Una táctica es un conjunto de técnicas que un arquitecto puede usar para conseguir influenciar o determinar un atributo de calidad. Por lo tanto, constituye una decisión de diseño fundamental, que influye o determina la manera en la que el sistema reacciona a un estímulo. Las tácticas se centran un atributo de calidad. Esto

simplifica el análisis de manera que las propiedades y el efecto de la táctica en el sistema se entienden con claridad. No tiene en cuenta las concesiones que hay que realizar entre atributos de calidad.

3.1.6.6 Categorías de decisiones de diseño

Las decisiones de diseño que constituyen un sistema se pueden clasificar como se sugiere a continuación. Usar este tipo de clasificaciones ayuda a los arquitectos a centrarse en las áreas qué se cree que van a resultar más problemáticas y tratar de influenciarlas usando tácticas.

1. *Asignación de responsabilidades*: Se identifican las responsabilidades, funcionalidades básicas, arquitectura de la infraestructura y como se van a satisfacer los atributos de calidad. Además, como se asignan las responsabilidades a los diferentes módulos, componentes y conectores.
2. *Modelo de coordinación*: decisiones relacionadas con la coordinación de los elementos de la arquitectura a través de los mecanismos definidos.
3. *Modelo de datos*: se define la abstracción de datos, las propiedades de las estructuras de datos. etc-
4. *Gestión de recursos compartidos*: define como se arbitran los recursos compartidos, tanto de hardware (Ej. CPU, batería, memoria, etc.) como los de software (Ej. system locks, buffers, threads, etc.)
5. *Asociación de los elementos de la arquitectura*: Hay dos tipos de decisiones relacionadas con la asociación de los elementos de la arquitectura:
    a. Las que afectan las estructuras software: relacionadas con las unidades de ejecución (Ej. hilos o procesos)
    b. Las que afectan los elementos software: relacionadas con los elementos del entorno (ej. asociando procesos con una CPU).
6. *Decisiones en momentos determinados del ciclo de vida*: Hay cierto tipo de decisiones que se toman en ciertos puntos del ciclo de vida, permitiendo así definir unos rangos de variación válidos asociados a para ese punto del ciclo de vida. Cada decisión debe tener definido un dominio, el punto en el ciclo de vida y el mecanismo para conseguir la variación. Un ejemplo de esto sería una tienda de aplicaciones que decide en tiempo de ejecución qué instancia de la aplicación es la correcta para que el usuario se descargue. Esta es una decisión que se toma en tiempo de ejecución (un punto específico del ciclo de vida). De la misma manera, otras decisiones se pueden definir para otros puntos del ciclo de vida.
7. *Elección de la tecnología*: Cada decisión de arquitectura debe implementarse con una tecnología en particular y por lo tanto la elección de la tecnología constituye también parte de las decisiones de diseño.

3.1.6.7 Caso de uso: El lenguaje de programación

Para ilustrar mejor las diferentes categorías de decisiones de diseño que existen se presenta un caso de uso donde se analiza por categoría el impacto de una decisión de diseño como es el seleccionar el lenguaje de programación con el cual se va a desarrollar el sistema.

1. *Asignación de responsabilidades*: El lenguaje de programación limita la asignación de responsabilidades, dependiendo del número de personas que se tengan en la organización, versadas en el lenguaje que se elija. En realidad, estos son dos elementos que se influyen mutuamente, y donde uno de los dos suele prevalecer. En ocasiones, se decidirá el lenguaje de programación en base al experto existente en la organización y a los equipos que se formarán alrededor del lenguaje de programación.
2. *Modelo de coordinación*: El lenguaje de programación afecta el modelo de coordinación porque es más complicado coordinar elementos de la arquitectura que se han implementado usando lenguajes de programación diferentes. Por lo tanto, los modelos de coordinación resultantes en esta situación serán más complejos.

3. *Modelo de datos*: El lenguaje de programación también afecta al modelo de datos. Por ejemplo, en lenguajes de programación no orientados a objetos, quizás no se puedan utilizar ciertas técnicas de abstracción de datos (ej. herencia, polimorfismo, etc.).
4. *Gestión de los recursos compartidos*: Influye porque diferentes lenguajes de programación tienen diferentes necesidades de recursos. Una aplicación desarrollada completamente en lenguaje ensamblador consumirá menos recursos que una desarrollada en Java y tenderá a gestionar los recursos más eficientemente. Así que a la hora de definir cómo se van a gestionar estos recursos las necesidades serán diferentes dependiendo del lenguaje de programación. Además, algunos lenguajes de programación tienen ya herramientas que facilitan la gestión de estos recursos mientras que en otros la gestión ha de realizarla el desarrollador.
5. *Asociación de los elementos de la arquitectura*: El lenguaje de programación también afecta a los elementos de la arquitectura, ya que diferentes lenguajes de programación necesitarán de diferentes elementos. Ej. No podemos desplegar una aplicación con J2EE en un servidor Apache, este tipo de aplicaciones debe ser asignada a un recurso compatible (ej. JBoss, etc.) mientras que sí podemos utilizar Apache con una aplicación desarrollada con PHP.
6. *Decisiones tomadas en momentos determinados del ciclo de vida*: La selección del lenguaje de programación es en sí misma una decisión que puede ser asociada a un determinado punto del ciclo de vida. No es estrictamente necesario decidir cuál va a ser el lenguaje de programación en fases de diseño o incepción ya que estás pueden ser independientes del lenguaje. Así que esta decisión podría tomarse al finalizar la fase de análisis.

3.1.6.8 El impacto de satisfacer determinados atributos de calidad

El conocido refrán, "*no se puede estar en misa y repicando*", nos recuerda lo frecuente que es que no se pueda realizar varias cosas a la vez. Esto no solo es que se aplique al diseño de arquitecturas software, sino que es un concepto clave. Es lo que en inglés se denominan *trade-offs*. En las secciones anteriores, se ha introducido una lista de atributos de calidad que una arquitectura puede requerir. La realización de un atributo de calidad (sea de los de la lista u otros no definidos en este documento) a un determinado nivel implicará la no realización de otro atributo a un nivel determinado.

Por ejemplo, para conseguir un gran desempeño en nuestro sistema, decidimos que vamos a desarrollarlo utilizando un lenguaje de más bajo nivel, como el lenguaje ensamblador. Otros atributos sufrirán como consecuencia de esta decisión. ¿Cuál será la modificabilidad del sistema? Muchísimo menor. No solo es más costoso modificar programas hechos en ensamblador, sino que hay muchísimos menos profesionales que puedan hacer esto. ¿Qué ocurriría con la portabilidad? Pues limitadísima, tendríamos que escribir el código del sistema desde cero para ejecutarlo en otro hardware.

Así que es al arquitecto al que le corresponde decidir, tomando en cuenta los objetivos de negocio y los requisitos de calidad, si se va a estar en misa o repicando. Es decir, qué concesiones se van a realizar. Esta no es una tarea trivial ya que el dominio no solo de los atributos sino de las tácticas disponibles para satisfacerlos es muy extenso. Para ayudar en esta tarea aparecen los patrones arquitectónicos, que veremos en la siguiente sección.

3.1.7 Patrones vs. Tácticas

Pasar de los requisitos de un sistema a diseñar una arquitectura ha sido históricamente un arte más que una ciencia. La experiencia del arquitecto o los arquitectos ha jugado un papel fundamental así que no es de extrañar que se hayan buscado mecanismos para preservar este conocimiento y reusarlo en otras situaciones. Es por eso que probablemente la literatura en los últimos años se ha dedicado mucho más al estudio de los patrones de software que a las tácticas. No obstante, los autores del libro abogan por estudiar las tácticas ya que los patrones de arquitectura:

⇒ son complejos y sus interacciones con otros patrones no siempre están claras

⇒ en ocasiones no están especificados lo suficiente, así que el arquitecto debe complementarlos para conseguir un diseño implementable
⇒ tienen asociado ya un rationale, es decir un conjunto de beneficios e inconvenientes (aunque estos también pueden depender del contexto donde se implementan)
⇒ puede ocurrir que no haya un patrón predefinido para el problema a cuál nos enfrentamos.

Ahí entran en acción las tácticas. Como son más sencillas de analizar (se centran en un atributo de calidad), sus propiedades y efectos son más sencillos de entender. Cuando se entienden bien todas las tácticas de un patrón, entonces el arquitecto está en mejor posición de entender el efecto de modificar ciertas partes del patrón. Si, por el contrario, se estudia el patrón como un paquete es más difícil comprender los efectos de un cambio en el patrón y el arquitecto no estará tan preparado para adaptarlo a las necesidades particulares de un sistema.

La idea de [1] es que, como arquitectos software, no debemos conformarnos con estudiar patrones, como el puente Bailey y sus diferentes configuraciones, sino que debemos estudiar sus componentes básicos para así entender el efecto que tienen en la satisfacción de los requisitos generales. Así estaremos en posición de elegir la configuración correcta, realizar modificaciones al patrón o diseñar el nuestro propio.

3.1.8 GPC - Evaluación de la arquitectura software

3.1.8.1 ATAM - Attribute Trade-off Analysis Method

Hasta ahora hemos visto las herramientas que tenemos a nuestra disposición para el diseño de arquitecturas de software y hemos razonado sobre ellas. Ahora analizaremos los métodos de evaluación de arquitecturas: que utilizaremos en el trabajo: ATAM, Lightweight ATAM y CBAM. Aunque en este trabajo se presenten de forma separada ATAM y CBAM se pueden integrar en un solo método.

ATAM, por sus siglas en inglés, es un método de evaluación y análisis de arquitecturas desarrollado por el Software Engineering Institute (SEI) perteneciente la universidad Carnegie Mellon. Estandariza el proceso de análisis y evaluación de arquitecturas. Permite que los evaluadores no estén familiarizados con la arquitectura, los objetivos empresariales, que el sistema no esté todavía construido y acomodar a un largo número de stakeholders. Aunque diferente literatura lo presenta como el estándar *de facto*, lo cierto es que es un proceso caracterizado por tener un coste asociado bastante alto por lo que dudo que tenga una amplia aceptación fuera el ámbito académico.

PARTICIPANTES

En ATAM toman parte los siguientes grupos:

- Equipo de evaluación: son un grupo de personas externas al proyecto el cuál la arquitectura es evaluada. Se les debe considerar competentes e imparciales.

  Dentro del Equipo de Evaluación se definen los siguientes roles:

  - Jefe de Equipo
  - Jefe de Evaluación
  - Responsable de Escenarios
  - Responsable del procedimiento
  - Cuestionador
- Decision-makers: son las personas que tienen la autoridad para decidir cambios en el proyecto y su desarrollo. Suele incluir al Director de Proyectos, quizás el cliente, etc. El arquitecto debe estar siempre incluido en este grupo.
- Architecture stakeholders: Se compone del grupo de personas que tienen un interés en que la arquitectura funcione de forma adecuada. Programadores, testers, integradores, usuarios, etc. Este grupo de personas no participa en todas las actividades de ATAM.

RESULTADOS A OBTENER

ATAM usa escenarios de atributos de calidad priorizados como base para la evaluación de la arquitectura. Es decir, que esos se han generado en algún ejercicio anterior como, por ejemplo, en una actividad ADD. Si no se han generado estos, se pueden generar durante ATAM. En todo caso, después de un ejercicio ATAM se deberá haber:

1. creado una presentación concisa de la arquitectura
2. articulado los objetivos empresariales
3. definido y priorizado los atributos de calidad, expresados mediante Escenarios de Atributos de Calidad.
4. establecido una lista de riesgos y no-riesgos
5. establecido una lista de grupos de riesgo, es decir se analizan los riesgos en busca de una relación entre ellos que permita identificar debilidades sistémicas.
6. relación entre las decisiones arquitectónicas y los requisitos de calidad
7. definido los puntos críticos (o sensibles) y las concesiones, siendo los puntos críticos esas decisiones de arquitectura que tienen un efecto marcado en uno o más atributos de calidad.

PROCESO DE EVALUACIÓN

Fase 0 – Preparación

Los responsables de la evaluación y los representantes del proyecto se reúnen para planificar el ejercicio ATAM. Se acuerdan temas relacionados con la logística: cuando se realizará, donde, etc. Los representantes pueden introducir el proyecto para que los responsables de la evaluación puedan elegir un equipo evaluador con la experiencia adecuada. El equipo de evaluación estudia la arquitectura para entender bien cuál es el propósito del sistema, la arquitectura y los atributos de calidad de más importancia.

Fase 1 - Evaluación

La fase 1 ya es parte de la evaluación de la arquitectura, y es la fase donde los evaluadores junto con los responsables del proyecto se reúnen para realizar el análisis y evaluación de la arquitectura. Tanto la fase 1 como la 2 tienen definido unos pasos que se han de realizar, y los cuáles se detallan a continuación:

1. Presentación del método ATAM
2. **Presentación de las motivaciones de negocio**: todos los que están involucrados en la evaluación de la arquitectura deben estar familiarizados con el contexto del sistema y las principales motivaciones para su desarrollo. Este es el objetivo de este paso del proceso. Información que se puede proporcionar como parte de este paso incluiría:
    a. Las funciones más importantes del sistema
    b. Limitaciones técnicas, de gestión, económicas o políticas
    c. Los objetivos empresariales y el contexto
    d. Quienes son los principales stakeholders
    e. Los requisitos arquitectónicamente significativos
3. **Presentación de la arquitectura**: en este punto el arquitecto hace una presentación de la arquitectura incluyendo la información que es relevante para el ejercicio de evaluación.
4. **Identificación de las estrategias de arquitectura:** se catalogan los patrones de arquitectura y tácticas que se han utilizado en el diseño de la arquitectura.
5. **Generación del árbol de atributos de calidad:** o lo que es lo mismo se asigna prioridad a los Escenarios de Atributo de Calidad.
6. **Análisis de las estrategias de arquitectura:** Se analizan los escenarios por el orden de prioridad definido en el árbol de atributos de calidad. El arquitecto explica a los evaluadores como la arquitectura permite satisfacer los requisitos y se discuten. En esta discusión lo normal es que se identifiquen las concesiones, los puntos críticos, riesgos y no riesgos del diseño de la solución.

Fase 2 - Evaluación

A esta fase se invita a un mayor número de stakeholders, a los cuáles se les presentará los resultados obtenidos. Por lo tanto, se repiten algunos pasos de la fase 1. Los pasos de esta fase son los siguientes:

7. Tormenta de ideas y priorización de escenarios
8. Análisis de las estrategias de arquitectura
9. Presentación de resultados

Fase 3 - Cierre

Durante la fase 3 se cierra el ejercicio ATAM, con la producción del informe del Equipo de Evaluación, validado por los stakeholders para asegurar que no ha habido malentendidos y después de validarlo se entrega a la persona o personas que encargaron la evaluación.

3.1.8.2 Lightweight Architecture Evaluation

El método ATAM que se ha descrito en la sección anterior es muy costoso para la organización. Según [1], se suele requerir un esfuerzo entre 20-30 pdays del equipo de evaluación y una cifra superior para el arquitecto y los stakeholders. ATAM es apropiado cuando el riesgo de un error de importancia en la arquitectura es inaceptable. En otras circunstancias, este método probablemente no tiene una proporción coste/ beneficio positiva. Por este motivo, se presenta el método Lightweight ATAM, que es una versión ligera de ATAM, donde la evaluación se realiza por el personal interno en uno o incluso medio día. Es decir, se invierte menos tiempo en la evaluación por parte del arquitecto y los stakeholders, y no se involucra a ningún personal externo. La concesión es que en este entorno se hace más difícil ser objetivos, ya que por un lado puede que haya intereses velados en los participantes que les hagan gravitar hacia ciertas opciones y por el otro habrá un menor número de opiniones diferentes, lo que puede llevar a comprometer la calidad de los resultados obtenidos.

3.1.8.3 Evaluación de la arquitectura GPC

Para evaluar la arquitectura de GPC se va a utilizar Lightweight ATAM ya que uno de los objetivos del trabajo es la reducción de los recursos empleados. Durante esta evaluación se identifican lo siguientes problemas arquitectónicos en el componente:

1. tiene difícil integración con servicios externos (interoperabilidad),
2. baja modificabilidad
3. problemas de desempeño en la gestión de roles

El resultado de la evaluación se incluye en la siguiente tabla.

Atributo	Evaluación
Disponibilidad	El sistema no presenta problemas específicos de disponibilidad aunque, cuando se presentan, estos afectan a todo el sistema, ya que todos los componentes están fuertemente acoplados.
Interoperabilidad	El sistema ofrece un servicio web utilizado por el componente de licitación electrónica que permite la creación de espacio de trabajo para contratos desde el módulo de licitación, así como un servicio de Single Sign On. Sin embargo, para un sistema de este tipo se considera que la interoperabilidad es demasiado baja ya que, siendo un sistema de gestión de contratos y proyectos, los indicadores de calidad KPIs, gestión de incidencias, información financiera u otros, suelen ser recogidos de sistemas externos (SAP, JIRA, etc.). Por este motivo se considera que el sistema debería mostrar una interoperabilidad más alta.

	En concreto GPC tiene un 20% de los casos de uso están expuestos como servicios, permitiendo su uso por otros componentes, pero no se utilizan tecnologías como Enterprise Java Beans, que facilitan la prestación de estos servicios. De datos históricos se sabe que el esfuerzo medio por servicio integrado son 10 pdays.
Modificabilidad	Aunque el sistema utiliza MVC, al estar fuertemente acoplado hace que las modificaciones en una parte del sistema afecten a todo el sistema, aumentando el esfuerzo requerido para introducir nuevas modificaciones. Esto presenta algunos inconvenientes, ya que las metodologías de gestión de contrato no están legisladas (al contrario que el proceso de licitación) y por lo tanto cambian ostensiblemente de organización en organización, con lo que hay que adaptar el sistema frecuentemente. Una arquitectura orientada a componentes sería más adecuada para un sistema de estas características. De datos históricos se sabe que la media de defectos por cambio efectuado es 15. El esfuerzo medio por Object Point [14], es decir por Pantalla Simple, es de 0.7 pdays.
Despempeño	El sistema tiene algunos problemas de desempeño debidos a la implementación específica del sistema de roles y la organización jerárquica de los contratos.
Seguridad	Los mecanismos de seguridad del sistema son adecuados para los requisitos y dominio del sistema.
Testabilidad	Existe un *testing coverage* muy bajo, del 20% del código.
Usabilidad	En diferentes exámenes de usabilidad, la usabilidad del sistema ha sido evaluada en test de usabilidad con puntuaciones muy bajas de entre el 40-50%.

Capítulo 2 Rediseño de la arquitectura software del componente GPC utilizando ADD + MDA

De la evaluación realizada en la sección anterior se derivan los objetivos de la mejora. Se han identificado deficiencias en interoperabilidad, modificabilidad, desempeño, testabilidad y usabilidad. Aunque todos son candidatos para la mejora, se seleccionan los atributos de interoperabilidad y modificabilidad ya que se consideran que son los que tendrán un mayor impacto en las licitaciones en las que participa la empresa, **facilitando la adjudicación de nuevos contratos**, como se justifica a continuación:

Interoperabilidad: Como se ha comentado anteriormente, parte de la información que se registra en el sistema suele provenir de sistemas externos, por lo que es frecuente que se necesite desarrollo adicional después que se adjudica un contrato, para integrar la plataforma con estos sistemas externos. Una mejor interoperabilidad reducirá el coste de integración, permitiendo a la empresa ofrecer precios más competitivos.

Modificabilidad: Como las metodologías de gestión de contrato no están legisladas, cambian ostensiblemente de organización en organización, con lo que hay que adaptar el sistema frecuentemente. Una mayor modificabilidad, reducirá el coste de estos desarrollos permitiendo a la empresa ofrecer precios más competitivos.

Desempeño: Los problemas de desempeño afectan principalmente a la gestión de roles y gestión de contratos. Esta funcionalidad está únicamente disponible para los usuarios con el rol de Administrador de Sistema, que son una minoría por cada instancia de la aplicación. Aún en proyectos donde la plataforma se ofrece a nivel nacional, el número de usuarios administradores de sistema no suele superar 10, por lo que no se considera crítico resolver estos como parte de esta mejora.

Testabilidad: Los procesos de contratación de sistemas de licitación electrónica no suelen tener nunca ningún requisito específico que exija un nivel de pruebas específico para la plataforma. Lo que suelen incluir los contratos son niveles de servicio, dentro de los cuales hay que resolver los defectos. Como hasta el momento no se ha incurrido en ninguna penalización, no se considera necesario incrementar el test coverage como parte de esta mejora.

Usabilidad: Al igual que con la testabilidad, los procesos de contratación de sistemas de licitación electrónica no suelen tener nunca ningún requisito específico que exija tener un nivel específico de usabilidad, por lo tanto no se aprecia ningún beneficio inmediato de mejorar la usabilidad por encima del nivel actual.

Por lo tanto, se han extraído los siguientes objetivos para la mejora del componente de Gestión de Proyectos y Contratos:

1. **Incrementar la interoperabilidad**: El objetivo será que el 50% de las operaciones esté expuesta como servicio, dando prioridad a las relacionadas con la gestión financiera.
2. **Incrementar la modificabilidad**: reducir el número medio de defectos a 2 y el esfuerzo medio por Object Point de 0.25.

Para la consecución de estos objetivos, se utilizarán las tácticas definidas en [1] para esos atributos.

3.2.1 El manifiesto ágil [15]

El manifiesto ágil captura los principios que deben seguir los procesos de desarrollo de software ágiles. De este manifiesto se extraen las principales prácticas que caracterizan a las metodologías ágiles:

- Accesibilidad del cliente (el cliente debe estar siempre disponible).
- Pequeñas y frecuentes versiones del sistema (feed-back continuo).
- Test Driven Development

El manifiesto ágil se organiza en los valores y principios que se describen en las siguientes secciones.

3.2.1.1 Valores

Los valores del manifiesto ágil son los 4 que se muestran a continuación. La idea que intenta transmitir es que, aunque los elementos de la derecha se valoran, se valoran más los de la izquierda.

- **Individuos e interacciones** sobre procesos y herramientas
- **Software funcionando** sobre documentación extensiva
- **Colaboración** con el cliente sobre negociación contractual
- **Respuesta ante el cambio** sobre seguir un plan

3.2.1.2 Principios

De esos 4 valores se extraen los siguientes principios:

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.

6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
10. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

3.2.1.3 ¿Qué es un proceso de software ágil?

Como se ha podido apreciar en la siguiente sección, Ágil no es una metodología de desarrollo. Simplemente describe que principios y valores que debería mostrar un proceso software ágil. En [16] se define una metodología de desarrollo ágil como una donde los "*requisitos y soluciones evolucionan a través de un esfuerzo colaborativo en equipos multifuncionales autoorganizados*".

3.2.2 Attribute Driven Design 3.0 (ADD Ágil)

ADD Ágil es un método recursivo que descompone el Sistema o uno de los elementos del Sistema aplicando patrones arquitectónicos y tácticas para satisfacer los requisitos de ese componente. Esto lo diferencia de otros métodos más tradicionales de la rama de la arquitectura software, que no iteran sobre los requisitos en diferentes fases del desarrollo.

ADD Ágil sigue el siguiente flujo: Planear, Hacer, Comprobar y Actuar. En la fase de planificación, se seleccionan los atributos de calidad y los requisitos que son aplicables para el componente siendo analizado y se toman las decisiones de diseño necesarias basadas en ese componente y en sus requisitos y atributos.

En la fase de Hacer, se aplican las tácticas y patrones seleccionados para satisfacer los requisitos del componente. En la fase de Comprobar, se analiza el diseño para verificar que todos los requisitos se han satisfecho y si al hacerlo se tiene un impacto en el sistema.

En la última fase, se actúa en base a la información recabada y si todo está bien entonces se puede pasar a diseñar otro componente. Si se ha encontrado algún problema, se vuelve a empezar desde el principio hasta que todos los requisitos se hayan satisfecho.

La entrada de ADD Ágil es el conjunto de requisitos del Proyecto, que incluye tanto los requisitos funcionales como los requisitos no funcionales. Los requisitos deberían ser descritos en detalle para poder comprobar si pueden ser satisfechos con una arquitectura particular.

La salida de ADD Ágil es un conjunto de decisiones de diseño hechas durante el proceso junto con un diseño inicial de la arquitectura software. Este diseño muestra como dividir el sistema en diferentes componentes, las responsabilidades de cada componente y las interacciones entre componentes.

El proceso en sí mismo se puede descomponer en 7 pasos [6].

1. Revisar las entradas
2. Establecer el objetivo de la iteración
3. El primero es elegir un elemento del sistema a diseñar. En la primera iteración, será el sistema completo y se dividirá el sistema completo en sus diferentes componentes. En las iteraciones siguientes se refinarán los componentes definidos en la primera acción, definiendo los módulos, sub-módulos, etc. y así sucesivamente.

4. Se identifican y seleccionan los requisitos **arquitectónicamente** más significativos del componente. Para ello se listan los requisitos y se les asigna una prioridad teniendo en cuenta 2 factores: su impacto a nivel de negocio y su impacto a nivel de la arquitectura. Los requisitos que tengan un alto impacto en los 2 factores serán los más significativos.
5. Se diseña una solución para el componente que satisfaga los requisitos seleccionados. Este paso está compuesto por las siguientes tareas:
    a. Decidimos qué patrones y tácticas se pueden utilizar para satisfacer los requisitos seleccionados. Aquí normalmente es cuando deberemos realizar concesiones y por lo tanto cuando se toman decisiones importantes.
    b. Se instancian las tácticas y los patrones seleccionados para el componente. Para ello se necesita tomar más decisiones. Por ejemplo, si se ha seleccionado un patrón para una arquitectura en capas, se decide cuántas capas se van a usar.
    c. Se asignan las responsabilidades a los elementos de la arquitectura. Para ello se utilizan los requisitos funcionales.
    d. Se definen las interfaces y cómo se van a utilizar, es decir se definen las relaciones entre los elementos y el intercambio de información.
6. Se crean las vistas y se documentan las decisiones tomadas en el paso anterior.
7. Se analiza el diseño resultante, se comprueba si se ha conseguido el objetivo de la iteración y el propósito del diseño.

3.2.3 El rediseño de GPC

Para el rediseño de la arquitectura se va a utilizar ADD Ágil combinado con MDA. Cuando se usa ADD Ágil el diseño arquitectónico se realiza incrementalmente en iteraciones a medida que avanza el desarrollo de un proyecto de software. Cada iteración de diseño puede tener lugar dentro de una fase de desarrollo de software, como un sprint cuando se utiliza SCRUM o una fase de construcción con una metodología basada RUP o AUP. En cada una de estas iteraciones se realiza el diseño que se considera necesario para la fase del proyecto. En este sentido, el proceso es ágil ya que no se realiza de forma completa el diseño de la arquitectura, sino que se va completando de forma incremental.

Aunque ADD Ágil proporciona una guía detallada de las tareas que deben realizarse dentro estas iteraciones de diseño, en este trabajo no se va a seguir estrictamente ya que hay situaciones en las que no es necesario seguir un proceso tan estructurado.

3.2.4 Iteración 1. Orientación a servicios

En la primera iteración se decidirá el rediseño de la arquitectura del componente. Como ya se ha mencionado los atributos de calidad que se quiere mejorar son los de interoperabilidad y modificabilidad.

3.2.4.1 Interoperabilidad (Interoperability)

La Interoperabilidad es la capacidad de intercambiar información entre diferentes sistemas, usando interfaces e interpretar la información que se intercambia de forma satisfactoria. Este intercambio de información puede que se realice de forma indirecta y la información que se intercambia ha de ser interpretada de forma correcta, sintáctica y semánticamente.

Las categorías de tácticas para conseguir mejorar la interoperabilidad son:

⇒ **Descubrimiento de los servicios**: un servicio de búsqueda de directorio que proporciona ciertas funcionalidades accesibles mediante una interfaz

⇒ Gestión de la respuesta: o el servicio responde al llamamiento, se envía a otro sistema o se publica un broadcast para que el servicio adecuado responda. La respuesta podría ser el resultado de una composición de servicios. Algunas tácticas para gestión de la respuesta son:
- *Orquestación*: Hay un mecanismo de control que coordina y gestiona la secuencia de invocación de los servicios. Se utiliza para esquemas de interoperabilidad complejos.
- *Tailor interface*: permite añadir o eliminar capacidades a una interfaz

La interoperabilidad es uno de los atributos fundamentales de las Arquitecturas Orientadas a Servicios.

3.2.4.2 Modificabilidad (Modifiability)

La Modificabilidad de un sistema es la facilidad con la cual se pueden efectuar cambios en el entorno del sistema o satisfacer nuevos requisitos o especificación de funcionalidades sin introducir defectos o degradar las cualidades existentes. Es decir, que mide el coste y el riesgo de introducir cambios en el sistema. Algunas de las categorías de tácticas empleadas para mejorar este atributo son:

⇒ Reducir el tamaño de los módulos: Aunque no se apueste al 100% por una arquitectura SOA con componentes independientes, ayuda a reducir el impacto de los cambios el evitar arquitecturas monolíticas.
⇒ Incrementar la cohesión: La cohesión de un módulo es el grado de organización semántica de la funcionalidad del módulo. Es decir que la funcionalidad incluida en un módulo está relacionada. Por ejemplo, las todas las funciones relacionadas con la gestión de usuarios están en el mismo módulo, esto es cohesión.
⇒ Minimizar las dependencias: reducir el nivel de dependencias entre los diferentes módulos (Ej. Usar interfaces)
⇒ Enlazar en diferido: Parametrizar cuando sea posible para que el sistema gestione parte de los cambios.

3.2.4.3 Orientación a servicios

Teniendo en cuenta estos atributos a mejorar y las tácticas existentes para ello, es evidente que se necesita rediseñar la arquitectura para que tenga una orientación a servicios más significante que la que tiene actualmente. Este cambio de orientación cumplirá la misiva de mejorar la interoperabilidad. Hemos definido la interoperabilidad como "*la capacidad de intercambiar información entre diferentes sistemas, usando interfaces*". A la misma vez, el uso de interfaces también se asocia a la táctica de modificabilidad "*Minimizar las dependencias*". Para minimizar dependencias, además de utilizar interfaces se realizará una descomposición de los módulos en componentes desplegables independientemente, proporcionando así al componente una independencia funcional. Esta independencia funcional as asocia a la táctica de "*Reducir el tamaño de los módulos*", ya que el componente GPC tiene una granularidad arquitectónica mayor y por lo tanto se reduce el impacto de los cambios realizados a partir del rediseño en los nuevos componentes surgidos.

Otra táctica que se podría utilizar para mejorar la modificabilidad es la de "*Incrementar la cohesión*". Para realizar esta táctica, teniendo en cuenta que GPC presenta una estructura modular, habría que estudiar las interacciones entre módulos que se producen durante la ejecución e intentar reducirlas. Esta táctica no forma parte de este trabajo ya que no se tiene acceso al código fuente, por lo que es imposible realizar este análisis.

Por lo tanto, el componente GPC se fragmentará en los siguientes componentes, que después del rediseño de la arquitectura se comunicarán entre ellos estrictamente usando interfaces:

1. Gestión y Seguimiento de Contratos. Este componente incluirá las funciones de los módulos:
   a. *Gestión y Seguimiento del Contrato*, sin incluir las relacionadas con la gestión financiera, que se organizarán en un componente independiente.

b. *Administración*. Este componente incluirá los módulos de Administración (usuarios, organizaciones y roles), Informes y Auditoría.

Básicamente, este componente es el componente GPC. Incluye la GUI web, la gestión de usuarios, autenticación, etc.

2. Gestor de Documentos. Incluirá las funciones del módulo de *Gestión de Documentos*.
3. Gestor de Eventos. Incluirá las funciones del módulo de *Gestión de Eventos*.
4. Gestor de Incidencias. Incluirá las funciones del módulo de *Gestión de Incidencias*.
5. Gestión Financiera. En este componente se incluirán las funciones relacionadas con presupuesto del proyecto, facturas, pagos, etc. que se incluían en el módulo de *Gestión y Seguimiento del Contrato.*

Para realizar la descomposición funcional que llevará a los nuevos componentes, se ha tomado como referencia el Manual de Usuario del componente GPC, donde los analistas de la empresa ya han seguido un proceso deductivo para analizar la funcionalidad y organizarla de forma cohesiva, de forma que el usuario pueda localizar la información necesaria en el manual de forma sencilla. Cada encabezado principal del manual de usuario se convertirá en un componente, con la excepción de la funcionalidad de auditorías e informes que seguirán en el componente principal de Gestión de Contratos.

Los nuevos componentes: el Gestor de Documentos, el Gestor de Eventos y el Gestor de Incidencias, después de la implementación de la mejora expondrán la funcionalidad disponible (casos de uso) como servicios, asegurando que el 50% de la funcionalidad estará expuesta como servicios, cumpliendo uno de los objetivos fijados para el atributo de interoperabilidad.

3.2.4.4 Identificar las estrategias de arquitectura de la plataforma

Teniendo en cuenta las tácticas disponibles para la mejora de los atributos de calidad, se seleccionan los siguientes patrones de arquitectura descritos en [1] como candidatos para la mejora.

Opción 1 - Arquitectura SOA

Las arquitecturas orientadas a servicios, comúnmente conocidas como SOA (por sus siglas en inglés), están formadas por componentes software distribuidos que proveen y/o consumen servicios. El consumidor y el proveedor de un servicio son independientes a tal grado que es posible que consumidor y proveedor estén implementados en lenguajes de programación y/o plataformas independientes. Aunque hay diferentes estrategias (Ej. API, interfaces, Enterprise Service Bus, Registro de Servicios, Servidores de orquestación, etc.) los componentes SOA siempre exponen los servicios que utilizan, de forma que puedan ser utilizados por terceros. En un entorno fuertemente orientado a servicios, los componentes exponen y a su vez son usuarios de servicios, lo que hace que las plataformas estén débilmente acopladas mejorando la interoperabilidad. Este acoplamiento débil por otro lado introduce intermediarios en las transacciones y por lo tanto hace que latencia y desempeño de la plataforma sea menor.

Los componentes se comunicarán siguiendo un modelo SOA, donde los servicios estarán expuestos y podrán ser utilizados por tanto por los componentes de la plataforma como por otros sistemas externos.

Como se menciona en [17], entre los principales factores determinantes para la adopción de SOA destacan aquellos relacionados con:

- la necesidad de flexibilidad y capacidad de cambio, como puede ser necesitar introducir regularmente nuevos servicios o tener que considerar procesos de negocio que cambien frecuentemente.
- la necesidad de integración tanto internamente como en relación a terceras partes
- la necesidad de reutilización, es decir la necesidad de distribuir el software a diferentes actores

Por este motivo, una arquitectura SOA parece la elección correcta para el diseño de la arquitectura de la plataforma, ya que son precisamente esas necesidades las que motivan el rediseño. Sin embargo, en [17] también se citan 2 elementos que pesan en este proyecto en particular:

- Subestimar la complejidad técnica de una SOA a gran escala.
- Subestimar la necesidad de disponer de un modelo de negocio de SOA. No existe ninguna aproximación "one size fits all" al gobierno de SOA y puede ser perjudicial tanto el exceso como la ausencia de suficiente gobierno.

El desarrollar una arquitectura 100% SOA introduce una complejidad técnica mayor, por lo que podría ser contraproducente hacer un cambio tan radical en la plataforma, con los riesgos que ello conlleva.

Opción 2 - Componentes independientes y comunicación a través de interfaces J2EE

Broker es un patrón de los conocidos como "Component-and-Connector" y que se ocupan de gestionar la comunicación entre componentes, con la intención de romper el acoplamiento entre estos. Otro patrón de este tipo sería el Modelo-Vista-Controlador.

Un bróker es simplemente un intermediario responsable de gestionar la comunicación entre componentes software, típicamente un cliente y un servidor. En términos de atributos de calidad, incrementa la modificabilidad y también la disponibilidad (Ej. hace más sencillo reemplazar un servidor por otro). Como siempre que se incrementa el nivel de abstracción, se obtiene un menor desempeño y mayor latencia.

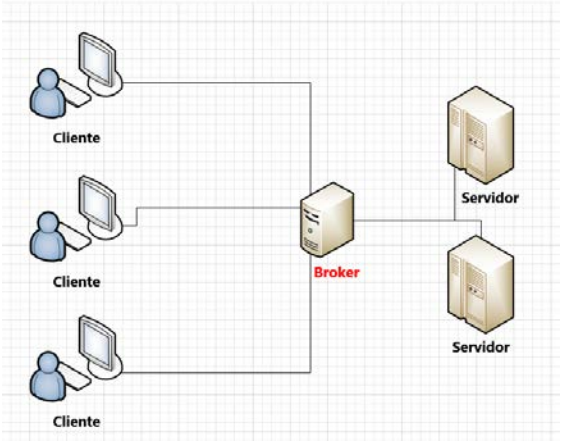

FIGURA 10: *REPRESENTACIÓN DEL PATRÓN BROKER*

En esta opción, en lugar de optar por una arquitectura completamente SOA, se divide el componente GPC en componentes independientes, pero que se comunican por medio de brokers en lugar de una arquitectura completamente SOA. El patrón de broker separa a los usuarios de servicios (clientes) de proveedores de servicios (servidores) a través de un intermediario, llamado bróker. Cuando un cliente necesita un servicio, consulta a un agente a través de una interfaz, que abstrae la identidad, localización y características del servidor.

Esto aunque representa una arquitectura intermedia en el sentido que proporciona un acoplamiento más débil que actualmente presente en el componente GPC pero, a su vez, más fuerte que una arquitectura SOA. El problema de usar el patrón bróker es que introduce la complejidad adicional de implementar la comunicación entre los clientes y el servidor y minimiza la disponibilidad del sistema, ya que el bróker constituye lo que se conoce en inglés como un *single point of failure*.

Opción 3 - Opción mixta (SOA y Bróker J2EE)

La última opción que se evaluará es la que se denomina como **mixta**. La mixta incluye implementar una arquitectura que utilice un bróker para ciertos componentes y SOA para otros. En lugar de decidir a nivel global cuál va a ser la arquitectura de comunicación, se realizará una evaluación componente a componente. Es decir que el componente de Gestión de Documentos podría utilizar un bróker mientras que el de Gestión Financiera se comunique de forma 100% independiente, usando una arquitectura SOA.

Esta opción simplificaría la complejidad técnica de SOA que se comentaba anteriormente y reconoce que quizás no se necesite para todos los componentes. Por otro lado, soportar 2 estrategias diferentes también es fácil que conlleve un esfuerzo asociado mayor.

3.2.4.5 Evaluación de estrategias de arquitectura

En esta iteración se van a evaluar las estrategias de arquitectura. Para realizar la evaluación se va a utilizar el método CBAM, que introduce los conceptos de coste, utilidad y ROI en el proceso de decisión. Se puede argumentar que CBAM no puede justificar una mejora (cualitativa o cuantitativa) ya que todo se supedita al ROI como medida final, por lo que si no se tiene medidas de costes de desarrollo (sólo estimaciones teóricas) se desarrolla un diseño en el que no está garantizada la mejora deseada (no es hasta que se implementan los cambios que se puede medir la mejora). Esto, sin embargo, no justifica dejarlos fuera del proceso de decisión. De hecho, históricamente, la mayoría de técnicas desarrolladas en la investigación de la arquitectura software se centran en los aspectos técnicos e ignoran el valor creado por esos cambios o decisiones arquitectónicas.

Pero dado el enorme impacto de la arquitectura software en un sistema, en literatura más reciente que [1] se hace evidente que no se pueden ignorar los aspectos económicos. En [18], ya se identifican claramente estos como parte del campo de la arquitectura software. Como se deja claro en [19], no tomarlos en cuenta simplemente empobrece el proceso de decisión, ya que hace que los arquitectos software tomen decisiones sin tener en cuenta aspectos como el coste, la flexibilidad o el riesgo, cuando estos impactan notablemente en el beneficio.

Además, los arquitectos software deberán justificar estas decisiones a la dirección o alta dirección de la organización, sea para obtener la asignación de recursos o simplemente para constatar el beneficio, cuando finalice la implementación. La dirección no está interesada en los detalles técnicos de las decisiones tomadas sino en el valor creado por ellas. Por estos motivos, para justificar la asignación de recursos para implementar la mejora en este trabajo se utilizará ATAM y CBAM, ya que proporcionan una visión del coste, utilidad y beneficio asociado a las diferentes decisiones arquitectónicas.

3.2.4.6 CBAM - Cost Based Analysis Method [20]

ATAM permite relacionar las decisiones arquitectónicas con los objetivos empresariales y los atributos de calidad, pero no proporciona un marco para guiar las decisiones tomando en cuenta los aspectos económicos de un proyecto, como el coste de una decisión de arquitectura o el impacto en el plan de proyecto. En la industria del software es difícil de imaginar un proyecto donde se puedan dejar de lado estos aspectos. Es por eso que, como parte de este trabajo, también se ha estudiado el método CBAM. Este método de evaluación parte básicamente donde ATAM termina, es decir puede utilizar los artefactos que produce como salida ATAM y añade una dimensión más al incluir el coste y el impacto en el plan de proyecto, lo que permite que la relación de ATAM, impacto de una decisión de arquitectura con los objetivos empresariales y atributos de calidad, convertirse en una relación que analiza el retorno de la inversión asociado a una decisión de arquitectura. CBAM también pude utilizarse como método independiente creando los escenarios específicamente para CBAM, sin utilizar ATAM para ello.

LA UTILIDAD DE LAS ESTRATEGIAS DE SOFTWARE

CBAM introduce el concepto de utilidad para permitir la capturar cuál es el beneficio asociado a una estrategia de arquitectura. Se crea una relación entre los valores del atributo a mejorar y la utilidad esperada cuando se realice la modificación. Es decir, los objetivos constituyen el eje x de la curva y la utilidad y= f(x).

Los puntos f(x) se calculan usando interpolación, así se puede obtener una medida de utilidad para el valor de los atributos que se espera conseguir, dependiendo de las estrategias utilizadas.

Una vez tenemos la utilidad actual y la utilidad esperada, podemos calcular el beneficio y entonces podemos relacionarlo con el coste. El método CBAM se compone de los siguientes pasos:

1. Crear escenarios
2. Refinar escenarios (definir los valores mejor, peor, actual y esperado de los objetivos)
3. Priorizar escenarios
4. Asignar utilidad (mejor, peor, actual, esperada)
5. Asociar las estrategias de la arquitectura con los escenarios y determinar la respuesta de los atributos de calidad
6. Determinar la utilidad esperada usando interpolación
7. Calcular el beneficio obtenido por el uso de una estrategia de arquitectura
8. Elegir las estrategias basándose en el ROI
9. Contrastar los resultados obtenidos con la intuición

En la práctica, la utilidad de una aplicación y el peso constituirá información que provenga del departamento de desarrollo de negocio cuando se solicite nueva funcionalidad o del departamento técnico cuando se evalúen decisiones de arquitectura que serán transparentes para el usuario final. De todas formas CBAM es una herramienta útil para capturar el beneficio, que es algo que la mayoría de métodos de estimación o evaluación omite.

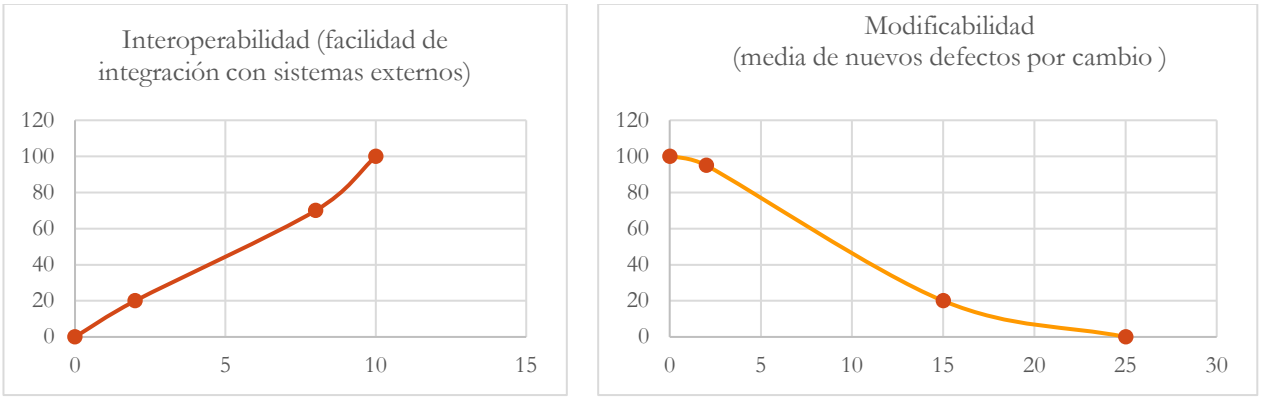

FIGURA 11: *EJEMPLO DE CURVAS DE UTILIDAD*

3.2.4.7 Evaluación utilizando CBAM - Cost Based Analysis Method

Como ya se ha justificado anteriormente se va a utilizar CBAM ya que se quiere incluir el coste como un elemento del proceso de decisión. En la primera tabla que se muestra a continuación se van a mostrar los objetivos generales de la mejora. El valor de los atributos de calidad que se desea conseguir se muestra en la columna "Deseado".

OBJETIVOS (X)						
Id	**Escenario**	**Peor**	**Actual**	**Deseado**	**Mejor**	**Peso**
1	Interoperabilidad (funcionalidad integrada con otros componentes)	0	20	90	100	35
2	Interoperabilidad (esfuerzo medio (pdays) por servicio integrado)	22	10	5	2	25
3	Modificabilidad (esfuerzo medio (pdays) por object point)	3	0.7	0.25	0.1	15

4	Modificabilidad (media de nuevos defectos por cambio efectuado)	25	15	2	0	25

La siguiente tabla muestra los resultados de la evaluación (la evaluación completa se puede consultar en el Anexo 1).

	AS	Escenario Afectado	Utilidad Actual	Utilidad Esperada	Beneficio bruto	Beneficio normalizado	Beneficio total	Coste	ROI	Ranking
1	Arquitectura SOA	1	20	90	70	2450	4002	633	6.328	3
		2	46	75	28	711				
		3	62	75	13	200				
		4	34	60	26	642				
2	Arquitectura Bróker	1	20	70	50	1750	3545	457	7.765	1
		2	46	61	14	355				
		3	62	72	10	155				
		4	34	86	51	1284				
3	Arquitectura mixta	1	20	85	65	2275	3974	589	6.753	2
		2	46	65	19	474				
		3	62	72	10	155				
		4	34	77	43	1070				

TABLA 3: *RESULTADO DE LA EVALUACIÓN DE ESTRATEGIAS DE ARQUITECTURA*

El resultado de la evaluación indica que la estrategia que resultará en un ROI mayor y por tanto la más óptima es el diseño de una arquitectura bróker.

3.2.5 Iteración 2. Objetivo: Diseño de la arquitectura de los componentes

Como parte final de la iteración 2 se realiza el diseño de la arquitectura seleccionada, utilizando una herramienta MDA que en este caso será la herramienta de IBM Rational Software Architect.

3.2.5.1 Model Driven Architecture (MDA)

Model Driven Architecture (MDA) es un paradigma para desarrollo de software propuesto por el Object Management Group (OMG) en 2001. La idea principal del MDA es construir aplicaciones de software a un nivel de abstracción superior al comúnmente utilizado, a través de modelos visuales en lugar de la utilización de un lenguaje de programación de alto nivel. MDA propone por lo tanto un cambio en el desarrollo *tradicional* de software, centrado en la programación de alto nivel, por uno centrado en el modelado del sistema.

PERSPECTIVA HISTÓRICA

Desde una perspectiva histórica, se puede considerar que el desarrollo de software ha venido siempre marcado por un incremento constante en el nivel de abstracción. Estos avances en el nivel de abstracción, han permitido a los desarrolladores pasar de desarrollar aplicaciones en lenguaje ensamblador, específicas

al hardware donde se iba a ejecutar, a desarrollar aplicaciones utilizando lenguajes de programación de alto nivel, completamente independientes del hardware donde se ejecutará. En la actualidad, la transformación que se realiza desde el lenguaje de alto nivel en el que se codifican los programas a a lenguaje máquina se hace de forma completamente automática por compiladores y ensambladores.

Siguiendo esta perspectiva histórica, MDA se presenta como la evolución natural en esta tendencia hacia la **abstracción**. Ahora el objetivo es construir aplicaciones que son solo son independientes del hardware sino de la plataforma de software donde se ejecutan y hacer de la generación del código fuente un proceso sino completamente automatizado, un proceso fuertemente automatizado.

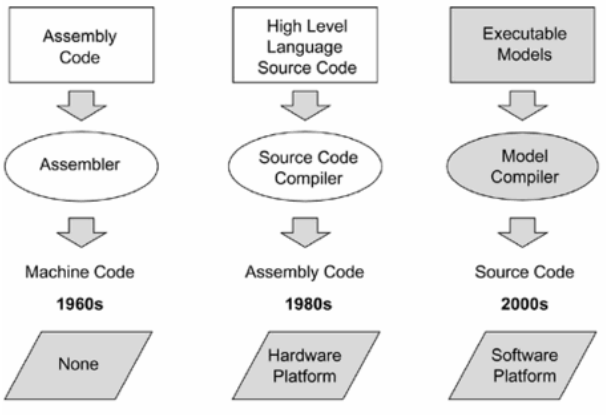

FIGURA 12: *EVOLUCIÓN HISTÓRICA DE LA ABSTRACCIÓN SOFTWARE*

Otra forma de observar históricamente la evolución de los paradigmas de desarrollo de software es considerar el esfuerzo puesto en la **reusabilidad**, como se muestra en la siguiente figura:

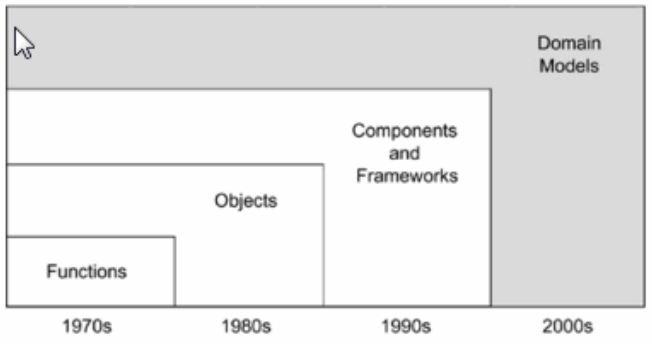

FIGURA 13: *EVOLUCIÓN HISTÓRICA DE LAS TÉCNICAS DE REUSABILIDAD DEL SOFTWARE*

La estandarización del uso de funciones, la programación orientada a objetos, los sistemas orientados a componentes, muestra como de forma constante el desarrollo del software ha evolucionado constantemente hasta maximizar la reusabilidad del software. En este sentido, los modelos de dominio propuestos por MDA también se presentan como la evolución natural hacia un grado mayor de reusabilidad.

LA NECESIDAD DE MDA

¿Qué es lo que motiva que el desarrollo de software progrese constantemente hacia una mayor abstracción y reusabilidad? En un principio, la motivación es puramente económica. Por un lado, desarrollar software es una actividad costosa, por lo que para desarrollar un sistema es frecuente necesitar de grandes inversiones. Otro aspecto a considerar es el rápido ritmo de cambio que afecta a las Tecnologías de la Información, lo que hace que los sistemas ya desarrollados se vuelvan obsoletos rápidamente. Cada vez que una nueva tecnología llega, hay que de nuevo invertir en rediseñar los sistemas de negocio o simplemente construirlos

desde cero. Así que hay que MDA se propone como un enfoque para proteger las inversiones en tecnología de los cambios y nuevos avances.

Es por esto que MDA separa la funcionalidad del sistema de su aplicación en una plataforma tecnológica específica. Para lograr esto, MDA se basa fuertemente en el modelado de software (mucho menos expuesto a cambios) definiendo dos modelos básicos:

1. EL Modelo Independiente de la Computación (CIM)
2. El Modelo Independiente de la Plataforma (PIM)
3. El Modelo de Plataforma específico (PSM)

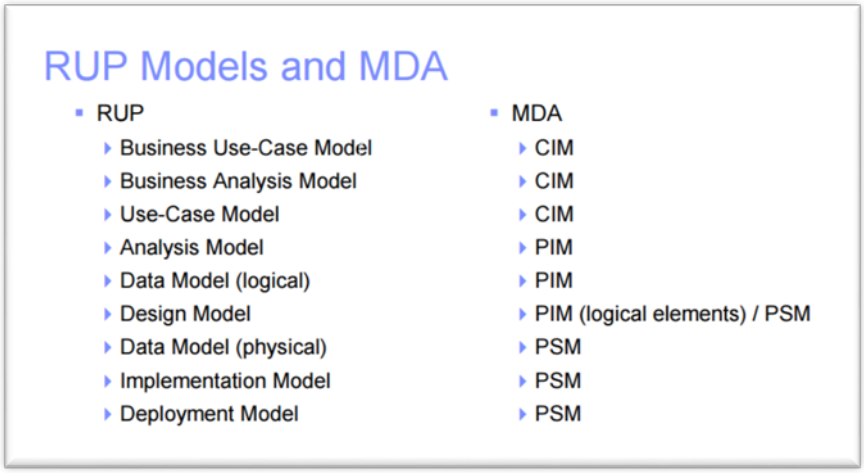

FIGURA 14: *RELACIÓN ENTRE RUP Y MDA* [21]

MDA considera al PIM como un modelo universal y de larga duración que sobrevivirá la tecnología cambiante y puede ser reutilizado en el futuro para adaptar los sistemas existentes a las nuevas tecnologías. El PIM se utiliza para generar modelos PSM, a partir de los cuales se genera el código fuente de la aplicación. Esas transformaciones se logran usando herramientas de desarrollo MDA.

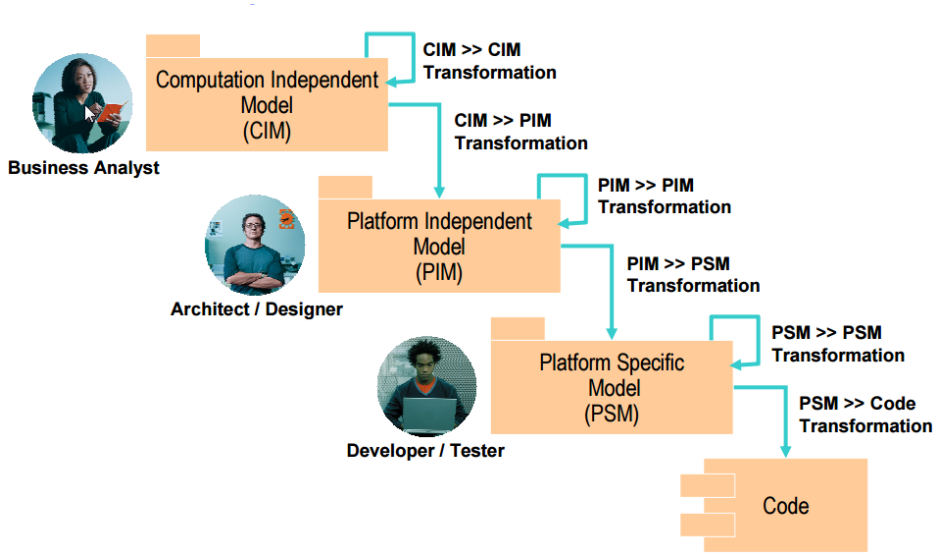

FIGURA 15: *MODELOS Y TRANSFORMACIONES DEFINIDOS EN LA TEORÍA MDA* [21]

LOS PRINCIPIOS MDA

En esta sección analizaremos los principios del desarrollo orientado a el modelo, MDA.

MODELOS

Un modelo es una serie de elementos que describen una realidad física, abstracta o hipotética. Un aspecto clave del desarrollo de modelos de sistemas es un uso adecuado de la abstracción y la clasificación. MDA permite crear diferentes modelos a diferentes niveles de abstracción y después relacionarlos de manera que juntos formen una implementación.

METAMODELOS

Es simplemente un modelo que se usa para describir otros modelos. Un metamodelo permite definir formalmente conceptos y reglas en un lenguaje de modelado, que utilizaremos para describir otros modelos. Un ejemplo de un metamodelo es UML.

La jerarquía de un metamodelo se puede definir en 3 capas que son universales. OMG ha estandarizado estas capas para facilitar la comunicación. La definición se presenta a continuación.

- M0: En la capa M0 están los datos de la aplicación. En este nivel no se habla de clases ni atributos, sino de entidades físicas que existen en el sistema. Por ejemplo, todas las instancias de los objetos de la aplicación en tiempo de ejecución o las filas de una base de datos.
- M1: en la capa M1 está la aplicación. Las clases de un sistema orientado a objetos, las definiciones de una base de datos relacional. El modelado de la aplicación tiene lugar a este nivel.
- M2: en la capa M2 está el metamodelo. Las herramientas que nos permiten definir un modelo se encuentran en esta capa.
- M3: es la capa donde se definen los meta-metamodelos. En esta capa se describen las propiedades que se pueden utilizar para definir un metamodelo. Las herramientas que permiten la definición de lenguajes y metamodelos, operan a este nivel.

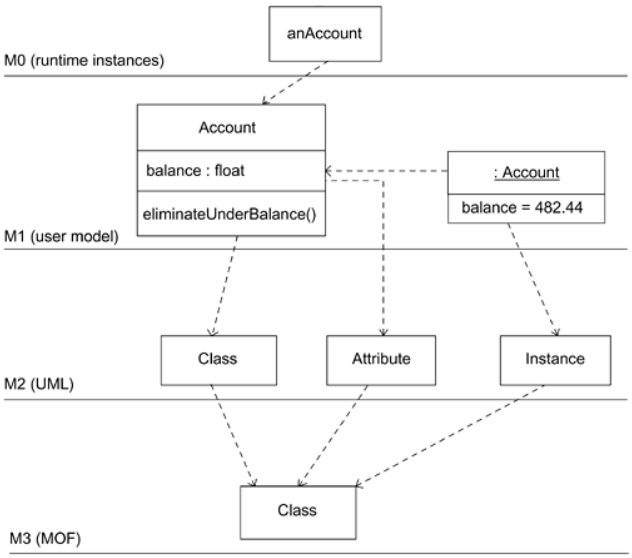

FIGURA 16: *EJEMPLO DE LA JERARQUÍA DE 4 CAPAS* [22]

LENGUAJES DE CONSTRUCCIÓN

Los lenguajes de construcción se crean, para comunicarse, de forma natural durante el desarrollo. MDA proporciona una forma de formalizar esos lenguajes, no solo para permitir la comunicación entre los miembros del equipo sino también para permitir que las asociaciones entre modelos se puedan expresar en esos idiomas (lo que da lugar a la automatización o comunicación entre máquinas).

ELABORACIÓN DE MODELOS

La elaboración de modelos simplemente captura la idea de que los modelos se pueden modificar, ya sea cambiando el modelo o modificando el código generado.

MODELOS EJECUTABLES

Los modelos ejecutables son modelos que tienen todo lo necesario para producir la funcionalidad deseada un dominio. No son el sistema completo, ya que se produce de la composición de varios modelos diferentes.

ASOCIACIÓN ENTRE MODELOS (MAPPING)

Como ya se ha comentado más arriba, una característica clave de MDA es la posibilidad de crear diferentes modelos para luego relacionarlos entre sí. Esto permite abstraernos de ciertos elementos en un modelo para describirlos en otro modelo. La asociación de modelos es la definición la reglas mediante funciones de asociación (que gobiernan las relaciones entre modelos. En la definición de estas reglas siempre aparece un modelo como entrada y otro como salida del proceso de mapping.

ESPECIFICACIÓN MDA DEL OMG

En esta sección se introducirá la especificación de MDA que proporciona el OMG, la cual se deriva de los principios de MDA que se han visto en la sección anterior. MDA es la arquitectura base de los estándares de esta organización desde 2001 y el objetivo que persiguen es el dotar de un estándar que, adhiriéndose a los principios MDA, unifique todas las fases del desarrollo y la integración de un sistema: el modelado de negocio, el modelado de la arquitectura de la aplicación, el desarrollo, la implementación, mantenimiento y evolución.

MOF (META-OBJECT FACILITY)

Como ya se ha comentado más arriba, el desarrollo de software en el MDA comienza definiendo un Modelo Independiente de la Plataforma (PIM) que describe la funcionalidad y comportamiento de una aplicación. Este modelo es construido utilizando un lenguaje de construcción de modelos. Así se obtiene un modelo estable ante la evolución de la tecnología, maximizando así el Retorno de la Inversión (ROI) del software.

Pues bien, MOF es un estándar para la creación de metamodelos. Definir el PIM de nuestro sistema usando un metamodelo basado en MOF, garantiza que los modelos se puedan almacenar en un repositorio MOF-compatible, ser analizado y transformado por las herramientas compatibles compatibles con MOF, y más tardes transmitido usando el estándar XMI a través de una red. Esto no limita los tipos de modelos que puede utilizar ya que los lenguajes basados en MOF son capaces de modelar la estructura, el comportamiento y los datos. UML es un buen ejemplo de un lenguaje de modelado basado en MOF, pero no es el único.

UML

OMG define una serie de lenguajes de modelado que son adecuados para definir PIMs o las PSMs. El más conocido y ampliamente utilizado es UML. Como parte de UML también encontramos:

- OCL: Lenguaje de Restricción de Objetos, es un lenguaje de consulta y de expresión para UML, que es una parte integral de la norma UML. El término "restricción", al igual que el término nebulosa en astronomía que nada tiene que ver con nubes, poco tiene que ver con la realidad. Hoy OCL es un lenguaje de consulta completa, comparable a SQL en su poder expresivo.
- Perfiles UML: permiten la adaptación de un metamodelo existente con constructos que son específicos para un dominio particular, plataforma, o método. Los elementos clave de perfiles son estereotipos, que se extienden del vocabulario básico del UML y restricciones, que especifican las condiciones dentro de un modelo que han de cumplirse para que el modelo sea "bien formado". El perfile se considera un subconjunto de UML con restricciones adicionales y adecuado para un uso específico. Utiliza la notación UML y OCL.

3.2.5.2 Creación de las vistas arquitectónicas

Para el diseño se diseñan 2 diagramas, un diagrama de componentes y un diagrama de comunicación, en los cuáles se documenta la opción elegida de una arquitectura bróker.

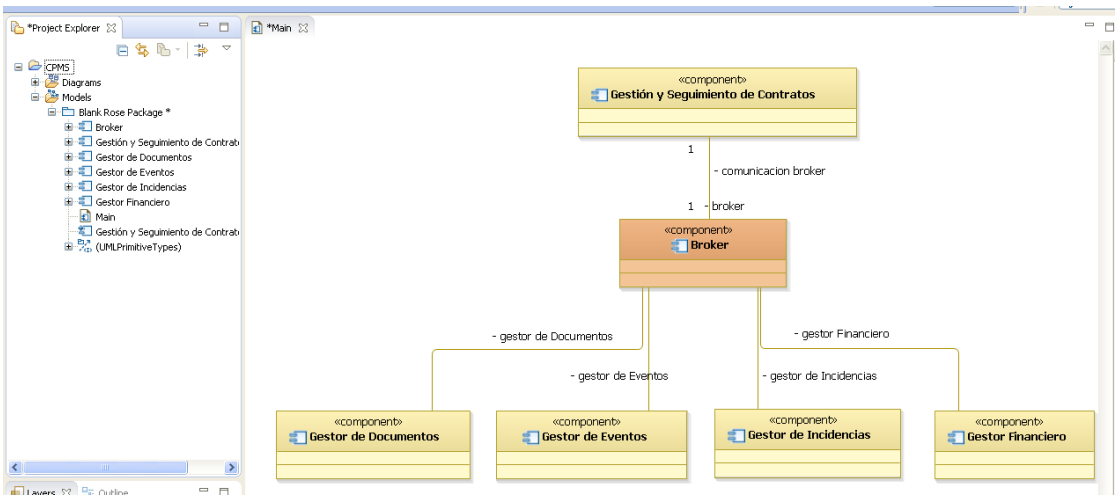

FIGURA 17: *DIAGRAMA DE COMPONENTES CON LA ARQUITECTURA SELECCIONADA PARA LA MEJORA*

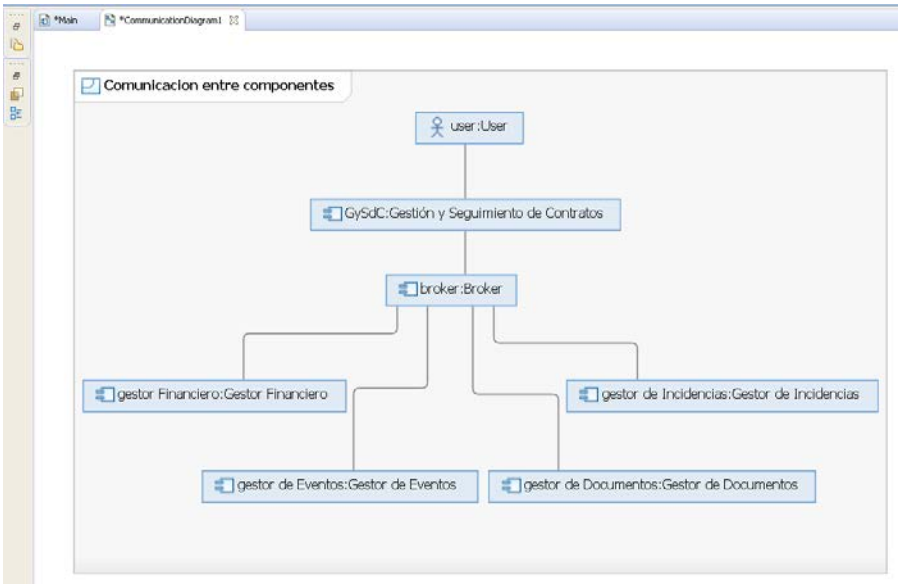

FIGURA 18: *DIAGRAMA DE COMUNICACIÓN CON LA ARQUITECTURA SELECCIONADA PARA LA MEJORA*

En el primer capítulo de esta parte, se han documentado las 2 iteraciones ADD iniciales que han ido diseñando la nueva arquitectura de la plataforma.

3.2.6 Transición a las iteraciones subsiguientes

Estas dos iteraciones iniciales han sido puramente analíticas y ya han justificado los cambios a realizar en la arquitectura de la plataforma de GPyC, en términos de los atributos de calidad que se van a mejorar. Las siguientes iteraciones se van a centrar más en el diseño de los componentes incrementalmente utilizando la herramienta MDA. Estas iteraciones deberán seguir los siguientes pasos, para realizar la modificación incremental de la plataforma:

1. **Modificación del modelo para incluir el diseño del componente**. Se identificarán los servicios que cada componente debe exponer usando los casos de uso incluidos en el Anexo. Se crearán los artefactos UML necesarios para ilustrar con más detalle cómo se comunicarán los componentes.
2. **Generación del código a partir del modelo diseñado**. Se crearán y aplicarán las trasformaciones necesarias para generar el código asociado al modelo.

3. **Desarrollo del componente**. El equipo técnico realizará los trabajos de programación necesarios para que el componente sea operacional.
4. Vuelta al paso 1

En este trabajo se van a documentar los pasos 1 y 2. El paso 3 correspondería al trabajo que los programadores deberán realizar para que el componente en particular sea completamente funcional y, evidentemente, este paso (que en la práctica estarían compuestos de varias sub-tareas) no se va a documentar ya que por un lado está fuera del objetivo del trabajo de fin de máster y por el otro ya se ha estimado que el esfuerzo de implementación de la mejora requiere 465 mdays.

Aunque se podría pensar dado que ya se ha justificado la mejora no es necesario documentar las siguientes iteraciones, en este trabajo se considera que es importante para mostrar la resolución de una de las ideas que se defiende en este trabajo: la introducción del diseño orientado a modelos en el proceso de desarrollo software.

Hay que puntualizar que tanto la evaluación CBAM de las estrategias arquitectónicas, como las posibles evaluaciones de las otras iteraciones, son estimaciones de planteamiento; no constatadas, en términos de repercusiones reales de costes y beneficios, ya que no se tiene acceso al código fuente ni derechos intelectuales para modificar el código, y por lo tanto no se pueden realizar las mediciones empíricas tras su ejecución y uso. Estas de todas formas justifican la mejora, es decir sirven para justificar el embarcarse en la tarea de rediseño, o lo que es lo mismo permiten a la organización asignar los recursos necesarios para la realización de la mejora. Lo más importante es que además permiten capturar en el proceso de decisión los aspectos económicos de las estrategias arquitectónicas. En cualquiera de las siguientes iteraciones que se documentan habría que realizar una evaluación, similar a la iteración 2, cuando se entienda que se está tomando una decisión arquitectónica (a nivel de componente), para ver el impacto en los atributos de calidad del sistema de acuerdo a los objetivos marcados.

3.2.7 Iteración 2. Diseño del Bróker

3.2.7.1 Modificación del modelo

En esta sección vamos se va a documentar la modificación del modelo. Se crea un modelo independiente para cada uno de los componentes que se va a diseñar, es decir en esta y las subsiguientes iteraciones se crearán los siguientes modelos en RSA:

- Broker Model
- Document Management Model
- Events Management Model
- Issues Management Model
- Financial Management Model

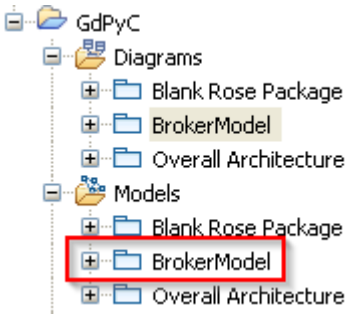

FIGURA 19: *CREACIÓN DEL MODELO PARA EL BRÓKER EN RSA*

El bróker funcionará utilizando una estructura similar a la de muchos frameworks, donde todas las peticiones se gestionan por un controlador principal, en este caso una clase llamada BrokerController. Esta clase en base a los parámetros recibidos delega las acciones pertinentes en un gestor (Handler) que decide que componente y servicio debe invocarse. Esto se describe en el siguiente diagrama de clases.

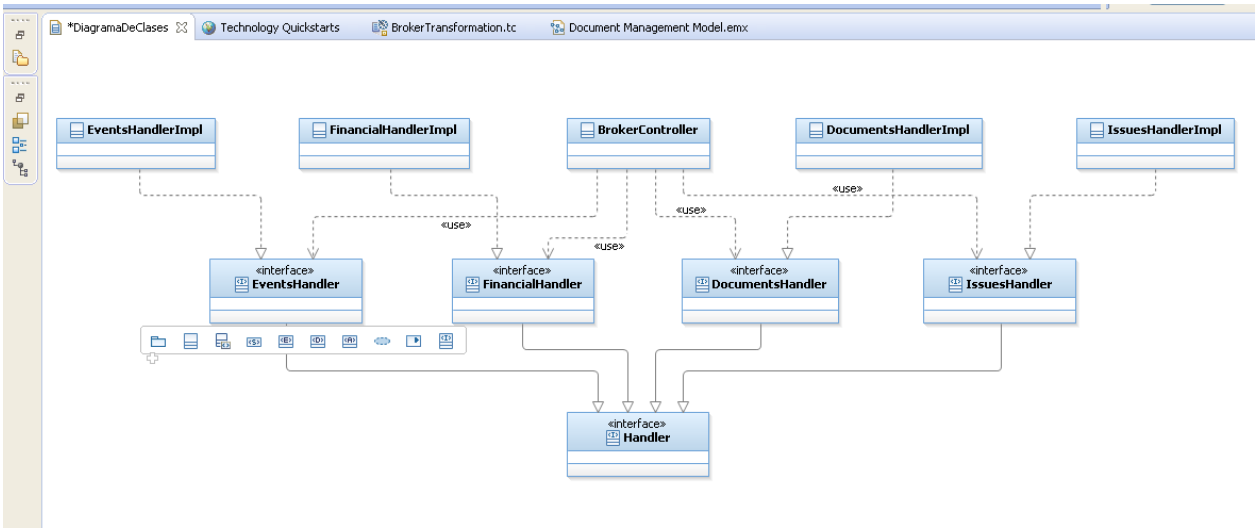

FIGURA 20: *DIAGRAMA DE CLASES DEL BRÓKER*

3.2.7.2 Generación del código

Para generar el código se ha creado una transformación UML a Java. Como ya se ha comentado se crea un proyecto independiente, llamado "bróker-project", como el destino de las transformaciones. Una captura de la transformación se incluye a continuación.

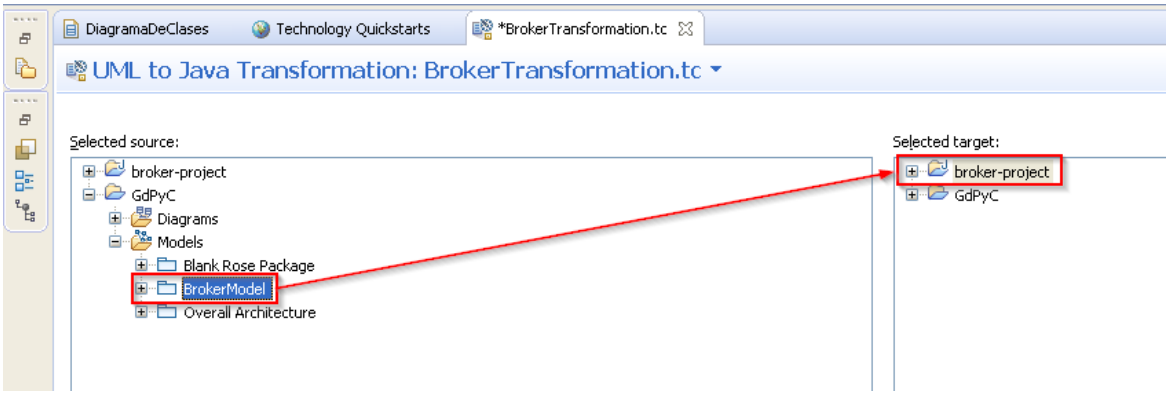

FIGURA 21: *DIAGRAMA DE CLASES DEL BRÓKER*

Y el resultado de la transformación se puede también ver la siguiente captura.

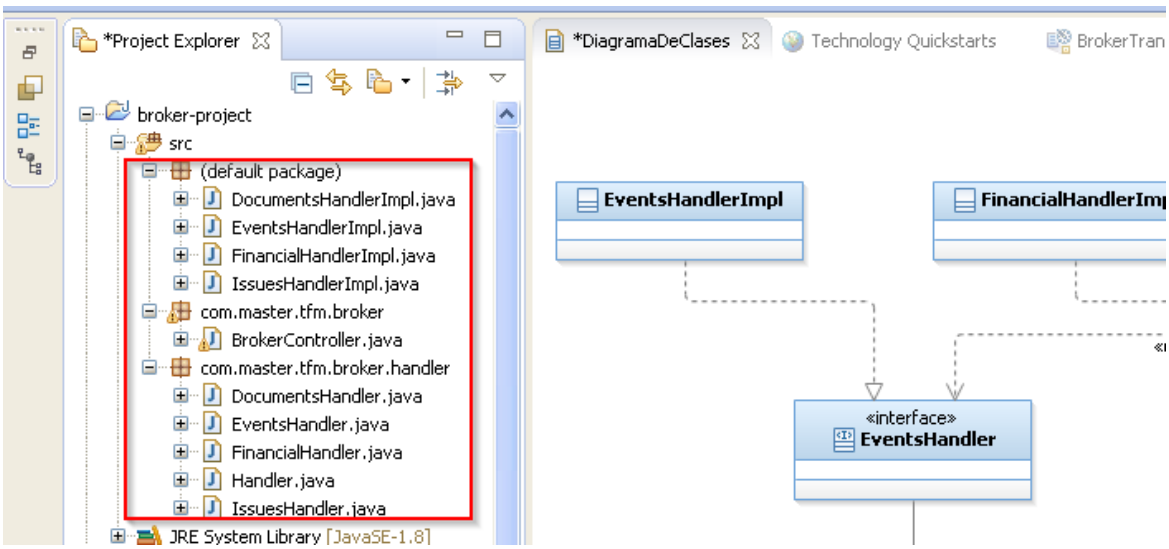

FIGURA 22: *CÓDIGO GENERADO*

Como se puede ver, se ha creado el esqueleto de la aplicación, que como se ha comentado en secciones anteriores es la base en la que los programadores pueden empezar a trabajar. A medida que se vayan modelando lo siguientes componentes, el modelo del bróker también se irá actualizando.

3.2.7.3 Implementación

En esta fase de implementación el equipo técnico puede trabajar en aspectos específicos del bróker, como como reaccionar cuando alguno de los componentes no está operativo, etc.

3.2.8 Iteración 3. Diseño del componente de Gestión de Documentos

3.2.8.1 Modificación del modelo

En esta iteración se modificará los modelos para diseñar el componente de Gestión de documentos. Los servicios disponibles que se incluyen en la modificación del modelo se han extraído tomando como base la lista de casos de uso proporcionada en los anexos. En particular, se modificará el Bróker y el Gestor de Documentos para incluir los siguientes métodos relacionados con la gestión de documentos:

- void updateDocument(Object user, Object document);
- void removeDocumentVersion(Object document, Object version);
- void getDocumentVersions(Object document);
- void getDocument(Object document);
- void deleteDocument(Object document);
- Object addDocument(Object document, Object folder);
- Object viewFolderContents(Object folder);
- void updateFolder(Object folderInfo);
- void deleteFolder(Object folder);
- void createFolder(Object parent);

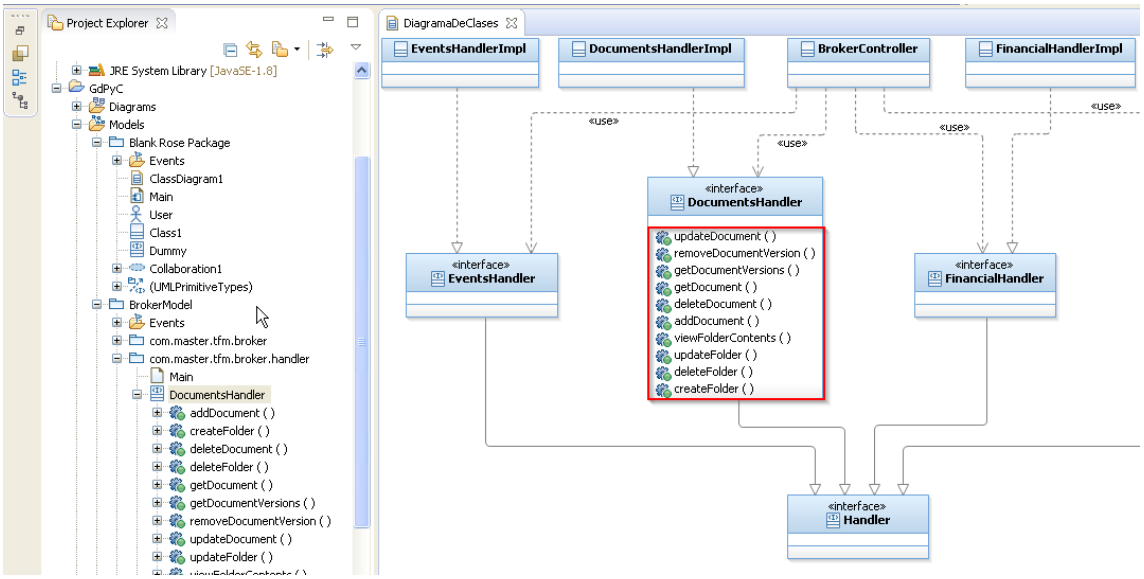

FIGURA 23: *MODIFICACIÓN DEL MODELO DEL COMPONENTE BRÓKER*

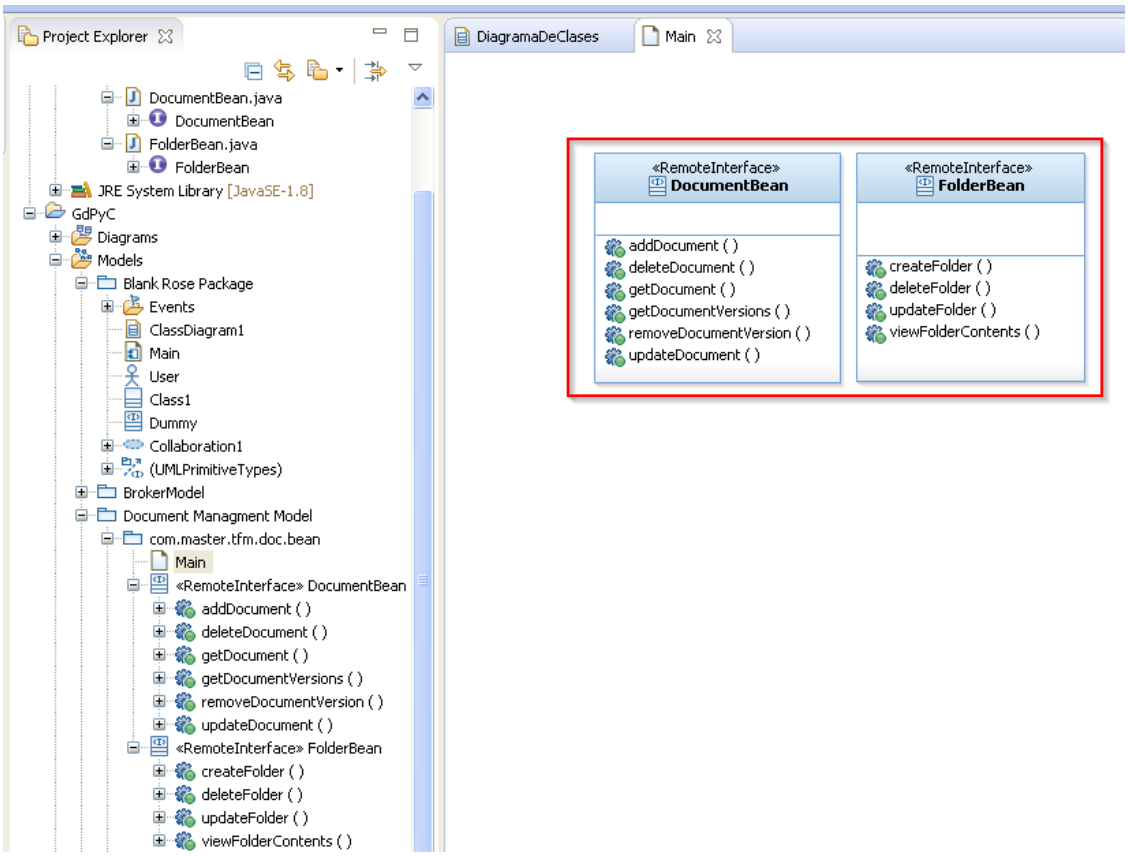

FIGURA 24: *ESQUELETO DEL MODELO DEL COMPONENTE DE GESTIÓN DE DOCUMENTOS*

3.2.8.2 Generación del código

Para generar el código se crean 2 transformaciones. Una en un proyecto Java genérico y otra para generar proyectos EJB3. Para el modelo del gestor de documentos, se ha utilizado el perfil UML para Java EJB. Por este motivo las clases que se muestran en la captura de pantalla incluyen la anotación "*RemoteInterface*". Las transformaciones disponibles permiten simplemente generar el mismo modelo en un proyecto Java genérico o en un proyecto Java EJB3. Esta práctica se mantendrá en el diseño de los siguientes componentes

para ilustrar una de las características claves de MDA y es la transformación en diferentes modelos a partir de un modelo base.

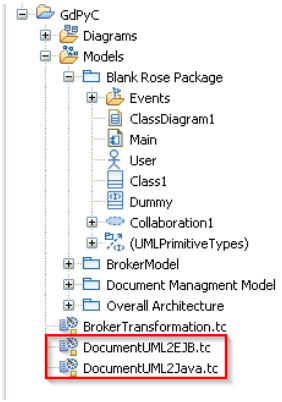

FIGURA 25: *TRANSFORMACIONES CREADAS*

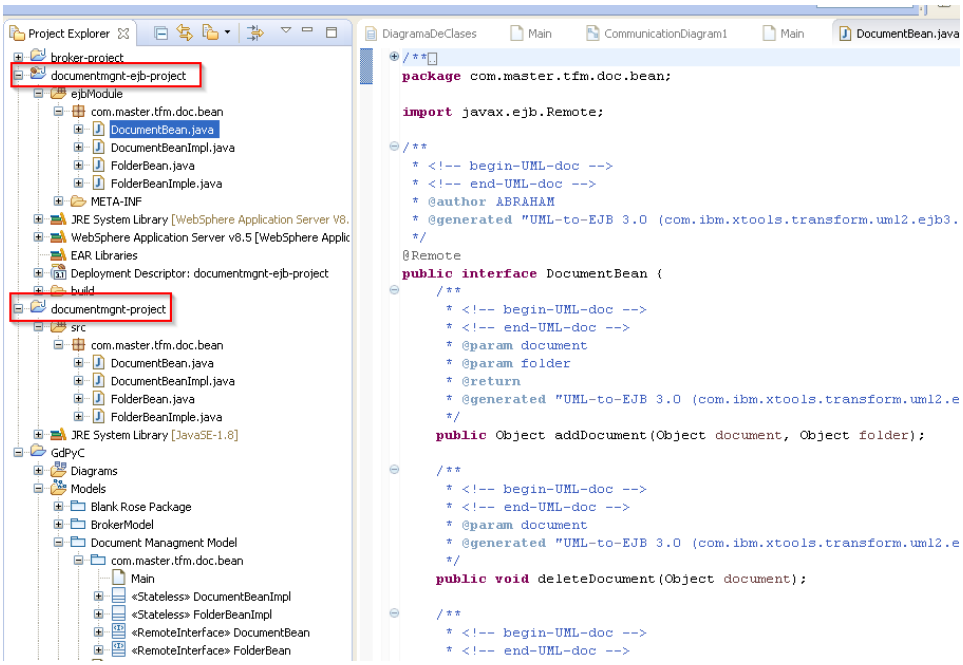

FIGURA 26: *PROYECTOS Y CÓDIGO GENERADOS*

3.2.8.3 Implementación

En esta fase de implementación gran parte del código que se va a utilizar era parte del componente GPC. El equipo técnico deberá transferir las clases necesarias al nuevo proyecto una vez que el esqueleto esté disponible, de forma que se soporten los servicios que ofrece el componente.

3.2.9 Iteración 4. Diseño del componente de Gestión de Eventos

3.2.9.1 Modificación del modelo

En esta sección se documenta el diseño del esqueleto del componente de gestión de eventos. El diseño consiste principalmente en la definición de los servicios que proporcionará este componente a través de las interfaces remotas que tendrá disponibles. Se definen los siguientes servicios:

- void addEvent(Object event, Object calendars);
- Object getEvent(Object eventId);

- void modifyEvent(Object eventId, Object list);
- void deleteEvent(Object eventId, Object calendars);
- void getEvents(Object calendar, Object from, Object to);
- Object getEventDocuments(Object eventId);
- Object getEventAttendees(Object eventId);
- void setEventAttendees(Object eventId, Object users);
- void setEventDocuments(Object eventId, Object documents);

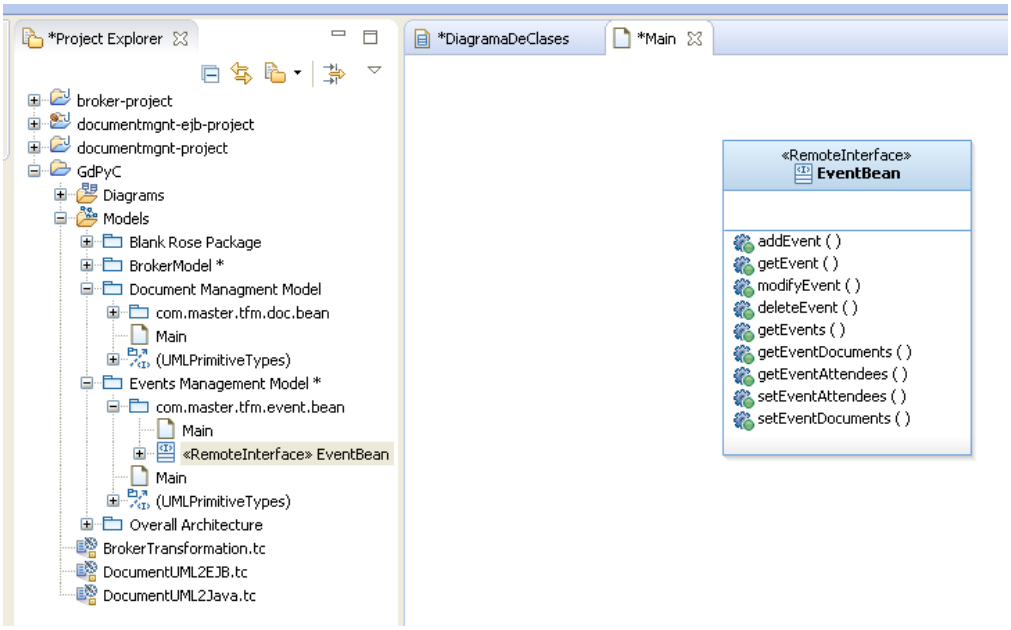

FIGURA 27: *ESQUELETO DEL MODELO DEL COMPONENTE DE GESTIÓN DE EVENTOS*

También se modifica el Handler apropiado en el Bróker para definir estos métodos.

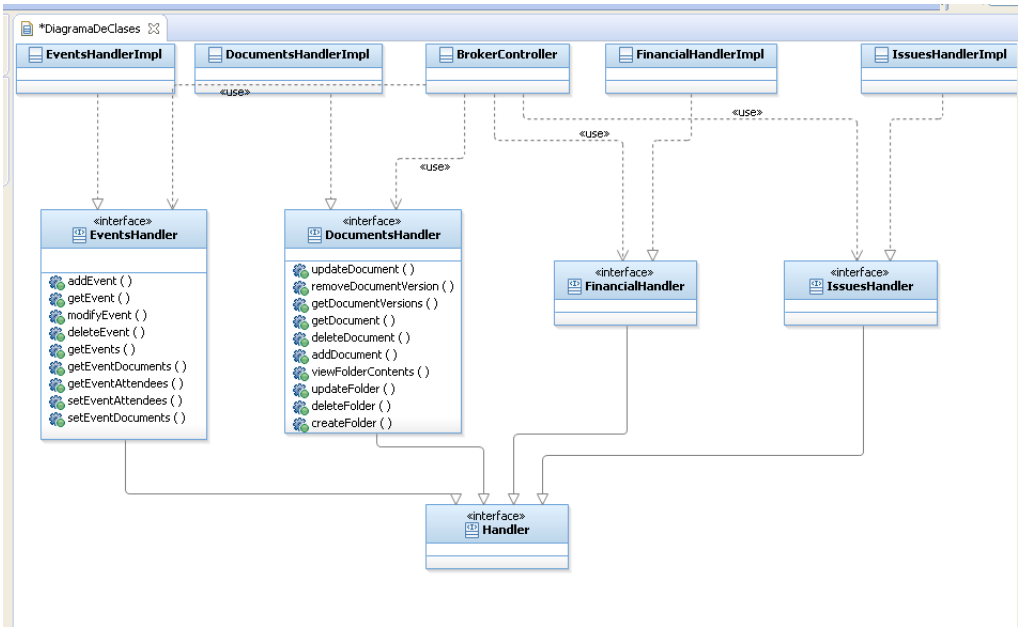

FIGURA 28: *MODIFICACIÓN DEL BRÓKER*

3.2.9.2 Generación del código

Se continúa con el mismo proceso y se generan 2 transformaciones una a un proyecto Java genérico y otra a un proyecto en Java EJB3.

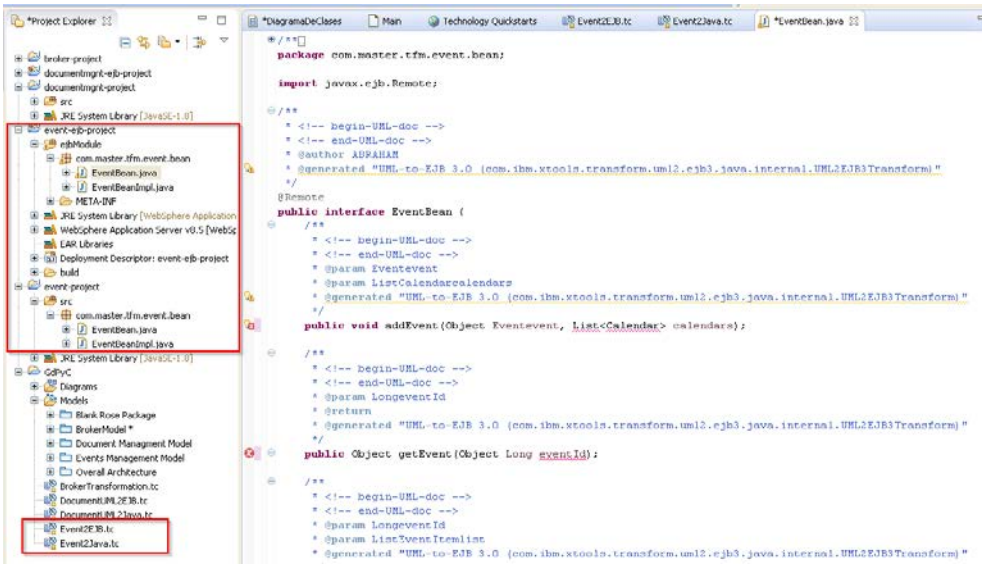

FIGURA 29: *TRANSFORMACIONES Y PROYECTOS CREADOS*

3.2.9.3 Implementación

La fase de implementación consistirá en integrar el código que pertenecía al componente GPC y se desarrollará el código que gestionará los servicios como remotos.

3.2.10 Iteración 5. Diseño del componente de Gestión de Incidencias

3.2.10.1 Modificación del modelo

En esta iteración, la más simple, se documenta el diseño en RSA del módulo de gestión de incidencias. Este simplemente consistirá de 3 servicios:

- Object viewIssue(Object issueId);
- void addIssue(Object issue);
- void modifyIssue(Object issueId);

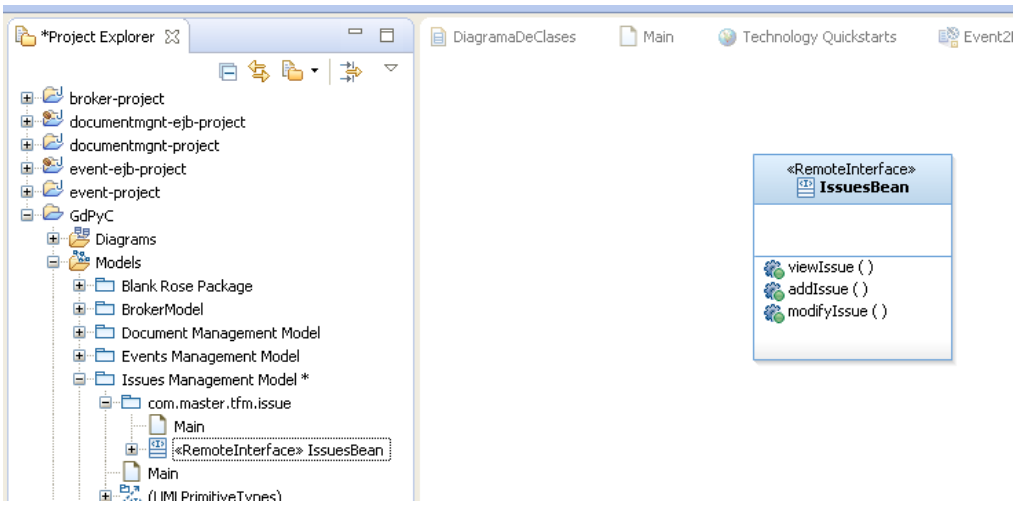

FIGURA 30: *ESQUELETO DEL MODELO DEL COMPONENTE DE GESTIÓN DE INCIDENCIAS*

Y la correspondiente actualización del Bróker.

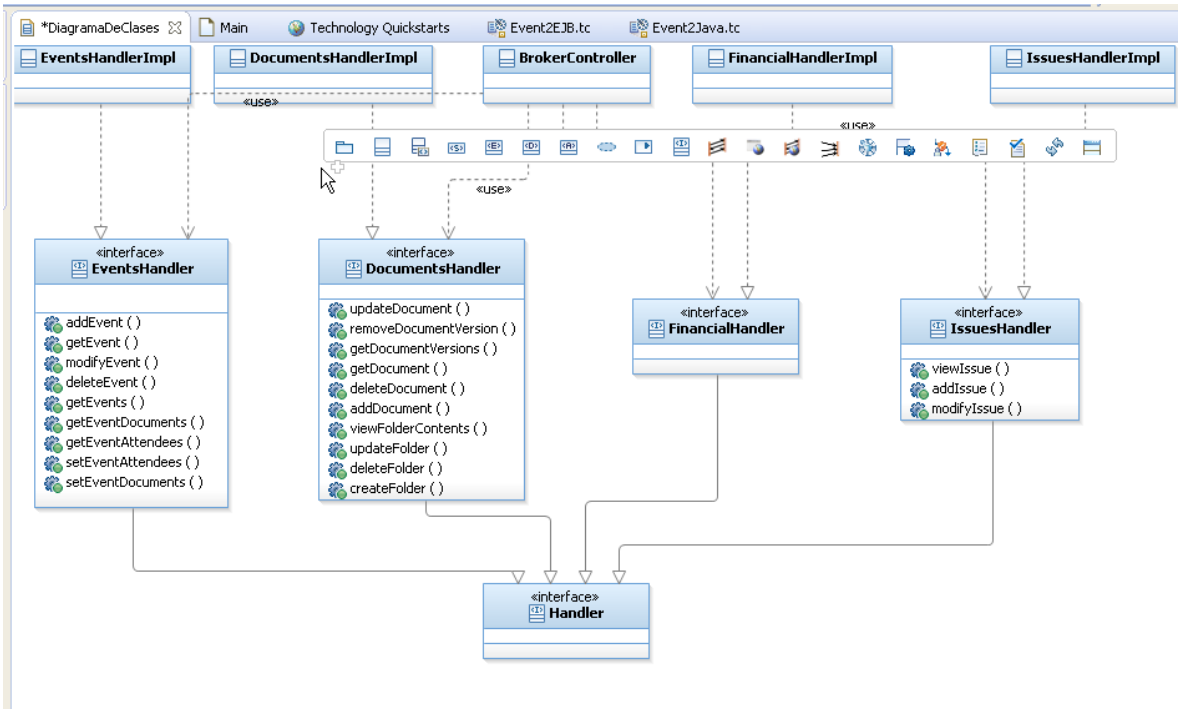

FIGURA 31: *MODIFICACIÓN DEL BRÓKER PARA INCLUIR LOS SERVICIOS DE GESTIÓN DE INCIDENCIAS*

3.2.10.2 Generación del código

Se vuelven a crear 2 transformaciones como en los ejemplos anteriores.

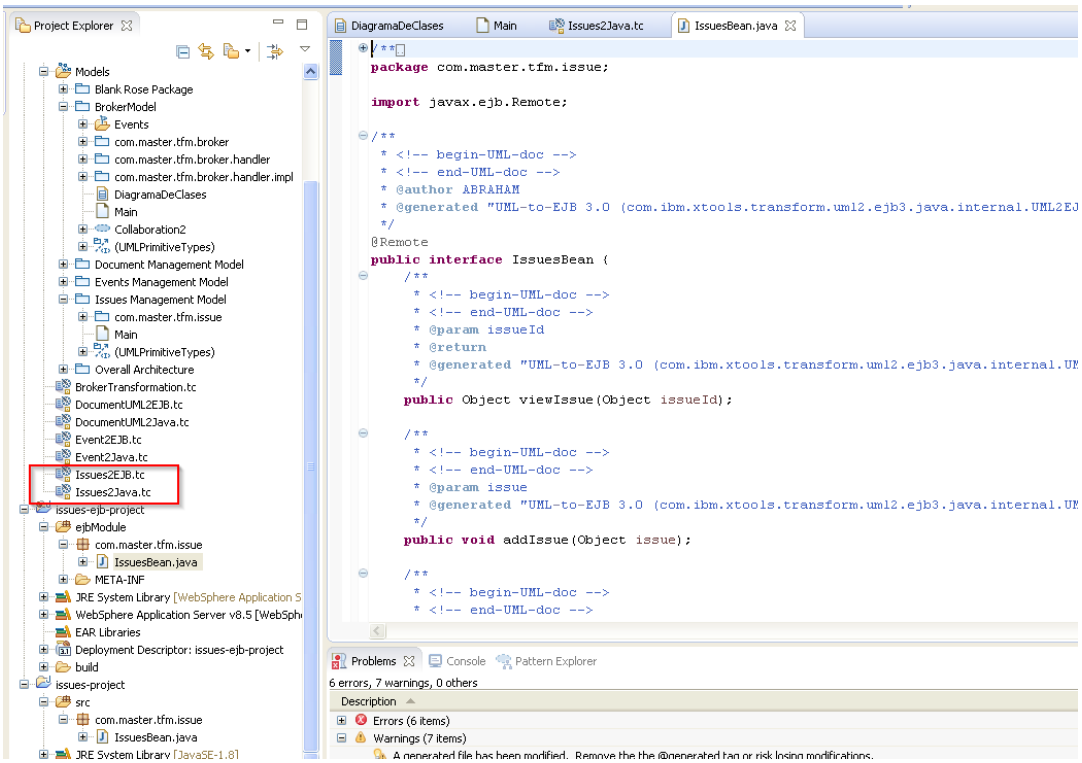

FIGURA 32: *TRANSFORMACIONES Y PROYECTOS CREADOS PARA EL COMPONENTE DE GESTIÓN DE INCIDENCIAS*

3.2.10.3 Implementación

La fase de implementación consistirá en integrar el código que pertenecía al componente GPC y se desarrollará el código que gestionará los servicios como remotos.

3.2.11 Iteración 6. Diseño del componente de Gestión Financiera

3.2.11.1 Modificación del modelo

El componente de Gestión Financiera estará proporcionará los siguientes servicios relacionados:

- Object viewFinancialData(Object contractId);
- void addMovement(Object movement, Object contractId);
- void modifyMovement(Object movementId, Object contractId);
- void cancelMovement(Object movementId);
- void createPayment(Object payment, Object contractId);
- void modifyPayment(Object paymentId);
- void createInvoice(Object invoice, Object paymentId, Object contractId);
- Object modifyInvoice(Object invoiceId);
- void cancelInvoice(Object invoiceId);
- void deletePayment(Object paymentId);

Estos servicios se dividirán en 2 interfaces remotas FinancialBean y PaymentBean..

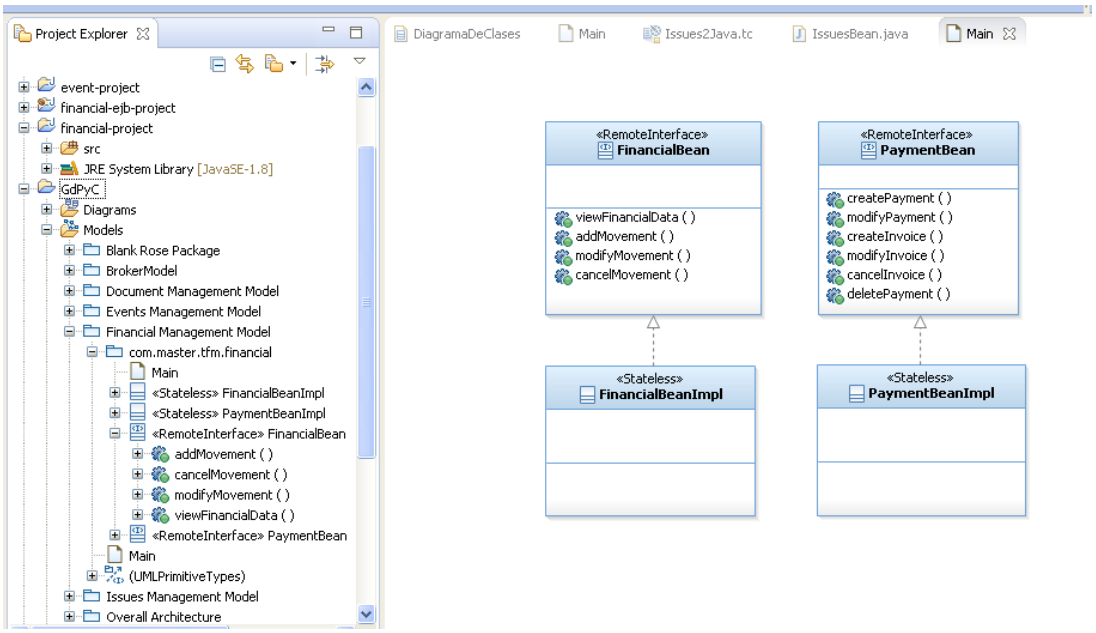

FIGURA 33: *ESQUELETO DEL MODELO DEL COMPONENTE DE GESTIÓN FINANCIERA*

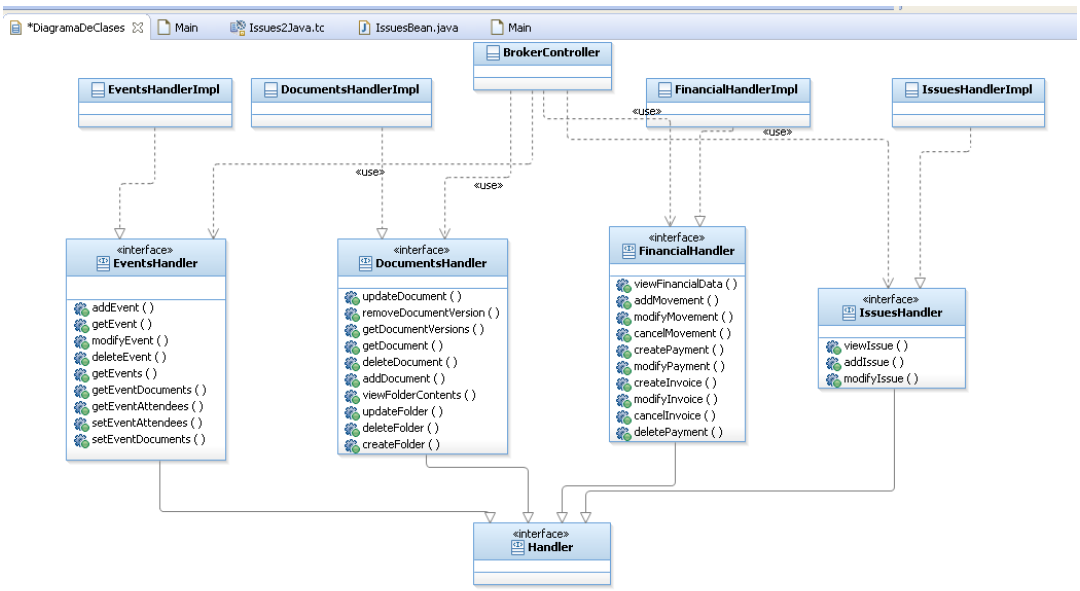

FIGURA 34: *MODIFICACIÓN DEL BRÓKER PARA INCLUIR LOS SERVICIOS DE GESTIÓN FINANCIERA*

3.2.11.2 Generación del código

En esta última iteración, se crean 2 nuevas transformaciones para este proyecto, como las de los ejemplos anteriores, pero también una 3 compuesta. Este tipo de transformación permite ejecutar secuencialmente un conjunto de transformaciones, que en este caso serán todas las transformaciones creadas en las iteraciones anteriores. Esta es una forma simple de regenerar todo el código.

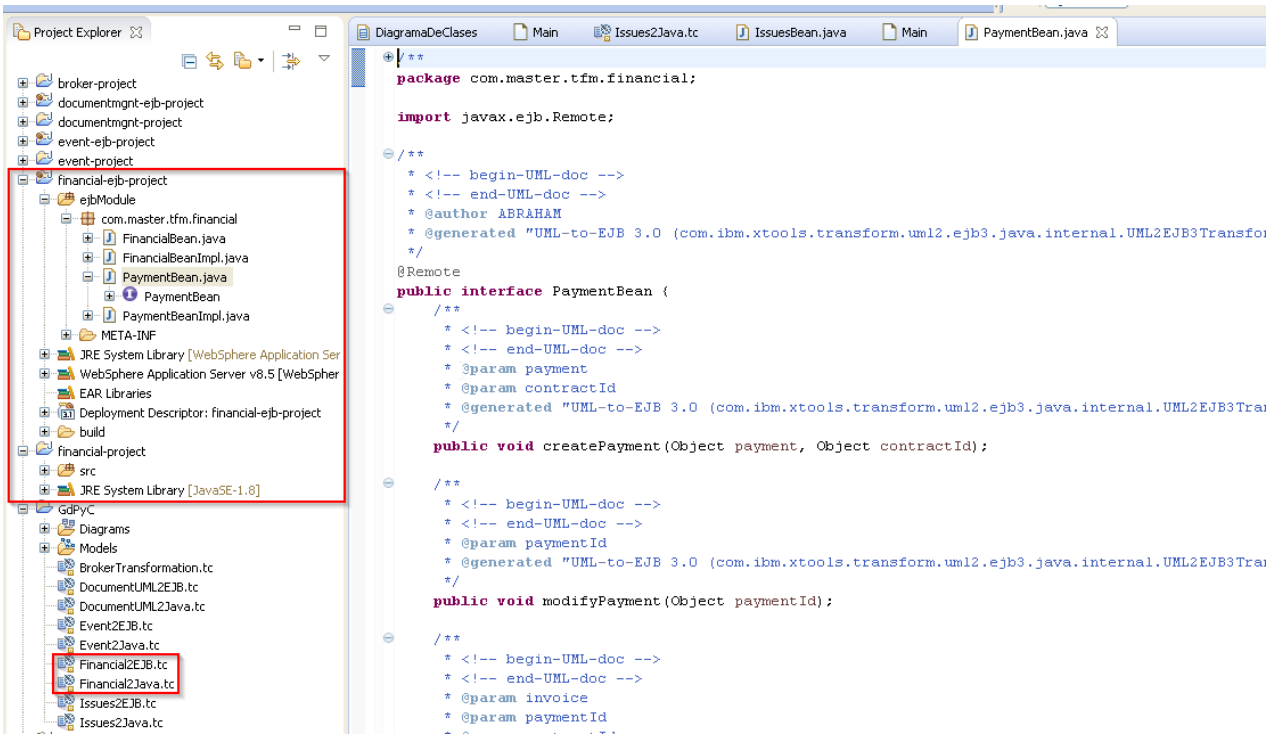

FIGURA 35: *TRANSFORMACIONES Y PROYECTOS CREADOS PARA EL COMPONENTE DE GESTIÓN FINANCIERA*

3.2.11.3 Implementación

La fase de implementación consistirá en integrar el código que pertenecía al componente GPC y se desarrollará el código que gestionará los servicios que se han definido como remotos.

Parte 4. Mejora de un Sistema de Licitación Electrónica – Conclusiones

En este trabajo se ha presentado la mejora de los atributos de interoperabilidad y modificabilidad de la plataforma de GPC, a través del rediseño de su arquitectura, cumpliendo con el primer objetivo del trabajo. La selección de la estrategia a seguir ha utilizado los atributos de calidad, pero también se ha incluido el ROI en el proceso de decisión, Además, se ha utilizado Lightweight ATAM que realiza una evaluación de la arquitectura existente menos costosa que ATAM. Esto cumple con el segundo objetivo del trabajo, el de reducir los recursos empleados para el rediseño.

Tanto los atributos de calidad objetivo de la mejora como ROI pueden ser medidos, aunque sea a posteriori y, por lo tanto, son medibles. Pese a esto, la mera aplicación de CBAM no garantiza que se consigan los resultados estimados. La realidad es que ninguna de la literatura estudiada presenta un método que garantice de forma inequívoca la consecución de los atributos de calidad a un nivel específico con anterioridad a la implementación/ rediseño el sistema. Menos aún, que garantice de antemano el beneficio suficiente para embarcarse en su desarrollo. Esto no significa que no haya que incluir estos elementos en el proceso de decisión, aunque constituyan estimaciones teóricas, como ya se ha justificado en el trabajo.

La modificación de la arquitectura se ha documentado usando una herramienta MDA, que permite que los modelos de la arquitectura generados constituyan artefactos útiles y accionables, ya que permiten generar el código fuente esqueleto de los componentes. Por lo tanto, el proceso utilizado se adecúa a los objetivos de negocio de la empresa, que sigue la tendencia de utilizar procesos ágiles o evolutivos. Como se ha mostrado en el trabajo, estas metodologías no solo reducen costes, sino que aumentan el ratio de éxito de los proyectos.

Las iteraciones del modelado de la arquitectura documentadas en este trabajo son una aproximación teórica y de alto nivel de abstracción. Son una propuesta para rediseñar la arquitectura utilizando un proceso de diseño incremental e iterativo, es decir ágil. Estas iteraciones constituyen el punto de partida de la planificación, pero, en la práctica, lo normal es que sufran adaptaciones y modificaciones cuando todos los "stakeholders" del proyecto se reúnan. Por ejemplo, es posible que se necesiten iteraciones adicionales para modelar otros detalles significativos de la arquitectura.

Pese a que todos los objetivos del trabajo se han cumplido, lo que quedaría por realizar es la constatación de la mejora cuando esta se implemente. La justificación de la mejora se ha de obtener antes de comenzar el rediseño y serviría para justificar la asignación de recursos y el comienzo la actividad de rediseño del componente GPC. Esta constatación no se ha incluido como objetivo del trabajo porque no se tiene acceso al código fuente ni derechos intelectuales para modificar el código, y por lo tanto no es posible realizar las mediciones empíricas que ofrecerán los resultados reales tras su implementación, ejecución y uso. Más aún, las métricas de modificabilidad miden el esfuerzo necesario para implementar nuevos cambios. Por lo tanto, no pueden siquiera ser medidas inmediatamente después que se implemente la mejora, sino que habría que esperar a que se realicen nuevos cambios en el sistema. De todas formas, aunque la constatación queda fuera de los objetivos marcados, es una parte fundamental del proyecto de rediseño. La constatación nos permitirá, por una parte, obtener el nivel real de mejora alcanzado por los atributos cualitativos y el ROI y, por otra parte, evaluar la precisión de las estimaciones realizadas. Esta información nos permitirá refinar el proceso seguido.

Parte 5. Anexos

Anexo 1. CBAM

Worksheet in C
Users Abraham Gime

Anexo 2. Lista de módulos y casos de uso

MÓDULOS DEL SISTEMA	*CASO DE USO*
MÓDULO DE BÚSQUEDAS	BÚSQUEDA RÁPIDA
	BÚSQUEDA AVANZADA
	RESULTADOS DE LA BÚSQUEDA
MÓDULO DE GESTIÓN DE CUENTAS	VER CUENTAS
	VER DETALLES DE UNA CUENTA
	CREAR UNA CUENTA
	MODIFICAR UNA CUENTA
	ACTIVAR / DESACTIVAR UNA CUENTA
	BORRAR UNA CUENTA
MÓDULO DE GESTIÓN DE GRUPOS	VER UN GRUPO DE CUENTAS
	VER UN GRUPO DE CUENTAS DETALLES
	CREAR UN GRUPO DE CUENTAS
	MODIFICAR UN GRUPO DE CUENTAS
	ELIMINAR UN GRUPO DE CUENTAS 28
MÓDULO DE GESTIÓN DE ROLES EN CONTRATOS	VER ROLES
	VER DETALLES DE UN ROL
	ASIGNAR UN ROL A UN ESPACIO DE TRABAJO
	MODIFICAR ROL EN ESPACIO DE TRABAJO
	ELIMINAR ROL EN ESPACIO DE TRABAJO
	ASIGNAR ROL/GRUPO A ESPACIO DE TRABAJO
MÓDULO DE GESTIÓN DE PERMISOS	VER ROLES
	VER PERMISOS DE UN ROL
	CREAR ROL
	MODIFICAR ROL
	ELIMINAR ROL
MÓDULO DE AUDITORÍA	
MÓDULO DE ESTADÍSTICAS	
MÓDULO DE GESTIÓN DE ORGANIZACIONES	BUSCAR
	VER DETALLES ORGANIZACIÓN
	CREAR ORGANIZACIÓN
	MODIFICAR ORGANIZACIÓN
	IMPORTAR/ EXPORTAR ORGANIZACIÓN
MÓDULO DE GESTIÓN y SEGUIMIENTO DEL CONTRATO	CREAR ESPACIO DE TRABAJO DEL CONTRATO
	VER INFORMACIÓN DEL CONTRATO
	CREATE CONTRACT WORKSPACE
	MODIFICAR EL ESPACIO DE TRABAJO

	ARCHIVAR CONTRATO
	ÁREA DE CONTRATOS ARCHIVADOS
	RESTAURAR ESPACIO DE TRABAJO ARCHIVADO
	VER TAREAS
	VER DETALLES DE LA TAREA
	CREAR TAREA
	MODIFICAR LA TAREA
	BORRAR TAREA
	ENTREGABLES
	VER LOS ENTREGABLES
	VER DETALLES ENTREGABLES 154
	CREATE ENTREGABLE
	MODIFICAR ENTREGABLE
	PRINCIPALES INDICADORES DE RENDIMIENTO
	INDICADORES DE RENDIMIENTO DE LAS CLAVES DEL CONTRATO
	INDICADOR DE RENDIMIENTO DE LA PANTALLA
	CREAR INDICADOR DE RENDIMIENTO CLAVE
	MODIFICAR EL INDICADOR DE RENDIMIENTO DE LA LLAVE
	REALIZACIÓN DE LA CLAVE DE EXPORTACIÓN
	IMPORTACIÓN DEL INDICADOR DE RENDIMIENTO CLAVE
	GESTIÓN FINANCIERA
	VER INFORMACIÓN FINANCIERA
	VER INFORMACIÓN FINANCIERA DETALLES
	CREAR INFORMACIÓN FINANCIERA
	MODIFICAR LA INFORMACIÓN FINANCIERA
	AÑADIR AHORROS A LA INFORMACIÓN FINANCIERA
	AÑADIR MARKETING PREMIA
	AÑADIR NOTA DE CRÉDITO A LA INFORMACIÓN FINANCIERA
	EXPORTAR INFORMACIÓN FINANCIERA PARA LA EXPORTACIÓN
	IMPORTAR INFORMACIÓN FINANCIERA DE IMPORTACIÓN
	PAGO / FACTURA
	VER PAGOS / FACTURAS
	VER LOS DATOS DE PAGO / FACTURA
	CREAR PAGO / FACTURA
	MODIFICAR EL PAGO / FACTURA
	BORRAR UN PAGO / FACTURA
MÓDULO DE INFORMES	INFORMES
	INFORMES DE CONTRATO
	CREAR INFORME ADICIONAL DE CONTRATO
	CREAR INFORME DE PROVEEDORES 59
	INFORMES DE MARKETING PREMIA
	INFORMES DE AHORROS
MÓDULO DE GESTIÓN DE DOCUMENTOS	GESTIÓN DE DOCUMENTOS
	VER CARPETAS
	CREAR LA CARPETA
	MODIFICAR LA CARPETA
	BORRAR LA CARPETA
	VER DOCUMENTOS

	REGISTRO Y SALIDA DE DOCUMENTOS MANUALES
	DEFINIR LOS DERECHOS DE ACCESO DOCUMENT / FOLDER
	ASIGNAR UN DOCUMENTO A UNA REUNIÓN / EVENTO
	ASIGNAR UN DOCUMENTO A UN FORO
	CARGAR DOCUMENTO
	DESCARGAR EL DOCUMENTO
	BORRAR DOCUMENTO
	VER LAS VERSIONES DEL DOCUMENTO
	CARGAR VERSIONES DEL DOCUMENTO
	MODIFICAR LAS VERSIONES DEL DOCUMENTO
	BORRAR LA VERSIÓN (S) DEL DOCUMENTO
	VER METADATOS
	MODIFICAR META-DATOS / PROPIEDADES
	NOTIFICACIONES DE CORREO ELECTRÓNICO
	VER UN DETALLES DE NOTIFICACIÓN DE CORREO ELECTRÓNICO
	CREAR UNA NOTIFICACIÓN DE CORREO ELECTRÓNICO
	MODIFICAR UNA NOTIFICACIÓN DE CORREO ELECTRÓNICO
	ACTIVAR / DESACTIVAR LA NOTIFICACIÓN
	BORRAR UNA NOTIFICACIÓN DE CORREO ELECTRÓNICO
	FILTRACIÓN DEL DOCUMENT MANAGER
	BÚSQUEDA DE CUENTA DE USUARIO
MÓDULO DE CALENDARIO	VER CALENDARIO
	EVENTOS DEL CALENDARIO ENTRE FECHAS
	CREAR UNA NOTIFICACIÓN PARA UN EVENTO
	ASIGNAR DERECHOS DE ACCESO A UN CALENDARIO TEMA
	ARTÍCULOS DEL CALENDARIO DE EXPORTACIÓN
	ARTÍCULOS DEL CALENDARIO DE IMPORTE
	VER DETALLES DEL EVENTO
	CREAR EVENTOS
	CREAR UN EVENTO RECURRENTE
	MODIFICAR EVENTO
	MODIFICAR UN EVENTO RECURRENTE
	REGLAS DE RECURRENCIA
	BORRAR EVENTO
	VER DETALLES DE LA REUNIÓN
	CREAR REUNIONES
	CREAR UNA REUNIÓN RECURRENTE
	RESPUESTA A UNA REUNIÓN
	MODIFICAR LA REUNIÓN
	MODIFICAR UNA REUNIÓN RECURRENTE
	ELIMINAR LA REUNIÓN
MÓDULO DE GESTIÓN DE INCIDENCIAS	VER INCIDENCIAS
	VER DETALLES DE INCIDENCIA
	AÑADIR INCIDENCIA
	MODIFICAR INCIDENCIA

Parte 6. Referencias

[1] L. Bass, P. Clements y R. Kazman, Software Architecture in Practice, Addison-Wesley Professional, 2012.

[2] J. A. P.-C. Atanasio, «Desarrollo de Software y de las Arquitecturas Software,» *Trabajos de Investigación en el Máster de Ingenería del Software,* 2010.

[3] Hewlett Packard, «Capitalize on current IT trends,» [En línea]. Available: https://www.hpe.com/h20195/v2/GetPDF.aspx/4AA6-4340ENW.pdf.

[4] A. Kaczorowska, «Traditional and agile project management in public sector and ICT,» de *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference*, 2015.

[5] Standish Group, «Standish Group 2015 Chaos Report,» 2015.

[6] R. Kazman y H. Cervantes, Designing Software Architectures: A Practical Approach, Addison-Wesley Professional, 2016.

[7] A. G. Á. P. B Cournède, «How to achieve growth-and equity-friendly fiscal consolidation? A proposed methodology for instrument choice with an illustrative application to OECD countries,» 2013.

[8] A. Cockburn, «Walking skeleton,» [En línea]. Available: http://alistair.cockburn.us/Walking+skeleton.

[9] ARQHYS.com, «Historia de la Torre de Pisa,» [En línea]. Available: http://www.arqhys.com/arquitectura/pisa-torre-historia.html.

[10] R. K. M. K. J. A. Mike Moore, «Quantifying the Value of Architecture Design Decisions:,» *Proceedings of the 25th International Conference on Software Engineering (ICSE 25),* 2003.

[11] ThinkDefence.co.uk, «UK Military Bridging – Equipment (The Bailey Bridge),» [En línea]. Available: http://www.thinkdefence.co.uk/2012/01/uk-military-bridging-equipment-the-bailey-bridge/.

[12] B. O. Bisso, «El puente que se instalará en la Av. Universitaria solo puede ser provisional,» [En línea]. Available: http://elcomercio.pe/blog/vidayfuturo/2013/02/el-puente-que-se-instalara-en.

[13] «Java EE - Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Java_EE. [Último acceso: 2017].

[14] R. K. R. a. K. R. Banker, «An Empirical Test of Object-Based Output Measurement,» de *Journal of Management Information Systems*, 1992.

[15] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland y D. Thomas, «Manifiesto por el Desarrollo Ágil de Software,» [En línea]. Available: http://www.agilemanifesto.org/iso/es/manifesto.html.

[16] K. W. Collier, Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing., 2011.

[17] J. G. Sopeña y F. J. S. Camino, «Informe de Vigilancia Tecnológica - Tecnologías software orientadas a servicios,» 2008.

[18] O. H. S. J. Kruchten P, «The past, present, and future for software architecture.,» de *Softw IEEE*, 2006.

[19] G. W. C. Y. H. B. Sullivan K, «The structure and value of modularity in software design,» de *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.

[20] R. Kazman, J. Asundi y M. Klein, «Making Architecture Decisions: An Economic Approach,» [En línea]. Available: http://resources.sei.cmu.edu/asset_files/TechnicalReport/2002_005_001_14084.pdf. [Último acceso: 11 2016].

[21] P. Eeles, «MDA and RUP,» IBM Software Group, 2004. [En línea]. Available: http://www.architecting.co.uk/presentations/MDA%20and%20RUP.pdf. [Último acceso: Feb 2016].

[22] S. J. Mellor, K. Scott, A. Uhl y D. Weise, MDA Distilled: Principles of Model-Driven Architecture, Addison-Wesley Professional, 2004.

[23] K. B. Erich Gamma, «JUnit A Cook's Tour,» [En línea]. Available: http://junit.sourceforge.net/doc/cookstour/cookstour.htm. [Último acceso: 03 2016].

[24] T. Vilaghy, V. Doddaballapur, S. Llaurency, H. Min y D. Tiwari, Writing Optimized Java Applications for z/OS, IBM Redbooks, 2002.

[25] R. Oshana y M. Kraeling, Software Engineering for Embedded Systems, Newnes.

[26] T. Sellarès, «The Model View Controller:a Composed Pattern,» [En línea]. Available: http://ima.udg.edu/~sellares/EINF-ES1/MVC-Toni.pdf. [Último acceso: 03 2006].

[27] «Model View Controller As An Aggregate Design Pattern,» [En línea]. Available: http://c2.com/cgi/wiki?ModelViewControllerAsAnAggregateDesignPattern. [Último acceso: 03 2016].

[28] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[29] C. G. Lasater, Design Patterns, Jones & Bartlett Learning, 2010.

[30] J. S. v. d. Ven y J. Bosch, «Busting Software Architecture Beliefs - A Survey on Success Factors in Architecture Decision Making,» de *2016 42th Euromicro Conference on Software Engineering and Advanced Applications*, 2016.

[31] E. W. Bahsoon R, «An economics-driven approach for valuing scalability in distributed architectures. Seventh Working IEEE/IFIP Conference on Software Architecture,» de *IEEE 2008; WICSA 2008;*.

[32] E. W. Bahsoon R, «Economics-driven architecting for nonfunctional requirements in the presence of middleware.,» de *Relating Software Requirements and Architectures*, 2011.

[33] J. D. McGregor, F. Bachman, L. Bass, P. Bianco y M. Klein, «Using an Architecture Reasoning Tool to Teach Software Architecture,» de *Software Engineering Education & Training, 2007. CSEET '07. 20th Conference on*, 2007.

--FIN DEL DOCUMENTO --