Jesús Vallecillos

COScore: una infraestructura de servicios para el despliegue de aplicaciones Mashup

Tesis Doctoral

Directores

Dr. Luis Iribarne
Dr. Nicolás Padilla
Dr. Javier Criado

Departamento de Informática
Grupo de Informática Aplicada

Departamento de Informática
Universidad de Almería

COScore: una infraestructura de servicios para el despliegue de aplicaciones Mashup

TESIS DOCTORAL

Presentada por

Jesús Vallecillos Ruiz

para optar al grado de
Doctor en Informática

Dirigida por

Dr. Luis Iribarne Martínez
Profesor Titular de Universidad
Departamento de Informática
Universidad de Almería

Dr. Nicolás Padilla Soriano
Profesor Titular de Universidad
Departamento de Informática
Universidad de Almería

Dr. Javier Criado Rodríguez
Doctor en Informática
Departamento de Informática
Universidad de Almería

Almería, a 16 de marzo de 2016

Escrito y editado por: Jesús Vallecillos Ruiz
Impresión: Murex Factoría de Color (Almería, España)

Marzo de 2016

Tesis Doctoral

COScore: una infraestructura de servicios para el despliegue de aplicaciones Mashup

Jesús Vallecillos Ruiz

Este documento ha sido generado con LaTeX.

Todas las Figuras y Tablas contenidas en
el presente documento son originales.

COSCore: una infraestructura de servicios
para el despliegue de aplicaciones Mashup

Jesús Vallecillos Ruiz
Departamento de Informática
Grupo de Investigación de Informática Aplicada (TIC-211)
Universidad de Almería
Almería, a 16 de marzo de 2016

http://acg.ual.es

A todo aquel que ha hecho posible
el desarrollo de este trabajo...

Agradecimientos

Que complicado escribir una página de agradecimientos... Resulta más difícil que escribir cualquier otra página del documento de tesis doctoral. Difícil quizás porque han podido ser muchas cosas, personas y variables las que me han traído hasta aquí. Finalizar este trabajo de tesis doctoral supone finalizar una etapa. Una etapa que me ha ayudado a crecer como persona y a conocer otros aspectos de mi que desconocía.

Me gustaría aprovechar la oportunidad de agradecer a mis padres, Francisco y Rufina, el esfuerzo realizado sobre mi para que siempre dispusiera de lo mejor. A mis hermanos Francisco y Javier por haber sido unos referentes para mi.

A mis amigos de toda la vida, Alex, Moises, José Luis y Silverio por haber sido una fuente de motivación y diversión constante.

Gracias a mis directores de tesis, Luis Iribarne, Nicolás Padilla y Javier Criado por todo lo que he podido aprender, tanto a nivel personal como a nivel profesional, a lo largo de esta etapa de tesis doctoral.

Gracias a toda la gente del grupo de investigación, Rosa Ayala, Antonio Jesús, José Andres, Antonio Corral, José Antonio Piedra, Moisés Espínola, Julio Gómez, Saturnino Leguizamóm, Francisco García y Diego Rodriguez. También agradecer a Ingenieros Alborada y sus miembros, Alfredo, Sergio, Luisa y Juan Jesús por esos desayunos que compartimos juntos.

Gracias a la Universidad de Almería por haberme nutrido de los conocimientos adquiridos durante todos estos años.

Gracias al proyecto de investigación "TDTrader: Una metodología para la interoperabilidad de servicios TDT-Web mediante la implantación de repositorios y modelos de mediación de componentes software MHP-COTS" (ref. TRA2009-0309). Ya que a partir de este proyecto de investigación comenzó mi historia en el grupo de investigación de informática aplicada de la Universidad de Almería.

Por supuesto agradecer al proyecto de investigación "ENIA: Desarrollo de un agente Web inteligente de información medioambiental" (P10-TIC6114). Sin la inversión de este proyecto de investigación de carácter regional no habría podido disfrutar de la beca de Formación de Personal Investigador (FPI).

Jesús Vallecillos Ruiz
Departamento de Informática
Grupo de Informática Aplicada
Universidad de Almería
Almería, 2016

© 2016 Vallecillos, J.

Contenido

PRÓLOGO xxi

1. REVISIÓN DE LA TECNOLOGÍA **1**
1.1. Introducción y conceptos relacionados 3
1.2. Aplicaciones Mashup 5
1.2.1. Tipos de aplicaciones Mashup 7
1.2.2. Componentes Mashup 8
1.2.2.1. Componentes de negocio 9
1.2.2.2. Componentes de datos 10
1.2.2.3. Componentes de interfaz de usuario 11
1.2.3. Comunicación entre componentes Mashup 12
1.3. Ingeniería de Componentes Software 13
1.3.1. Componentes 14
1.3.2. Componentes COTS 15
1.3.3. Arquitecturas de componentes 17
1.3.4. Especificación de componente software 19
1.3.5. Tecnologías de comunicación entre componentes 22
1.4. Ingeniería de Modelos 24
1.4.1. Metamodelado 25
1.4.2. Modelos 27
1.4.3. Modelos en tiempo de ejecución 28
1.5. Ingeniería de servicios 29
1.5.1. Servicios web SOAP 29
1.5.2. Servicios web RESTful 33
1.5.3. Microservicios 34
1.6. Resumen y conclusiones 36

2. COScore: MODELO DE INFRAESTRUCTURA **37**
2.1. Introducción y conceptos relacionados 40
2.2. Capa cliente 44
2.3. Capa dependiente de la plataforma 46
2.3.1. Gestión de componentes propios 49
2.3.2. Gestión de componentes de repositorios externos 52
2.4. Capa independiente de la plataforma 54

2.4.1. Arquitecturas de las aplicaciones *mashup* 54
2.4.2. Especificación de componentes 57
2.5. Escenarios de ejemplo base 60
2.5.1. Escenario de mapas temáticos 61
2.5.2. Escenario de domótica 62
2.6. Dependencias entre componentes *mashup* 63
2.6.1. Espacio de trabajo y componentes 63
2.6.2. Patrones de composición de componentes 69
2.6.3. Matriz de regeneración de dependencias 75
2.7. Relaciones entre componentes *mashup* 76
2.7.1. Relaciones binarias 77
2.7.2. Relaciones n-arias 79
2.8. Trabajo relacionado 81
2.9. Resumen y conclusiones 83

3. COScore: MODELO DE SERVICIOS Y OPERACIONES 85
3.1. Introducción y conceptos relacionados 87
3.2. Soporte de servicios 92
3.2.1. Bases de datos y controladores 92
3.2.2. Módulos 99
3.3. Servicios privados 104
3.3.1. Servicio *User Service* 104
3.3.1.1. Operación *Query User* 104
3.3.1.2. Operación *Update User* 109
3.3.1.3. Operación *Delete User* 113
3.3.1.4. Operación *Create User* 116
3.3.1.5. Operación *Query Profile* 120
3.3.2. Servicio *Manage Architecture Service* 123
3.3.2.1. Operación *Export AAM from String* 124
3.3.2.2. Operación *Export CAM from String* 126
3.3.2.3. Operación *Withdraw CAM* 129
3.3.3. Servicio *Manage Component Service* 131
3.3.3.1. Operación *Export CC from String* 131
3.3.3.2. Operación *Export CC from params* 134
3.3.3.3. Operación *Withdraw CC* 136
3.4. Servicios públicos 139
3.4.1. Servicio *Session Service* 139
3.4.1.1. Operación *Login* 139
3.4.1.2. Operación *Logout* 144
3.4.1.3. Operación *Init User Architecture* 147
3.4.1.4. Operación *Default Init* 153
3.4.2. Servicio *Communication Service* 158
3.4.2.1. Operación *Get Link Components* 158
3.4.3. Servicio *Component Service* 162
3.4.3.1. Operación *Update Architecture* 162

3.4.4. Servicio *Interaction Service* 170
3.4.4.1. Operación *Register Interaction* 170
3.5. Trabajo relacionado 174
3.6. Resumen y conclusiones 176

4. ESCENARIO Y EXPERIMENTACIÓN 177
4.1. Introducción y conceptos relacionados 180
4.2. Interfaz *mashup* ENIA 182
4.2.1. Componente COTSget 183
4.2.2. Escritorio de ENIA 184
4.2.3. Menú de ENIA 186
4.2.4. Panel de componentes de ENIA 187
4.2.5. Acciones sobre la interfaz ENIA 188
4.3. Escenarios de prueba 190
4.3.1. Escenario 1: Usuario anónimo 190
4.3.2. Escenario 2: Registro de usuario 192
4.3.3. Escenario 3: Usuario registrado 194
4.3.4. Escenario 4: Reconfiguración de la interfaz 197
4.3.5. Escenario 5: Cierre de sesión 199
4.4. Experimentación y pruebas 200
4.5. Resumen y conclusiones 206

5. RESULTADOS Y CONCLUSIONES FINALES 209
5.1. Aportaciones a la comunidad científica 213
5.2. Limitaciones de la infraestructura desarrollada 216
5.3. Líneas de investigación abiertas 217
5.4. Publicaciones derivadas de la tesis doctoral 218

A. PRUEBAS DE LOS SERVICIOS PÚBLICOS A-1
A.1. Pruebas de Session Service A-3
A.2. Pruebas de Communication Service A-6
A.3. Pruebas de Component Service A-8
A.4. Pruebas de Interaction Service A-23

B. INTERFAZ MASHUP ENIA B-1

ACRÓNIMOS I-1

BIBLIOGRAFÍA II-1

© 2016 Vallecillos, J.

Índice de Figuras

1.1. Tecnologías para el desarrollo de la infraestructura de servicios 5
1.2. Posicionamiento de *mashups* comparado con otras prácticas de integración 6
1.3. Niveles de modelado en MDA 24
1.4. MBE, MDE, MDD y MDA 25
1.5. Metamodelo ejemplo 26
1.6. Niveles de modelado 27
1.7. Modelos de ejemplo 28
1.8. Roles de servicio web y operaciones 30
1.9. Pila conceptual de servicios web 31
1.10. Ejemplo de arquitectura de microservicios 34

2.1. Interfaz *mashup* de Netvibes de ejemplo 40
2.2. Uso de repositorios de componentes para la construcción de las interfaces 42
2.3. Capas de la infraestructura 43
2.4. Inicialización de las aplicaciones *mashup* en la capa cliente 44
2.5. Operaciones de inserción y comunicación entre componentes (capa cliente) 45
2.6. Aplicación *mashup* de ejemplo desde la perspectiva de la capa cliente . . . 46
2.7. Capa dependiente de la plataforma: (a) con un único servidor basado en eventos y (b) con un servidor por cada plataforma soportada por la infraestructura 47
2.8. Modelo de datos de componentes 48
2.9. Estructura W3C de un *widget* mapa 50
2.10. Estructura del componente Java 52
2.11. Metamodelo de arquitectura basada en componentes 55
2.12. Metamodelo de componente 57
2.13. Especificación de las interfaces funcionales 58
2.14. Estructura de un componente: (a) vista de interfaces y (b) vista de puertos 60
2.15. Componentes del escenario base SIG 61
2.16. Escenario de domótica: (a) vista de interfaces y (b) vista de puertos . . . 62
2.17. Aplicación *mashup* con dos mapas y dos listas de capas 64
2.18. Aplicación *mashup* con dos mapas que contienen su lista de capas 65
2.19. Patrón de visualización #1: (a) vista de puertos y (b) vista de interfaces . 70
2.20. Patrón de visualización #2: (a) vista de puertos y (b) vista de interfaces . 71

2.21. Patrón de visualización #3: (a) vista de puertos y (b) vista de interfaces . 71
2.22. Patrón de visualización #4: (a) vista de puertos y (b) vista de interfaces . 72
2.23. Patrón de visualización #5: (a) vista de puertos y (b) vista de interfaces . 73
2.24. Patrón de visualización #6: (a) vista de puertos y (b) vista de interfaces . 73
2.25. Patrón de visualización #7: (a) vista de puertos y (b) vista de interfaces . 74
2.26. Patrón de visualización #8: (a) vista de puertos y (b) vista de interfaces . 75
2.27. Patrón de visualización #9: (a) vista de puertos y (b) vista de interfaces . 76
2.28. Relaciones binarias entre los componentes de los escenarios de ejemplo . . 78
2.29. Relaciones n-arias: (a) observación, (b) jerarquía y (c) *trading* 80
2.30. Relaciones n-arias: (a) secuencia, (b) control y (c) sumidero 81

3.1. Estructura de servicios, módulos, controladores y bases de datos 88
3.2. (a) Tablas de la base de datos de modelos de arquitectura y usuarios, (b) metamodelo de arquitectura de las aplicaciones *mashup* 93
3.3. (a) Tablas de la base de datos de especificaciones de componentes concretos (b), junto al metamodelo de especificaciones de componentes concretos 99
3.4. Operación *Query user* de *User Service* 107
3.5. Operación *Update user* de *User Service* 111
3.6. Operación *Delete user* de *User Service* 115
3.7. Operación *Create user* de *User Service* 118
3.8. Operación *Query profile* de *User Service* 121
3.9. Operación *Export AAM from String* de *Manage Architecture Service* . . . 125
3.10. Operación *Export CAM from String* de *Manage Architecture Service* . . . 127
3.11. Operación *Withdraw CAM* de *Manage Architecture Service* 130
3.12. Operación *Export CC from Srting* de *Manage Component Service* 132
3.13. Operación *Export CC from params* de *Manage Component Service* 135
3.14. Operación *Withdraw CC* de *Manage Component Service* 138
3.15. Operación *Login* de *Session Service* 142
3.16. Operación *Logout* de *Session Service* 145
3.17. Operación *Init user architecture* de *Session Service* 151
3.18. Operación *Default Init* de *Session Service* 156
3.19. Operación *Get link components* de *Communication Service* 160
3.20. Operación *Update architecture* de *Component Service* 166
3.21. Operación *Register interaction* de *Interaction Service* 172

4.1. Vista *Wireframe* de la interfaz *mashup* ENIA 183
4.2. Estructura de la organización de las fuentes de ENIA 184
4.3. Una vista de la interfaz *mashup* ENIA 185
4.4. Panel de inicio/registro 186
4.5. Paneles de menú de componentes de ENIA 187
4.6. Componentes COTSgets OGC 189
4.7. Interfaz *mashup* ENIA en el acceso de un usuario anónimo 191
4.8. Interfaz *mashup* ENIA para el registro de nuevos usuarios 193
4.9. Interfaz *mashup* ENIA para la identificación de usuarios 195
4.10. Interfaz *mashup* ENIA de un usuario registrado 196

4.11. Interfaz *mashup* ENIA del usuario una vez reconfigurado el escritorio . . . 197
4.12. Inserción de un nuevo componente en el escritorio de la interfaz *mashup* . 198
4.13. Cierre de sesión en la interfaz *mashup* ENIA 200
4.14. Puntos de medición en el proceso de evaluación 201
4.15. (a) Tiempos de respuesta en las capas A, B y C; (b) Inicialización de la aplicación *mashup* con tamaños de modelos y acoplamientos diferentes . . 202
4.16. Inicialización de la aplicación variando el tamaño y el acoplamiento 204
4.17. (a) Tiempos más bajos (primer usuario que recibe la respuesta); (b) Tiempos más altos (último que recibe la respuesta); (c) Usando 1, 2 y 3 servidores; (d) Evaluación de la comunicación con diferentes tamaños de la interfaz *mashup* 205

5.1. Versiones de COScore 211

© 2016 Vallecillos, J.

Índice de Tablas

1.1. Ejemplo de definición de interfaces con IDL 20
1.2. Estructura de una definición WSDL 20
1.3. Definición de esquema de un documento COTS 22
1.4. Una instancia ejemplo de un documento COTS 22
1.5. Estructura de un servicio web a través de su WSDL 32
1.6. Mensaje SOAP de entrada a la operación `login` 32
1.7. Mensaje SOAP de respuesta de la operación `login` 32

2.1. Archivo de configuración de un *widget* W3C 51
2.2. Elemento `body` de un componente web externo adaptado 53
2.3. Elemento `head` de un componente web externo adaptado 54
2.4. Restricción de relación binaria 56
2.5. Restricción bidireccional entre dos componentes 56
2.6. Restricción para un puerto de entrada 57
2.7. Componentes gráficos SIG base y sus interfaces 68
2.8. Patrones de visualización 70
2.9. Matriz de regeneración de dependencias 77
2.10. Principales relaciones binarias 78
2.11. Principales relaciones n-arias 79

3.1. Servicios públicos del COScore 89
3.2. Servicios privados del COScore 89
3.3. Descripción de los elementos de la infraestructura COScore 90

4.1. Acciones que se pueden realizar sobre ENIA 188
4.2. Inicialización de la aplicación variando el número de usuarios concurrentes 203

Prólogo

En el campo de las ciencias de la computación y, en especial, en lo que respecta al desarrollo de sistemas de información, cada vez es más común la combinación de diversas tecnologías, por ejemplo, utilizando diferentes lenguajes de programación, haciendo uso de múltiples marcos de trabajo y librerías, o conectando elementos que pertenecen a distintos fabricantes. Este tipo de ensamblaje es un mecanismo asumido cuando se hace referencia a la construcción de sistemas *hardware* (elementos físicos de un sistema de computación), de manera que, cada una de las piezas que se utilizan están descritas a través de algún tipo de especificación que incluye, además, la información necesaria para poder conectar dicha pieza con el resto de elementos del sistema. Como consecuencia, existe una amplia variedad de estándares para la construcción de estos componentes, los cuales son asumidos por la gran mayoría de los fabricantes. Parte de estos estándares están orientados a la descripción de piezas que se utilizan para el montaje del producto final (perspectiva interna), mientras que otra parte de ellos están enfocados en la definición de cómo dicha pieza puede ser conectada con otro dispositivo o de cómo puede ser utilizada (perspectiva externa o de interfaz). Como ejemplo, pensemos en los teclados de ordenador disponibles actualmente en el mercado. Por un lado, cada dispositivo de este tipo dispone de una ficha de especificación técnica que describe cómo ha sido construido y por qué piezas está formado. Por otro lado, todos los teclados tienen un conector de tipo USB o PS/2 (y no cualquier otro tipo de conector de los existentes en la industria), permitiendo la interoperabilidad con otros dispositivos del sistema. De esta manera, todos los ordenadores que deseen utilizar un teclado, deben ofrecer un conector del tipo correspondiente.

En el caso de la construcción de sistemas *software* (elementos lógicos de un sistema de computación) ocurre algo parecido, pero con ciertas características particulares que deben ser tenidas en cuenta. Desde los comienzos de la Ingeniera del Software (SE, *Software Engineering*), el diseño y construcción de arquitecturas basadas en componentes reutilizables ha representado un principio esencial en el desarrollo de aplicaciones software. En particular, el campo de la Ingeniería del Software Basada en Componentes (CBSE, *Component-Based Software Engineering*) tiene como objetivos principales (*i*) el desarrollo de sistemas a partir de un conjunto de partes, (*ii*) el desarrollo de cada una de las partes como entidades que pueden ser reutilizadas, y (*iii*) el mantenimiento y actualización de los sistemas mediante la configuración o reemplazo de dichas partes [Crnkovic, 2001]. Dentro de esta disciplina, los componentes software constituyen las piezas por las cuales están formados los sistemas y, por tanto, deben ser descritos de manera que cualquier desarrollador, fabricante o usuario que quiera utilizar dicho com-

ponente, disponga de la información necesaria para su correcto funcionamiento. Para ello, las especificaciones de los componentes deben incluir, al menos, una definición de cómo se puede hacer uso del componente y de qué acciones son necesarias en el caso de que se quiera conectar dicho elemento software con otro componente del sistema (perspectiva de "caja negra" de un componente). Adicionalmente, el componente puede ofrecer una descripción acerca de cómo se implementa su lógica interna (perspectiva de "caja blanca" de un componente).

Al contrario de lo que ocurre en el hardware, los componentes software pueden ser actualizados sin la necesidad de volver a realizar un nuevo proceso de fabricación adicional que cambie dicha pieza o parte del software. Del mismo modo, los sistemas software pueden ser modificados y adaptados sin tener que ensamblar de nuevo cada una de las partes que los constituyen. En el caso de los componentes, si se tiene acceso a la implementación, es posible realizar un proceso de actualización automático de su lógica interna, por ejemplo, modificando el código. Por el contrario, si se trata de un componente de "caja negra", existen técnicas para la creación de un código envoltorio (*wrapper*) que también pueden ser automatizadas para realizar dicha actualización. En el caso de sistemas software basados en componentes, existen diferentes mecanismos para llevar a cabo su adaptación, como por ejemplo, actualizar alguno de sus componentes, reconectar sus piezas de forma diferente a la original, insertar nuevos componentes, eliminar partes que no son necesarias o reemplazar componentes por otros que cumplan mejor con una funcionalidad requerida. Dichos mecanismos, por lo tanto, se encargan de realizar algún cambio en la arquitectura del sistema, la cual está definida por medio de sus componentes y las relaciones que existen entre ellos.

Esta flexibilidad que aportan los sistemas software basados en componentes permite modificarlos una vez que han sido desarrollados, tanto en tareas destinadas al mantenimiento y evolución del software, como en tareas de adaptación en tiempo de ejecución. Algunas de las estrategias de adaptación en tiempo de ejecución pueden ser diseñadas e implementadas durante el desarrollo del sistema, por ejemplo, como operaciones de reconfiguración que, bajo determinadas circunstancias (relativas al tipo de usuario que interactúa con el software, a los recursos disponibles, a la disponibilidad de nuevos componentes, etc.) cambien la arquitectura del sistema. Sin embargo, pueden existir otras estrategias que, por no haber sido identificadas en fases previas del desarrollo o por no disponer de la información necesaria, no hayan sido incluidas en la implementación del mecanismo encargado de **adaptar** el sistema. Por lo tanto, en el caso de que se quiera conseguir un sistema software basado en componentes que se adapte de forma dinámica y flexible, tanto a situaciones contempladas previamente como a las nuevas circunstancias que surjan, se pone de manifiesto la necesidad de poder aplicar nuevas opciones de **reconfiguración** que sean adicionales a la lógica de adaptación implementada.

Un ejemplo de nuevas alternativas de adaptación que deben ser incorporadas tras el desarrollo de un sistema son aquellas que provienen de la interacción que se realiza con las aplicaciones software una vez que son **desplegadas**. A partir de dicha interacción, que puede haber sido realizada por los usuarios o como origen de una comunicación de un elemento software, es posible extraer, deducir o inferir nuevo conocimiento que sirva para enriquecer y mejorar las opciones de cambio de las arquitecturas que definen dichas aplicaciones. La aplicación de nuevas alternativas de reconfiguración puede

ser abordada desde distintas aproximaciones, como el reemplazamiento de una lógica de adaptación por otra, la modificación de las reglas que definen el comportamiento de dicha lógica, o la incorporación de diferentes puntos de origen con la capacidad de reconfigurar las arquitecturas software, entre otros posibles ejemplos. En cualquier caso, la infraestructura que de soporte a cualquiera de estas alternativas (o a cualquier otra propuesta de adaptación de sistemas basados en componentes) debe proporcionar las operaciones de reconfiguración necesarias, así como aquellas acciones que permitan gestionar las arquitecturas de dichos sistemas.

Una **infraestructura** que proporcione un soporte a la adaptación de software basado en componentes debe tener en cuenta aspectos de interoperabilidad e integración de las diferentes piezas que lo constituyen. De manera ligada a dichos aspectos y como parte de una nueva tendencia en la que los diferentes componentes que constituyen un sistema software son considerados como servicios que deben ser conectados entre sí, surge el concepto de aplicaciones *mashup* [Daniel and Matera, 2014]. En esta nueva corriente, el término ***mashup*** no se refiere únicamente a servicios web ni su ámbito de aplicación está limitado al dominio Web sino que, por el contrario, pretende agrupar a aquellas aplicaciones software desarrolladas y desplegadas como resultado de la composición de elementos software heterogéneos. La composición de elementos heterogéneos es una técnica bien conocida en la Ingeniería de Software Basada en Componentes, no obstante, en las aplicaciones *mashup* se le otorga una perspectiva de integración e interoperabilidad más vinculada a las etapas de despliegue y ejecución de una aplicación que a las etapas de diseño y construcción (en contraposición a la perspectiva de los paradigmas tradicionales). Como otros aspectos adicionales, el concepto de *mashup* está relacionado con desarrollo rápido de aplicaciones software gracias a la utilización de servicios y APIs (*Application Programming Interfaces*) que pueden haber sido desarrollados por terceros y que, además, son abiertos y están disponibles para que el nuevo software que se construya pueda hacer uso de ellos de manera sencilla y desde cualquier ubicación.

El uso de componentes y servicios que han sido desarrollados tanto por la organización que construye nuevo software (propios) como por organizaciones externas (de terceros) resulta en un amplio catálogo de piezas que pueden ser utilizadas para generar nuevas aplicaciones. Este hecho deriva en una gran variabilidad de opciones para construir sistemas software, lo cual puede suponer una ventaja (puesto que se dispone de gran variedad de alternativas) pero también implica la necesidad de gestionar todos los elementos disponibles y las operaciones de reconfiguración.

En la actualidad, existen cada vez más trabajos de investigación enfocados en la combinación, integración e interoperabilidad de los servicios que constituyen las aplicaciones *mashup*. Como casos prácticos de aplicación, también existen propuestas de catálogos de servicios, APIs y repositorios de componentes cuyo objetivo es poder llevar a cabo la construcción de *mashups*. A modo de ejemplo, aplicaciones *mashup* de interfaz de usuario son ofrecidas en forma de tablero de mandos o panel de instrumentos (*dashboard*) para que los usuarios de dichas aplicaciones puedan configurarse una interfaz con los componentes que desean o que más utilizan. Casos específicos de este tipo de aplicaciones puede verse en productos actuales ofrecidos por Netvibes, Geckoboard o MyYahoo, y también suponen la base de algunos proyectos que ya han finalizado, como es el caso de las interfaces basadas en *widgets* de iGoogle. En cualquier caso, todavía existe un

© 2016 Vallecillos, J.

vacío en lo que respecta a propuestas de infraestructuras que ofrezcan un soporte global a todas la funcionalidad relacionada con la gestión de aplicaciones *mashup* y con su despliegue, incluyendo operaciones de alta/baja/modificación de nuevos servicios, reconfiguración de las arquitecturas que definen las aplicaciones o gestión de la comunicación entre los servicios desplegados, como algunos posibles ejemplos de dicha funcionalidad.

Desde nuestro punto de vista, las metodologías y técnicas comentadas pueden utilizarse de manera conjunta para el desarrollo de una infraestructura que de solución a toda la funcionalidad requerida por las aplicaciones *mashup*. Para ello, la combinación de la Ingeniería de Software Basada en Componentes y la tecnología *mashup* nos permite (*i*) construir aplicaciones software complejas mediante el ensamblaje de piezas más simples, (*ii*) que dichas piezas puedan desarrollarse por terceras partes y ser utilizadas posteriormente por cualquier aplicación *mashup*, (*iii*) que los servicios y componentes que constituyen estas piezas se encuentren disponibles y accesibles (*i.e.*, en la red) para su utilización e incorporación a las aplicaciones, (*iv*) que la comunicación entre los componentes sea resuelto mediante técnicas de interoperabilidad de servicios, (*v*) que los distintos componentes puedan integrarse en una misma aplicación, y (*vi*) que las arquitecturas software que constituyen las aplicaciones puedan reconfigurarse modificando su estructura para adaptarse a lo largo del tiempo.

El trabajo de investigación desarrollado en la presente tesis doctoral tiene como objetivo principal el desarrollo de una infraestructura que de solución al despliegue y gestión de aplicaciones *mashup*, incluyendo la funcionalidad mencionada anteriormente. Además, la infraestructura debe dar soporte a aplicaciones *mashup* pertenecientes a distintos dominios e, incluso, que se desplieguen en dispositivos de plataformas diferentes. Con esta finalidad, la infraestructura se basa en un proceso de abstracción que nos permite definir las aplicaciones *mashup* como arquitecturas software en las que se representan cada uno de los componentes que forman parte de ellas, así como sus relaciones. De esta manera, las distintas piezas de las aplicaciones se gestionan de forma similar, independientemente del dominio y de la plataforma, y únicamente se ejecutan operaciones específicas de una plataforma cuando hay que generar el código necesario para el despliegue de las aplicaciones. Cada una de estas piezas ha sido nombrada como componente COTSget, de la combinación del concepto de componente COTS (*Commercial Off-The-Shelf*) para hacer referencia a componentes que pueden haber sido desarrollados por terceras partes, y del término *gadget*, que se refiere a un artefacto software que encapsula cierta funcionalidad y que permite llevar a cabo una tarea.

El modelo de **infraestructura** propuesta está constituida por tres capas. La capa que se encuentra en el nivel inferior es la capa independiente de la plataforma y contiene toda la funcionalidad que es común a todos los dispositivos para los que se pueden construir aplicaciones *mashup*. La capa superior está formada por las propias aplicaciones *mashup* y por los clientes que hacen uso de ellas, entre los que se encuentran, por ejemplo, los navegadores (en el caso de que las aplicaciones de tipo web) o los contenedores de aplicaciones (en el caso de las aplicaciones de tipo Java). Realizando el rol de intermediario, existe una capa dependiente de la plataforma encargada de conectar la capa cliente con la capa independiente de la plataforma. El objetivo de esta capa intermedia es permitir la comunicación entre ambas capas, de manera que las aplicaciones que se encuentran desplegadas en la capa cliente puedan ser gestionadas por la lógica de las operaciones de

la capa independiente (teniendo en cuenta la información que proporciona la primera) y que las decisiones tomadas en la capa inferior puedan modificar las aplicaciones en tiempo de ejecución y de forma dinámica. Esta flexibilidad se consigue, por ejemplo, sin llevar a cabo una recarga de toda la aplicación y actualizando únicamente aquellas partes de la arquitectura que han sido modificadas.

La capa dependiente de la plataforma también contiene los distintos repositorios de componentes que pueden formar parte de las aplicaciones *mashup* y, dependiendo del tipo de plataforma utilizada en cada momento, se hará uso de una colección u otra. Teniendo en cuenta que la capa cliente no es una aportación propia del trabajo de investigación, sino que únicamente se utiliza como soporte para el despliegue de las aplicaciones, las otras dos capas de la infraestructura (dependiente e independiente de la plataforma) constituye el núcleo de la propuesta de esta tesis doctoral y juntas reciben el nombre de COScore (*COTSget-based architecture Operating Support core*). Uno de los aspectos que caracterizan este núcleo de la infraestructura es que está desarrollado como un conjunto de **servicios**. Cada servicio, a su vez, agrupa un conjunto de operaciones relacionadas entre sí. Por ejemplo, existe un servicio que contiene todas las operaciones relacionadas con la gestión de usuarios (creación, eliminación, modificación, etc.). De esta forma, el desarrollo de la infraestructura ha generado una API que ofrece toda la funcionalidad necesaria para poder llevar a cabo el despliegue de aplicaciones *mashup* y que, además, permite realizar todas las acciones de gestión relacionadas con el uso de dichas aplicaciones.

Objetivos y contribución

El desarrollo de una infraestructura de servicios para poder llevar a cabo el despliegue y la gestión de aplicaciones mashup tiene, como consecuencia, la necesidad de completar los siguientes objetivos de carácter genérico:

1. En primer lugar, es necesario que exista un modelo de infraestructura que de soporte a aplicaciones basadas en componentes, en particular, aplicaciones tipo *mashup*. El soporte ofrecido debe permitir tanto el despliegue de las aplicaciones, como la gestión posterior de las acciones necesarias para su correcto funcionamiento y para poder realizar operaciones de reconfiguración en tiempo de ejecución.

2. En segundo lugar, la infraestructura creada debe permitir el despliegue y gestión de aplicaciones *mashup* de distintos tipos y en distintas plataformas, siempre que cumplan con unos determinados requisitos.

3. En tercer lugar, para que sea posible llevar a cabo este enfoque genérico de las aplicaciones *mashup*, es necesario utilizar técnicas formales en forma de especificaciones para la descripción tanto de los componentes como de las arquitecturas que constituye las aplicaciones.

4. En cuarto lugar, se establece que toda la funcionalidad ofrecida por la infraestructura debe estar proporcionada en forma de servicios. De esta manera, las aplicaciones *mashup* harán uso de dichos servicios para llevar a cabo su despliegue y

© 2016 Vallecillos, J.

gestión. Adicionalmente, este tipo de desarrollo en forma de servicios debe generar como resultado un conjunto de operaciones que puede ser utilizada, tanto por los desarrolladores de la infraestructura como por usuarios externos.

5. Por último, es necesario realizar un proceso de pruebas, validación y evaluación de la infraestructura propuesta. Para ello, se deben construir escenarios y casos de estudio suficientes para poder determinar que el trabajo desarrollado es válido para el despliegue de aplicaciones *mashup* y que los resultados obtenidos (en lo que respecta a tiempos de respuesta) son adecuados.

La consecución de los objetivos mencionados tiene como resultado el desarrollo de una infraestructura que permite el despliegue de aplicaciones *mashup*. Dichas aplicaciones deben estar descritas en forma de arquitecturas software en las que cada una de sus piezas es un componente COTSget. Recordemos que el término que califica dichos componentes proviene de la combinación del concepto de componente COTS (pieza de una arquitectura que puede haber sido desarrollada por terceras partes) y el término *gadget* (elemento software que encapsula la funcionalidad necesaria para llevar a cabo una tarea). Por tanto, estos componentes son componentes de granularidad gruesa, es decir, no se trata de componentes simples sino que tienen cierta complejidad. Además del despliegue de aplicaciones *mashup*, la infraestructura permite la gestión de dichas aplicaciones, incluyendo operaciones de reconfiguración para modificar la estructura y composición de las arquitecturas que las definen. La aportación del trabajo de investigación realizado se puede dividir en las diferentes contribuciones que se listan a continuación:

- Se ha construido una infraestructura estructurada en tres capas para dar soporte a aplicaciones *mashup* de distintos dominios y plataformas. Dicha infraestructura ha sido desarrollada en un proceso incremental que ha originado diferentes versiones. La versión que describe en esta tesis es *COScore 2.0.0*. Las capas por las cuales está construida la infraestructura son la capa cliente, la capa dependiente de la plataforma y la capa independiente de la plataforma. En la capa cliente se encuentran las aplicaciones *mashup*, en la capa dependiente de la plataforma se localiza un servidor JavaScript encargado de hacer de mediador para gestionar los procesos de comunicación entre la capa cliente y la capa independiente de la plataforma. Además, en la capa dependiente se encuentran los repositorios de componentes que son utilizados para el despliegue en las aplicaciones *mashup*. Por último, la capa independiente de la plataforma contiene el núcleo funcional de la infraestructura.

- Se han gestionado una serie de repositorios indispensables dentro de la infraestructura. Se trata de los repositorios de componentes, tanto propios como de terceras partes. Con respecto a los repositorios de terceras partes, se han construido carcasas en forma de código de envoltorio (*wrapper*) para aquellos componentes localizados en bases de datos que no pertenecen a nuestro desarrollo. A partir de estos componentes adaptados, los componentes desarrollados por terceras partes pueden ser gestionados para ser integrados en las aplicaciones *mashup*. Por otro lado, los repositorios de componentes propios contienen los componentes construidos íntegramente siguiendo la especificación de componente aquí desarrollada.

- Se ha definido una especificación de componente por medio del uso de técnicas de metamodelado. Este metamodelo de componente se utiliza para poder construir las arquitecturas de componentes que definen las aplicaciones *mashup*. Esta especificación contiene todos los elementos necesarios para que un componente pueda ser integrado en la arquitectura de componentes de una aplicación. Además, se ha construido una especificación para describir formalmente las aplicaciones *mashup*, en forma de arquitecturas software constituidas por componentes y relaciones.

- Se ha definido un conjunto de tipos de relaciones que pueden existir entre los componentes. Para realizar este desglose de tipos, en primer lugar se ha creado una serie de escenarios simples de ejemplo. De ese modo, se ha creado un escenario de domótica y otro escenario relacionado con un sistema de información geográfica. A partir de estos escenarios, se han creado dos grandes grupos de relaciones, las binarias y las n-arias. Las relaciones binarias relacionan dos componentes de la arquitectura entre sí y las n-arias relacionan tres o más componentes entre sí.

- Con el objetivo de que la infraestructura pueda dar soporte a las aplicaciones *mashup*, se ha creado un conjunto de servicios públicos por medio de los cuales las aplicaciones se conectan y acceden a la funcionalidad ofrecida. Dichos servicios públicos han sido nombrados como *Register interaction* (encargado de gestionar procesos de registro de la interacción producida en la aplicación), *Update architecture* (el cual gestiona la actualización de las arquitecturas de componentes que definen las aplicaciones), *Get link components* (cuya finalidad es gestionar los procesos de comunicación entre los componentes) y *Session* (encargado de gestionar los procesos de inicio de sesión para los usuarios de las aplicaciones). Cada servicio, a su vez, está compuesto por un conjunto de operaciones.

- También se necesitan operaciones de carácter privado para gestionar la infraestructura y poder dar así funcionalidad a las aplicaciones *mashup*. Para ello, se ha definido un conjunto de servicios privados: *User* (encargado de gestionar los usuarios que hacen uso de aplicaciones *mashup*), *Manage Architecture* (que gestiona las especificaciones de las arquitecturas utilizadas para describir las aplicaciones *mashup*) y *Manage Component* (el cual gestiona las especificaciones de los componentes que pueden ser utilizados por la infraestructura). De forma similar a los servicios públicos, cada servicio privado está compuesto por un conjunto de operaciones.

- Se ofrecen detalles acerca de cómo se han implementado los servicios pertenecientes a la infraestructura. Para ello, se describe cada una de las operaciones que constituyen los servicios y, además se describen ejemplos para ilustrar de qué manera se deben consumir dichos servicios.

- Se han realizado distintas pruebas y experimentos con el objetivo de poder analizar los tiempos de respuesta que tienen lugar dentro de la infraestructura. Para realizar estas pruebas, se han tenido en cuenta diferentes escenarios de ejemplo en los cuales se ha evaluado el impacto de diferentes factores que afectan a los resultados obtenidos, como por ejemplo, el número de componentes que forman parte de la aplicación *mashup* o el número de usuarios que acceden en el mismo instante de tiempo a un recurso.

© 2016 Vallecillos, J.

– Finalmente, se ha analizado el principal caso de estudio que ha sido desarrollado para la validación del trabajo de investigación realizado. Se trata de una aplicación *mashup* de tipo web para la explotación de datos pertenecientes a un sistema de información geográfica. Dicho sistema recibe el nombre de ENIA (*ENvironmental Information Agent*) y se enmarca dentro de un proyecto de investigación regional, cuyo objetivo principal es la explotación de información geográfica basada en mapas y obtenida a partir de servicios OGC (*Open Geospatial Consortium*) ofrecidos por la REDIAM (*Red de Información Ambiental de Andalucía*).

Marco y líneas futuras

El desarrollo del presente trabajo de tesis se enmarca dentro de un proyecto de investigación de excelencia de ámbito regional financiado por la Junta de Andalucía y cuya referencia es P10-TIC6114. Este proyecto se titula "ENIA: Desarrollo de un agente Web inteligente de información medioambiental". El objetivo principal de este proyecto ha consistido en permitir la gestión de información medioambiental perteneciente a un Sistema de Información Geográfica (SIG) a través de interfaces gráficas de usuario que pudieran adaptar su estructura dependiendo de las circunstancias. Dichas circunstancias incluían, por ejemplo, el tipo de usuario que interactúa con el sistema, las dependencias que existen entre los componentes que forman parte de las interfaces, o los recursos disponibles. En este sentido, dicho proyecto de investigación constituye el caso de estudio ideal para la validación de la infraestructura propuesta. Las interfaces gráficas de usuario de este proyecto son un ejemplo de aplicaciones *mashup* (en este caso, de tipo web) que pueden desplegarse y gestionarse gracias al uso de nuestra infraestructura.

Adicionalmente, el trabajo de investigación realizado ha sido desarrollado gracias a la concesión de una beca de Formación de Personal Investigador (FPI) financiada por la Junta de Andalucía. Durante esta beca predoctoral se han realizado tanto labores de investigación relacionadas con el proyecto mencionado anteriormente, como la elaboración del trabajo de investigación relacionado con esta tesis doctoral. Por otra parte, el trabajo de tesis ha sido desarrollado dentro del Programa de Doctorado en Informática Ref. 8908 del RD99/11 de la Universidad de Almería.

Una vez desarrollado este trabajo de tesis y finalizado el proyecto de investigación en el que se enmarca, quedan aún abiertas determinadas líneas de investigación que serán acometidas en un futuro próximo. Por ejemplo, sería interesante poder aplicar y validar la infraestructura en otros dominios distintos a las aplicaciones *mashup* de tipo web o de tipo Java. Además, la interacción capturada de las aplicaciones y que proviene de la capa cliente puede ser utilizada por la infraestructura para inferir nuevas estrategias de modificación y de reconfiguración. De esta manera, sería posible hacer uso de los comportamientos que los usuarios tienen con las aplicaciones para adaptar sus arquitecturas de manera que se optimice su utilización. También resulta interesante mejorar la infraestrutura para que la interacción con las aplicaciones pueda realizarse desde distintos medios y a partir de vistas diferentes, incluyendo además una perspectiva que incorpore algún tipo de interfaz natural de usuario (NUI, *Natural User Interface*). Algunas de

estas líneas forman parte de otros trabajos de investigación que están siendo llevados a cabo por el grupo de investigación de Informática Aplicada de la Universidad de Almería (TIC-211). Otras líneas de investigación se encuentran incluidas en el proyecto de investigación TIN2013-41576-R, financiado por el MINECO (Ministerio de Economía y Competitividad). Dicho proyecto titulado "Evolución de Sistemas Dinámicos en la Nube: un escenario marco hacia las Interfaces de Usuario Inteligentes", está siendo llevado a cabo actualmente por el grupo de investigación TIC-211 de la Universidad de Almería.

Estructura de la tesis doctoral

El trabajo de tesis doctoral está constituido por cuatro capítulos, además de un anexo, un listado de acrónimos y el conjunto de referencias bibliográficas que han sido utilizadas.

El *Capítulo 1* contiene una revisión de las principales tecnologías y paradigmas que han sido utilizados para poder llevar a cabo el desarrollo de la infraestructura propuesta para el despliegue de aplicaciones *mashup*. Dichas tecnologías incluyen: las propias aplicaciones *mashup*, la ingeniería de componentes software, la ingeniería de modelos y la ingeniería de servicios. El *Capítulo 2* detalla el modelo de infraestructura propuesto, incluyendo la definición de las tres capas (cliente, dependiente de la plataforma e independiente de la plataforma) y la descripción de las dependencias y relaciones que pueden existir dentro de las arquitecturas de las aplicaciones *mashup*.

El *Capítulo 3* describe el modelo de servicios y operaciones que se ha desarrollado como solución de implementación de la infraestructura propuesta. Para ello, en primer lugar se describe el soporte de los servicios proporcionados (bases de datos, controladores y módulos). Posteriormente, el capítulo detalla el conjunto de servicios (públicos y privados) que han sido implementados. De cada una de las operaciones de los servicios, se detalla (*i*) la interfaz, (*ii*) los parámetros de entrada/salida, (*iii*) la descripción, (*iv*) el comportamiento, (*v*) la implementación, (*vi*) un ejemplo de petición/respuesta, y (*vii*) los posibles mensajes de respuesta de su ejecución. Por último, el *Capítulo 4* contiene la parte del trabajo de investigación realizado que está relacionada con la validación y evaluación de la infraestructura propuesta. En este sentido, el capítulo incluye la descripción del caso de estudio principal (aplicaciones *mashup* del proyecto ENIA), ejemplos de ejecución de las distintas operaciones disponibles y un análisis acerca de los tiempos de respuesta obtenidos en escenarios de prueba experimentales.

Los *Capítulos* del *1* al *4* de este documento se estructuran de la siguiente forma. En cada uno de ellos se comienza con una primera sección de *Introducción y conceptos relacionados*, se continúa con la estructura de secciones propias para cada capítulo y se finaliza con una sección de *Resumen y conclusiones*. Para los *Capítulos 1*, *2* y *3*, se incluye una sección de *Trabajos relacionados*. Este documento incluye un *Anexo A* con pruebas realizadas como parte del desarrollo de la infraestructura y que no han podido ser incluidas en los *Capítulos 3* y *4* debido a su extensión. También incluye un *Anexo B* con la guía rápida de usuario de la interfaz ENIA. Además, este trabajo contiene un listado de *Acrónimos* con el objetivo de definir los principales términos y siglas que son utilizadas en el presente documento. Por último, las referencias bibliográficas que han sido usadas se encuentran en el apartado de *Bibliografía*.

© 2016 Vallecillos, J.

Capítulo 1

Revisión de la tecnología

© 2016 Vallecillos, J.

Capítulo 1

Revisión de la tecnología

Contenidos

1.1. Introducción y conceptos relacionados 3
1.2. Aplicaciones Mashup 5
1.2.1. Tipos de aplicaciones Mashup 7
1.2.2. Componentes Mashup 8
1.2.2.1. Componentes de negocio 9
1.2.2.2. Componentes de datos 10
1.2.2.3. Componentes de interfaz de usuario 11
1.2.3. Comunicación entre componentes Mashup 12
1.3. Ingeniería de Componentes Software 13
1.3.1. Componentes 14
1.3.2. Componentes COTS 15
1.3.3. Arquitecturas de componentes 17
1.3.4. Especificación de componente software 19
1.3.5. Tecnologías de comunicación entre componentes 22
1.4. Ingeniería de Modelos 24
1.4.1. Metamodelado 25
1.4.2. Modelos 27
1.4.3. Modelos en tiempo de ejecución 28
1.5. Ingeniería de servicios 29
1.5.1. Servicios web SOAP 29
1.5.2. Servicios web RESTful 33
1.5.3. Microservicios 34
1.6. Resumen y conclusiones 36

El objetivo principal del trabajo de investigación que aquí se presenta es el desarrollo de una infraestructura de servicios para dar soporte a aplicaciones de tipo *mashup*. Cuando se habla de revisión de la tecnología se hace referencia al conjunto de paradigmas tecnológicos que han hecho posible el desarrollo de dicha infraestructura de servicios. A partir de este conjunto de técnicas, se puede llevar a cabo tanto la construcción de las aplicaciones cliente como el desarrollo de la infraestructura que dará soporte a dichas aplicaciones a través de los servicios. Por lo tanto, a lo largo del capítulo se hace un pequeño estudio de las diferentes áreas que dan soporte al diseño, construcción y despliegue de la infraestructura propuesta en el presente trabajo de investigación.

Este capítulo está estructurado en seis partes. Se comienza con la introducción del trabajo de investigación y con la presentación de cuáles son los bloques principales sobre los que se sustenta. A continuación, se revisa cada uno de estos bloques. En primer lugar, se lleva a cabo una descripción de las aplicaciones *mashup* realizando una definición de las mismas, describiendo los componentes por los cuales están formadas las aplicaciones y los procesos de comunicación que tienen lugar entre los componentes. Después, se justifica el aporte que tiene para nuestra propuesta la utilización de técnicas de *Ingeniería de Software Basada en Componentes*. Posteriormente, se presentan algunas de las técnicas relacionadas con la *Ingeniería de Modelos*, poniendo de manifiesto su utilidad para dar soporte a las arquitecturas por las cuales están formadas las aplicaciones *mashup* y los componentes que las constituyen. Por último, se expone la *Ingeniería de Servicios*, paradigma utilizado para establecer un canal de comunicación entre las aplicaciones y la infraestructura de servicios. El capítulo finaliza con un resumen breve y con las principales conclusiones extraídas de la revisión tecnológica.

1.1. Introducción y conceptos relacionados

En una *Sociedad de la Información* tan cambiante como la que vivimos en la actualidad se demanda cada vez más entornos complejos capaces de resolver tareas muy dispares. La resolución de estas tareas debe poder realizarse desde cualquier sitio y en el menor tiempo posible. Con esto, surge la idea de recopilar y reutilizar "trozos" de aplicaciones diferentes, desarrollados por terceros, residentes en repositorios públicos o privados, y de integrarlas para dar soporte a la resolución de un conjunto de tareas concretas. De esta necesidad de aglutinar diversa funcionalidad, surge el concepto de "aplicación *mashup*".

Cuando se construye una aplicación *mashup* se crea un entorno que tiene una estructura fija formada por ciertos componentes *mashup*. ¿Pero qué ocurre si dichas aplicaciones debieran ser modificadas en el tiempo? Podría suceder que los usuarios necesitaran añadir o eliminar componentes del entorno para que éste cambie según sus necesidades. O desde una perspectiva más autónoma, podría darse el caso de que el entorno necesitara realizar algún cambio porque se haya detectado la necesidad de una modificación en base a ciertas reglas establecidas. Esta cuestión lleva también a plantearse ¿qué ocurre si el usuario quisiera abandonar lo que están haciendo en este momento? Podría ser interesante que su entorno de trabajo y su estado fuera persistente. ¿Qué ocurría

© 2016 Vallecillos, J.

además si los componentes necesitaran intercambiar información entre sí independientemente de la tecnología en la que fueron desarrollados? Quizás pudiera ser necesario que un componente informara a otro en tiempo de ejecución de que se debe realizar una acción en concreto, o que la aplicación *mashup* tenga que cambiar alguna característica o propiedad de un componente del entorno, como podría ser el tamaño del mismo.

Todas estas cuestiones hacen que se plantee la necesidad de usar unas tecnologías y definir una infraestructura que permita y facilite el soporte de aplicaciones *mashup* dinámicas en tiempo de ejecución. Sería interesante que las aplicaciones pudieran estar construidas por piezas pudiendo quitarlas o ponerlas en función de las necesidades del momento. Además, estas piezas tendrían que cumplir algunas características, como poder enviar o recibir mensajes para dar la posibilidad de llevar a cabo procesos de comunicación entre ellas. Habría que tener en cuenta además que estos elementos (o piezas) de software pueden estar desarrollados por otras personas (o terceros) que no son conscientes de dónde y cómo serán usadas ni quién se comunicará con ellas.

Por otro lado, el uso de técnicas de abstracción puede facilitar la tarea de diseño y desarrollo de las aplicaciones junto a los componentes que las forman, así como los cambios que se pueden producir en tiempo de ejecución dentro de la aplicación. Estos procesos de abstracción también pueden servir para almacenar la persistencia de las aplicaciones, además de permitir realizar una traza de la evolución del entorno de trabajo.

Con todo ello, también puede ser interesante que las aplicaciones pudieran estar conectadas a algún tipo de *infraestructura* que permitiera darles soporte. Por ese motivo, el uso de servicios facilita que un determinado sistema, en este caso una aplicación *mashup*, pueda realizar o solicitar operaciones a otro sistema. De esta forma, adquiere sentido que este soporte de las aplicaciones esté ubicado de forma remota, al cual las aplicaciones se conectarán.

Por las causas mencionadas con anterioridad, la infraestructura para el despliegue de las aplicaciones *mashup* que aquí se propone se sustenta principalmente en tres áreas de la *Ingeniería del Software*:

- *Ingeniería del Software Basada en Componentes*: se utiliza para la construcción de las aplicaciones con el objetivo de realizar un desarrollo basado en piezas. Concretamente, se hace uso de componentes de granularidad gruesa para la construcción de arquitecturas de componentes de mayor complejidad.

- *Ingeniería de Modelos*: los modelos son utilizados para la descripción de los componentes desarrollados, y para definir las arquitecturas de las aplicaciones formadas a partir de los componentes. Las técnicas de modelado ayudan en la manipulación de estos modelos, por ejemplo, facilitando la construcción de las aplicaciones a partir de los componentes y dando soporte a las aplicaciones con respecto a la persistencia, además de describir cuáles serán los caminos de la comunicación entre los componentes.

- *Ingeniería de Servicios*: los servicios software permiten la comunicación entre software desarrollado con diferentes tecnologías. La interoperabilidad que aportan los servicios se consigue a través de la adopción de estándares abiertos que definen cómo debe ser dicha comunicación. En este caso, se han implementado servicios web debido a su expansión y la facilidad de ser integrados en cualquier aplicación desarrollada.

En la Figura 1.1 se muestra la relación entre las distintas áreas que darán soporte al despliegue de las aplicaciones *mashup*. La Ingeniería del Software Basada en Componentes se apoya en la Ingeniería de Servicios para la gestión de los componentes dentro de la aplicación. Ambas se relacionan también con la Ingeniería de Modelos que proporcionan los mecanismos de representación y las operaciones que permiten realizar cambios en las aplicaciones. La Ingeniería de Modelos también permitirá llevar a cabo procesos de persistencia e informar de cuáles son los caminos para realizar los procesos de comunicación entre los componentes. Estos tres elementos han sido estudiados desde una perspectiva general para conocer qué son capaces de aportar a la hora de desplegar aplicaciones *mashup*. En su combinación con el dominio de aplicación, se constituye el paradigma global en el cual se enmarca el trabajo de investigación desarrollado. Cada uno de los bloques se describen en las siguientes secciones del capítulo.

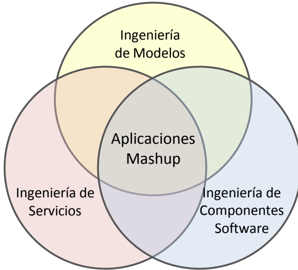


Figura 1.1: Tecnologías para el desarrollo de la infraestructura de servicios

1.2. Aplicaciones Mashup

Como se ha comentado, el presente trabajo de tesis doctoral tiene como objetivo la definición de un *modelo de infraestructura* que permita y facilite el soporte y despliegue de aplicaciones *mashup*. ¿Pero en qué consisten realmente las aplicaciones *mashup*? El concepto *mashup* es relativamente reciente y todavía este término genera ciertas dudas sobre lo que es exactamente. En la literatura existen diversas definiciones sobre el término *mashup*. Una de ellas aparece en [Florian and Maristella, 2014] la cual se apoya en los trabajos de [Hoyer and Stanoevska-Slabeva, 2009] y [Jin et al., 2008]:

Definición 1.1 (*Mashup*) *Una aplicación mashup (también denominada Web mashup) es una aplicación que integra dos o más componentes* mashup *en cualquiera de las capas de aplicación, es decir, en la capa de datos, en la capa de negocio o en la capa de presentación; y en algunos casos, permitiendo la comunicación entre ellas.*

Según esta definición, se destaca que los recursos reutilizables (los componentes *mashup* en este caso) son elementos básicos para construir la aplicación. El énfasis que pone la definición en las diferentes capas de la aplicación refleja que son un aspecto importante. Esto permite la heterogeneidad de las tecnologías, modelos y semántica por las que

© 2016 Vallecillos, J.

son desarrollados los componentes que forman las aplicaciones. Así que, este alto grado de heterogeneidad entre las tecnologías, modelos y semánticas de los componentes ha sido una de las aportaciones del paradigma *mashup*. Concretamente, ha tenido un importante impacto en el desarrollo del software basado en componentes para la computación orientada a servicios o la integración de datos, tendencias que se centran en integrar componentes homogéneos para potenciar algún tipo de recurso disponible o accesible en la red [Benslimane et al., 2008]. Dada la amplia gama de tecnologías existentes hoy en día para el desarrollo de componentes, se puede llegar a construir aplicaciones basadas en componentes heterogéneas (entendiendo por heterogénea que no se distinguen las capas de aplicación por las que está formada la aplicación *mashup*). Pero el desarrollo de la aplicación *mashup* es más compleja cuando se integran componentes que están localizados en diferentes capas de aplicación (Figura 1.2), por ejemplo, si se desarrolla una aplicación *mashup* que mezcla componentes pertenecientes a la capa de datos con componentes pertenecientes a la capa de presentación. Integrar componentes que pertenecen a diferentes capas de aplicación es una de las ventajas de las aplicaciones *mashup*.

La definición vista anteriormente entonces acota el término concretando que las aplicaciones *mashup* realizan nuevas aportaciones a través de un conjunto de recursos reutilizables que pueden trabajar de forma conjunta mediante procesos de computación. Aunque, al igual que ocurre en la combinación de servicios web, no es suficiente con poder invocar los servicios; es importante conectar de forma correcta la salida de un servicio con la entrada de otro.

En la Figura 1.2 (inspirada en [Florian and Maristella, 2014]), se observan dónde se encuentran los tipos de aplicaciones que se consideran *mashup* con respecto a aplicaciones de misión *no crítica*, aplicaciones *transaccionales* y aplicaciones de misión *crítica*. Se conoce como aplicación de misión crítica aquella aplicación que en caso de fallar tendría un impacto significativo en el funcionamiento de cualquier empresa, organización o institución que dependa de su información. Estos tres tipos de aplicaciones (no críticas, transacciones o críticas) pueden ir desde el correo electrónico, hasta sistemas de administración de una empresa, como cálculo de inventario, transacciones bancarias, nóminas y flujos de efectivo, o bien la gestión de finanzas o del sistema de pensiones por parte de un gobierno, por poner algunos ejemplos. Por tanto, las dos dimensiones por las cuales está construido el gráfico son la complejidad de la aplicación (eje de abscisas)

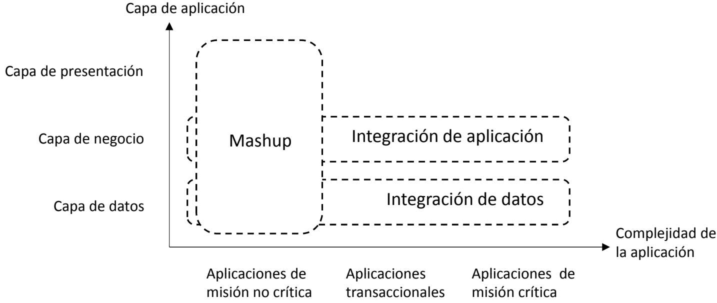


Figura 1.2: Posicionamiento de *mashups* comparado con otras prácticas de integración

y la pila de la capa de aplicación (eje de ordenadas). A través de esta figura se puede observar la contribución del desarrollo *mashup* para integrar componentes en la capa de presentación [Daniel et al., 2007].

Respecto al desarrollo de aplicaciones de misión crítica, las aplicaciones *mashup* no están pensadas para este tipo de desarrollo. Las razones son múltiples. Por un lado, hay aplicaciones *mashup* que están pensadas para ser desarrolladas a partir de un conjunto de componentes de tipo web; obviamente, podría ser difícil encontrar en la web todos los componentes que son necesarios para el desarrollo de aplicaciones de misión crítica. Por otro lado, las aplicaciones *mashup* no nacieron con este objetivo. Su principal objetivo era crear una aplicación simple a partir de otras. Para ello, estas aplicaciones se ensamblarían durante el tiempo necesario para encontrar una respuesta a una petición determinada o consulta que, una vez encontrada, las aplicaciones podrían ser eliminadas porque ya no serían necesarias. Muchas herramientas *mashup* pretenden también que los usuarios finales sin conocimiento de desarrollo de software, puedan crear sus propias aplicaciones. Con lo cual, todo esto aleja a las aplicaciones *mashup* para ser de misión crítica.

El resto de esta sección se divide de la siguiente forma. En primer lugar, se realiza un resumen sobre las aplicaciones *mashup*. En este resumen se lleva a cabo una recopilación de los cuatro tipos de aplicaciones *mashup* con los que se puede encontrar (de datos, de negocio, de presentación, e híbridas). Para cada tipo se habla de cuál es el objetivo de esa aplicación *mashup* y se describen distintos ejemplos. Por otro lado, se dedica una subsección para explicar los componentes *mashup*, donde se dan ejemplos para este tipo de componentes. Para finalizar la sección, se tratan aspectos de la comunicación y de interacción entre componentes *mashup*.

1.2.1. Tipos de aplicaciones Mashup

En la Figura 1.2 se puede observar la ubicación de las aplicaciones *mashup* dentro de una de las tres capas de aplicación [Florian and Maristella, 2014]. Esta clasificación de las aplicaciones *mashup* es muy similar a la categorización de los tipos de componentes *mashup* existentes. Sin embargo, los tipos de aplicaciones *mashup* no están directamente relacionadas con el tipo de componente que utiliza una aplicación *mashup*. Según [Florian and Maristella, 2014], se distinguen cuatro tipos de aplicaciones *mashup*: de capa de datos, de capa de negocio, de capa de presentación, y aplicaciones híbridas.

Aplicaciones Mashup de la capa de datos: operan dentro de la capa de datos. Estas aplicaciones se encargan de recuperar datos de diferentes servicios de datos o recursos, los procesan, y devuelven un conjunto de resultados integrados (la salida de datos *mashup*). De esta forma, estas aplicaciones hacen posible realizar operaciones de mediación de datos, tales como la modificación o integración de datos. También permiten realizar transformaciones de datos y llevar a cabo integraciones heterogéneas de estructuras de datos. Todas estas tareas se realizan por medio de una vista unificada que contiene el conjunto de datos a través de componentes individuales. Las aplicaciones *mashup* de datos están normalmente publicados como recursos web de fácil acceso, por ejemplo, archivos RSS [Board, 2009] o servicios web RESTful [Richardson and Ruby, 2008].

© 2016 Vallecillos, J.

Aplicaciones Mashup de la capa de negocio: estas aplicaciones proporcionan funcionalidad que se publica por medio de algún componente de negocio o de datos. Por ese motivo, debe existir heterogeneidad entre los diferentes tipos de componentes que forman la aplicación. También se debe controlar la coherencia de los datos que viajan desde unos componentes a otros. Las *mashup* de negocio contienen componentes que gestionan procesos y que son normalmente componentes de negocio, como objetos *JavaScript* o servicios SOAP. Se usa el término de aplicación *mashup* de negocio para distinguir este tipo de aplicación *mashup* de la composición de servicios web tradicionales, aunque se entiende que crear composición de servicios es la principal tarea de las aplicaciones *mashup* de negocio. Las *mashup* de negocio también hacen uso de componentes que no son servicios web, como objetos *JavaScript* o formularios *wrapper* HTML.

Aplicaciones Mashup de la capa de presentación: se localizan en la capa de presentación. Estas aplicaciones gestionan componentes de interfaz de usuario (IU). Combinan los componentes de interfaces de usuario de aplicaciones nativas en una interfaz de usuario integrada, donde los componentes que forman la IU posiblemente están sincronizados con otros componentes. Desarrollar aplicaciones de IU *mashup* conlleva realizar previamente una recopilación de componentes de IU procedentes de páginas web, además de tener que añadir sincronización entre los componentes que las forman. Las aplicaciones de IU *mashup* son normalmente publicadas como aplicaciones web donde los usuarios pueden interactuar con los componentes que las forman.

Aplicaciones Mashup híbridas: abarcan múltiples capas de aplicación. Contienen todos los tipos de componentes dentro de una misma aplicación *mashup*. La integración de los componentes *mashup* se lleva a cabo dentro de la mayoría de las capas que forman la aplicación. El principal objetivo de las aplicaciones *mashup* híbridas son la comunicación entre las tres capas, lo cual se consigue por medio de la integración de dichas capas. Las aplicaciones *mashup* híbridas son las que aportan mayor riqueza computacional ya que son las más complejas por permitir desarrollar aplicaciones completas, como pueden ser aplicaciones web interactivas.

Las aplicaciones *mashup* de negocio, en forma de servicios web, suelen estar integradas en aplicaciones *mashup* híbridas (por ejemplo, en servicios de geocodificación son utilizadas para traducir direcciones en coordenadas geográficas dentro de las aplicaciones *mashup* basadas en mapas), pero las aplicaciones *mashup* de negocio se centran más en la computación orientada a servicios y composición de servicios web.

1.2.2. Componentes Mashup

Uno de los pilares de las aplicaciones *mashup* es construir aplicaciones a través de trozos (o piezas) ya existentes. Generalmente a estos elementos que permiten el desarrollo de aplicaciones *mashup* se les denomina *componentes mashup*. En el ámbito del desarrollo basado en componentes es posible encontrar diferentes tipos de componentes. Pero un *componente mashup*, tal y como se puede comprobar en [Florian and Maristella, 2014], no puede ser cualquier tipo de componente, como se indica en la siguiente definición de *componente mashup*.

Definición 1.2 (*Componente Mashup*) *Un componente mashup es una pieza de datos, aplicación lógica o interfaz de usuario que puede ser reutilizada y es accesible ya sea de forma local o remota.*

Como se puede comprobar, la definición es muy amplia, en lo que a términos tecnológicos se refiere. Los *mashup* a nivel de composición de aplicaciones no están limitados a ningún componente tecnológico específico, pudiendo darse los componentes *mashup* en forma de servicios web SOAP [Bernstein and Haas, 2008] [Papazoglou, 2008], o *widgets* de interfaz de usuario, entre otros. Esta gran variedad de tecnología hace que las aplicaciones *mashup* puedan llegar a ser muy complejas, aunque esta complejidad a nivel de tecnología hace que las aplicaciones *mashup* sean atractivas e interesantes tanto para los usuarios que las utilizan como para aquellos que las desarrollan. Desarrollar componentes *mashup* de utilidad requiere dominar un gran número de tecnologías, de lógica de componentes, o formatos de datos, entre otros elementos. El objetivo de este apartado es realizar una pequeña recopilación de los tipos de componentes en función de las tecnologías que los definen y en función de cuál será su uso. Dicha recopilación está basada en la agrupación de componentes propuesta en [Florian and Maristella, 2014] (componentes de negocio, componentes de datos y componentes de interfaz de usuario).

1.2.2.1. Componentes de negocio

Estos componentes aportan lógica de negocio o funcionalidad. Por ejemplo, un componente de negocio puede proporcionar funcionalidad de pago (como un servicio PayPal) o lógica computacional por medio de reutilización de algoritmos. Para poder hacer uso de un componente de negocio hay que resolver cómo interactuar con él, esto es, cómo proporcionarle la información de entrada y de qué manera obtener la de salida. A continuación, se tratan los componentes de negocio como son los *servicios web*, los *servicios web RESTful*, *librerías* y *APIs JavaScript*, *APIs de dispositivos* y *extracción de APIs*:

(a) Los *servicios web* son probablemente la mejor forma estandarizada hoy día de computación distribuida, especialmente dentro de la Web. Hay muchas definiciones sobre lo que es un servicio web. Técnicamente, se podría decir que un servicio web es un objeto remoto con métodos de entrada y salida de datos que se ejecuta en la Web, el cual se identifica por medio de una URL y con el cual se puede interactuar a través de un mensaje basado en invocación de métodos. Esta invocación utiliza el estándar *Simple Object Access Protocol* (SOAP) [W3C, 2007]. SOAP es un protocolo de la capa de aplicación basado en XML para el intercambio de mensajes en servicios web. Los servicios web SOAP serán tratados en la subsección 1.5.1.

(b) Los *servicios web RESTful* son muy similares a los servicios web con la única diferencia de que estos no están basados en objetos SOAP. El concepto RESTful se aplica a aquellos servicios que cumplen con una arquitectura REST (*REst State Transfer*). Son servicios web que no guardan el estado del servicio tras una solicitud. Los servicios web RESTful sólo contienen información de estado en la invocación de sus mensajes y en sus respuestas. Los mensajes de entrada y salida ya contienen la información necesaria para gestionar los recursos. Los servicios RESTful son tratados más adelante con más detalle en la Subsección 1.5.2.

© 2016 Vallecillos, J.

(c) Las *librerías* y las *APIs JavaScript* son otra forma de reutilizar lógica de negocio en la Web. Se llama API (*Application Programming Interface*) a todas estas interfaces de programación proporcionadas a través de los navegadores. Por ejemplo, el objeto *XMLHttpRequest* habilita de forma asíncrona las llamadas HTTP en segundo plano en el navegador del cliente, con el objetivo de almacenar datos en el servidor. Se llaman librerías a todos aquellos archivos *JavaScript* que se pueden descargar en el cliente a través de la Web. Estos archivos proporcionan el código para ser usado en el navegador del cliente, el cual aporta nueva funcionalidad.

(d) Las *APIs de dispositivos* relacionadas con los dispositivos móviles. Debido a la gran potencia computacional de los dispositivos móviles se ha llegado a un nuevo tipo de API conocidas como *APIs de dispositivos.* Las APIs de dispositivos son *APIs JavaScript* que permiten a las aplicaciones web interactuar con otros servicios, como pueden ser calendarios, contactos, mensajería, cámaras, geolocalización, acelerómetros, etc. De esta manera, las APIs de dispositivos pretenden proporcionar funcionalidad a las aplicaciones que se ejecutan de forma local. El término "API de dispositivo" puede llevar a confusión, ya que puede parecer que sólo están disponibles para dispositivos *smartphone*, lo cual no es del todo cierto. Algunas APIs de dispositivos también están disponibles para equipos de escritorio, portátiles o tabletas, dependiendo de la funcionalidad que proporcione.

(e) La *Extracción de APIs.* En una típica aplicación web, se obtienen datos como entrada, que son procesados y producen nuevos datos de salida. Para encontrar un componente en la Web se comprueba si existe un área de desarrolladores y luego se busca, a través de la lista de APIs disponibles, algún componente lógico que se adapte a nuestras necesidades. En ocasiones, es necesario construir un *wrapper* que acepte datos por medio de algún mecanismo de entrada a la interfaz. Un ejemplo de *wrapper* podría ser un objeto *JavaScript* autónomo localizado en la parte cliente que se encarga de interactuar con una aplicación remota para que la parte cliente y servidor puedan comunicarse entre sí.

1.2.2.2. Componentes de datos

Estos componentes proporcionan datos por medio de sus interfaces. Los datos pueden ser estáticos, como simples archivos RSS o Atom, o dinámicos, los cuales requieren de una consulta de entrada, como pueden ser los servicios web. Básicamente existen cuatro tipos de componentes de datos: RSS (*Really Simple Syndication*), Atom (*Atom Syndication Format*), *extracción de datos de la Web* y *fragmentos enriquecidos.* A continuación, se describe cada uno de ellos:

(a) *Really Simple Syndication* (RSS) es una forma de sincronizar contenido web; es un formato de datos usado para generar publicidad y contenido dentro de la Web. El formato de datos, detallado en [Board, 2009], es un formato XML desarrollado específicamente para todo tipo de sitios que se actualiza con frecuencia, por medio del cual se puede compartir la información y usarla en otros sitios web o aplicaciones.

(b) *Atom Syndication Format* (Atom) permite unificar contenido con un mismo formato y es muy similar al RSS. Se creó con el objetivo de resolver algunas deficiencias del RSS, tales como la limitación de contenido, o no poder soportar modularidad dentro de la información. Otro objetivo del Atom era crear una especificación estándar, para poco a poco ir reemplazando el RSS. Aunque todavía a día de hoy no se ha conseguido reemplazar a RSS porque este llegó antes al mercado, ganando consistencia y soporte en muy poco tiempo.

(c) *Extracción de datos de la Web.* Como en el caso de los componentes lógicos, no siempre se puede encontrar el componente de datos que se está buscando. Podría ocurrir que la información que se necesita se encuentre de una aplicación web. De esta forma, se podrían extraer los datos desde el sitio web, obteniendo aquellos fragmentos en los que se está interesado, y volviendo a publicar estos datos dentro de algún componente. La extracción de datos se puede hacer de forma específica, lo que significa que se necesita llevar a cabo un proceso de reprogramación, lo que además implica conocer la estructura de las páginas web construidas en HTML.

(d) *Fragmentos enriquecidos* contienen información que es publicada en formato HTML [Allsopp, 2007]. Los fragmentos heredan la construcción de anotaciones del código HTML permitiendo crear estructuras de datos que puedan ser definidas por encima de los datos publicados en formato HTML. De esta forma, se facilita la extracción de datos desde páginas web. Por lo tanto, se podría decir que los fragmentos enriquecidos heredan código HTML con el propósito de crear metadatos que pueden ser utilizados. Existen tres técnicas de fragmentos enriquecidos. La primera son los llamados *Microformatos*, que son porciones de código HTML (o XHTML) que permiten estructurar información aprovechando los atributos `id` o `class`. Por otro lado, están los *Microdatos* los cuales son un conjunto de notificaciones en etiquetas construidos por la W3C. Como tercer tipo de técnica, las *anotaciones RDF* son fragmentos de código basados en un formato de anotación RDF simple.

1.2.2.3. Componentes de interfaz de usuario

Estos componentes se incrustan en una interfaz de usuario (IU) y dan la posibilidad de hacer uso de componentes lógicos que se pueden comunicar además con componentes de datos como, por ejemplo, Google Maps. Dentro de este tipo de componentes, se pueden encontrar: *fragmentos de código y librerías IU JavaScript*, *portlets*, *widgets y gadgets*, *recortes web y extracción de componentes de IU*. Veamos a continuación una pequeña descripción de cada uno de ellos:

(a) *Fragmentos de código y librerías IU JavaScript.* El tipo más simple de interfaz de usuario basada en componentes que se puede tener en la Web es una interfaz de usuario monolítica, como podría ser una imagen o una página web completa, que se integra en otra página. Un ejemplo son los *banners* publicitarios de terceras partes en una aplicación web. La forma más sencilla de usar este componente de IU era utilizar un elemento imagen de HTML (`img`) y referenciarlo como una imagen externa. Posteriormente esta forma de reutilizar componentes de tipo IU ha sido a través del elemento `iframe`.

© 2016 Vallecillos, J.

(b) Los *portlets* son también componentes IU basados en *Java* que pueden procesar solicitudes procedentes de los usuarios y generar contenido de manera dinámica[1].

(c) Los *widgets y gadgets* son una estandarización reciente de componentes de IU ubicados en el lado cliente. La estandarización ha sido realizada por W3C [Group, 2012]. Estos componentes son similares a los *gadgets* de OpenSocial [OpenSocial, 2011], los *gadgets* de Google[2] o los *widgets* de Yahoo![3]. Los *widgets* W3C son aplicaciones simples desplegadas en aplicaciones cliente similares a los *portlets* Java. Los *widgets* se basan en *JavaScript* y se ejecutan por tanto en la aplicación local.

(d) *Recortes web y extracción de componentes de IU.* Se puede dar el caso de necesitar extraer componentes de IU a partir de aplicaciones web ya existentes. En otros casos, se puede necesitar obtener un trozo de una página HTML y reutilizarla en otras páginas. Esto fue una práctica relativamente efectiva en los comienzos de la Web, donde las páginas web eran estáticas y no había lógica de negocio que estaba dinámicamente cambiando elementos en las aplicaciones cliente. Pero actualmente realizar recortes de páginas web ya no es una práctica interesante, especialmente porque la presencia de diferentes hojas de estilos (CSS) y lógica *JavaScript* hace que dicha tarea sea más compleja.

En la investigación desarrollada en esta tesis doctoral, se ha optado por utilizar *widgets* como tipo de componente de interfaz de usuario para la construcción de interfaces de usuario *mashup* híbridas. Este hecho nos ha permitido embeber código de terceros, como han sido las implementaciones de componentes geográficos basados en servicios OGC (*Open Geospatial Consortium*)[4], componentes de redes sociales o componentes específicos desarrollados por la REDIAM (*Red de Información Ambiental de Andalucía*)[5], como parte de un proyecto de investigación para la explotación de información ambiental a través de interfaces *mashup*.

1.2.3. Comunicación entre componentes Mashup

Un aspecto importante en los componentes *mashup* es la comunicación entre estos. Este aspecto está condicionado al tipo de acceso a dichos elementos (pudiendo ser local o remoto). Acceder de forma local a un componente implica comunicarse con él en la misma aplicación donde se está ejecutando. Por otro lado, el acceso a un componente de forma remota implica la obtención de un componente en sí o la ejecución de alguna de sus operaciones a partir de llamadas a un servidor externo. La comunicación entre componentes *mashup* se puede realizar de las dos formas tradicionales de comunicación entre objetos: síncrona y asíncrona.

En una interacción síncrona, denominada también como *interacción bloqueante*, un componente permanece bloqueado hasta que recibe una respuesta de otro. Para que dicho

[1]JSR 286: Portlet Specification 2.0 – `https://jcp.org/en/jsr/detail?id=286`
[2]Google Gadgets – `https://developers.google.com/gadgets/`
[3]Yahoo! widgets – `http://widgets.yahoo.com/`
[4]OGC – `http://www.opengeospatial.org/`
[5]REDIAM – `http://www.juntadeandalucia.es/medioambiente/site/rediam`

proceso pueda tener lugar, la llamada debe producirse mientras el servicio permanece a la escucha, para que se le pueda dar soporte a la invocación síncrona. Ejemplos de comunicaciones síncronas son la invocación de funciones de una librería *JavaScript*, o una operación de petición/respuesta a través de un servicio web.

En una interacción asíncrona (o *no bloqueante*), el componente que inicia la comunicación no permanece bloqueado a la espera de una respuesta. Pueden existir diversas respuestas de devolución por parte del componente invocado, pero la lógica del componente continúa funcionando igual. Una vez recibida la respuesta, puede ser que esta información recibida sea añadida al funcionamiento del componente. Las interacciones asíncronas no necesitan controlar que en el momento de la interacción ambas partes integradas deban estar activas a la vez. Los eventos DOM *JavaScript* o las notificaciones de servicios web son ejemplos de comunicaciones asíncronas.

En la infraestructura desarrollada en este trabajo de investigación, los componentes se ejecutan de forma local aunque se obtienen de forma remota de un servidor. Además, la comunicación entre los componentes se realiza de forma asíncrona, de esta forma, permitimos la definición de componentes con interfaces que permitan invocar sus operaciones según las necesidades de estos.

1.3. Ingeniería de Componentes Software

Como se ha comentado anteriormente, uno de los pilares de las aplicaciones *mashup* es que están construidas en base a componentes. Por este motivo, es necesario realizar una revisión de los conceptos de componente y arquitectura de componentes, así como de las implicaciones que tiene la utilización de la *Ingeniería del Software Basada en Componentes* (ISBC), también conocida en la literatura por su denominación en inglés CBSE (*Component-Based Software Engineering*).

Concretamente, en el desarrollo de software basado en componentes se parte de una descripción de la aplicación mediante una arquitectura de componentes software, la cual contiene una visión abstracta de la aplicación. A partir de una colección de componentes reales, residentes en un repositorio de componentes, las técnicas de ISBC tratan de localizar las mejores soluciones que casan con las necesidades impuestas en la arquitectura calculadas a partir de la combinación de componentes que hay en dicho repositorios de componentes. Por tanto, el desarrollo de software basado en componentes sigue una serie de etapas, como las propuestas en [Dean and Vigder, 1997]: (a) una primera fase de *búsqueda* de componentes software en los repositorios de componentes para satisfacer las necesidades de los usuarios o de una arquitectura de entrada, (b) una segunda fase de *evaluación* de los componentes a partir de unos criterios definidos, (c) una tercera fase de *adaptación* de los componentes que van a ser utilizados, y (d) una última fase de *ensamblaje* de los componentes. Incluso en otras propuestas de investigación, como la de [Barnes et al., 2014] se establece otra etapa adicional basada en un mecanismo de patrones para definir cuál puede ser la mejor *configuración* de componentes que cumpla con los objetivos de negocio de la organización.

En el trabajo de tesis doctoral que aquí se describe se sigue una propuesta ISBC para el desarrollo de los componentes y el manejo de las arquitecturas de componentes

© 2016 Vallecillos, J.

de las aplicaciones *mashup*. En las siguientes subsecciones se definen los principales elementos que intervienen en la propuesta. En primer lugar, se realiza una revisión de los conceptos de **componente** y **arquitectura de componentes**. Posteriormente, se describe cómo se lleva a cabo la reutilización en los sistemas basados en componentes. Para ello, se describen algunos de los mecanismos de especificación de componentes usados en operaciones de búsqueda y selección. Por último, la sección presenta algunas de las tecnologías existentes relacionadas con el uso de componentes.

1.3.1. Componentes

En el desarrollo de software se utiliza el término *componente* para hacer referencia a tres elementos: módulos por los que está formado el sistema, clases por las que está formado el sistema, o las funciones o procedimientos del sistema. Al igual que ocurre con la definición de *mashup*, es difícil encontrar una definición para *componente software* aceptada por todos los autores. Veamos algunas de las más extendidas a lo largo de la literatura. Quizás, una de las más conocidas y aceptadas haya sido la de [Szyperski, 2002].

Definición 1.3 (Componente software [Szyperski, 2002]) *Un componente es una unidad de composición con una interfaz especificada de manera contractual y dependencias del contexto explícitas. Un componente software puede ser desplegado de forma independiente y está sujeto a la composición de terceros.*

De esta forma, la definición se centra en la división entre la especificación de su interfaz (o interfaces) y su implementación. Esto quiere decir que, para usar un componente dentro de una aplicación, no se tiene en cuenta la forma en la cual fue implementado el componente. Además, la actualización del componente, o la modificación en su lógica de negocio, no tiene que afectar a la reconexión con la aplicación de la que forma parte [Crnkovic, 2001]. Existen otras definiciones de componente como la que proporciona el SEI (*Software Engineering Institute*) [Brown, 1999]:

Definición 1.4 (Componente software [Brown, 1999]) *Un componente software es un fragmento de un sistema software que puede ser ensamblado con otros fragmentos para formar piezas más grandes o aplicaciones completas.*

Otra definición de componente bien conocida es la de EDOC (*Enterprise Distributed Object Computing*) [OMG, 2004] de OMG (*Object Management Group*).

Definición 1.5 (Componente software [OMG, 2004]) *Un componente es algo que se puede componer junto con otras partes para formar una composición o ensamblaje.*

Otra definición de componente quizás más relacionada con las aplicaciones *mashup* es la que se usa para identificar los componentes Web.

Definición 1.6 (Componente software [Cooney, 2014]) *Un componente Web es una pieza de interfaz de usuario reutilizable que ha sido creada usando tecnología Web de código "abierto". Estos componentes se añaden en una aplicación como trozos de interfaz de usuario, los cuáles no necesitan usar librerías externas.*

La especificación de este tipo de componentes tiene actualmente algunas propuestas de implementación[6], entre las cuales se encuentra Polymer[7], que es un conjunto de librerías desarrolladas por Google para construir componentes Web.

A partir de las definiciones vistas anteriormente se puede decir que un componente software se puede combinar con otros elementos del mismo tipo para crear un sistema de mayor complejidad. El nivel de abstracción y complejidad de un componente determina cuál es su *granularidad*. En este sentido, un componente de *granularidad gruesa* es un componente que está compuesto por una agrupación de componentes, o puede tratarse de una pequeña aplicación que encapsula cierta funcionalidad y que es utilizada para construir otras aplicaciones o sistemas de tamaño superior. Si la complejidad de un componente es más simple, se acerca más a un componente de *granularidad fina*. La combinación de estos componentes, da lugar a una *arquitectura de componentes software*.

Pero cuando se trabaja en el ámbito de la ingeniería de componentes es indispensable contar con un tipo o definición de componente esencial: los componentes comerciales o componentes COTS (*Commercial Off-The-Shelf*). Estos se caracterizan por ser componentes desarrollados por terceros, que cuentan con una definición (o especificación) conocida, accesible y localizada por el resto desde repositorios públicos. Debido a que este tipo de componente ha sido capital en el desarrollo de la propuesta de investigación realizada en esta tesis doctoral, nos vamos a centrar con algo más de detalle en el siguiente apartado.

1.3.2. Componentes COTS

Históricamente hablando, el término COTS se remonta al primer lustro de los años 90, cuando en Junio de 1994 el Secretario de Defensa americano, William Perry, ordenó hacer el máximo uso posible de especificaciones y estándares comerciales en la adquisición de productos (hardware y software) para el Ministerio de Defensa. En Noviembre de 1994, el Vicesecretario de Defensa para la Adquisición y Tecnología, Paul Kaminski, ordenó utilizar estándares y especificaciones de sistemas abiertos como una norma extendida para la adquisición de sistemas electrónicos de defensa. A partir de entonces, los términos “comercial”, “sistemas abiertos”, “estándar” y “especificación” han estado muy ligados entre sí (aunque un término no implica los otros), estando muy presentes en estos últimos años en ISBC [Iribarne, 2003].

El término “componente comercial” puede ser referido de muy diversas formas, como por ejemplo, software *Commercial Off-The-Shelf* (COTS), o *Non-Developmental Item* (NDI), o incluso *Modifiable Off-The-Shelf* (MOTS) [Carney, 2000]. En realidad existen unas pequeñas diferencias entre ellos. Por ejemplo, un componente COTS es un software que (a) existe a priori, posiblemente en repositorios; (b) está disponible al público en general; y (c) puede ser comprado o alquilado. Un componente NDI se trata de un software desarrollado (inicialmente sin un interés comercial) por unas organizaciones para cubrir ciertas necesidades internas, y que puede ser requerido por otras organizaciones. Por tanto es un software que (a) existe también a priori, aunque no necesariamente en repositorios conocidos; (b) está disponible, aunque no necesariamente al público en general;

[6] WebComponents – `http://webcomponents.org/`
[7] Polymer – `https://www.polymer-project.org/`

© 2016 Vallecillos, J.

y (c) puede ser adquirido, aunque más bien por contrato. Por último, un componente MOTS es un tipo de software *Off-The-Shelf* donde se permite tener acceso a una parte del código del componente, a diferencia del componente COTS, cuya naturaleza es de caja negra, adquirido en formato binario, y sin posibilidad de acceder al código fuente.

Como sucede para el caso de los componentes software, en la literatura existen diversas definiciones para el término COTS. Una definición del término la podemos encontrar con la definición de elemento "COTS" del SEI, que dice lo siguiente:

Definición 1.7 (Elemento COTS del SEI, [Brown, 1999]) *Un elemento COTS se refiere a un tipo particular de componente software, probado y validado, caracterizado por ser una entidad comercial, normalmente de grano grueso y que reside en repositorios software, y que es adquirido mediante compra o alquiler con licencia, para ser probado, validado e integrado por usuarios de sistemas.*

Existen otros autores, como en [Meyers, 2001], que también consideran que un componente comercial no tiene necesariamente que ser adquirido mediante compra o licencia, sino que también puede ser adquirido como software de dominio público o desarrollado fuera de la organización.

En nuestro trabajo de investigación, para el uso de componentes "mashup", hemos adoptado la definición de componente COTS establecida por [Iribarne, 2003].

Definición 1.8 (Componente COTS, [Iribarne, 2003]) *Un componente COTS es una unidad de elemento software en formato binario, utilizada para la composición de sistemas de software basados en componentes, que generalmente es de grano grueso, que puede ser adquirido mediante compra, licencia, o ser un software de dominio público, y con una especificación bien definida que reside en repositorios conocidos.*

Por regla general, existe una gran diversidad de parámetros que caracterizan a un componente COTS, pero sin embargo, dos son los más comunes en la literatura de componentes COTS. En primer lugar, un componente COTS suele ser de grano grueso y de naturaleza de "caja negra" sin posibilidad de ser modificado o tener acceso al código fuente. Una de las ventajas de un software comercial es precisamente que se desarrolla con la idea de que va a ser aceptado como es, sin permitir modificaciones. Hay algunos desarrolladores de componentes que permiten la posibilidad de soportar técnicas de personalización que no requieren una modificación del código fuente, por ejemplo mediante el uso de *plug-ins* y *scripts*. Y en segundo lugar, un componente COTS puede ser instalado en distintos lugares y por distintas organizaciones, sin que ninguna de ellas tenga el completo control sobre la evolución del componente software.

Son muy numerosas las ventajas —aunque también lo son los inconvenientes— de utilizar componentes COTS en lugar de componentes de "fabricación propia". Una de las ventajas más claras es el factor económico, relacionado con el coste de desarrollo. Puede ser mucho más barato comprar un producto comercial, donde el coste de desarrollo ha sido amortizado por muchos clientes, que intentar desarrollar una nueva "pieza" software. Por otro lado, el hecho de que un componente COTS haya sido probado y validado por el vendedor y por otros usuarios del componente en el mercado, suele hacer que sea aceptado como un producto mejor diseñado y fiable que los componentes construidos

por uno mismo. Otra ventaja es que el uso de un producto comercial permite integrar nuevas tecnologías y nuevos estándares más fácilmente y rápidamente que si se construye por la propia organización.

En cuanto a las desventajas, destacamos principalmente dos, aunque estas derivan en otras más. En primer lugar, los desarrolladores que han adquirido un componente comercial no tienen posibilidad de acceso al código fuente para modificar la funcionalidad del componente. Esto significa que en las fases de análisis, diseño, implementación y pruebas, el componente es tratado como un componente de caja negra, y esto puede acarrear ciertos inconvenientes para el desarrollador, como por ejemplo no saber cómo detectar y proceder en caso de fallos; o que el sistema requiera un nivel de seguridad no disponible en el componente, entre otros problemas. Además, los productos comerciales están en continua evolución, incorporando el fabricante nuevas mejoras al producto y ofreciéndoselo a sus clientes (por contrato, licencia o libre distribución). Sin embargo, de cara al cliente desarrollador, reemplazar un componente por uno actualizado puede ser una tarea laboriosa e intensiva: el componente y el sistema deben pasar de nuevo unas pruebas (en el lado cliente).

Otra gran desventaja es que, por regla general, los componentes COTS no suelen tener asociados ninguna especificación de sus interfaces, ni de comportamiento, de los protocolos de interacción con otros componentes, de los atributos de calidad de servicio, y otras características que lo identifiquen. En algunos casos, las especificaciones que ofrece el fabricante de componentes COTS puede que no sean siempre correctas, o que sean incompletas, o que no sigan una forma estándar para escribirlas (las especificaciones). Otras veces, aunque el vendedor de componentes COTS proporcione una descripción funcional del componente, puede que ésta no satisfaga las necesidades del integrador, y que necesite conocer más detalles de la especificación del comportamiento y de los requisitos del componente.

Para nuestros propósitos de investigación hemos adoptado el modelo de documentación de componente COTS definido por [Iribarne, 2003]. El modelo de documentación recoge información funcional (interfaces, protocolos y comportamiento), información extra-funcional, información de implementación e implantación, e información de marketing. El modelo de documentación de componentes COTS está soportado por plantillas en XML que están basadas en el esquema COTS-XMLSchema desarrollado a partir del lenguaje *XMLSchema Language* del W3C. Esta plantilla de definición de componente COTS será detallada más adelante en el Capítulo 2.

1.3.3. Arquitecturas de componentes

El propósito del desarrollo de software basado en componentes es poder construir sistemas más complejos a partir de piezas ya existentes. Con este tipo de desarrollo, lo que se pretende es realizar un símil con el desarrollo de sistemas mecánicos donde se usan piezas mecánicas (tales como engranajes o muelles) para construir una máquina que realice alguna función. Basándonos en esta idea, el objetivo de una arquitectura de componentes es utilizar componentes software para desarrollar un sistema más complejo que cumpla con una determinada tarea. A continuación se ofrece una definición del término *arquitectura software.*

© 2016 Vallecillos, J.

Definición 1.9 (Arquitectura software [Bass et al., 2003]) *La arquitectura software de un programa o de un sistema de computación es la estructura o las estructuras de dicho sistema y consta de los siguientes elementos: a) los componentes software, b) las propiedades de dichos componentes y que son visibles de forma externa, y c) las relaciones entre ellos.*

Se puede decir que una *arquitectura software* se encuentra entre el proceso de requisitos e implementación, ya que permite realizar una abstracción del sistema para exponer ciertas propiedades del mismo y ocultar otras. Las arquitecturas software son importantes en al menos seis aspectos del desarrollo de software según [Garlan, 2000]:

(a) *Comprensión del sistema*: una arquitectura facilita entender lo que hace el sistema, pues permite expresarlo con un alto nivel de abstracción en donde los aspectos de diseño pueden ser fácilmente comprendidos.

(b) *Reutilización*: las descripciones arquitectónicas permiten que se puedan llevar a cabo procesos de reutilización, generalmente de componentes y de marcos de trabajo (*frameworks*).

(c) *Construcción*: un planteamiento arquitectónico permite tener una visión parcial del sistema que se desea construir, describiendo los componentes y sus relaciones.

(d) *Evolución*: permite crear un límite de la parte funcional de un componente. Por tanto, la división hace más fácil que se puedan realizar cambios en la arquitectura por motivos de interoperabilidad, prototipado y reutilización.

(e) *Análisis*: con la arquitectura software se puede hacer un nuevo análisis y refinar así los requisitos identificados.

(f) *Decisión*: permite desvelar ciertos detalles que permiten decidir qué estrategia de implementación utilizar, modificar o incluir en los requisitos.

En una *arquitectura software* se definen los detalles de diseño del conjunto de componentes y sus relaciones, lo cual forma una vista abstracta del sistema modelado.

Existe una gran variedad de elementos arquitectónicos que simplifican las tareas de diseño en la construcción de una arquitectura. Estos elementos que forman parte de la arquitectura se conocen como *estilos arquitectónicos*. Un estilo arquitectónico está compuesto por un conjunto de estilos de componentes a nivel arquitectónico y por unas descripciones de *patrones* de interacción entre ellos [Bass, 2007]. Estos componentes son utilizados para modelar las interacciones que hay dentro de una infraestructura de componentes. De la misma forma que los patrones de diseño de [Gamma et al., 1994] ayudan a los desarrolladores a diseñar sus clases, los estilos arquitectónicos sirven de ayuda en las tareas de diseño de componentes en una arquitectura software.

Tradicionalmente una *arquitectura software* se centra en describir y analizar estructuras estáticas. Pero en entornos complejos donde intervienen componentes de grano grueso, el sistema puede necesitar patrones arquitectónicos complejos como en [Cuesta, 2002],

donde se propone una arquitectura de software dinámica que utiliza un enfoque reflexivo para permitir la reconfiguración automática de la arquitectura como respuesta a la evolución del sistema.

Otro aspecto importante en la construcción de arquitecturas software es el proceso de evaluación que determina la validez de la estructura realizada. Esta tarea, aunque está centrada en la resolución de la estructura de los componentes de la arquitectura, conlleva una serie de procesos asociados. El proceso central es la resolución de la funcionalidad requerida en la arquitectura. Además, también es necesario abordar la resolución de otro tipo de requisitos no funcionales, relacionados con algún caso de restricción o con aspectos de calidad de servicio (QoS, *Quality of Service*) [Buschmann et al., 2012]. En este sentido, surgen técnicas específicas para el cálculo de atributos de calidad vinculados con las arquitecturas de componentes, en lo que se denomina *calidad de servicio* de la arquitectura (QoSA, *Quality of Software Architectures*) [Buhnova et al., 2014].

1.3.4. Especificación de componente software

Un componente software debe ser descrito por medio de una especificación a través de la cual se puede identificar y localizar al componente. Una especificación de componente debe contener la información suficiente para poder ser seleccionado dentro de un conjunto de componentes, para ser luego evaluado y ensamblado junto con el resto de componentes. Por tanto, el objetivo principal de una especificación es permitir que se pueda determinar si un componente puede formar parte de una arquitectura de componentes, y que pueda interactuar con el resto de componentes al contar con la funcionalidad que se necesita dentro del sistema.

La especificación de un componente debe ser descrita cumpliendo con un estándar existente, para que éste sea interpretable por todo tipo de usuarios del sistema, pero en especial, por procesos automáticos de búsqueda. Según el trabajo de [Han, 1999], un componente software puede ser definido a partir de una parte sintáctica y otra semántica. La *parte sintáctica* está formada por sus *atributos*, *operaciones* (métodos) y *eventos*. Por otro lado, la *parte semántica* (o dinámica) describe el *comportamiento* de los elementos mencionados, como por ejemplo los *protocolos* de interacción del componente. También debe existir una sección de *propiedades* que es necesaria para describir el conjunto de características extra-funcionales, como puede ser el tamaño del componente o si su tamaño puede ser modificado.

Un aspecto importante de un componente es la definición de su interfaz, descrita en su parte sintáctica. Definir una interfaz permite aislar la parte de implementación de la parte funcional del componente [Crnkovic, 2001]. Una interfaz es una abstracción de un servicio que define las operaciones proporcionadas por dicho servicio, pero no sus implementaciones. Por ejemplo, en CORBA (*Common Object Request Broker Architecture*) [OMG, 2006] de OMG, las interfaces están constituidas por un conjunto de operaciones de entrada y de salida, y atributos. Las operaciones de entrada son operaciones que el componente implementa. Las de salida son operaciones requeridas por el componente para que éste funcione correctamente. Las primeras se describen dentro de un tipo de interfaces denominadas *interfaces proporcionadas* (*provided*) por el componente, mientras que las segundas se describen en las denominadas *interfaces requeridas* (*required*).

© 2016 Vallecillos, J.

En la Tabla 1.1 se puede observar una descripción de un componente con dos interfaces, una proporcionada (`GestionarCuenta`) y otra requerida (`InfoCuenta`). Se ha hecho uso del lenguaje IDL (*Interface Description Language*) de CORBA para describir dichas interfaces. La interfaz `GestionarCuenta` tiene dos operaciones `comprarAccion` y `venderAccion`, las cuales reciben como entrada el identificador de la cuenta. La operación `comprarAccion` devuelve un entero que identifica el registro de la compra mientras que la operación `venderAccion` no devuelve nada. Este componente hace uso de la operación `validarCuenta` de otro componente para llevar a cabo la validación de la cuenta de un usuario. Para ello se envía el identificador de la cuenta y su contraseña.

Interfaz proporcionada	Interfaz requerida

`interface GestionarCuenta {`
`  int comprarAccion (in int id);`
`  void venderAccion (in int id);`
`};`
	`interface InfoCuenta {`
	`  boolean validarCuenta (`
	`          in String idCuenta,`
	`          in String password);`
	`};`

Tabla 1.1: Ejemplo de definición de interfaces con IDL

Otro lenguaje para realizar descripciones de interfaces es WSDL (*Web Services Description Language*) para definir componentes como servicios web. Según la Tabla 1.2, una definición WSDL se compone principalmente de dos partes, una parte abstracta y otra parte concreta. La parte abstracta sirve para definir la estructura de sus operaciones, mientras que la parte concreta permite describir dónde está ubicado un servicio y de qué manera puede ser invocado. De esta forma, el estándar WSDL (un estándar basado en XML) se puede utilizar para describir las interfaces de componentes software.

```
<!-- Estructura de una definición WSDL  -->
<definitions name="ServiceName"
  targetNamespace="http://example.org/serviceName/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
   <!-- parte abstracta -->
   <types> ...
   <message> ...
   <portType> ...

   <!-- parte concreta -->
   <binding> ...
   <service> ...
</definition>
```

Tabla 1.2: Estructura de una definición WSDL

Como se ha comentado, la descripción de un componente queda establecido por medio de la especificación de las interfaces. Pero no siempre esta información es suficiente para definir un componente por completo. Es necesario también una definición de su comportamiento. La definición del *comportamiento* de un componente se usa para añadir información semántica que describa el funcionamiento de los métodos [Vallecillo et al., 1999]. Para realizar descripciones sobre el comportamiento de los componentes existen lengua-

jes tales como Larch [Dhara and Leavens, 1996], OCL (*Object Constraint Language*) [Warmer and Kleppe, 1998] o JML (*Java Modeling Language*) [Kilov et al., 1999]. Es decisión del desarrollador del componente usar un lenguaje u otro para definir la parte semántica o de comportamiento.

En una descripción de componente también están las propiedades extra-funcionales, como por ejemplo, características de calidad de servicio (*Quality of Service*, QoS). La información extra-funcional se puede clasificar de diferentes formas. Algunos ejemplos de este tipo de propiedades son la *usabilidad*, la *portabilidad*, la *facilidad de prueba* o la *facilidad de instalación*. También hay otra información que puede resultar valiosa (y que puede considerarse como extra-funcional), como la referencia a datos relacionados con la implementación, información de empaquetamiento u otra información necesaria para el mercado o la industria de componentes. Algunos ejemplos específicos de estos tipos de propiedades son el precio de uso de un componente, el lenguaje en el que está implementado, o la localización donde se encuentra, entre otros elementos [Iribarne et al., 2004].

En resumen, una especificación de un componente queda establecida por un documento que contiene la descripción de las siguiente cuatro partes:

(a) *información sintáctica* de los atributos, las operaciones (o métodos) y los eventos de las interfaces de un componente;

(b) *información semántica* acerca del comportamiento de estos operadores;

(c) *información de protocolos*, que determina la interoperabilidad del componente con otros componentes;

(d) y un conjunto de *propiedades extra-funcionales* del componente, como propiedades de seguridad, fiabilidad o rendimiento, entre otras.

Para nuestros propósitos de investigación, hemos adoptado y extendido el modelo de documentación de componentes COTS denominado `COTScomponent` establecido en [Iribarne, 2003]. Dicho modelo de documentación está soportado por un lenguaje que permite definir documentos COTS. El lenguaje está descrito con un esquema gramatical en XML, utilizando la notación XML-Schema del W3C[8]. El esquema gramatical determina cómo se debe escribir un documento COTS del modelo en una plantilla (*template*) XML para recoger la especificación de un componente comercial.

A modo de ejemplo, en la Tabla 1.3 se muestra la definición de esquema de un documento COTS, y en la Tabla 1.4 se muestra el esqueleto de una instancia de plantilla de un documento COTS.

Como se puede comprobar, una plantilla de documento COTS comienza por el elemento `COTScomponent`, y dentro de éste se definen sus cuatro partes. Además, el elemento principal `COTScomponent` también puede contener uno o más atributos, siendo `name` el único atributo obligatorio utilizado para recoger el nombre del componente que se está documentando. Los dos atributos `xmlns` son dos espacios de nombres, el primero de ellos se utiliza para establecer el lugar donde reside la gramática del lenguaje de un documento COTS (`COTS-XMLSchema.xsd`) y el segundo permite utilizar los tipos de datos definidos en la gramática base `XMLSchema` del W3C.

[8]XMLSchema – `http://www.w3.org/2000/10/XMLSchema`

© 2016 Vallecillos, J.

Documento COTS: <COTScomponent>

```
<xsd:element name="COTScomponent" type="exportingType"/>
<xsd:complexType name="exportingType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:sequence>
    <xsd:element name="functional" type="functionalType"/>
    <xsd:element name="properties" type="propertiesType" minOccurs="0"/>
    <xsd:element name="packaging" type="packagingType" minOccurs="0"/>
    <xsd:element name="marketing" type="marketingType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Tabla 1.3: Definición de esquema de un documento COTS

```
1:  <COTScomponent name="OnePlaceBuffer"
2:                 xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd"
3:                 xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
4:    <functional> ... </functional>
5:    <properties> ... </properties>
6:    <packaging> ... </packaging>
7:    <marketing> ... </marketing>
8:  </COTScomponent>
```

Tabla 1.4: Una instancia ejemplo de un documento COTS

Como se ha comentado, en el presente trabajo de investigación se ha adoptado y extendido la definición de componente `COTScomponent` para especificar los componentes *mashups* usados por el modelo de infraestructura propuesto para el despliegue de aplicaciones *mashup*. En el siguiente capítulo se explicará el uso dado a esta plantilla y cómo ha sido adoptada a partir de la definición de un metamodelo de componentes.

1.3.5. Tecnologías de comunicación entre componentes

Desde sus inicios, en la Ingeniería del Software Basada en Componentes se han propuesto diferentes modelos de componentes, cada uno con sus propias peculiaridades. Como características comunes a todos ellos se encuentran (a) los tipos de componentes que se pueden construir, (b) el tipo de arquitecturas que se pueden construir con ellos, (c) los procesos de comunicación que tienen lugar entre los componentes, y (d) las plataformas o sistemas en los que se pueden aplicar [Crnkovic et al., 2011]. Algunos ejemplos de modelos de componentes son: *Entreprise JavaBeans* (EJB) [DeMichiel and Keith, 2006], *Microsoft Component Object Model* (COM) [Box, 1998], *CORBA Component Model* (CCM) [OMG, 2006], *Open Services Gateway* (OSGi) [OSGi Alliance, 2007], o Fractal [Bruneton et al., 2006], entre otros.

Los procesos de comunicación entre componentes son importantes para que el conjunto de componentes del sistema puedan realizar las tareas por las cuales se construyó. Para cada modelo de componente existe una manera de llevar a la práctica los procesos de comunicación. Para el caso del modelo COM, los componentes permiten la comunica-

ción de manera transparente, de tal forma que los clientes no son conscientes de dónde se están ejecutando estos componentes, pudiendo estar ubicados en la misma máquina, o en máquinas diferentes. A todos los componentes COM se accede a través de sus propias interfaces. La interfaz del componente debe estar disponible para que cuando se produzca una llamada a dicha interfaz, éste esté listo para poder responder. En el caso de que el componente no esté disponible, se busca otro a través de un *proxy* ubicado en un servidor que pueda proporcionar dicha información [Box, 1998].

El modelo de componentes CORBA (*Common Object Request Broker Architecture* de OMG responde a la siguiente forma de comunicación [OMG, 2006]. La comunicación entre dos componentes se realiza a través de un objeto ORB (*Object Request Broker*) el cual actúa como mediador para localizar la referencia de uno sobre el otro. Para un componente que juega el papel de servidor (o productor), debe estar previamente registrada una referencia del mismo en el ORB. Seguidamente otro componente que juega el papel de cliente (o de consumidor), se conecta con el objeto ORB solicitando una referencia del componente destino sobre el cual se desea conectar. El ORB localiza y recupera la referencia del componente servidor, que se la ofrece al cliente para establecer una comunicación directa cliente/servidor entre ambos componentes. Tanto en el lado cliente como en el del servidor, debe existir funcionando una instancia del objeto ORB, el cual opera como una especie de bus virtual de comunicación entre ambos componentes.

En el caso del modelo OSGi se hace uso de eventos para llevar a cabo los procesos de comunicación entre los componentes. En este modelo de componente existe un servicio de administración para manejar los eventos, en el cual se publican los componentes y los eventos, describiendo qué componente está relacionado con cada evento. Para cada evento publicado también se define su tipo, además de las propiedades del evento que serán tenidas en cuenta durante el proceso de comunicación [OSGi Alliance, 2007].

Respecto al modelo Fractal, la comunicación entre componentes sólo es posible si sus interfaces están conectadas entre sí. En Fractal este tipo de conexión se llama *binding*. En un *binding*, una interfaz cliente se conecta con una interfaz servidor. Puede darse el caso de que para conectar una interfaz cliente con una interfaz servidor sea necesario colocar un *middleware* construido a partir de un conjunto de componentes que harán de intermediarios entre ambas interfaces. El concepto de *binding* es lo que en otros modelos de componentes se conoce como *connector* [Bruneton et al., 2006].

Por otro lado está la tecnología para la comunicación de componentes web, como la utilizada en *Portlets*. Un *Porlet* es un componente web incrustado en una página web cuyo objetivo es darle una funcionalidad extra a la aplicación. Para este tipo de componente existe una especificación donde se define cómo deben ser construidos los componentes *Portlet* y cómo deben ser sus procesos de comunicación [Hepper, 2008]. En OMELETTE [Chudnovskyy et al., 2013], se expone cómo extender el contenedor de componentes Apache Wookie y *Shinding* para establecer procesos de comunicación entre las interfaces de los componentes *widgets*. El objetivo de esta extensión es añadir automáticamente un código fuente a las instancias de los *widgets* para que estén a la escucha de eventos de tipo DOM y de esta forma los componentes puedan intercambiar información. En [Sire et al., 2009], se construye una API con el propósito de comunicar *widgets* entre sí mediante eventos de mensajes. El *widget* origen gestiona su envío de eventos a través del método *addWidgetEventListener* y el método *removeEventListener*.

© 2016 Vallecillos, J.

1.4. Ingeniería de Modelos

Otro de los pilares fundamentales en este trabajo de tesis doctoral es la *Ingeniería de Modelos*, ya que en dicha propuesta se utilizan modelos y metamodelos para la definición de los elementos principales que intervienen en la infraestructura para el despliegue de aplicaciones *mashup*.

Cuando se habla de Ingeniería de Modelos, es interesante aclarar diferentes conceptos que son utilizados dentro de este ámbito, como son los conceptos de Arquitectura Dirigida por Modelos o *Model-Driven Architecture* (MDA), Desarrollo Dirigido por Modelos o *Model-Driven Development* (MDD), Ingeniería Dirigida por Modelos o *Model-Driven Engineering* (MDE) e Ingeniería Basada en Modelos o *Model-Based Engineering* (MBE). En determinadas ocasiones, el uso de los conceptos MDA y MDD genera incertidumbre por su similitud. MDD es un paradigma de desarrollo donde los modelos son el pilar fundamental, al contrario de otros paradigmas de desarrollo donde el elemento fundamental son los programas. De forma general, MDD se utiliza para crear código a partir del uso de modelos. Dentro de MDD las principales operaciones para la manipulación de los modelos son las transformaciones.

En cambio, MDA es la iniciativa de *Object Management Group* (OMG) para llevar a la práctica la estandarización de MDD. Nace en el año 2001 para establecer un conjunto de tecnologías que fueran interoperables entre sí para aplicar de forma efectiva el desarrollo dirigido por modelos. MDA establece una serie de estándares para gestionar modelos, como por ejemplo, *Unified Modeling Language* (UML) para el modelado, *Meta-Object Facility* (MOF) para el metamodelado, o *XML Metadata Interchange* (XMI) para el intercambio de modelos. Además, MDA propone utilizar los modelos en tres niveles de abstracción: *Computation Independent Model* (CIM), *Platform Independent Model* (PIM) y *Platform Specific Model* (PSM) (ver Figura 1.3). Los modelos CIM describen los requisitos del sistema, sin tener en cuenta detalles de la estructura ni de la funcionalidad. Los modelos PIM detallan la funcionalidad sin entrar en la plataforma donde se ejecuta el software. Para finalizar, los modelos PSM añaden información sobre la implementación del software para ser ejecutado en una plataforma concreta. De esta forma, MDA se encuentra dentro de MDD.

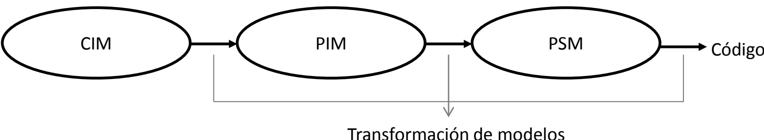


Figura 1.3: Niveles de modelado en MDA

El concepto MDE engloba a MDD, ya que las actividades que contiene esta ingeniería va más allá de las enfocadas únicamente al desarrollo del software, como por ejemplo, hacer uso de modelos en tiempo de ejecución, o para ayudarse de los modelos en alguna tarea relacionada con el mantenimiento del software. Para el caso de MBE, este se utiliza cuando se quiere dar una connotación a grandes rasgos de los modelos. O sea, los modelos están presentes y son un elemento importante e incluso clave, pero las actividades no

están "dirigidas" por los modelos. En ciertas ocasiones, estos elementos se podrían haber cambiado por otras alternativas de representación; en estos casos, las acciones que se encargan de manipularlos no están enfocadas en los modelos. La Figura 1.4 muestra la relación entre los cuatro conceptos. Vemos que MDA está dentro de MDD, el cual se considera dentro del subconjunto de MDE. MBE, como se observa, es el superconjunto que engloba al resto de conceptos. La clasificación que se ve en esta subsección es un tema abierto aún en la comunidad[9] y analizada en algunos trabajos existentes en la literatura, como [Ameller, 2009].

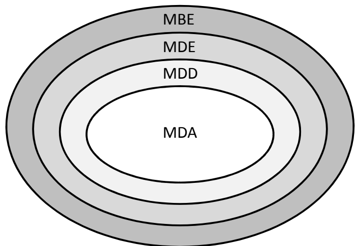


Figura 1.4: MBE, MDE, MDD y MDA

El presente trabajo de investigación hace uso del concepto MBE para mantener los modelos de las aplicaciones *mashup* y los modelos a través de los cuales se representan los componentes que forman dichas aplicaciones *mashups*. Para la definición formal de estos modelos, es necesario cumplir con sus correspondientes metamodelos, en los que se describe cómo deben ser construidos dichos elementos.

1.4.1. Metamodelado

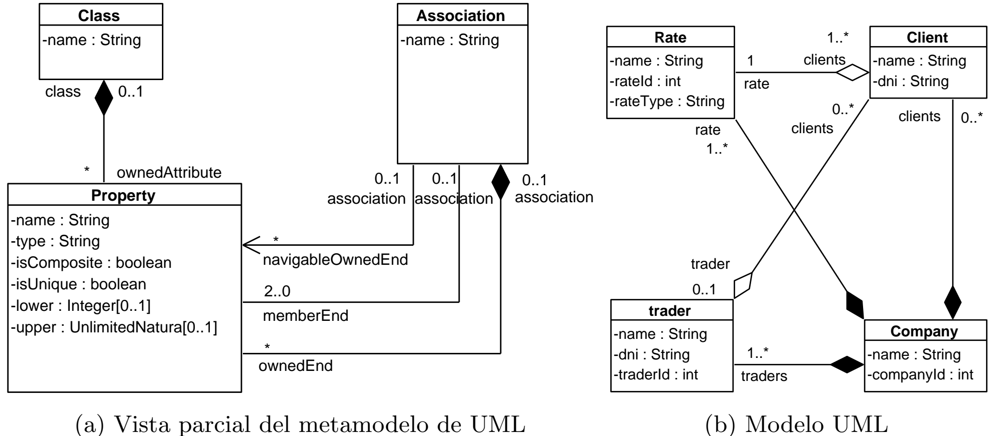
Conforme a su significado, el prefijo "meta" se utiliza para definir algo que está por encima, o dicho de otra forma "que trasciende". En la línea de este significado, el uso de este prefijo cobra un significado más "reflexivo" en algunos ámbitos de la ciencia y en especial, en el campo de la informática. De esta manera, se utiliza el concepto *metadatos* para hacer referencia a datos acerca de los datos, al igual que se utiliza el concepto metaintérprete para describir un intérprete de un intérprete (o un programa).

El concepto *metamodelado* define el proceso mediante el cual se crean modelos para representar modelos (*i.e.*, metamodelos). Desde un punto de vista global, se utilizará el concepto de "metamodelado" para hacer referencia al análisis, diseño y construcción de los metamodelos que se usan para solucionar ciertos problemas. Estos metamodelos representan los conceptos de un dominio en términos de metaclases y de las relaciones entre ellas (metareferencias o meta-asociaciones). De forma análoga a los diagramas de clases de UML, los metamodelos permiten definir la sintaxis abstracta de un nuevo lenguaje para representar la estructura de objetos que se pueden representar dentro de un dominio, o sea, las entidades, los atributos, las cardinalidades y los roles.

[9] `http://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/`

© 2016 Vallecillos, J.

Por ejemplo, en la Figura 1.5a se muestra una vista parcial del metamodelo de UML para escribir diagramas de clases. Mediante este lenguaje, se pueden crear modelos en los cuales intervienen tres tipos de elementos: clases, propiedades y asociaciones. La Figura 1.5b expone un modelo construido a partir del metamodelo anterior. Por ejemplo, la clase *Cliente* tiene cuatro propiedades: *dni*, *nombre*, *comercial* y *tarifa*. Las propiedades *comercial* y *tarifa* pertenecen a la asociación que la clase *Cliente* mantiene con las clase *Comercial* y *Tarifa*, respectivamente.



(a) Vista parcial del metamodelo de UML (b) Modelo UML

Figura 1.5: Metamodelo ejemplo

Como se ha indicado, los metamodelos permiten definir lenguajes para construir modelos. Dichos lenguajes pueden ser de propósito general, como UML, o pueden ser creados con un propósito específico, como son los lenguajes específicos del dominio o DSL (*Domain-Specific Language*) [Whittle et al., 2014]. En los dos casos, tal y como sucede para crear modelos, se necesita un metamodelo. Para la creación de un metamodelo se necesita un lenguaje de metamodelado, el cual recibe el nombre de metamodelo (ver Figura 1.6). Para que un lenguaje se pueda utilizar como metamodelo, debe poder crearse él mismo con su propio lenguaje. El estándar MOF (*Meta-Object Facility*) [ISO/IEC, 2014], propuesto por OMG, es el metamodelo más usado por la comunidad MDE para definir lenguajes de modelado. De esta forma, por ejemplo, el propio lenguaje UML se define mediante el uso de MOF.

Hay diferentes herramientas para crear metamodelos, todas basadas en el estándar MOF. Algunas de las más conocidas son [Steinberg et al., 2008] *Eclipse Modeling Framework* (EMF), MetaEdit+ [Tolvanen and Kelly, 2009], Oslo (*Microsoft SQL Server Modeling*, SSM) [Brunelière et al., 2010] y *Architecture of integrated Information Systems* (ARIS) [Kern and Kühne, 2007]. De todas estas, EMF es una de las herramientas más usadas por la comunidad por su grado de madurez, soporte y desarrollo de este *framework*. EMF ofrece editores gráficos para crear lenguajes de modelado, mecanismos para la validación y depuración, además de la posibilidad de generar de forma automática APIs basadas en *Java* para manipular metamodelos y sus modelos asociados.

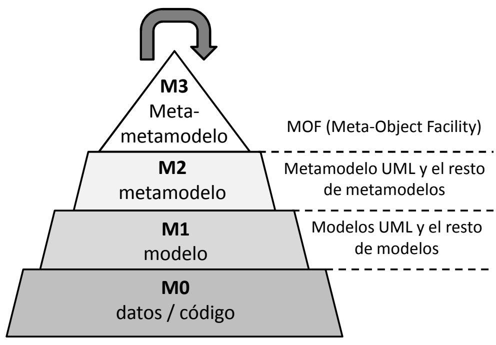


Figura 1.6: Niveles de modelado

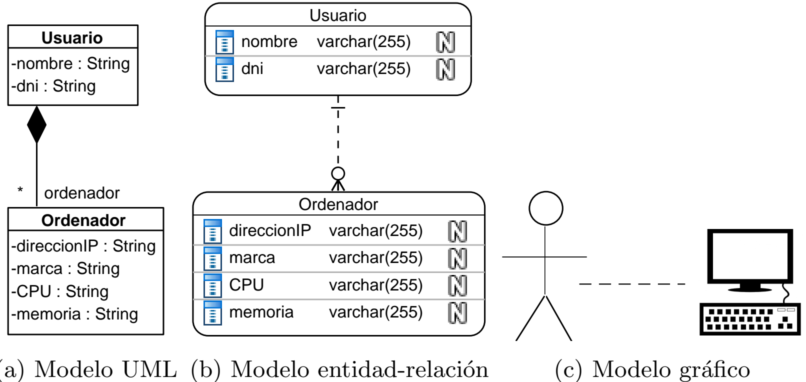
1.4.2. Modelos

Una vez construido el metamodelo o los metamodelos que se usarán para las diferentes actividades de ingeniería de software, a partir de ellos se definen los modelos que describen el sistema que está tratando de modelar. Para MDE, "todo es un modelo", ya que cualquier elemento del mundo se puede representar por medio de este paradigma.

El uso de modelos simplifica el proceso de desarrollo del software, y mejora la interoperabilidad de los sistemas y la comunicación entre los desarrolladores y otras personas que forman parte del equipo. Esto es debido gracias al mecanismo de abstracción que proporcionan los modelos, permitiendo representar parte de la realidad de forma simplificada, y de forma que sea sencillo para el usuario entender el software. Por ejemplo, supongamos que se quiere representar la relación que hay entre una persona y su ordenador mediante un modelo. La Figura 1.7 muestra tres posibles modelos que identifican y describen los conceptos que forman parte de este dominio. Cada modelo define la información relacionada con el lenguaje de modelado (metamodelo) con el que se creó. La Figura 1.7a es un modelo UML representado por un diagrama de clases, la Figura 1.7b muestra un modelo entidad-relación, y la Figura 1.7c muestra un modelo gráfico que describe la información a partir de formas y figuras. Aunque en la Figura 1.7 se muestran tres modelos diferentes, existe otra alternativa en la cual el modelo que describe la información es igual para los tres casos, y lo único que cambia es el diagrama que lo representa. Mientras que el término modelo se usa para definir la descripción y abstracción de una parte del sistema utilizando un lenguaje de modelado, el término diagrama hace referencia a la representación gráfica de un modelo o un conjunto de modelos.

En función del lenguaje que se use para construir modelos, la representación del modelo puede cambiar. Para usar UML con el objetivo de modelar, existen multitud de herramientas para crear, visualizar y modificar diagramas asociados a los modelos. Sin embargo, utilizar otros lenguajes de modelado de propósito general o de lenguajes específicos de dominio, requiere el desarrollo de herramientas para dar soporte a su creación o visualización. Con respecto a la codificación interna de los modelos, hay un consenso para utilizar el estándar XMI (*XML Metadata Interchange*) para representar los modelos en los diagramas correspondientes.

© 2016 Vallecillos, J.



(a) Modelo UML (b) Modelo entidad-relación (c) Modelo gráfico

Figura 1.7: Modelos de ejemplo

1.4.3. Modelos en tiempo de ejecución

El paradigma de la ingeniería dirigida por modelos utiliza los modelos y los procesos que los gestionan para transformarlos en tiempo de diseño y en tiempo de desarrollo. Aunque en los últimos años ha surgido una nueva corriente para la aplicación de las técnicas y la metodología MDE al contexto en tiempo de ejecución [Blair et al., 2009]. El propósito de los modelos que se gestionan en tiempo de ejecución es el mismo que el de los modelos en tiempo de diseño o en tiempo de desarrollo, aunque hay diferentes consideraciones que se deben tener en cuenta. Por ejemplo, las dimensiones de la utilización de recursos, de la eficiencia y del rendimiento, suelen estar presentes en las estrategias que manipulan los modelos. La información y dependencia respecto al contexto es otro elemento que puede jugar un papel importante. De esta forma, las características relacionadas con la personalización o la adecuación de los modelos a las preferencias de los usuarios o a su tipo de perfil, también están relacionadas con los modelos en tiempo de ejecución.

Que un modelo tenga que ser manipulado en tiempo de ejecución suele implicar que la información que el modelo representa está cambiando a lo largo del tiempo, y que por tanto, su gestión debe ser asumida de manera automática por el sistema software. Por eso, el concepto de modelo en tiempo de ejecución se relaciona con la adaptación dinámica de aquello que está siendo representado por dichos modelos. Uno de los dominios de aplicación en la literatura es la adaptación de sistemas en respuesta a cambios del contexto, del entorno o de los requisitos [Cheng et al., 2009]. De esta forma, se puede separar el proceso de adaptación, de aquellos elementos que se necesitan adaptar y que se pueden representar por los modelos en tiempo de ejecución [Salehie and Tahvildari, 2009].

Existen propuestas que utilizan los modelos en tiempo de ejecución para llevar a cabo procesos de adaptación, como en [Garlan and Schmerl, 2004] [Morin et al., 2009] [Blair et al., 2009] [Derakhshanmanesh et al., 2014]. Hay otras propuestas, como la descrita en [Yuan et al., 2014] que utilizan la información obtenida de la interacción de los componentes del sistema para crear un modelo probabilista que dé la posibilidad de crear nuevas operaciones de adaptación; o la propuesta de [Heinrich et al., 2015], donde se muestra un programa para adaptar y evolucionar (en tiempo de ejecución) sistemas

basados en la nube por medio de la observación y el análisis de características relacionadas con la calidad. En esa propuesta, las técnicas de Ingeniería de Modelos se usan para diferentes objetivos, entre los que están la monitorización de la información y las transformaciones de modelos. Utilizar modelos en tiempo de ejecución favorece la aparición de nuevas tendencias para manipular los modelos. De esta forma, también hay una consecuencia en la mejora y actualización de algunas de las herramientas relacionadas directamente con la ingeniería de modelos, como puede ser EMF. Las versiones actuales de estas librerías permiten su uso en tiempo de ejecución.

En el presente trabajo de tesis doctoral se utilizan los modelos en tiempo de ejecución. Los elementos principales que se definen a través de modelos y que son manejados en tiempo de ejecución son las arquitecturas de componentes, que describen las aplicaciones *mashup*. Además, los componentes también están representados mediante modelos, y son almacenados en un repositorio que también se gestiona en tiempo de ejecución. Por último, estos modelos también dan soporte para poder llevar a cabo procesos de persistencia y comunicación entre los componentes que forman parte de las aplicaciones, todo ello, como veremos en los próximos capítulos.

1.5. Ingeniería de servicios

Por último, otro de los pilares fundamentales que soporta la metodología propuesta en esta tesis doctoral es la *Ingeniería de Servicios*. El uso de servicios consigue aportar interoperabilidad entre las aplicaciones software, independientemente de las características de las plataformas sobre las que fueron desplegadas. Además, facilitan el acceso a la información, pues los servicios fomentan el uso de estándares y protocolos de comunicación. Por norma general, los servicios tratan de hacer uso de estándares para lograr una mayor integridad entre los elementos software que los componen.

En el trabajo de tesis se usan servicios web para ofrecer un conjunto de operaciones a las aplicaciones *mashup*, y poder construir un entorno más accesible desde cualquier lugar y para cualquier dispositivo. En las siguientes subsecciones se definen los principales tipos de servicios web. En primer lugar, se realiza una revisión de los servicios web tradicionales basados en SOAP, que son los servicios desplegados por la infraestructura definida. Posteriormente, se describen los servicios RESTful, que son una alternativa a los servicios web SOAP tradicionales. Para finalizar la sección, se introducen los microservicios, un concepto de servicio que ha surgido recientemente en la literatura.

1.5.1. Servicios web SOAP

Un servicio web SOAP es una interfaz que describe un conjunto de operaciones a las cuáles se tiene acceso mediante un mensaje XML [Kreger et al., 2001] [Papazoglou, 2008] [Alonso et al., 2010]. Una descripción de un servicio ofrece detalles sobre cómo hay que hacer uso del servicio, cómo debe ser el formato de los mensajes que consumen las operaciones, cómo deben ser los protocolos de transporte y cómo deben ser los protocolos de comunicación con el servicio web. La interfaz consigue ocultar detalles sobre cómo ha sido implementado el servicio, permitiendo a los programadores usarlo, inde-

© 2016 Vallecillos, J.

pendientemente de la plataforma hardware y software en la cual se ha implementado, e independientemente del lenguaje de programación en el que se construyó. Esta forma de desplegar funcionalidad y recursos computacionales ha sido utilizada en este trabajo de investigación de tesis doctoral, ya que es idónea para realizar el despliegue de aplicaciones construidas en base a componentes de terceros (componentes comerciales o COTS), como lo son las arquitecturas de componentes que forman las aplicaciones *mashup*.

La arquitectura de servicios web se basa en la interacción de tres tipos de roles (muy parecido al modelo ORB de CORBA visto anteriormente): *proveedores* de servicios, *registro* de servicios y *consumidores* de servicios, como se puede ver en la Figura 1.8 (inspirada en [Curbera, 2001]). Estos roles actúan sobre los servicios web haciendo uso del conjunto de operaciones. En un escenario típico de servicio web, un proveedor de servicio alberga un conjunto de funcionalidades o de capacidades, que son accesibles a través de la red. El proveedor del servicio define una descripción de servicio para el servicio web, haciéndolo público para el resto de los consumidores y para el objeto de Registro de servicios. Cuando un consumidor de servicios quiere hacer uso de una operación, obtiene la localización del servicio donde está la operación. Para obtener esta localización, el consumidor de un servicio se comunica con el objeto que juega el papel de “registro de servicios”, y a partir de la información que éste le devuelve, el consumidor se conecta con el proveedor del servicio invocando las operaciones del servicio web que desea consumir en ese determinado instante.

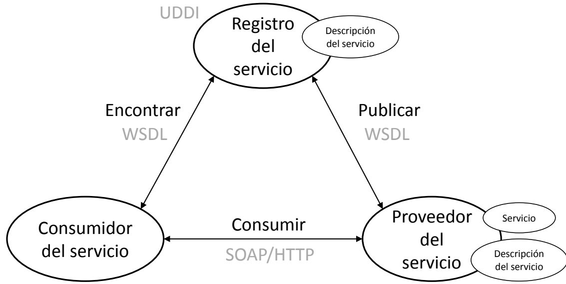


Figura 1.8: Roles de servicio web y operaciones

Los servicios web se estructuran mediante una arquitectura de capas. Para comprender cada uno de los elementos que forman la arquitectura y conocer su ubicación dentro de ella, se muestra una imagen con una pila conceptual (Figura 1.9 inspirada en [Curbera, 2001]). Para comunicar cada una de las capas se hace uso de estándares. Las capas más altas se sirven de las capas más bajas. El texto de la izquierda de la figura, representa la tecnología estándar que se aplica en cada capa de la pila.

La capa más baja de la arquitectura es la *capa de red*. Los servicios web deben estar en localizaciones que sean accesibles para poder ser invocados por los consumidores de servicios. Los servicios web públicos usan protocolos de red que son estándares, como por ejemplo, el protocolo HTTP, o los protocolos de Internet, como SMTP o FTP, entre otros. La siguiente capa es la *capa de mensajería*, basada en XML. El protocolo que se utiliza en esta capa es SOAP debido a que:

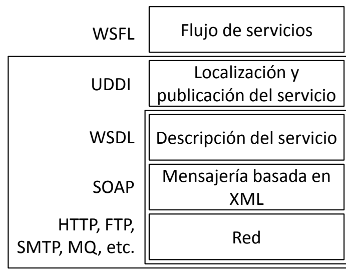


Figura 1.9: Pila conceptual de servicios web

(a) es el mecanismo envolvente estándar para la comunicación de llamadas a procedimientos remotos y mensajes que contienen documentos de tipo XML;

(b) es simple, básicamente un mensaje POST HTTP que contiene XML; y

(c) los mensajes SOAP dan soporte a la publicación y localización de las operaciones en la arquitectura de servicios web.

Por otro lado, la *capa de descripción del servicio* hace uso del estándar WSDL (*Web Services Description Language*) para describir cómo se debe consumir el servicio web. Por medio de WSDL se define cómo es la interfaz y cómo son los mecanismos de interacción con el servicio. Para describir el servicio a este nivel, también se necesita una descripción adicional para dar detalles sobre el contexto de negocio y relaciones servicio a servicio. Estas descripciones de servicios definidas en WSDL son almacenadas en un servicio de registro que permite la publicación y localización de servicios en la red. Este servicio de registro se puede definir a través de UDDI (*Universal Description, Discovery and Integration*). Por último, la composición de servicios y el flujo de información se describen a través de un documento WSFL (*Web Services Flow Language*). En resumen, se puede acceder a un servicio web que está disponible en la red por medio del protocolo SOAP y se representa por una descripción de servicio.

A modo de ejemplo, supóngase el servicio web descrito por el documento WSDL de la Tabla 1.5. El servicio web se denomina *SessionWSImplService*, cuyo nombre queda establecido en el elemento `wsdl:service` del documento WSDL. El servicio contiene una operación llamada `login` visible dentro del elemento `wsdl:operation`. Esta operación tiene como entrada el tipo complejo (*complexType*) `login` y como devolución el tipo complejo `loginResponse`. El tipo `login` contiene a su vez el tipo complejo `loginSessionParams` el cual está formado por un elemento llamado *userName* y *userPassword*. Por otro lado, la operación `login` devuelve `loginSessionResult`, el cual contiene un elemento llamado `validation`, un elemento `userID` y otro elemento llamado `message`. Para comunicarse con la operación `login` del servicio web es necesario enviarle un mensaje SOAP como el descrito en la Tabla 1.6. Como se puede observar, los parámetros que contiene el mensaje son `userName` y `userPassword`. Para finalizar, la operación responde con otro mensaje SOAP visible en la Tabla 1.7. Este mensaje de respuesta está formado por un campo `validation`, `userID` y `message`.

© 2016 Vallecillos, J.

```
<wsdl:definitions xmlns:xsd="..." xmlns:wsdl name="SessionWSImplService" targetNamespace="...">
 <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" ... version="1.0">
     <xsd:element name="login" type="tns:login"/>
     <xsd:complexType name="login">
        <xsd:sequence> <xsd:element name="params" type="tns:loginSessionParams"/> </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="loginSessionParams">
        <xsd:sequence>
            <xsd:element name="userName" type="xsd:string"/>
            <xsd:element name="userPassword" type="xsd:string"/>
        </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="loginResponse">
        <xsd:sequence> <xsd:element name="result" type="tns:loginSessionResult"/> </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="loginSessionResult">
        <xsd:sequence>
            <xsd:element name="validation" type="xsd:boolean"/>
            <xsd:element name="userID" type="xsd:string"/>
            <xsd:element name="message" type="xsd:string"/>
        </xsd:sequence>
     </xsd:complexType>
    </xsd:schema>
 </wsdl:types>
 <wsdl:message name="login"> <wsdl:part element="tns:login" name="parameters"> ... </wsdl:part>
 </wsdl:message>
 <wsdl:portType name="SessionWS">
    <wsdl:operation name="login">
        <wsdl:input message="tns:login" name="login"></wsdl:input>
        <wsdl:output message="tns:loginResponse" name="loginResponse"> ... </wsdl:output>
    </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="SessionWSImplServiceSoapBinding" type="tns:SessionWS"> ... </wsdl:binding>
 <wsdl:service name="SessionWSImplService">...</wsdl:service>
</wsdl:definitions>
```

Tabla 1.5: Estructura de un servicio web a través de su WSDL

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:ws="...">
  <soapenv:Body>
    <ws:login>
      <params>
        <userName> "ejemplo nombre" </userName> <userPassword> "ejemplo password" </userPassword>
      </params>
    </ws:login>
 </soapenv:Body>
</soapenv:Envelope>
```

Tabla 1.6: Mensaje SOAP de entrada a la operación `login`

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <ns2:loginResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
    <result>
       <validation> "ejemplo true o false" </validation> <userID> "ejemplo ID o -1" </userID>
       <message> "ejemplo mensaje de éxito o de error"</message>
    </result>
  </ns2:loginResponse>
 </soap:Body>
</soap:Envelope>
```

Tabla 1.7: Mensaje SOAP de respuesta de la operación `login`

1.5.2. Servicios web RESTful

Los servicios web RESTful se han diseñado siguiendo el estilo de diseño *REpresentational State Transfer* (REST) [Belqasmi et al., 2011]. REST es un estilo de Arquitectura Software Orientada a recursos (*Resource-Oriented Architecture*, ROA) para sistemas hipermedia distribuidos. Una arquitectura diseñada bajo el modelo REST proporciona un conjunto de reglas y una serie de procedimientos paso a paso para desarrollar una tarea. Los servicios web RESTful pueden ser descritos usando el lenguaje WADL (*Web Application Description Language*) [W3C, 2009].

Los servicios RESTful se basan en tres características principales: direccionamiento, interfaces uniformes y la carencia de estados [Richardson and Ruby, 2008]. Para el direccionamiento, REST modela el conjunto de datos que son ofrecidos como recursos, e identifica cada recurso a través de una referencia URI (*Uniform Resource Identifier*). Un recurso es un tipo de información al que se puede tener acceso y hacer referencia, como por ejemplo, un documento, un registro de una base de datos o un resultado de búsqueda. Los recursos que son ofrecidos por medio de REST tienen la característica de que son accesibles por medio de una interfaz uniforme y estándar. Una interfaz uniforme ofrece ciertas ventajas, como la facilidad de uso y la interoperabilidad. Los servicios RESTful no guardan su estado ya que cada solicitud sobre el servicio contiene toda la información necesaria para el servidor. Tampoco se guarda en el servidor los datos de sesión de los clientes; es decir, el servidor nunca se basa en información previa para responder ante una nueva solicitud. Todo esto facilita el desarrollo de aplicaciones simples y escalables que permiten realizar de forma sencilla un balanceo de la carga computacional.

Tanto REST como ROA soportan diferentes formatos de comunicación, incluyendo texto plano, HTML, XML, y JSON (*JavaScript Object Notation*). ROA usa HTTP como protocolo de comunicación. Así, la interfaz uniforme ROA está formada por operaciones HTTP, y dentro del conjunto de operaciones HTTP se usan GET, PUT, POST y DELETE. Se puede diseñar un servicio web RESTful a partir de ROA siguiendo los siguientes pasos. En primer lugar hay que definir el conjunto de datos sobre el cual el servicio debe funcionar, y después llevar a cabo una división de los recursos. Tras ello, para cada recurso se procede de la siguiente manera: (a) primero, se nombra el recurso utilizando una URI; (b) a continuación, se identifica el tipo de interfaz específica que se ofrece a través del servicio; (c) seguidamente, se diseña cómo se representará el recurso que será solicitado luego por los clientes, y cómo se enviará dicho recurso al cliente; (d) y finalmente, se define cómo será el comportamiento del servicio y qué ocurre durante una ejecución satisfactoria del mismo.

En nuestro caso, para el trabajo de investigación desarrollado en esta tesis doctoral, todos los servicios web de la infraestructura propuesta han sido desarrollados como servicios web SOAP, tal y como se describirá en los Capítulos 3 y 4. Se ha optado por utilizar la perspectiva tradicional basada en estándares para el desarrollo de los servicios web de dicha infraestructura, motivo por el cual los servicios siguen los estándares SOAP. No obstante, nuestras aplicaciones *mashup*, y como consecuencia, cada uno de sus componentes, utilizan servicios de terceras partes que pueden estar implementados como servicios web RESTful, como es el caso de los servicios que permiten acceder al repositorio de componentes tipo Web.

© 2016 Vallecillos, J.

1.5.3. Microservicios

En las aplicaciones actuales hay una tendencia al desarrollo orientado a servicios, ya que estos aportan interoperabilidad, fomentan el uso de estándares y de protocolos basados en texto, además de aportar un acceso fácil a la funcionalidad. Recientemente, ha surgido una nueva tendencia en este ámbito basada en el uso de "microservicios".

Para comprender mejor el significado de un *microservicio*, veamos antes algunas definiciones. En [Thones, 2015] se define un microservicio como una pequeña aplicación que se puede desplegar de forma independiente, que es escalable y que puede ser usada de manera aislada. Cada microservicio contiene una funcionalidad simple y fácil de comprender. Esto implica que un microservicio pueda ser modificado o reemplazado por otro de manera sencilla. En otros trabajos como en [Namiot and Sneps-Sneppe, 2014] se describe el concepto de "arquitectura de microservicios" como una infraestructura para desarrollar aplicaciones software a través de una serie de pequeños servicios totalmente independientes. Dentro de esta arquitectura software, los microservicios no funcionan aisladamente sino que pueden comunicarse entre sí para cumplir un objetivo común, funcionando como un sistema. En [Uckelmann et al., 2011] se usa el protocolo HTTP como una de las formas principales de comunicación entre microservicios.

Un microservicio debe gestionar un mínimo de funcionalidad que suele ser utilizada por algún otro microservicio o una aplicación cliente. Los microservicios que forman parte de la arquitectura, pueden escribirse en lenguajes de programación diferentes, ya que los procesos de entrada y salida de información se realizan a través de la interfaz del propio microservicio. Por tanto, según lo comentado hasta el momento, se puede decir que un microservicio sólo tiene las operaciones individuales que aparecen dentro de un servicio web. De tal forma que estas operaciones son las que se ejecutan dentro del microservicio y que se comunican entre ellas para llevar a cabo tareas de forma conjunta.

Si se examina la Figura 1.10 se observa un ejemplo de una arquitectura basada en microservicios. Cada área funcional mostrada en la figura se implementa como un mi-

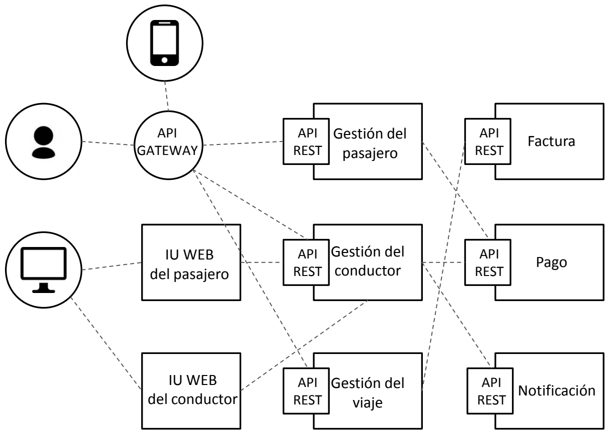


Figura 1.10: Ejemplo de arquitectura de microservicios

croservicio, y se encuentra representada por un recuadro en dicha figura. De esta forma, cada aplicación web se divide en un conjunto de aplicaciones web más simples (como son las IU del pasajero y del conductor). Así, es más fácil desplegar distintas aplicaciones para cada usuario específico, teniendo también en cuenta cada dispositivo. En este caso, para llevar a cabo la comunicación con cada servicio se usan servicios REST. Como se puede ver, para la "Gestión del conductor" se usan las notificaciones del servidor para informar sobre conductores disponibles para un viaje potencial. Los servicios pueden ser asíncronos, basando la comunicación en mensajes. Las APIs REST también están disponibles para la aplicación móvil que usan los conductores y los pasajeros. Sin embargo, la aplicación no tiene acceso directo a lo que hay tras servicios. La comunicación se lleva a cabo a través de la "API Gateway" que se encarga de tareas tales como el balanceo de la carga, el control de acceso y la monitorización.

Respecto al hecho de usar microservicios frente a los servicios tradicionales en el desarrollo de arquitecturas, existen ciertas *ventajas* [Thones, 2015]. En primer lugar, el despliegue es más sencillo si se compara con una solución monolítica. Al construir una arquitectura por partes, es más sencillo escalar el sistema, pudiendo añadir o quitar elementos en el futuro, si fuera necesario. Además, se permite explotar más aún el balanceo de carga computacional. Cuando se produce algún problema dentro de la arquitectura, es más simple localizar en qué parte se produjo el problema, por el propio desglose de la arquitectura. Otra ventaja de las arquitecturas basadas en microservicios es que son más sencillas de entender cómo funcionan, por el hecho de estar construidas en base a fragmentos funcionales (servicios sencillos). También, esta forma de construcción basada en trozos funcionales hace más fácil la realización de modificaciones de la arquitectura, ya que se llega a tener una comprensión más rápida de la funcionalidad global del sistema. Además, un desarrollo basado en fragmentos facilita la incorporación de nuevos desarrolladores a un proyecto durante el proceso de desarrollo. Por estos motivos, la tendencia es a construir aplicaciones a partir del desarrollo de colecciones de pequeños servicios (pequeñas funcionalidades) que pueden ser fácilmente ensamblados.

Por contra, una arquitectura basada en microservicios también tiene una serie de *inconvenientes*. Por ejemplo, la división del sistema. En ocasiones es complicado saber cómo se realiza la división del sistema, en qué partes se desglosa, y cuáles de ellas se pueden llevar a implementar como servicios independientes. Otro inconveniente es que en una arquitectura de microservicios es complejo realizar cambios sobre un microservicio usado por otros elementos de la arquitectura. Por ejemplo, cuando se requieren cambios en los servicios A, B y C, donde A depende de B y B depende de C. En una arquitectura monolítica se puede cambiar de manera sencilla, integrar los cambios y desplegar el sistema. Pero en un patrón de arquitectura basada en microservicios, se necesita planificar y coordinar los cambios en cada servicio. Es decir, en este caso se necesita actualizar el servicio A, después el servicio B y por último el servicio C. Llevar a cabo también el despliegue de una aplicación basada en microservicios es más compleja. En una arquitectura monolítica es muy fácil desplegar los servicios, estando todos en un mismo servidor, donde se hace además un balanceo de carga tradicional. En una arquitectura basada en microservicios cada instancia tiene su propia ubicación dentro de la infraestructura de servicios, teniendo que controlar las bases de datos que maneja cada microservicio y la gestión de mensajes entre los microservicios dentro de la arquitectura.

© 2016 Vallecillos, J.

1.6. Resumen y conclusiones

En este primer capítulo se han presentado cada uno de los pilares principales sobre los cuáles se basa este trabajo de tesis doctoral: las aplicaciones *mashup*, la Ingeniería del Software Basada en Componentes, la Ingeniería de Modelos y la Ingeniería de Servicios. La combinación de estos bloques dan la posibilidad de desarrollar la infraestructura basada en servicios propuesta en esta tesis para el despliegue de aplicaciones *mashup*, la cual se presenta en los siguientes capítulos de esta memoria.

La primera parte de este capítulo describió las aplicaciones *mashup* a las cuales se les da soporte a través de la infraestructura. Un aspecto importante en esta descripción es la utilización de los componentes *mashup* como pieza unitaria, reutilizable y accesible de forma local y remota, que permite llevar a cabo la construcción de estas aplicaciones.

En la segunda parte se presentó la Ingeniería del Software Basada en Componentes. En este bloque se han revisado las bases y los conceptos de esta ingeniería, con el objetivo de poner de manifiesto las ventajas que aportan en el desarrollo de aplicaciones a través del ensamblaje de partes más pequeñas y de la reutilización de componentes. Además, se han visto diferentes opciones para describir (o especificar) los componentes.

Posteriormente, se han revisado los elementos y los mecanismos más utilizados de la Ingeniería de Modelos y se ha justificado cómo estas técnicas pueden ser utilizadas para la representación de las aplicaciones *mashup* y los componentes que las forman. Un aspecto fundamental que se ha tenido en cuenta es que estos elementos pueden ser tratados en tiempo de ejecución.

Finalmente, se ha revisado la Ingeniería de Servicios mostrando los distintos tipos de servicios existentes y destacando los servicios web SOAP, por ser la opción utilizada en el desarrollo de este trabajo de investigación, y los microservicios, puesto que suponen una estructura similar a la infraestructura propuesta.

Capítulo 2

COScore: modelo de infraestructura

© 2016 Vallecillos, J.

Capítulo 2

COSCORE: MODELO DE INFRAESTRUCTURA

Contenidos

2.1. Introducción y conceptos relacionados 40
2.2. Capa cliente 44
2.3. Capa dependiente de la plataforma 46
2.3.1. Gestión de componentes propios 49
2.3.2. Gestión de componentes de repositorios externos 52
2.4. Capa independiente de la plataforma 54
2.4.1. Arquitecturas de las aplicaciones *mashup* 54
2.4.2. Especificación de componentes 57
2.5. Escenarios de ejemplo base 60
2.5.1. Escenario de mapas temáticos 61
2.5.2. Escenario de domótica 62
2.6. Dependencias entre componentes *mashup* 63
2.6.1. Espacio de trabajo y componentes 63
2.6.2. Patrones de composición de componentes 69
2.6.3. Matriz de regeneración de dependencias 75
2.7. Relaciones entre componentes *mashup* 76
2.7.1. Relaciones binarias 77
2.7.2. Relaciones n-arias 79
2.8. Trabajo relacionado 81
2.9. Resumen y conclusiones 83

En la sociedad actual donde todo evoluciona a gran velocidad, las aplicaciones software deben también adecuarse a estos continuos cambios, adaptando su estructura y/o funcionamiento. Esto sucede, por ejemplo, en las aplicaciones *mashup* (dominio de aplicación de esta investigación) que pueden cambiar en el tiempo por estar constituidas y construidas en base a "piezas software" (*i.e.*, componentes), permitiendo con ello crear nuevos entornos de manera sencilla y rápida para adaptarse en tiempo de ejecución a las necesidades de los usuarios.

Con este propósito, en el trabajo de investigación se ha desarrollado una infraestructura basada en servicios denominada COScore (*COTSget-based architecture Operating Support core*) para permitir el despliegue de aplicaciones *mashup*. Dicha infraestructura ha sido implementada mediante una arquitectura de *microservicios* y siguiendo un *modelo a tres capas* para facilitar con ello la propiedades de separación, modularidad, escalabilidad e interoperabilidad, como requiere toda aplicación basada en servicios, como lo es la propuesta COScore presentada en esta tesis doctoral. Para permitir estas propiedades, en la propuesta:

(a) se define una *capa cliente* donde se despliegan las aplicaciones *mashup*,

(b) una *capa dependiente* de la plataforma que contiene las operaciones específicas de cada tipo de dispositivo, y por último,

(c) una *capa independiente* de la plataforma donde se definen las operaciones comunes a todo tipo de aplicación *mashup* soportada por esta infraestructura.

El capítulo se estructura en nueve secciones. La Sección 2.1 analiza las distintas partes que forman parte de la infraestructura encargada de dar soporte a las aplicaciones *mashup*. A continuación, la Sección 2.2 muestra la primera parte por la cual está formada la infraestructura propuesta (*i.e.*, la capa cliente), describiendo cómo se despliegan las aplicaciones *mashup* en esta capa. En la Sección 2.3 se detalla la capa dependiente de la plataforma, la cual permite conectar cada aplicación *mashup* con los servicios ofrecidos por la capa independiente de la plataforma. En esta sección se incluye también una descripción de los repositorios de componentes que se utilizan para el despliegue de las aplicaciones. En la Sección 2.4 se expone la capa independiente de la plataforma, parte central del trabajo de tesis doctoral. Como parte de la explicación de este nivel de la infraestructura, se describen las partes por las cuales está formada esta capa, analizando los repositorios y las especificaciones que alberga. Se continúa con la Sección 2.5, donde se exponen dos escenarios básicos de aplicación de la infraestructura propuesta. Estos escenarios se utilizarán para definir los tipos de dependencias existentes entre los componentes. La Sección 2.6 establece los principios para resolver las dependencias entre los componentes que forman parte de las aplicaciones y, posteriormente, la Sección 2.7 describe cómo resolver dichas dependencias a través de la definición de relaciones entre los componentes. Para finalizar, en la Sección 2.8 se discuten algunos de los trabajos relacionados con modelos de infraestructura similares como el que aquí se trata. El capítulo finaliza con un resumen y con las conclusiones extraídas.

© 2016 Vallecillos, J.

2.1. Introducción y conceptos relacionados

En primer lugar, para ayudar en la exposición de la infraestructura desarrollada en este trabajo de investigación, se va a partir de la interfaz de usuario mostrada en la Figura 2.1. En la parte izquierda de dicha figura, podemos ver una interfaz *mashup* correspondiente a la plataforma Netvibes[1] formada por varias "piezas", cada una de las cuales se utiliza con una finalidad distinta. En este caso, la interfaz está formada por (a) un componente de cabecera con el menú, (b) un componente del tiempo que muestra información meteorológica, (c) un componente para realizar búsquedas en la Web, (d) un componente para la gestión de una red social de fotos, (e) un componente para la gestión del correo electrónico, etc. Estos son algunos ejemplos de componentes que pueden ser incluidos en la interfaz, la cual puede ser reconfigurada (insertando nuevos componentes, eliminando componentes innecesarios o modificando las propiedades de componentes existentes) dependiendo del usuario que está haciendo uso de ella o del propósito para el cual ha sido "creada".

Inspirándonos en esta idea de interfaz *mashup* que puede ser reconfigurada dependiendo de las preferencias de los usuarios, el trabajo de investigación desarrollado en esta tesis doctoral propone una infraestructura para el despliegue de interfaces *mashup* que, además, permite llevar a cabo la gestión de dichas interfaces y de las operaciones de reconfiguración en tiempo de ejecución. No obstante, es necesario remarcar que esta infraestructura no está orientada al soporte de cualquier tipo de aplicaciones *mashup*, sino que únicamente es válida para aplicaciones que presenten las siguientes características:

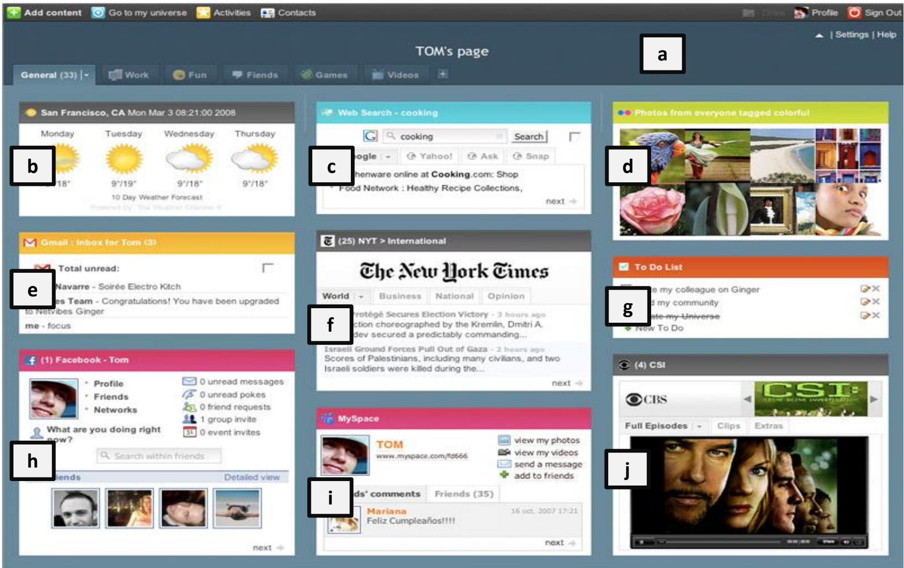


Figura 2.1: Interfaz *mashup* de Netvibes de ejemplo

[1]Netvibes – `https://www.netvibes.com/`

— Las aplicaciones deben poder ser descritas a través de la construcción de **arquitecturas** de componentes.

— Puesto que son parte de un arquitectura, los componentes de las aplicaciones no son independientes entre sí, sino que pueden presentar **dependencias** entre ellos.

— Las dependencias entre componentes deben poder resolverse mediante un proceso de comunicación basado en **envío de mensajes** de forma **asíncrona**.

— Los componentes que conforman estas aplicaciones *mashup* son componentes de **granularidad gruesa** que encapsulan una funcionalidad de cierta complejidad. A modo de ejemplo, en el dominio de interfaces de usuario, se trata de componentes tipo *widget* que permiten llevar a cabo una tarea, y no se trata de simples botones, etiquetas o campos de entrada de texto.

— Las arquitecturas que definen estas aplicaciones son **diferentes** dependiendo del **usuario** que vaya a hacer uso de ellas y de la **tarea** que se esté realizando.

— Las aplicaciones pueden ser **modificadas en tiempo de ejecución** para ser adaptadas a las diferentes **situaciones** de uso y para que puedan adaptarse a las **preferencias** de los usuarios.

Un ejemplo de este tipo de aplicaciones *mashup* son las interfaces de usuario dinámicas de la propuesta presentada en [Criado, 2015], trabajo que ha servido de base para esta tesis doctoral. De esta manera, el presente trabajo de investigación ofrece una infraestructura que da soporte a este tipo de interfaces de usuario *mashup* (construidas a partir de componentes web de tipo *widget*) pero que, además, también permite desplegar otros tipos de aplicaciones *mashup* que cumplan con los principios enumerados anteriormente. Para ello, esta infraestructura ha sido también validada en el dominio de la domótica, permitiendo desplegar aplicaciones *mashup* implementadas como arquitecturas de componentes Java que simulan instalaciones sencillas de hogar digital.

Tal y como se ha comentado, las aplicaciones *mashup* gestionadas por la infraestructura están representadas por medio de una arquitectura de componentes. Para ello, es necesario realizar un proceso de abstracción que permita describir cada una de las piezas de dicha arquitectura (*i.e.*, componentes) y cómo están relacionadas entre sí, pero que oculte el resto de detalles de implementación que no son necesarios para la utilización de los componentes, ni para la ejecución o interpretación de las arquitecturas. En nuestro caso, se propone hacer uso de modelos para llevar a cabo dicho proceso de abstracción, de manera que sea posible trabajar con estas representaciones abstractas (resolviendo los caminos de comunicación, modificando su estructura, etc.) hasta el momento del despliegue, en el que habrá que hacer uso de los componentes "reales" para construir las aplicaciones *mashup* que se ofrecen como resultado.

Además, la infraestructura propuesta permite la construcción de aplicaciones *mashup* a partir tanto de **repositorios propios** de componentes (almacenados y gestionados de forma local), como de **repositorios externos** (desarrollados por terceras partes y accesibles de forma remota), tal y como se muestra en la Figura 2.2. Para ello, todos los componentes deben estar descritos por una especificación que permita inspeccionar

© 2016 Vallecillos, J.

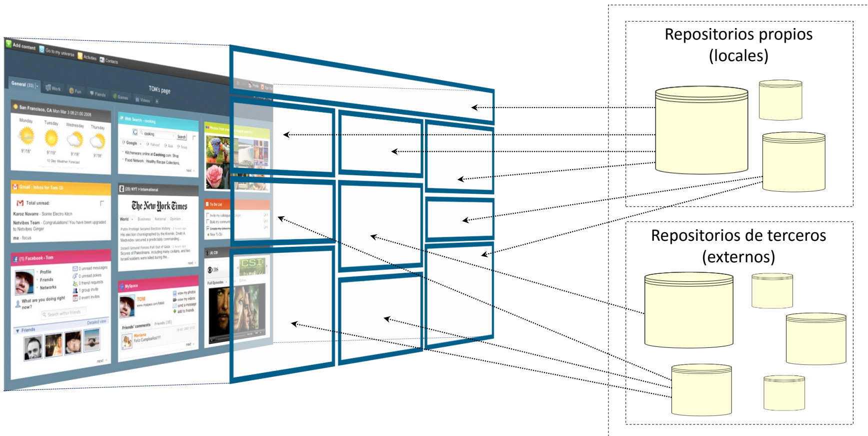


Figura 2.2: Uso de repositorios de componentes para la construcción de las interfaces

los repositorios disponibles para llevar a cabo una búsqueda de los componentes que se ajusten mejor a la arquitectura definida. Además, estas especificaciones también aportan información necesaria para el despliegue, lo cual se consigue a través de la definición de sus interfaces funcionales y de parte de información extra-funcional que también debe ser tenida en cuenta durante la ejecución y que se encuentra descrita en sus propiedades extra-funcionales (en forma de restricciones, calidades y otra información adicional). Los componentes utilizados en la arquitectura, reciben el nombre de componentes COTSget, término que procede de la combinación componentes COTS (*Commercial Off-The-Self*) y *gadget*, entendiendo como tal "un artefacto software que encapsula la funcionalidad necesaria para llevar a cabo una tarea".

Independientemente del dominio, las aplicaciones *mashup* deben ser gestionadas de forma similar, puesto que la mayoría de las operaciones (por ejemplo, añadir un nuevo componente en la arquitectura, conectar dos componentes para que se comuniquen, o crear una copia de una arquitectura por defecto para un nuevo usuario) van a ser realizas sobre su representación abstracta (modelo) sin tener en cuenta la plataforma en la que se despliegan las aplicaciones. No obstante, aquellas operaciones que dependen de la plataforma (como por ejemplo, crear una instancia de un componente, obtener los objetos o el código correspondiente para el despliegue de las aplicaciones, o dar de alta un nuevo componente en el repositorio) son gestionadas de manera específica. Para ello, la infraestructura dispone del mismo tipo de operación. Teniendo en cuenta estas características, la infraestructura propuesta se estructura en tres capas: capa cliente, capa servidor dependiente de la plataforma y capa independiente de la plataforma. En la capa cliente se realiza el despliegue de las aplicaciones *mashup*, mientras que las otras dos capas que se encuentran en el servidor conforman el núcleo (COScore, *COTSget-based architecture Operating Support core*) de la insfraestructura (ver Figura 2.3).

La **capa cliente** es la capa más alta de la infraestructura definida. Desde el lado cliente, se hace uso de los servicios que la infraestructura despliega a través de solicitudes que son gestionadas por la capa dependiente de la plataforma. Algunos ejemplos de la

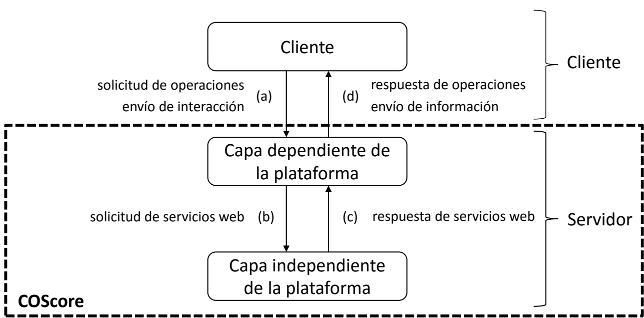


Figura 2.3: Capas de la infraestructura

funcionalidad que se resuelve a través de estas solicitudes son los procesos de inicialización, los procesos de comunicación entre elementos de la capa cliente (*i.e.*, entre los componentes de la aplicación), o dar soporte de persistencia a las aplicaciones.

La **capa dependiente de la plataforma** constituye la capa intermedia de la infraestructura. Como se ha mencionado, esta capa se encargar de recibir la interacción llevada a cabo en el cliente y de procesar las solicitudes de aquellas operaciones a las que se da soporte (Figura 2.3(a)). Para poder dar respuesta a dichas solicitudes e informar al sistema de la interacción realizada, esta capa intermedia se encarga de invocar los servicios web correspondientes proporcionados por la capa independiente de la plataforma (Figura 2.3(b)). La respuesta de los servicios web invocados es devuelta a la capa dependiente de la plataforma (Figura 2.3(c)), la cual se encarga de propagar la respuesta hacia el cliente (Figura 2.3(c)), enviándole la información necesaria para la actualización de las aplicaciones *mashup* (en el caso de que fuera necesario).

La **capa independiente de la plataforma** es la capa más baja de la infraestructura. Como se ha introducido previamente, esta capa contiene las operaciones comunes para todos los tipos de aplicaciones *mashup* y aquellas operaciones específicas para cada una de las plataformas a las que se da soporte. De esta manera, una aplicación *mashup* de tipo web requiere un soporte distinto al que requiere una aplicación de tipo Java. Por ejemplo, la construcción inicial de ambos tipos de aplicación es distinta, debido a que las operaciones de consulta de los repositorios e instanciación los componentes son diferentes. Sin embargo, a pesar de las diferencias en la implementación de dichas operaciones, el tipo de operación que hay que realizar es el mismo, y por ello, todas las operaciones se resuelven en la capa independiente de la plataforma.

El hecho de que la resolución de toda la funcionalidad se lleve a cabo en la capa independiente de la plataforma es un aspecto que ha sido tenido en cuenta para que no suponga un cuello de botella en la infraestructura. En nuestro caso, se ha optado por realizar un balanceo de carga entre distintos servidores que pertenecen a esta capa y que ofrecen todos los servicios contemplados en la infraestructura. Además, la capa independiente de la plataforma ha sido desarrollada de forma modular y escalable, pudiendo añadir gradualmente funcionalidad a cada nueva plataforma para la que se de soporte. En las siguientes secciones del capítulo se describen cada una de las capas que forman parte de la infraestructura propuesta.

© 2016 Vallecillos, J.

2.2. Capa cliente

La capa cliente de la insfraestructura es la encargada de alojar y ejecutar el software correspondiente a las aplicaciones *mashup*. Aunque esta capa no constituye en sí una aportación del trabajo de investigación realizado y, por tanto, no se incluye en el núcleo (COScore) de la propuesta, es necesario describirla como parte de la infraestructura, destacando el papel que juega dentro del despliegue de nuestras aplicaciones. Veamos a continuación dos tipos de capas cliente relacionadas con los dos escenarios de ejemplo mencionados y que sirven para la validación y evaluación de la infraestructura. En el caso de aplicaciones *mashup* de tipo web, la capa cliente consiste básicamente en el código HTML (junto con los archivos CSS y JS) que se ejecuta en un navegador web. Para el caso de aplicaciones *mashup* de tipo Java, la capa cliente está formada por los objetos Java que se ejecutan en un servidor de aplicaciones o servidor web que realiza las funciones de cliente de la infraestructura.

Como parte de cada aplicación *mashup*, la capa cliente contiene los componentes que forman parte de su arquitectura. No obstante, cuando una aplicación *mashup* inicia su despliegue, no tiene información acerca de cuáles son los componentes que forman parte de dicha arquitectura. Por el contrario, la aplicación únicamente aporta información sobre el usuario y la plataforma sobre la que se está desplegando (Figura 2.4, paso 1). Una vez resuelta la operación de inicialización de la aplicación *mashup* que se le mostrará al usuario, la capa cliente recibe el conjunto de componentes que conforman la arquitectura que define dicha aplicación (Figura 2.4, paso 2). Por último, en la capa cliente se construye la aplicación *mashup*, realizando un despliegue de los componentes y configurando sus propiedades (Figura 2.4, paso 3). De forma ilustrativa, algunas de las propiedades que se configuran para el caso de aplicaciones *mashup* tipo web son el ancho, alto y posición de cada componente.

Una vez que la aplicación *mashup* ha sido inicializada, la capa cliente debe dar soporte al resto de operaciones de comunicación que se realizan. Estas operaciones incluyen: (*i*) comunicación entre componentes, (*ii*) envío de mensajes a la capa dependiente de la

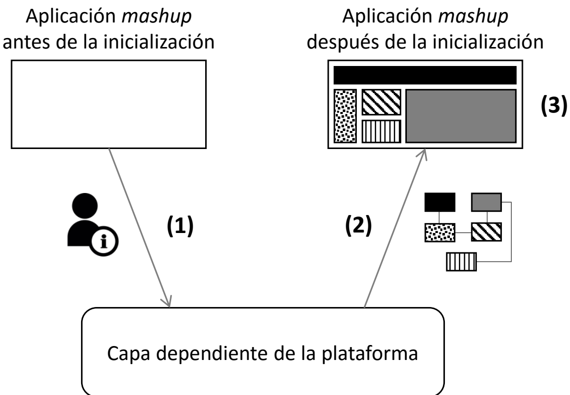


Figura 2.4: Inicialización de las aplicaciones *mashup* en la capa cliente

plataforma (para la invocación de las operaciones de los servicios que ofrece el COScore), y (*iii*) recepción de mensajes de la capa dependiente de la plataforma (para recibir las respuestas de los servicios del COScore o para recibir notificaciones en el caso de que fuera necesario). El primer caso es el único que se podría resolver directamente en la capa cliente, sin utilizar el resto de capas de la infraestructura pero, sin embargo, esta operación también se resuelve en el COScore. Para ello, la resolución de la comunicación entre componentes es un servicio más que ofrece la infraestructura desarrollada. Por lo tanto, resulta clave el mecanismo llevado a cabo para comunicar la capa cliente con la capa dependiente de la plataforma. En nuestro caso, las aplicaciones *mashup* se conectan a la capa dependiente (y viceversa) mediante el uso de *sockets* utilizando un servidor *JavaScript* (ubicado en la capa dependiente) que hace de mediador entre la capa cliente y la capa independiente.

Desde el momento de su inicialización, las aplicaciones *mashup* son modificadas o actualizadas mediante la gestión de las piezas que las conforman, *i.e.*, a través de la gestión de sus arquitecturas. Puesto que, tal y como se ha mencionado, las dependencias y comunicaciones entre componentes no se resuelven directamente en la capa cliente, el software (ya sea en forma de código, objetos, etc.) resultante del despliegue de las aplicaciones *mashup* no contiene información acerca de estas relaciones. Por lo tanto, desde el punto de vista de la capa cliente, las aplicaciones son un conjunto de "trozos" que no están conectados entre sí, sino que se conectan (para el envío y recepción de información) a la capa dependiente a través del servidor *JavaScript* mencionado anteriormente. En la Figura 2.5 se muestran dos ejemplos de operaciones que se llevan a cabo en las aplicaciones *mashup* y su comportamiento respecto a la capa cliente. En el caso de la inserción de un nuevo componente (parte izquierda de la Figura 2.5), la aplicación envía un mensaje a la capa dependiente para solicitar la incoporación de un nuevo elemento. Como resultado, la capa dependiente envía un mensaje de respuesta con el componente (en forma de código, en forma de objeto, etc.) que debe ser insertado. En el caso de la comunicación entre componentes (parte derecha de la figura), la aplicación envía a la

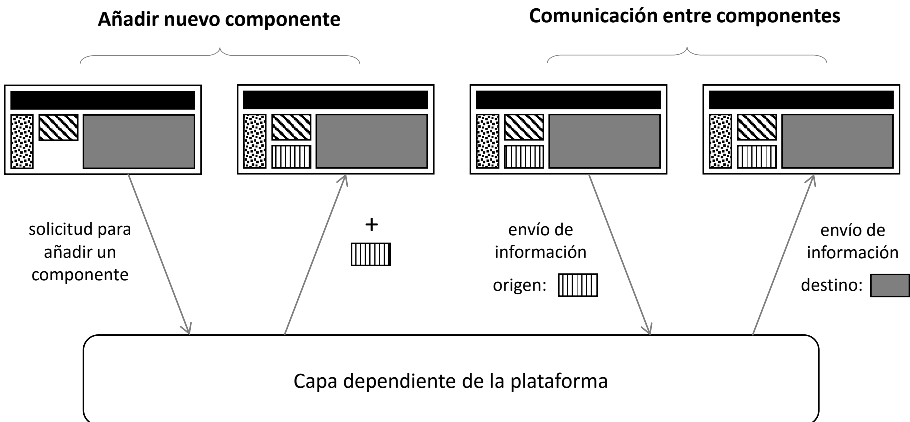


Figura 2.5: Operaciones de inserción y comunicación entre componentes (capa cliente)

© 2016 Vallecillos, J.

capa cliente tanto el identificador del componente origen como el contenido del mensaje que quiere enviar a otro componente. Una vez resuelto el camino de comunicación, la capa dependiente envía al componente (o componentes, en el caso de que existan varios receptores) destino los datos enviados por parte del componente emisor.

Como ha sido mencionado previamente, para el trabajo de tesis doctoral se han utilizado dos escenarios de ejemplo, uno que hace uso de aplicaciones *mashup* basadas en la Web para el desarrollo de interfaces de usuario, y otro que utiliza aplicaciones *mashup* implementadas en Java para el control de un sistema de hogar digital. Con el objetivo de ilustrar al lector acerca del aspecto de este tipo de aplicaciones desde el punto de vista de la capa cliente, la Figura 2.6 muestra una aplicación sencilla perteneciente al primero de los escenarios. En particular, la aplicación mashup permite la interacción con componentes de un Sistema de Información Geográfica (SIG), entre los cuales se encuentran mapas, visores, aplicaciones para el análisis de datos geográficos, componentes de redes sociales, etc. En esta interfaz de usuario de ejemplo, se muestra una aplicación con cinco componentes: una cabecera para la gestión de la sesión del usuario, tres mapas con información de capas geográficas y un componente de Twitter. La parte derecha de la Figura 2.6 muestra el código de dicha interfaz. Como se puede ver, la aplicación consiste básicamente en un archivo `html` que contiene tantos elementos de tipo `iframe` como los componentes que la constituyen. Cada *iframe* hace referencia (a través del atributo `src`) a las instancias de los componentes, las cuales se encuentran alojadas en uno de los repositorios de componentes gestionados por la infraestructura desarrollada.

2.3. Capa dependiente de la plataforma

Como ya se ha mencionado anteriormente, el objetivo de la infraestructura es poder dar soporte a diferentes tipos de aplicaciones *mashup* que hayan sido desarrolladas por

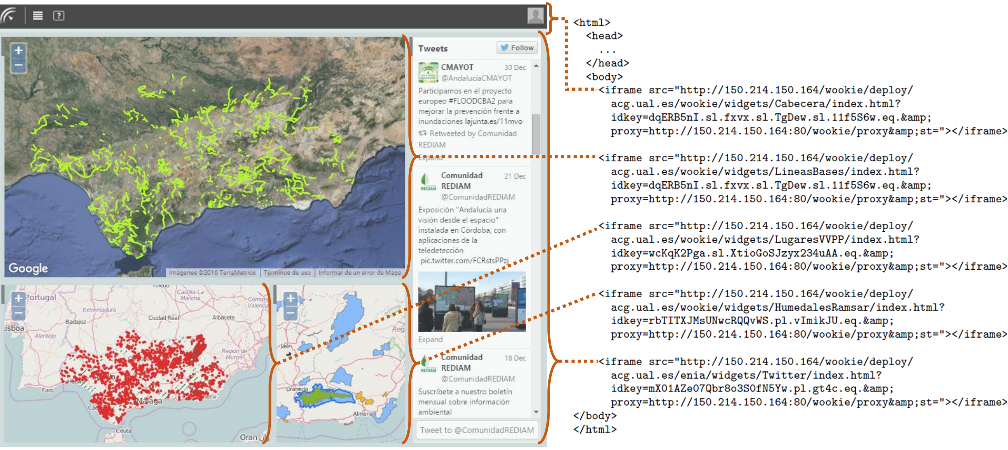


Figura 2.6: Aplicación *mashup* de ejemplo desde la perspectiva de la capa cliente

medio de diferentes tecnologías. Consecuentemente, la infraestructura debe dar soporte a los distintos tipos de componentes por los cuales están formadas dichas aplicaciones. Por este motivo, se propone utilizar una capa intermedia que tenga una vinculación con las plataformas específicas a las que se da soporte, y que sirva como intermediaria entre la capa cliente (aplicaciones *mashup*) y los servicios ofrecidos por la infraestructura desarrollada (que son utilizados por las aplicaciones).

Esta capa intermedia, nombrada como capa dependiente de la plataforma, está formada por bloques principales: un conjunto de **repositorios de componentes** (agrupados a su vez por el tipo de plataforma) y un **servidor** (o varios servidores) **basado en eventos**. El objetivo de los repositorios de componentes es obvio: se encargan de almacenar y ofrecer los elementos que formarán parte las aplicaciones *mashup*. En lo que se refiere al servidor (o servidores) basado en eventos, su finalidad es (*i*) recibir peticiones que provienen de la capa cliente, (*ii*) resolver estas peticiones a través de la invocación de servicios web proporcionados por la capa independiente de la plataforma, y (*iii*) enviar los mensajes de respuesta, notificación o actualización correspondientes a las aplicaciones *mashup*. Dependiendo de la estrategia de implementación, es posible utilizar un único servidor basado en eventos que atienda las peticiones de todas las plataformas a las que la infraestructura da soporte (Figura 2.7a) o, por el contrario, desplegar un servidor por cada plataforma (Figura 2.7b).

Para la implementación de este tipo de servidores basados en eventos, se ha optado por Node.js[2], que está implementado con JavaScript y que permite realizar la conexión entre las capas de la infraestructura tanto de forma síncrona como asíncrona. Para ello, este tipo de servidor se conecta con la capa cliente a través de *sockets*, haciendo uso de Socket.io[3]. El uso de este tipo de servidor presenta una serie de ventajas. En primer lugar, cuando se produce un evento en alguno de los componentes de la aplicación *mas-*

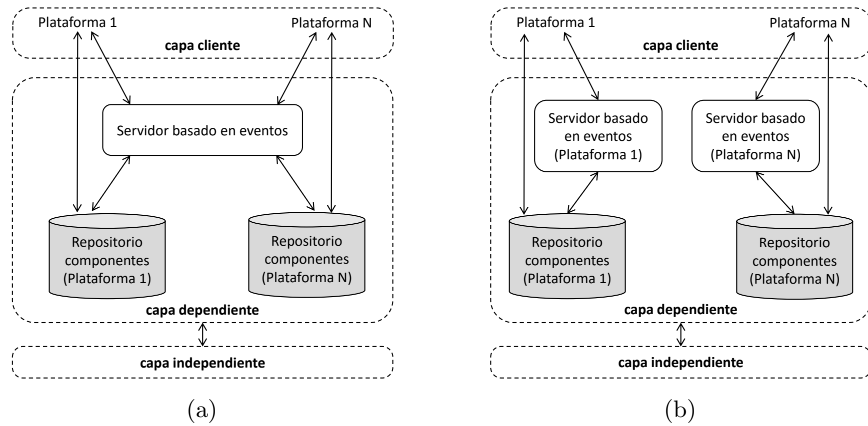


Figura 2.7: Capa dependiente de la plataforma: (a) con un único servidor basado en eventos y (b) con un servidor por cada plataforma soportada por la infraestructura

[2]Node.js – `https://nodejs.org/`
[3]Socket.io – `http://socket.io/`

© 2016 Vallecillos, J.

hup, inmediatamente el servidor puede reaccionar comunicándose, si es necesario, con la parte independiente e invocando los servicios web correspondientes. Además, cuando se producen procesos de comunicación entre componentes que forman parte de la aplicación, la interacción entre componentes no impide que la aplicación pueda ser actualizada de forma paralela, ya que los componentes siguen a la escucha para poder recibir eventos procedentes del servidor JavaScript. Por otro lado, la aplicación puede ser actualizada sólo en aquellas partes que son modificadas, y no es necesario realizar una recarga de la aplicación *mashup* al completo. Este hecho permite que el estado actual del resto de elementos que no cambian no se vea afectado. Adicionalmente, el uso de *sockets* para la comunicación permite conectar componentes de cualquier tipo y plataforma, siempre que puedan hacer uso de esta tecnología. Esto significa que dentro de la infraestructura se pueden integrar componentes de tipo web, de tipo Java, o de cualquier otro tipo de plataforma. Ese tipo de servidor *JavaScript* también ofrece la posibilidad de consumir los servicios web y las operaciones que despliega la capa independiente.

Respecto a los repositorios de componentes, en la infraestructura se propone hacer uso del modelo de datos que se muestra en la Figura 2.8. Aunque el modelo de datos

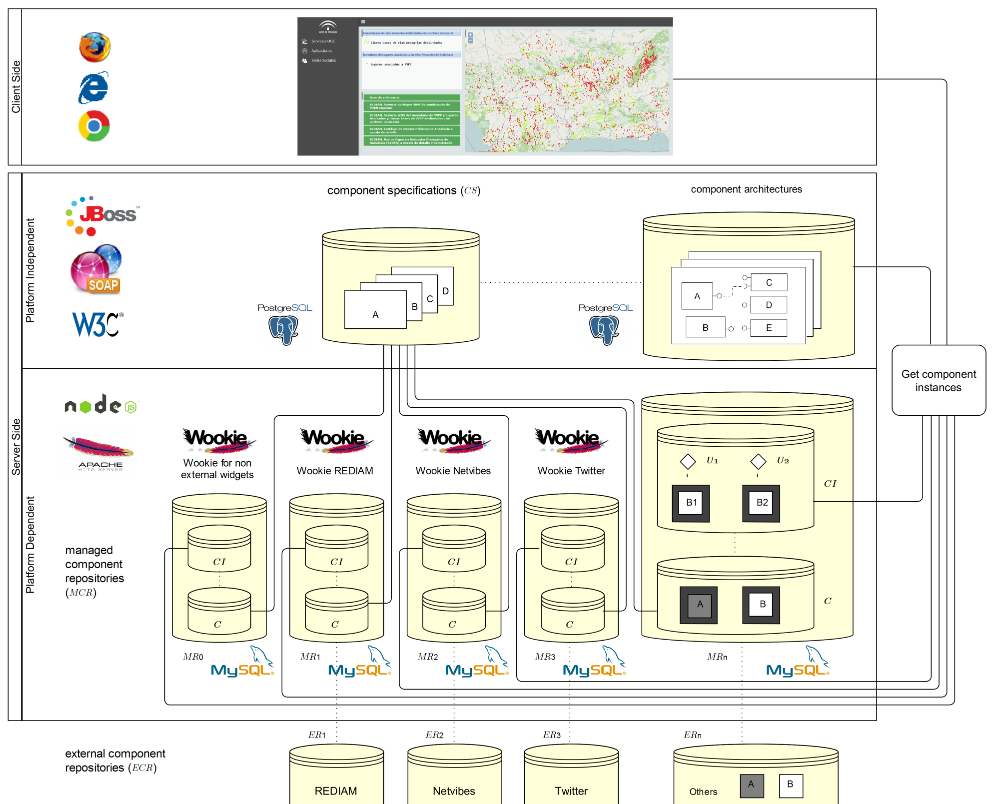


Figura 2.8: Modelo de datos de componentes

mostrado se basa en aplicaciones *mashup* de tipo web, su estructura es idéntica para cualquier tipo de plataforma. El motivo de haber utilizado la plataforma web para ilustrar el modelo de datos es que, de esta manera, es posible incluir un ejemplo de las tecnologías específicas que se han utilizado para el desarrollo de la capa dependiente de la plataforma. En la Figura 2.8 se muestran las tres capas de la infraestructura: la capa cliente (*Client side*), la capa dependiente de la plataforma (*Platform dependent*) y la capa independiente de la plataforma (*Platform independent*). Los repositorios de los componentes "reales" que se utilizan para construir las aplicaciones *mashup* se encuentran en la capa dependiente. No obstante, existen otros repositorios de datos que son importantes para el funcionamiento de la infraestructura: las especificaciones de componentes (*component specifications*) y las arquitecturas de componentes (*component architectures*). Aunque estos elementos son mencionados en esta sección para la descripción del modelo de datos de la capa dependiente, se describen con más detalle en la Sección 2.4.

En la capa dependiente de la plataforma están los repositorios de componentes propios (*managed component repositories*, *MCR*), que ofrecen los componentes (C) que son utilizados en la aplicaciones *mashup* desplegadas en la capa cliente (ver Figura 2.8). Además, estos repositorios almacenan las instancias (CI) de los componentes, que permiten identificar y almacenar el estado de un componente que forma parte de una aplicación *mashup* y que pertenece a un usuario específico. Para que los componentes desarrollados por terceras partes que se encuentren en repositorios externos a la infraestructura puedan ser utilizados por nuestras aplicaciones *mashup*, es necesario incorporar estos componentes como parte del conjunto de repositorios de componentes propios (*MCR*). En la Figura 2.8 se muestran tres repositorios de componentes externos (*external component repositories*, *ECR*) que son incorporados al modelo de datos. Para poder realizar esta incorporación, es necesario construir un código envoltorio (*wrapper*) que permita almacenar los componentes en los repositorios *MCR* y obtener instancias de dichos componentes. Este código envoltorio contendrá además al componente externo y/o hará referencia a aquellos recursos externos necesarios para su ejecución.

En las siguientes subsecciones se describen los distintos repositorios de componentes mostrados en la Figura 2.8. Esta descripción incluye información sobre la estructura y composición de los repositorios de componentes propios (*MCR*). Además, se ofrecen detalles acerca de cómo los componentes ubicados en repositorios externos (*ERC*) son adaptados para poder ser utilizados dentro de la infraestructura propuesta.

2.3.1. Gestión de componentes propios

Los repositorios de componentes propios son aquellos repositorios de los cuales se nutren las aplicaciones *mashup* que son construidas y desplegadas por la infraestructura desarrollada. Tal y como se observa en el modelo de datos de la Figura 2.8, existen diferentes repositorios de componentes que constituyen el conjunto de repositorios de componentes propios, denotado como $MCR = \{MR_1, \ldots, MR_n\}$. Cada uno de los repositorios de este conjunto (MR_i) está formado por un conjunto de componentes (C) y un conjunto de instancias, de manera que $MR_i = \{C_i, CI_i\}$. A su vez, es posible definir el conjunto de componentes como $C_i = \{c_1, \ldots, c_j\}$. De igual forma, el conjunto de instancias se representa mediante la expresión $CI_i = \{i_1, \ldots, i_k\}$. Por un lado, cada componente contiene

© 2016 Vallecillos, J.

el código necesario para poder ser ejecutado dentro de una aplicación *mashup*. Por otro lado, una instancia de un componente se crea a partir de la información que identifica al usuario al cual pertenece la aplicación *mashup* en la que se encuentra dicho componente. Cada instancia tiene asociada cierta información relativa al estado del componente y que lo diferencia del resto de instancias del mismo componente, tal y como se puede observar para las instancias `B1` y `B2` del componente `B` de la Figura 2.8.

Al igual que para el modelo de datos de la Figura 2.8, para describir la gestión de los componentes propios de la infraestructura, también se va a hacer uso de la plataforma web como escenario de ejemplo. En dicho dominio, el tipo de repositorio escogido para alojar los componentes de tipo web es Apache Wookie[4]. Este tipo de repositorio permite almacenar *widgets* que cumplen con la especificación de W3C[5]. Este tipo de componente cumple con los principios establecidos para los componentes COTSgets de las aplicaciones *mashup* que forman parte de nuestra infraestructura.

Para comprender mejor cómo está estructurado un *widget* de este tipo, veamos un componente de ejemplo que forma parte de uno de los repositorios de componentes propios. En este caso, se trata de un componente mapa del repositorio MR_0 (repositorio de componentes propios que no está relacionado con un repositorio de componentes externos). Este componente mapa permite visualizar una capa geográfica que contiene información sobre los espacios naturales protegidos de la región de Andalucía (España). Según la definición de W3C, este *widget* debe contener (como mínimo) en el directorio raíz un archivo **`index.html`** y un archivo **`config.xml`**. Para el caso de este componente, además existe un archivo de imagen que se utiliza como icono para identificar el componente en el repositorio Wookie. Por otro lado, el componente está formado por los directorios **`content`** e **`interface`**, que contienen la funcionalidad del componente codificada en JavaScript (ver Figura 2.9). La estructura de carpetas que contienen la implementación de la funcionalidad de los componentes no viene determinada por la especificación de *widget* de W3C, sino que ha sido propuesta para la definición de nuestros componentes COTSgets como parte del trabajo de investigación realizado.

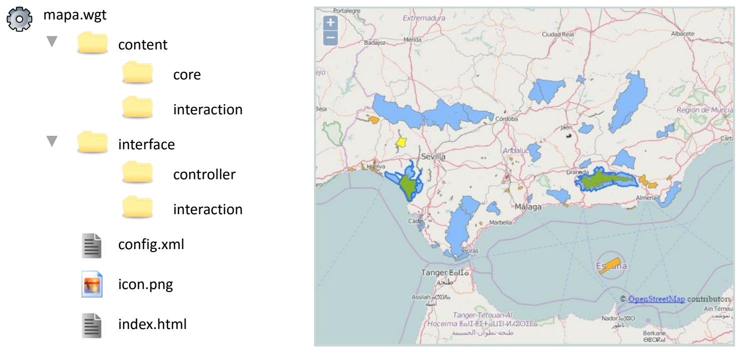


Figura 2.9: Estructura W3C de un *widget* mapa

[4]Apache Wookie – `http://wookie.apache.org/`
[5]Widgets de W3C – `https://www.w3.org/TR/widgets/`

```
<?xml version="1.0" ?>
<widget xmlns="http://www.w3.org/ns/widgets" id="http://acg.ual.es/
    wookie/widgets/Mapa" version="1.1.0" width="844" height="508">
  <!-- Autor del Widget -->
  <author>Jesús</author>
  <!-- Página de arranque -->
  <content src="index.html" />
  <!-- Descripción del widget -->
  <description>...</description>
  <!-- Nombre del widget -->
  <name>OGC-RENPA-EENNPP</name>
  <!-- Icono por defecto del widget -->
  <icon src="icon.png" />
</widget>
```

Tabla 2.1: Archivo de configuración de un *widget* W3C

La información del archivo `config.xml` se utiliza para gestionar los componentes que han sido dados de alta en el repositorio de componentes de tipo web (Apache Wookie). La Tabla 2.1 muestra un ejemplo de archivo `config.xml` perteneciente al componente mapa comentado anteriormente. En dicha tabla se puede ver la estructura de un *widget* compuesto por el espacio de nombres (`xmlns`), un identificador del componente (`id`), una versión de componente (`version`), el ancho (`width`) y alto (`height`) por defecto, el autor (`author`), la página base del componente (`content`), la descripción (`description`), el nombre (`name`) y el icono del componente (`icon`).

Una vez que los componentes han sido dados de alta en Wookie, dichos repositorios tienen la capacidad de proporcionar instancias de los componentes. Para solicitar la creación de estas instancias de componentes, los repositorios ofrecen un servicio web RESTful que se encarga de crearlas, o de devolver una dirección con su localización, en el caso de que la instancia solicitada ya estuviera creada. Como resultado, el repositorio de componentes devuelve una URL con la dirección de la instancia del componente, la cual está asociada a una aplicación *mashup* de un usuario específico. Un ejemplo de URL de una instancia creada para el componente ejemplo anterior es:

`http://150.214.150.164/wookie/deploy/acg.ual.es/wookie/widgets/OGC-RENPA-EENNPP/index.html?idkey=yxHz1SZZoC1JA750Fm0a3KnCODs.eq.&proxy=http://150-.214.150.164:80/wookie/proxy&st=`.

En la URL anterior se encuentra el identificador de la instancia dentro del parámetro `idkey`. En caso de querer situar este componente dentro de una página web HTML es suficiente con hacer uso del elemento `iframe` y añadir dentro del atributo `src` la URL de la instancia (`<iframe src = “...”/>`).

Además de los repositorios de componentes propios utilizados para el caso de la plataforma web, en el desarrollo de la infraestructura propuesta también se han construido repositorios de componentes propios para la plataforma Java, la cual permite desplegar aplicaciones *mashup* para el control de sistemas de hogar digital. Para dar soporte a estos componentes, se ha creado un servidor de componentes Java. Además, para la

© 2016 Vallecillos, J.

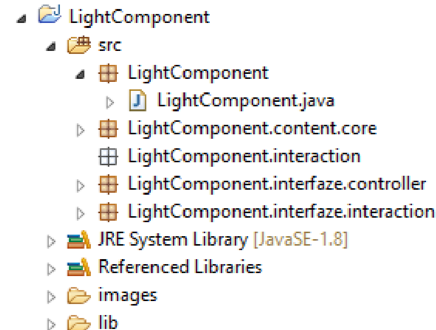


Figura 2.10: Estructura del componente Java

implementación de este tipo de componentes, se ha seguido la estructura definida anteriormente para el caso de los componentes de tipo web. En la Figura 2.10 se muestra la estructura de la implementación de un componente de tipo *bombilla* perteneciente a una instalación domótica.

Para poder dar de alta este tipo de componentes en el repositorio de componentes Java, se debe generar un archivo `.jar` y situarlo en un directorio específico para que el servidor lo pueda ofrecer y gestionar. Cuando un cliente pretende hacer uso de un componente, se pone en contacto con este servidor. Al igual que en el caso de los repositorios de componentes web, el servidor genera una instancia a partir de la información aportada por el usuario. La instancia que se devuelve al usuario (a la aplicación *mashup* que se le ofrece al usuario) es un archivo que contiene un objeto en formato binario. El identificador de la instancia es el nombre del archivo en sí, por lo que para el caso del componente visto en la Figura 2.10, el nombre de archivo de la instancia podría ser `2_app2_LightComponent_Light.obj`. El número “2” define el identificador del usuario, “app2”describe la aplicación a la cual pertenece el componente, “LightComponent” es el identificador del componente y “Light” es el nombre del componente.

2.3.2. Gestión de componentes de repositorios externos

De manera adicional a los componentes desarrollados específicamente para nuestras aplicaciones *mashup*, la infraestructura desarrollada permite que puedan utilizarse componentes desarrollados por terceros. Dichos componentes se encuentran almacenados en repositorios externos. Es importante destacar que estos componentes no son extraídos de sus repositorios de origen para llevar a cabo su integración, si no que se construye un código envoltorio (*wrapper*) que redirecciona el acceso hacia dicho componente para poder ser integrado en la aplicación *mashup*. Existen otros casos en los que no se realiza una redirección, sino que se replica una copia del componente en la cual se incrusta parte de su comportamiento. Es cierto que esta técnica depende del funcionamiento correcto y de la disponibilidad de los componentes externos, lo cual no puede ser controlado por la infraestructura. No obstante, el trabajo de investigación propuesto permite que la aplicación *mashup* siga funcionando a pesar de la “desconexión” de dichos componentes.

```
<body>
    <a class="twitter-timeline" href="https://twitter.com/
      ComunidadREDIAM" data-widget-id="449117979691585536">
      Tweets por @ComunidadREDIAM</a>
    <script>!function(d,s,id){var js,fjs=d.getElementsByTagName(s)
      [0],p=/^http:/.test(d.location)?'http':'https';if(!d.
      getElementById(id)){js=d.createElement(s);...
    </script>
</body>
```

Tabla 2.2: Elemento `body` de un componente web externo adaptado

En la Figura 2.8, podemos ver que el conjunto de *repositorios de componentes externos* (*External Component Repositories*, *ECR*) se encuentra fuera del COScore. Estos componentes se encuentran almacenados en su lugar de origen. El nivel *ECR* está formado por un conjunto de n repositorios, de manera que $ECR = \{ER_1, ..., ER_n\}$; es decir, existe uno por cada plataforma que se quiera adaptar en el sistema. Para cada uno de estos repositorios existirá su correspondiente repositorio de componentes gestionados (MR_i) en la capa dependiente de la plataforma de la infraestructura para adoptar y adaptar dichos componentes en el sistema. A modo de ejemplo, en la Figura 2.8 se pueden ver tres repositorios de componentes de tipo web que pertenecen a organizaciones externas (como REDIAM, Netvibes y Twitter) y que están siendo integradas en el sistema a través de repositorios de tipo Apache Wookie.

Con el propósito de comprender cómo funciona este proceso de integración de los componentes desarrollados por terceras partes en la infraestructura COScore, se aporta un caso de integración paso a paso. Para el caso de los componentes de tipo web considérese, por ejemplo, la integración de un componente de Twitter en una aplicación web. Al tratarse de un componente de tipo web, la carcasa (*wrapper*) se realiza mediante la creación de un componente *widget* que sigue la especificación de W3C. Tal y como ha sido mencionado anteriormente, la especificación de este tipo de *widget* determina que es necesario crear un archivo `index.html`, el cual constituye el elemento base del componente. En este caso de ejemplo, el elemento `body` del archivo HTML está constituido como se observa en la Tabla 2.2.

Para que este componente externo funcione correctamente en nuestras aplicaciones, ha sido necesario añadir cierto código con el objetivo de que el contenido del componente se ajuste a las dimensiones específicas establecidas para el componente en su especificación. Este código ha sido incorporado como parte del elemento `head` del archivo `index.html`, tal y como muestra la Tabla 2.3.

Una vez construido el `index.html`, se genera la estructura de directorios definida según la especificación de W3C, donde será guardado dicho archivo junto con el resto de archivos JavaScript y CSS necesarios para su funcionamiento. Posteriormente, se genera un archivo comprimido que contendrá todos los elementos del componente *widget*. Finalmente, ese componente se da de alta en el repositorio de componentes de Wookie. En la Figura 2.8, dicho repositorio de componentes propios se corresponde con *Wookie Twitter* (MR_3), el cual contiene los componentes propios que han sido adaptados a partir de los componentes externos que pertenecen a ER_3.

© 2016 Vallecillos, J.

```
<head>
  ...
  <script>
    $(window).bind('load', function() {
      $('body').css('height','100%');
      setTimeout(function(){
        var divTwitter = document.getElementById('twitter-widget-0').
          contentWindow.document.body.firstChild;
        var iframeTwitter = document.getElementById('twitter-widget-0');
        $(iframeTwitter).css('width','100%');
        $(iframeTwitter).css('height','100%');
        $(iframeTwitter).css('position','absolute');
        $(divTwitter).css('width','100%');
        $(divTwitter).css('height','100%');
      }, 1000);
    });
  </script>
</head>
```

Tabla 2.3: Elemento `head` de un componente web externo adaptado

2.4. Capa independiente de la plataforma

La capa independiente de la plataforma pretende, como su propio nombre indica, ofrecer una serie de servicios a todas las aplicaciones *mashup* independientemente de la plataforma utilizada para ejecutar los componentes. Para conseguir esta independencia, el sistema considera la aplicación *mashup* como una abstracción de la misma. Para ello, se han desarrollado dos metamodelos que definen los elementos principales de dicho tipo de aplicaciones: un metamodelo de arquitectura y un metamodelo de componentes. Por tanto, para dar soporte a las aplicaciones *mashup* y puesto que han sido descritas haciendo uso de modelos, la capa independiente de la plataforma ofrece una serie de servicios para la manipulación de dichos modelos, permitiendo de esta manera la vinculación con la aplicación desplegada en la capa cliente.

En la siguiente subsección se muestra el metamodelo de las arquitecturas que representan las aplicaciones *mashup*, describiendo cada una de las partes por las cuales está formado, además de las restricciones definidas para construir modelos a partir del mismo. Posteriormente, se describe el metamodelo utilizado para definir los componentes que forman parte de dichas arquitecturas, ofreciendo detalles acerca de su estructura y de las propiedades que se incluyen en esta definición. Con respecto a los servicios desplegados en esta capa, en el Capítulo 3 se exponen cada una de las operaciones ofrecidas.

2.4.1. Arquitecturas de las aplicaciones *mashup*

Esta sección describe el metamodelo utilizado para definir las aplicaciones *mashup*, las cuales se basan en arquitecturas de componentes COTSget. Las instancias de este metamodelo de arquitectura (*i.e.*, los modelos de las arquitecturas) se guardan en un repositorio específico dentro de la capa independiente de la plataforma de la infraestructura propuesta (repositorio *component architectures* del modelo de datos de la Figura 2.8).

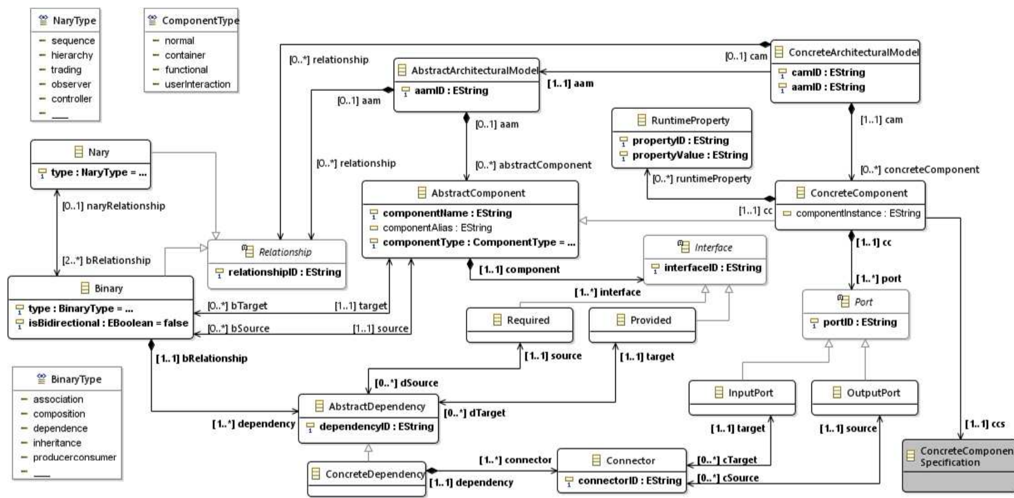


Figura 2.11: Metamodelo de arquitectura basada en componentes

Para el modelado, se utilizan técnicas de diseño de MDE con el objetivo de construir un metamodelo de arquitecturas, tal y como muestra la Figura 2.11. Esta representación ayuda a comprender las diferentes partes que forman la arquitectura.

El metamodelo de la Figura 2.11 permite definir las arquitecturas en dos niveles de abstracción, de manera que es posible crear dos tipos de elementos: modelos de arquitecturas abstractas (*AbstractArchitecturalModel*) y modelos de arquitecturas concretas (*ConcreteArchitecturalModel*). Los modelos de arquitecturas abstractas identifican la forma y estructura de las aplicaciones *mashup*, además de determinar los tipos de componentes que deben ser incluidos para que sea considerada correcta. Por otro lado, los modelos de arquitecturas concretas identifican los componentes concretos (que se corresponden con las definiciones de los componentes reales que pueden ser ejecutados en las aplicaciones) que han sido seleccionados como solución para los tipos de componentes definidos en los modelos de arquitecturas abstractas. No obstante, el presente trabajo de investigación se centra únicamente en el nivel concreto de las arquitecturas, puesto que se corresponde con el nivel de abstracción necesario para la gestión y el despliegue de las aplicaciones *mashup*. El hecho de tener en cuenta también el nivel abstracto de las arquitecturas se debe a que la infraestructura propuesta en esta tesis doctoral se basa en el trabajo presentado en [Criado, 2015], además de que supone una solución tecnológica de la metodología descrita.

Según el metamodelo de la Figura 2.11, cada modelo de arquitectura concreta está formado por un conjunto de componentes concretos (*ConcreteComponent*) y un conjunto de relaciones (*Relationship*) que existen entre dichos componentes. Cada componente tiene una propiedad que describe su tipo (*ComponentType*). El tipo de componente contenedor (*container*) identifica un componente que se usa para contener otros componentes. Esto hace posible construir componentes más complejos a partir de otros. El tipo de componente funcional (*functional*) se usa para construir componentes funciona-

© 2016 Vallecillos, J.

les que no incluyen una interfaz para la interacción con el usuario, permitiendo ejecutar código en segundo plano (*background*). El tipo de componente *userInteraction* se usa para construir componentes que incluyen interacción con el usuario o que, simplemente, permiten visualizar cierta información. Finalmente, el tipo de componente *normal* es un componente que une las características de los tipos *functional* y *userInteraction*.

Los modelos de arquitecturas también incluyen las relaciones que existen entre sus componentes. Cada relación conecta dos o más componentes simultáneamente y está formada por un conjunto de elementos de tipo conector (*Connector*). Los conectores representan los enlaces que existen entre los puertos de salida de unos componentes y los puertos de entrada de otros componentes, y constituyen la representación de los caminos de comunicación que existen dentro de la arquitectura. Las relaciones pueden ser de dos tipos: binarias (*Binary*) y n-arias (*Nary*). Las relaciones binarias son las relaciones que existen entre dos componentes (por ejemplo, asociación, composición, etc.). Las relaciones n-arias se componen al menos de dos relaciones binarias y, por lo tanto, relacionan al menos tres componentes de la arquitectura (por ejemplo, herencia, secuencia, etc.). Más adelante se ofrecerán detalles de las relaciones.

No todas las restricciones del modelo de arquitectura pueden ser expresadas en un metamodelo. Por esta razón, se ha definido un conjunto de restricciones OCL (*Object Constraint Language*), las cuales ayudan a formalizar las restricciones descritas y mejorar el modelo, dando coherencia y confiabilidad. En una arquitectura sólo se permite una relación simple entre componentes, es decir, una relación binaria cuyo origen es un componente A y un componente destino B. Esta restricción se ve en la Tabla 2.4.

```
context ConcreteArchitecturalModel
  inv: not(relationship -> exists(r1 : Binary, r2 : Binary | r1 <> r2
    and r1.oclIsTypeOf(Binary) and r2.oclIsTypeOf(Binary)
    and r1.source = r2.source and r1.target = r2.target));
```

Tabla 2.4: Restricción de relación binaria

Esta restricción es básica en el modelo de arquitectura y ayuda a restringir el número de posibles relaciones entre componentes, haciendo el modelo más manejable y útil. La siguiente restricción indica que si la relación entre dos componentes A y B es bidireccional, entonces hay al menos dos conectores: uno cuyo origen es A y el destino es B, y otro conector cuyo origen es B y destino A. Así, la restricción en OCL se puede ver en la Tabla 2.5.

```
context Binary
  inv: isBidirectional = true implies (connector->exists(
    c1, c2 | c1 <> c2 and
    c1.source.component = c2.target.component));
```

Tabla 2.5: Restricción bidireccional entre dos componentes

El código de la restricción indica que al menos un puerto de salida del primer componente debe ser conectado a un puerto de entrada del segundo, y viceversa. Asimismo,

se ha establecido que un componente debe tener al menos un puerto de entrada. La correspondiente restricción OCL se puede observar en la Tabla 2.6.

```
context ConcreteComponent
  inv: port->exists(p | p.oclIsTypeOf(InputPort));
```

Tabla 2.6: Restricción para un puerto de entrada

De este modo, se comprueba que hay al menos un puerto de entrada (*InputPort*). A lo largo de este capítulo se ofrecen más detalles acerca de la estructura interna de los componentes y de las relaciones que pueden ser establecidas entre ellos.

2.4.2. Especificación de componentes

El propósito de hacer uso de un metamodelo para la definición de los componentes (COTSgets) que conforman las aplicaciones *mashup* es conseguir que el desarrollo de componentes tenga una estructura definida formalmente. De esta manera, cuando un desarrollador pretenda construir un componente para integrarlo dentro de la infraestructura, sabrá qué elementos debe contener para funcionar correctamente. Dentro de la capa independiente de la plataforma, las instancias de este metamodelo (*i.e.*, los modelos de componentes) se almacenan en un repositorio específico (*component specifications*) del modelo de datos de la Figura 2.8.

En la Figura 2.12 se puede ver parte del metamodelo propuesto (adaptado del modelo de componente COTScomponent de [Iribarne et al., 2004]) que describe la estructura interna de los componentes. Un componente está formado por cuatro bloques princi-

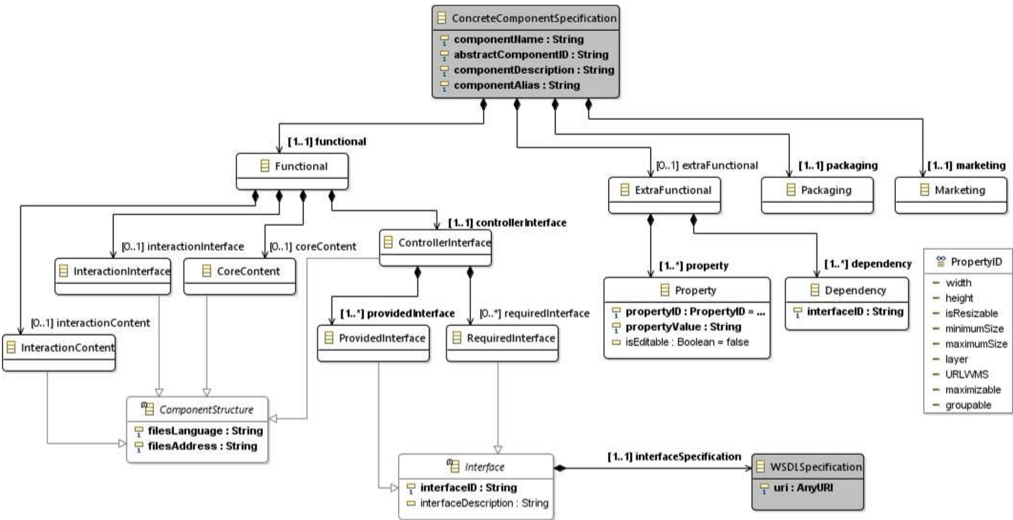


Figura 2.12: Metamodelo de componente

© 2016 Vallecillos, J.

pales: parte funcional (*Functional*), parte extra-funcional (*ExtraFunctional*), parte de empaquetado (*Packaging*) y parte de mercado (*Marketing*). La parte *Marketing* de un componente identifica la información relacionada con las entidades que desarrollan el componente, tales como el nombre de la organización, nombre de contacto, etc. La parte *Packaging* proporciona información relacionada con el empaquetado del componente, centrándose en identificar el repositorio donde el componente se localiza, el lenguaje de programación usado, etc. La parte *ExtraFunctional* identifica el conjunto de propiedades extra-funcionales que un componente puede tener. Estas propiedades proporcionan información no funcional (incluyendo tanto aspectos de calidad de servicio como otros atributos relevantes), por ejemplo, propiedades relacionadas con la apariencia del componente como la anchura o la altura. Además, el bloque extra-funcional incluye el conjunto de dependencias que un componente tiene con respecto a otros componentes. Finalmente, la parte *Functional* describe con detalle la funcionalidad del componente.

La parte funcional es el bloque fundamental para comprender la estructura de un componente COTSget. En un componente se diferencia la funcionalidad relacionada con la interacción del usuario (*InteractionContent*) y la propia interna del componente (*CoreContent*). Por otro lado, la parte funcional está compuesta por un conjunto de interfaces que permiten comunicar el componente con el exterior. A través de la interfaz de interacción (*InteractionInterface*) tienen lugar las acciones entre el usuario que hace uso del componente y el propio componente. Por lo tanto, en esta interfaz se gestionan los eventos que son soportados, los cuales están condicionados por la plataforma en la que se despliega la aplicación. El desarrollador del componente debe tener en cuenta cuál es su entorno (entornos móviles, aplicaciones de escritorio, etc.) para implementar la funcionalidad acorde a los posibles eventos. Por otro lado, la interfaz controlador (*ControllerInterface*) se utiliza para gestionar la comunicación entre componentes (o la invocación de un componente desde la propia infraestructura) y es inaccesible para el usuario. Esta interfaz permite la comunicación de los componentes a través de un conjunto de interfaces proporcionadas (*ProvidedInterfaces*) y de un conjunto de interfaces requeridas (*RequiredInterfaces*).

Las interfaces proporcionadas definen toda la funcionalidad que el componente establece como visible para el exterior, es decir, describen métodos que pueden ser invocados para hacer uso de algunas operaciones pertenecientes al componente. Las interfaces requeridas de un componente describen qué operaciones pertenecen a otros componentes pero que son invocados por este componente para funcionar de forma correcta. Cada interfaz se describe a través del elemento *WSDLSpecification*, que cumple con la estruc-

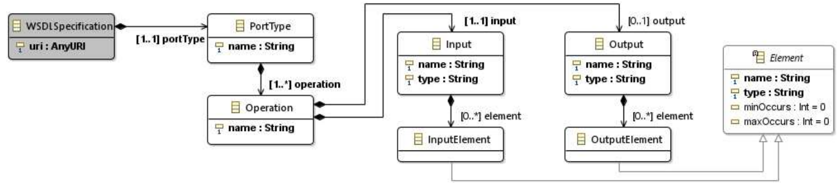


Figura 2.13: Especificación de las interfaces funcionales

tura de un descriptor WSDL (*Web Service Description Language*)[6]. Esta especificación usa el concepto *portType* como elemento raíz para describir cada una de las interfaces. En la Figura 2.13 se puede ver el fragmento del metamodelo que se corresponde con dicha especificación. Cada interfaz tiene un nombre (para que las interfaces dentro del mismo componente puedan ser referidas de manera única) y un conjunto de operaciones (*Operation*). La información que devuelven como respuesta estas operaciones está reflejado en su definición de salida (*Output*), mientras que la información que reciben como parámetro, viene definido por su entrada (*Input*). Una operación siempre define el tipo de información de entrada y, opcionalmente, define el tipo de información de salida (si no existe *Output* significa que la operación no devuelve información como respuesta).

Dependiendo del tipo de implementación realizada para un componente y, como consecuencia, del número de elementos de la parte funcional que son definidos, existen los distintos tipos de componentes que se describen a continuación:

— *Componente funcional*: estos componentes sólo implementan la funcionalidad interna (*CoreContent*), sin implementar funcionalidad relacionada con la interacción de usuario (*InteractionContent*). Puede ser desarrollado para ejecutar tareas en *background*, como por ejemplo, acceder a una base de datos.

— *Componente de interacción de usuario*: este componente sólo incluye la funcionalidad asociada con la interacción de usuario (*InteractionContent*). Este tipo de componente puede ser utilizado para visualizar alguna información de tipo gráfica o para interaccionar con el usuario.

— *Componente contenedor*: este componente no tiene ni funcionalidad interna ni funcionalidad relacionada con la interacción del usuario. Un contenedor es un componente que está formado por varios componentes para, conjuntamente, desarrollar una tarea o propuesta común.

— *Componente normal*: este tipo de componente es un híbrido entre el componente de interacción y el componente funcional. Permite gestionar la interacción de usuario, pero además implementa funcionalidad interna.

En la Figura 2.14a se muestra un componente de tipo *normal*, puesto que implementa tanto la funcionalidad asociada con la interacción de usuario (*InteractionContent*) como la funcionalidad interna (*CoreContent*). Además, en dicha figura se puede observar una representación conceptual de cómo están estructurados el resto de bloques funcionales de un componente. En esta representación, el hecho de que un bloque sea adyacente a otro significa que ambos bloques pueden comunicarse entre sí. Por ejemplo, la interfaz de interacción (*InteractionInterface*) es la frontera para establecer la interacción con el usuario, y puede comunicarse tanto con la implementación de *InteractionContent* como con la interfaz que gestiona la comunicación con los componentes (*ControllerInterface*).

La Figura 2.14b muestra los puertos que el componente del ejemplo anterior debe implementar para dar solución a las interfaces descritas en la Figura 2.14a. Para el caso de las interfaces proporcionadas, las operaciones que sólo tienen entrada se representan

[6]Web Services Description Language – `http://www.w3.org/TR/wsdl`

© 2016 Vallecillos, J.

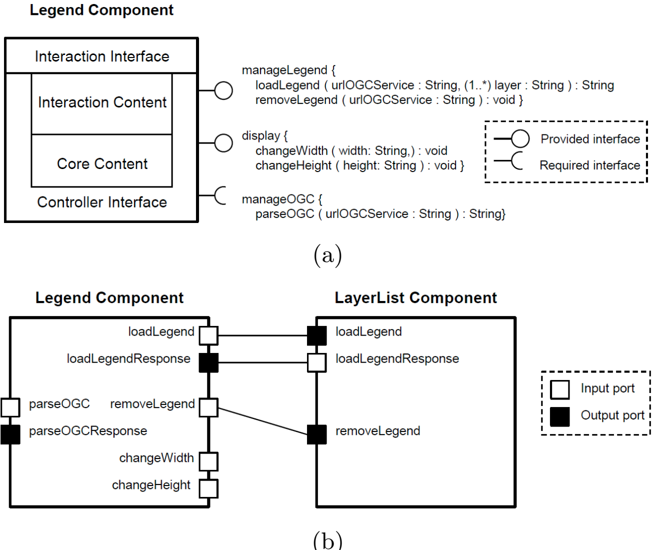


Figura 2.14: Estructura de un componente: (a) vista de interfaces y (b) vista de puertos

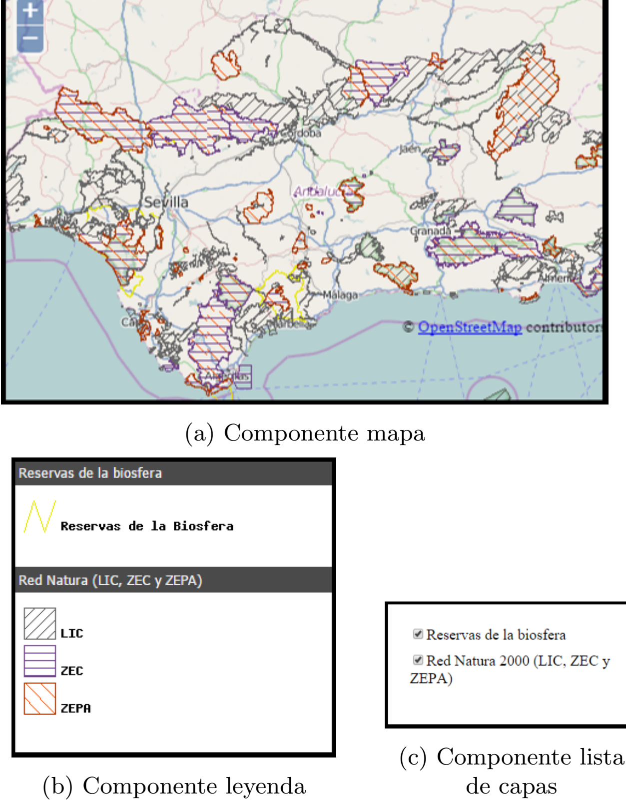
por un puerto de entrada (con el mismo nombre de la operación). Las operaciones que sí devuelven información como resultado se representan mediante un puerto de entrada (con el mismo nombre de la operación) y un puerto de salida (cuyo nombre es la combinación del nombre de la operación y del término *Response*). En el caso de las interfaces requeridas, la equivalencia entre operaciones y puertos se realiza al contrario. En la figura se incluye también un ejemplo de conexión entre los puertos de este componente con otro componente de la aplicación *mashup*, lo que deriva en la definición de las relaciones que existen entre los componentes de una arquitectura.

2.5. Escenarios de ejemplo base

Con el objetivo de explicar los diferentes tipos de dependencias entre los componentes de las aplicaciones *mashup*, así como las distintas relaciones que surgen de dichas dependencias, se van a utilizar dos escenarios de ejemplo. Por un lado, un escenario basado en un Sistema de Información Geográfica (SIG), donde existe un componente mapa sobre el cual se cargan capas geográficas, además de una leyenda sobre la que se observan las capas cargadas sobre el mapa. Para cargar la información sobre el mapa y la leyenda hay otro componente que contiene la lista de capas que se pueden cargar en el mapa. El segundo escenario consistirá en un sistema domótico que pretende controlar el encendido y apagado de una bombilla de una instalación domótica. El encendido o apagado de la bombilla se puede realizar tanto de manera física, tocando el interruptor, como de forma virtual interaccionando con un botón. A continuación, se describen ambos escenarios.

2.5.1. Escenario de mapas temáticos

Este primer escenario forma parte de un SIG y está constituido por una aplicación *mashup* de tipo web cuya arquitectura está formada por tres componentes (Figura 2.15). Por un lado, en el componente *Mapa* se cargan y visualizan distintas capas con información geográfica. Por otro lado, un componente *Leyenda* ofrece información acerca de las capas que han sido cargadas en el mapa. Por último, un componente *Lista de capas* se encarga de visualizar el conjunto de capas que están disponibles para ser visualizadas en el mapa, permitiendo seleccionarlas y deseleccionarlas (para su incorporación o eliminación, respectivamente). Para tener una idea de cuál es el aspecto de dichos componentes, la Figura 2.15a muestra el componente *Mapa*, la Figura 2.15b muestra el componente *Leyenda* y la Figura 2.15c muestra la *Lista de capas*. Existe un cuarto componente en este escenario, el componente *Contenedor* que no es visible. Su objetivo es albergar a varios componentes del entorno. Dependiendo de los componentes incluidos en el contenedor y de las relaciones establecidas entre estos se pueden establecer diferentes configuraciones de visualización entre componentes, descritas en las Secciones 2.6 y 2.7.



(a) Componente mapa

(b) Componente leyenda

(c) Componente lista de capas

Figura 2.15: Componentes del escenario base SIG

© 2016 Vallecillos, J.

2.5.2. Escenario de domótica

Para comprender con más detalle una arquitectura de componentes y sus relaciones se ha definido un segundo escenario de un sistema domótico sencillo. El problema que se pretende resolver en este escenario es controlar el encendido y apagado de una bombilla en un hogar. El encendido o apagado de la bombilla se puede realizar tanto de manera física (tocando el interruptor) como de forma virtual (interaccionando con un botón). Por lo tanto, la arquitectura que define la aplicación *mashup* de este escenario se compone del conjunto de componentes necesarios para resolver esta tarea (ver Figura 2.16).

Uno de estos componentes es el componente *Bombilla*, encargado de simular el estado actual de la bombilla. Otro es el componente *Interruptor* virtual para encender o apagar la bombilla dentro del entorno. Se dispone de un componente que controla el encendido y apagado procedente del entorno real o virtual, llamado *Controlador de luz*. Además, hay un componente llamado *RaspberryPi2GPIO* que se encarga de conectar el entorno real con el virtual. *RaspberryPi2GPIO* comunica los impulsos eléctricos de encendido y apagado con la parte virtual y transforma la información virtual en impulsos eléctricos. Por último, la arquitectura está compuesta por un componente *Logger* que registra la ejecución en un archivo de texto con el objetivo de seguir una traza de lo que sucede en la aplicación. Estos componentes no son visuales, sino que están ejecutándose en *background* en una aplicación *mashup* de tipo *Java*.

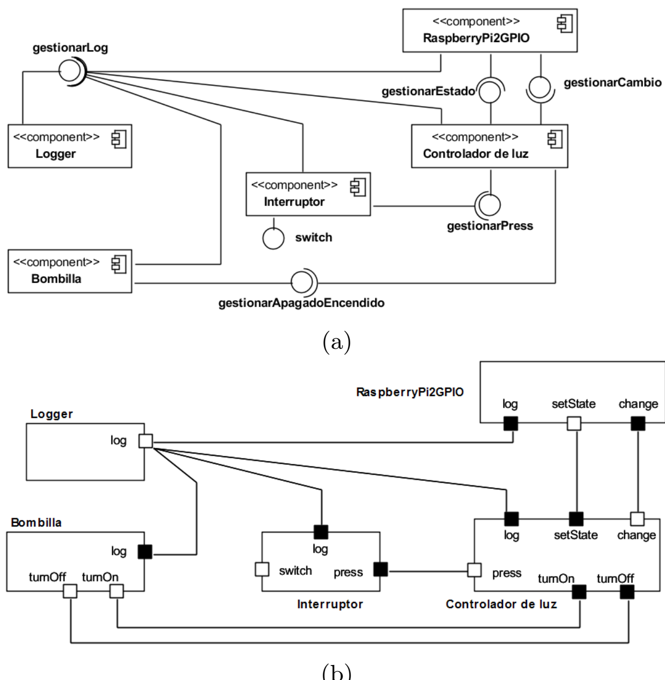


Figura 2.16: Escenario de domótica: (a) vista de interfaces y (b) vista de puertos

En la Figura 2.16a se muestran las interfaces de cada componente y cómo están relacionadas entre sí. En la Figura 2.16b se muestran los puertos de los componentes y cómo se conectan para permitir las comunicaciones entre componentes. Por ejemplo, se puede ver cómo la interfaz proporcionada `gestionarLog` del componente *Logger* tiene un método `log` que se resuelve mediante un puerto de entrada con el mismo nombre. El resto de componentes que hacen uso de la operación de dicha interfaz (a través de la conexión con una interfaz requerida) tienen un puerto de salida también nombrado como `log` y que se conectan con el puerto de entrada anterior. De igual forma, la interfaz proporcionada `gestionarApagadoEncendido` del componente *Bombilla* está formada por dos métodos (de sólo entrada) que se resuelven mediante dos puertos de entrada (`turnOn` y `turnOff`), los cuales están relacionados con dos puertos de salida (con el mismo nombre que los anteriores) de la interfaz requerida del componente *Controlador de luz*.

2.6. Dependencias entre componentes mashup

En las secciones anteriores de este capítulo se ha descrito ampliamente la infraestructura COScore construida para dar soporte a las aplicaciones *mashup* basadas en componentes COTSgets. Estos componentes pueden ser vistos como "cajas negras" puesto que se desconoce la funcionalidad interna que cada uno de ellos implementa. La única información disponible es el conjunto de interfaces (proporcionadas y/o requeridas) que cada componente ha definido a través de su correspondiente especificación. Esta visión de "caja negra" permite que un desarrollador tenga cierta libertad para organizar (o estructurar) una aplicación. Dependiendo de cómo se realice esta organización, las dependencias entre los componentes serán distintas e incluso, podrá influir en la calidad del producto final, afectando, por ejemplo, a aspectos relacionados con la usabilidad.

A lo largo de las próximas subsecciones se va a analizar esta organización de los componentes y su influencia en el resultado final de un producto. Para facilitar su comprensión, en todas las subsecciones se van a utilizar el escenario de mapas temáticos. Así, se empezará describiendo el espacio de trabajo donde tiene lugar la ejecución de los componentes. A continuación se describirán diferentes implementaciones de los componentes utilizados en el escenario de ejemplo. Estas implementaciones nos permitirán analizar una serie de patrones de composición de componentes que podrían utilizarse en el escenario. Por último, se mostrará una matriz de regeneración de dependencias que mostrará posibles soluciones que se podrían utilizar por parte de un desarrollador para tomar una buena decisión de diseño.

2.6.1. Espacio de trabajo y componentes

El espacio de trabajo de una aplicación (en adelante W), está formado por el entorno donde se visualizan los componentes con interfaz de usuario. Este entorno se encuentra en el sistema del cliente. En la sección anterior, se mostró que el sistema de mapas temáticos estaba formado por un conjunto de componentes interrelacionados, formados básicamente por los componentes Mapa, Leyenda y Lista de capas. Por supuesto, el sistema descrito permite que existan diversos mapas con sus respectivas leyendas y lista

© 2016 Vallecillos, J.

de capas. En una situación de este tipo (donde se muestren varios mapas con sus respectivas leyendas y lista de capas) el usuario podría tener cierta dificultad en identificar qué componentes están vinculados con otros. El COScore no puede detectar este tipo de circunstancias. Eso hace que dependiendo de cómo se establezcan las dependencias entre los componentes, el desarrollador podrá ofrecer diferentes soluciones para el usuario e influir directamente en la calidad del producto realizado.

Un ejemplo de esta situación aparece en la Figura 2.17. En dicha figura se muestra un W compuesto por dos mapas y dos listas de capas. Como se puede apreciar, existe una cierta dificultad en determinar qué lista pertenece a cada mapa. Esta circunstancia ocurre porque los cuatro componentes son independientes, desde el punto de vista visual, y permite que cada componente pueda ser posicionado en cualquier lugar del W. En la Figura 2.18 se muestra otro ejemplo de W donde aparecen los mismos componentes que en el caso anterior. Sin embargo, en este caso existe una dependencia "visual" entre cada lista de capas con su respectivo mapa. Esta dependencia provoca que en caso de mover un mapa, éste "arrastre" a su correspondiente lista de capas. Esta solución, en principio, no debe producir confusión en el usuario.

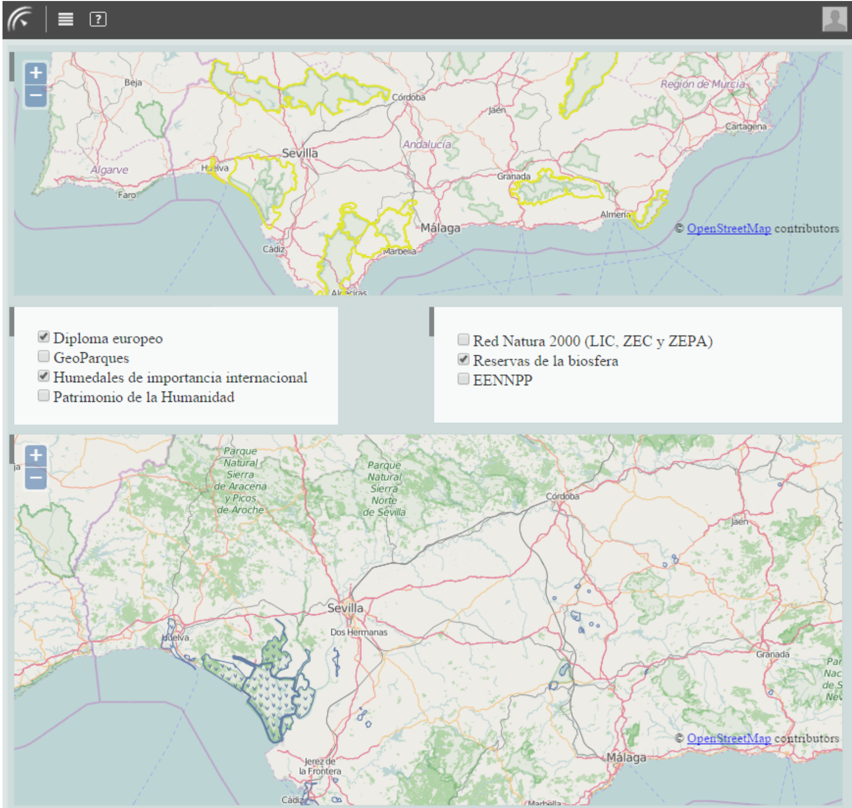


Figura 2.17: Aplicación *mashup* con dos mapas y dos listas de capas

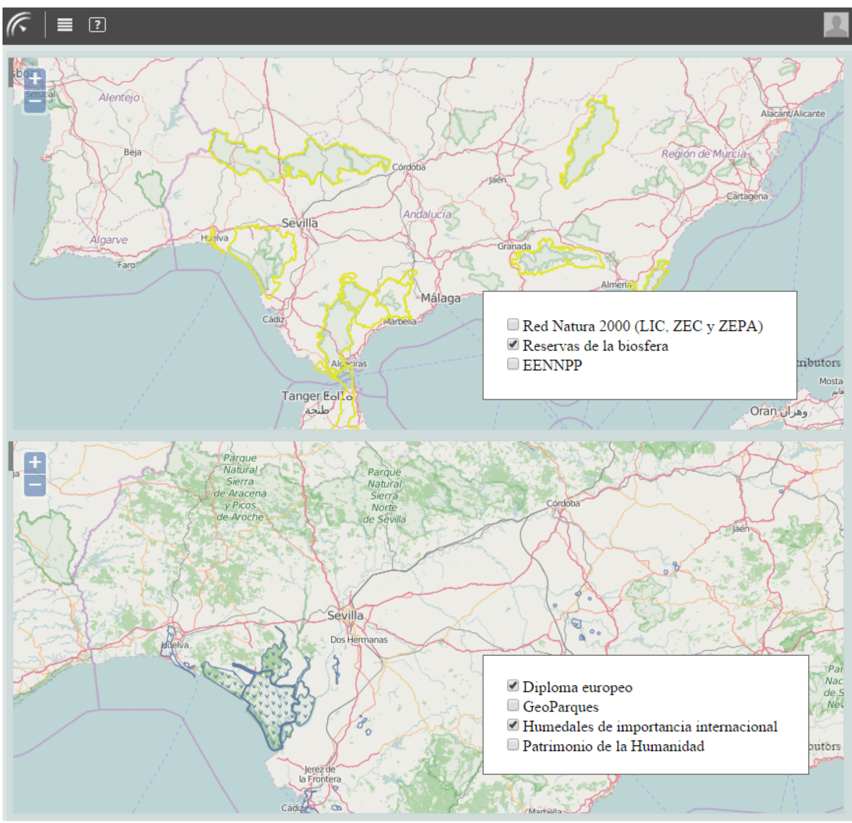


Figura 2.18: Aplicación *mashup* con dos mapas que contienen su lista de capas

En la Tabla 2.7 se pueden observar diferentes implementaciones de los componentes del escenario. Esta lista, no exahustiva, incluye 6 implementaciones del componente Mapa, 2 de Lista de capas, 1 de Leyenda y 3 de Contenedor. En la primera columna de la tabla se indica un código para identificar la implementación (*e.g.*, M4 representará la implementación 4 del componente Mapa), en la segunda columna se muestra el diagrama en UML del componente que se está implementando y, en la tercera columna, las interfaces que incluye dicho componente. A continuación vamos a describir brevemente cada una de las implementaciones.

La implementación M1 identifica a un componente Mapa que, en principio, puede funcionar aisladamente de otros componentes. Dicho componente incluye una interfaz proporcionada (llamada `gestionarCapas`) por donde se puede añadir/borrar (utilizando los métodos `addLayer`/`deleteLayer`) la capas mostradas en el mapa. La segunda implementación de un mapa (M2) es una extensión de la anterior que incluye una interfaz requerida (llamada `gestionarLista`) utilizada para indicar a un componente Lista de capas las capas que están visibles en el mapa. Esta interfaz utiliza los métodos `addLayerList` y `deleteLayerList` para dar dicha información. La implementación

© 2016 Vallecillos, J.

M3 es similar a M2, sin embargo, en este caso la información que envía el mapa (a través de los métodos `addLegend` y `deleteLegend` de la interfaz `gestionarLeyenda`) está relacionada con la leyenda de cada capa. La implementación M4 es la unión de M2 y M3, donde un componente Mapa podrá pasar información a los componentes Lista de capas y Leyenda. Las implementaciones M5 y M6 son muy similares a M4 en cuanto a funcionalidad. La diferencia aparece en que los mapas de M5 y M6 se comunican con componentes Contenedor para informarles de la lista de capas y leyendas actuales respectivamente.

Id.	Componente	Interfaces

M1	gestionarCapas
	<<component>>
	Mapa
		gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}

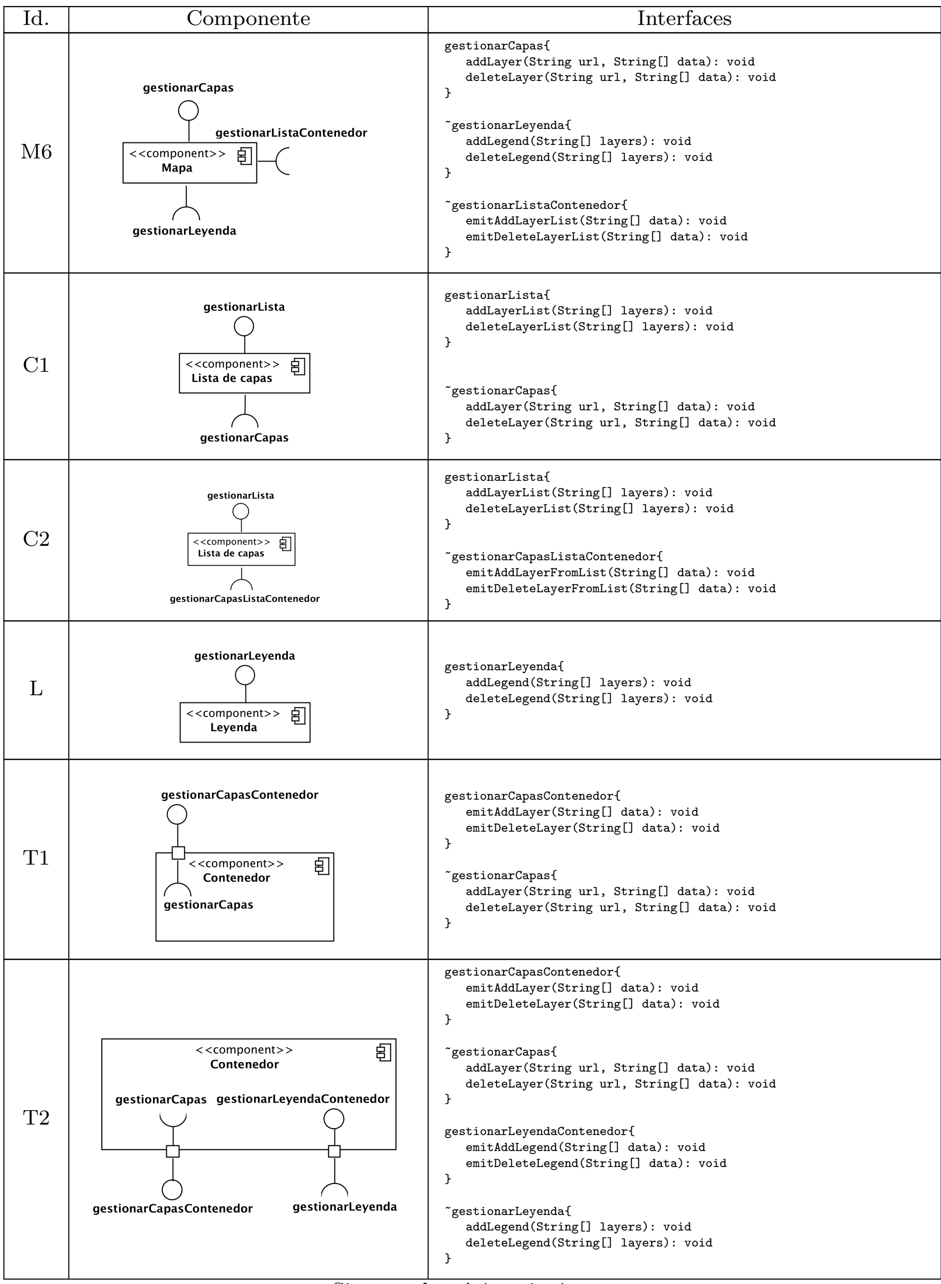
M2	gestionarCapas
	<<component>>
	Mapa
	gestionarLista
		gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		~gestionarLista{
		addLayerList(String[] layers): void
		deleteLayerList(String[] layers): void
		}

M3	gestionarCapas
	<<component>>
	Mapa
	gestionarLeyenda
		gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		~gestionarLeyenda{
		addLegend(String[] layers): void
		deleteLegend(String[] layers): void
		}

M4	gestionarCapas
	<<component>>
	Mapa
	gestionarLista
	gestionarLeyenda
		gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		~gestionarLeyenda{
		addLegend(String[] layers): void
		deleteLegend(String[] layers): void
		}
		~gestionarLista{
		addLayerList(String[] layers): void
		deleteLayerList(String[] layers): void
		}

M5	gestionarCapas
	<<component>>
	Mapa
	gestionarLista
	gestionarLeyendaContenedor
		gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		~gestionarLeyendaContenedor{
		emitAddLegend(String[] data): void
		emitDeleteLegend(String[] data): void
		}
		~gestionarLista{
		addLayerList(String[] layers): void
		deleteLayerList(String[] layers): void
		}

Sigue en la página siguiente.



Id.	Componente	Interfaces

M6	gestionarCapas
	gestionarListaContenedor
	<<component>>
	Mapa
	gestionarLeyenda
		gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		~gestionarLeyenda{
		addLegend(String[] layers): void
		deleteLegend(String[] layers): void
		}
		~gestionarListaContenedor{
		emitAddLayerList(String[] data): void
		emitDeleteLayerList(String[] data): void
		}

C1	gestionarLista
	<<component>>
	Lista de capas
	gestionarCapas
		gestionarLista{
		addLayerList(String[] layers): void
		deleteLayerList(String[] layers): void
		}
		~gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}

C2	gestionarLista
	<<component>>
	Lista de capas
	gestionarCapasListaContenedor
		gestionarLista{
		addLayerList(String[] layers): void
		deleteLayerList(String[] layers): void
		}
		~gestionarCapasListaContenedor{
		emitAddLayerFromList(String[] data): void
		emitDeleteLayerFromList(String[] data): void
		}

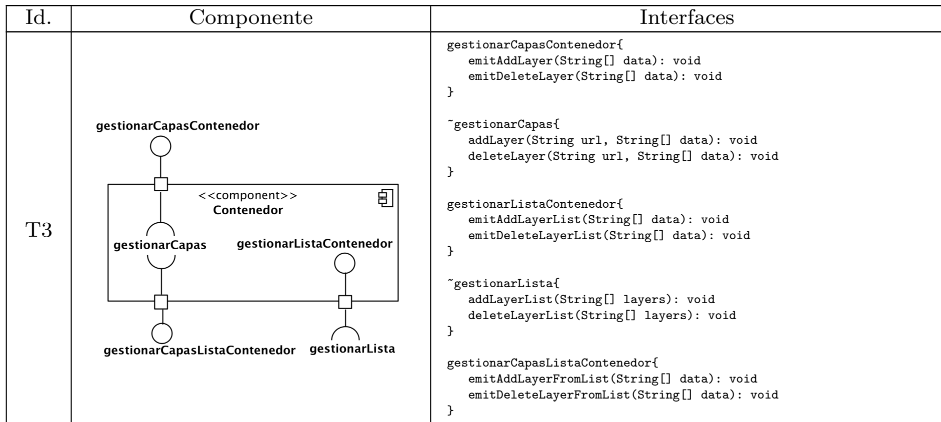
L	gestionarLeyenda
	<<component>>
	Leyenda
		gestionarLeyenda{
		addLegend(String[] layers): void
		deleteLegend(String[] layers): void
		}

T1	gestionarCapasContenedor
	<<component>>
	Contenedor
	gestionarCapas
		gestionarCapasContenedor{
		emitAddLayer(String[] data): void
		emitDeleteLayer(String[] data): void
		}
		~gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}

T2	<<component>>
	Contenedor
	gestionarCapas
	gestionarLeyendaContenedor
	gestionarCapasContenedor
	gestionarLeyenda
		gestionarCapasContenedor{
		emitAddLayer(String[] data): void
		emitDeleteLayer(String[] data): void
		}
		~gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		gestionarLeyendaContenedor{
		emitAddLegend(String[] data): void
		emitDeleteLegend(String[] data): void
		}
		~gestionarLeyenda{
		addLegend(String[] layers): void
		deleteLegend(String[] layers): void
		}

Sigue en la página siguiente.

© 2016 Vallecillos, J.



Id.	Componente	Interfaces

T3	gestionarCapasContenedor
	<<component>>
	Contenedor
	gestionarCapas
	gestionarListaContenedor
	gestionarCapasListaContenedor
	gestionarLista
		gestionarCapasContenedor{
		emitAddLayer(String[] data): void
		emitDeleteLayer(String[] data): void
		}
		
		~gestionarCapas{
		addLayer(String url, String[] data): void
		deleteLayer(String url, String[] data): void
		}
		
		gestionarListaContenedor{
		emitAddLayerList(String[] data): void
		emitDeleteLayerList(String[] data): void
		}
		
		~gestionarLista{
		addLayerList(String[] layers): void
		deleteLayerList(String[] layers): void
		}
		
		gestionarCapasListaContenedor{
		emitAddLayerFromList(String[] data): void
		emitDeleteLayerFromList(String[] data): void
		}

Tabla 2.7: Componentes gráficos SIG base y sus interfaces

El componente Lista de capas incluye 2 implementaciones. La implementación C1 utiliza la interfaz proporcionada `gestionarLista`, para obtener la lista de capas a desplegar en la interfaz de usuario, y la interfaz requerida `gestionarCapas`, para comunicar a un componente Mapa la lista de capas que puede visualizar. Con esta implementación, el componente Lista de capas será el encargado de controlar al componente Mapa, al contrario de lo que representa la implementación M2 (donde el componente Mapa tiene el control e informa al componente Lista de capas). La segunda implementación (C2) es similar a la anterior, pero en este caso, la interfaz requerida `gestionarCapasContenedor` emite información a un contenedor.

La implementación L describe un componente Leyenda con la interfaz proporcionada `gestionarLeyenda` por donde se obtiene la información relacionada con cada capa. En el escenario de ejemplo propuesto, un componente leyenda de este tipo siempre estará vinculado a otro componente que haga uso de su interfaz proporcionada para añadir y/o eliminar información de leyenda, por ejemplo, de las capas que se están visualizando.

La implementación T1 describe un componente Contenedor formado por un componente Mapa. Su única interfaz visible al exterior es la interfaz proporcionada `gestionar-CapasContenedor`. Esta interfaz se utiliza para proporcionar al componente Mapa la información necesaria para su actividad. Por supuesto, este contenedor puede agrupar a otros componentes (distintos del mapa). Sin embargo, estos componentes sólo podrán comunicarse internamente entre sí, o con el componente Mapa, pero no con el exterior. La implementación T2 describe un componente Contenedor formado por, al menos, los componentes Mapa y Leyenda. Este contenedor obtiene información para la leyenda a través de la interfaz `gestionarLeyenda` (utilizando los métodos `addLegend` y `deleteLegend`) y ofrece información del componente Mapa a un componente Lista de capas externo al contenedor, a través de la interfaz `gestionarCapasContenedor` (utilizando los métodos `emitAddLayer` y `emitDeleteLayer`). Por último, tenemos la implementación T3. Esta implementación es una extensión del componente T2, pero incluyendo al componente Lista de capas como parte del contenedor. Para que este tipo de componente pueda obtener información del exterior, el contenedor incluye la interfaz

`gestionarCapasListaContenedor` (formado por los métodos `emitAddLayerFromList` y `emitDeleteLayerFromList`).

A partir de las implementaciones descritas anteriormente, un desarrollador podrá crear un W con diferentes situaciones. Analicemos en la siguiente subsección algunos de los casos posibles y ver cómo influye en el resultado final de un producto.

2.6.2. Patrones de composición de componentes

A partir de los ejemplos de implementación descritos anteriormente, podemos establecer diferentes combinaciones entre los componentes. Un efecto inmediato de estas combinaciones es que la ubicación de determinados componentes en la interfaz de usuario pueden variar. Así, según las combinaciones establecidas, un componente puede ser libremente colocado en cualquier posición de W. Por otro lado, su posición puede depender de la posición relativa de otro componente. A estas combinaciones se las ha llamado *patrones de visualización*. En la Tabla 2.8 podemos ver algunos de los posibles patrones que pueden aparecer a partir del escenario de ejemplo.

Patrón	Nombre	Estilo
#1	Unitario	M
#2	Binario independiente tipo A	M C
#3	Binario independiente tipo B	M L
#4	Ternario independiente	M C L
#5	Binario dependiente tipo A	T M L
#6	Binario dependiente tipo B	T M C
#7	Ternario parcialmente dependiente tipo A	T M C L

Sigue en la página siguiente.

© 2016 Vallecillos, J.

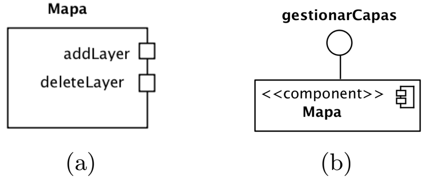
Patrón	Nombre	Estilo
#8	Ternario parcialmente dependiente tipo B	T M L C
#9	Ternario dependiente	T M C L

Tabla 2.8: Patrones de visualización

A continuación se describe con detalle cada uno de los patrones de la Tabla 2.8. Para cada uno de ellos se muestran dos figuras, describiendo los componentes implicados tanto en la vista de puertos con sus correspondientes conectores, como en la vista de las interfaces con sus correspondientes dependencias. En las figuras correspondientes a las vistas de puertos se utilizan el símbolo □ para denotar un puerto de entrada, ■ para un puerto de salida, y los símbolos □■ y ■□ para agrupar una entrada y una salida o una salida y una entrada, respectivamente.

Patrón de visualización #1: unitario

El patrón de visualización unitario (Figura 2.19) es aquel que está formado únicamente por el mapa (utilizando la implementación M1) y no tiene más componentes conectados. En este mapa se pueden cargar las capas geográficas. En la Figura 2.19a se puede observar los puertos de entrada `removeLayer` y `loadLayer` asociados con este componente, que se corresponderán con los métodos `addLayer` y `deleteLayer` de la interfaz proporcionada `gestionarCapas` de la implementación M1.



(a) (b)

Figura 2.19: Patrón de visualización #1: (a) vista de puertos y (b) vista de interfaces

Patrón de visualización #2: binario independiente tipo A

Para el patrón de visualización binario independiente de tipo A (Figura 2.20), el componente Lista de capas (implementación C1 según Tabla 2.8) se conecta con el componente Mapa (M2). Estos componentes están relacionados entre sí por medio de dos interfaces, la interfaz `gestionarLista` y la interfaz `gestionarCapas` (Figura 2.20b). La interfaz `gestionarLista` tiene los métodos `addLayerList` y `deleteLayerList`, y la interfaz

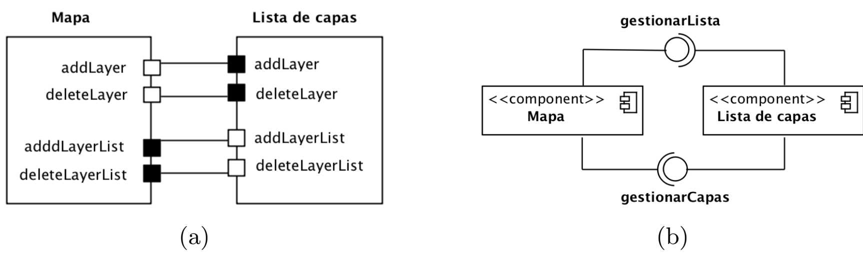


Figura 2.20: Patrón de visualización #2: (a) vista de puertos y (b) vista de interfaces

`gestionarCapas` está formada por los métodos `addLayer` y `deleteLayer`. Los puertos asociados con estos métodos se encuentran en la Figura 2.20a, junto con sus conectores.

En este patrón, el componente Mapa tiene cuatro puertos. El puerto de salida `addLayerList` (correspondiente al método `addLayerList` de la interfaz `gestionarLista`), está conectado con el puerto de entrada `addLayerList` del componente Lista de capas. De igual forma, se conecta el puerto `deleteLayerList` del mapa con su homólogo. Por otro lado, el puerto de entrada `addLayer` (correspondiente al método `addLayer` de la interfaz `gestionarCapa`), está conectado con el puerto de salida `addLayer` del componente Lista de capas. De igual forma se conecta el puerto `deleteLayer` del mapa con su homólogo.

Patrón de visualización #3: binario independiente tipo B

En el patrón de visualización binario independiente de tipo B (Figura 2.21), el componente Mapa (M3) está relacionado con la Leyenda (L). Para ello, la interfaz proporcionada `gestionarLeyenda` del componente Leyenda está conectada con la interfaz requerida ∼`gestionarLeyenda` del componente Mapa. Para comprender mejor la relación que existe entre cada componente, se puede observar la Figura 2.21a, dónde se pueden ver los componentes Mapa y Leyenda conectados entre sí a través de los puertos. El componente Mapa tiene cuatro puertos y el componente Leyenda tiene dos puertos. En el componente Mapa el puerto de salida `addLegend` permite ejecutar la función `addLegend` a través de la interfaz ∼`gestionarLegenda`. El puerto de salida `deleteLegend` capacita a la función `deleteLegend` para que pueda ser ejecutada por medio de la interfaz

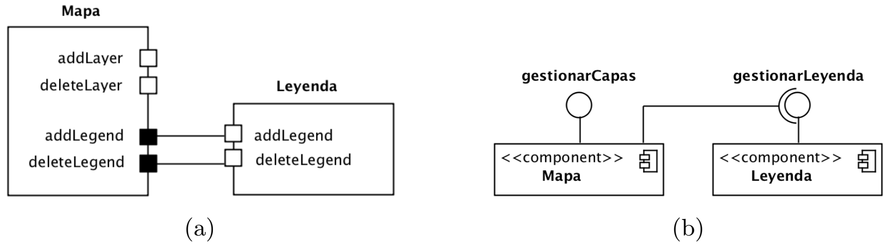


Figura 2.21: Patrón de visualización #3: (a) vista de puertos y (b) vista de interfaces

© 2016 Vallecillos, J.

∼`gestionarLegenda`. En Leyenda, el puerto de entrada `addLegend` posibilita la entrada de información al método `addLegend` de la interfaz `gestionarLeyenda`. Por último, el puerto `deleteLegend` permite el paso de información al método `deleteLegend` de la interfaz `gestionarLeyenda`.

Patrón de visualización #4: ternario independiente

En este patrón (Figura 2.22) se observa que hay tres componentes: Mapa (M4), Leyenda (L) y Lista de capas (C1). El componente Mapa está relacionado con la Lista de capas a través de las interfaces `gestionarLista` y `gestionarCapas` y con la Leyenda por medio de la interfaz `gestionarLeyenda` (Figura 2.22b). Estas relaciones han sido descritas en los patrones #2 (Figura 2.20) y #3 (Figura 2.21). En dichos patrones se especificaron los métodos que forman cada interfaz y cómo han sido conectados estos métodos a través de los puertos.

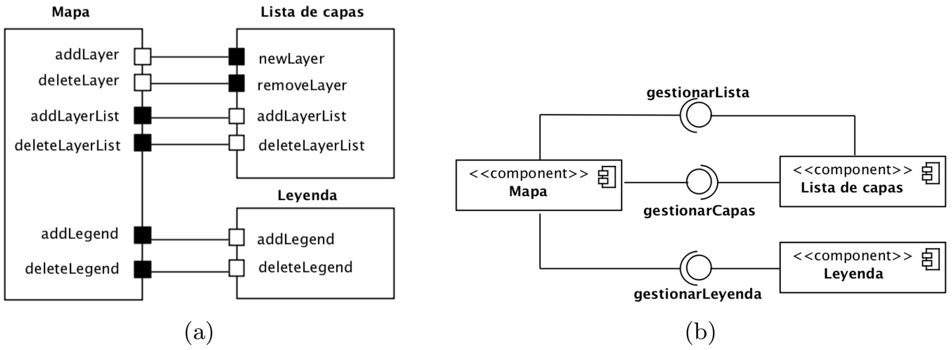


Figura 2.22: Patrón de visualización #4: (a) vista de puertos y (b) vista de interfaces

Patrón de visualización #5: binario dependiente tipo A

El patrón de visualización binario dependiente de tipo A (Figura 2.23) se encuentra relacionado con los componentes Mapa (M3) y Leyenda (L) y, como se puede observar, la Leyenda y el mapa están dentro de un componente Contenedor (T2). El componente Contenedor tiene una interfaz proporcionada llamada `gestionarCapasContenedor` con los métodos `emitAddLayer` y `emitDeleteLayer` que gestiona las capas a cargar en el componente Mapa (Figura 2.23b). Fijándonos en la Figura 2.23a, se establece una conexión entre los puertos `deleteLayer`, `addLayer` (asociados a la interfaz proporcionada `gestionarCapas` de un componente Mapa) con `deleteLayer` y `addLayer` (asociados a la interfaz requerida ∼`gestionarCapas` del componente Contenedor). El puerto de entrada `deleteLayer` permite obtener información del puerto `emitDeleteLayer` (a través del puerto de salida `deleteLayer` del Contenedor) y el puerto de entrada `addLayer` permite la obtención de información del puerto `emitAddLayer` (a través del puerto de salida `addLayer` del Contenedor). La otra relación que hay en este patrón, es la relación del componente Mapa con el de Leyenda que se ha descrito en el patrón #2.

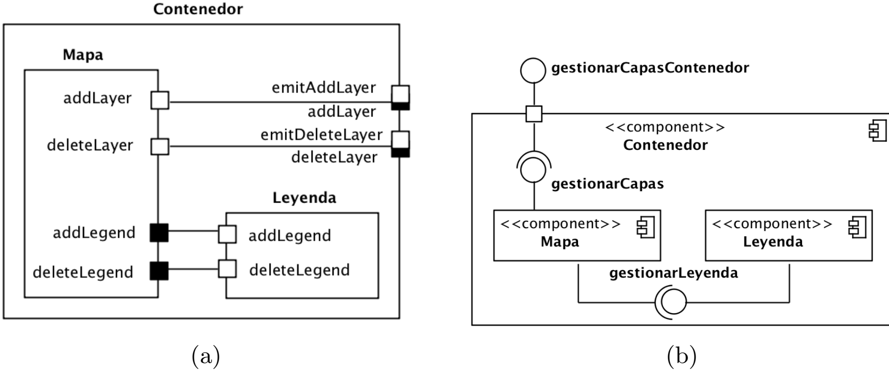


Figura 2.23: Patrón de visualización #5: (a) vista de puertos y (b) vista de interfaces

Patrón de visualización #6: binario dependiente tipo B

En este patrón (Figura 2.24) se observa que el componente Lista de capas (C1) está junto al componente Mapa (M2) dentro de un componente Contenedor (T1). La interfaz proporcionada `gestionarCapasContenedor` del componente Contenedor ha sido descrita en el patrón #5. Las interfaces `gestionarCapas` y `gestionarLista` que vinculan los componentes Mapa y Lista de capas se describieron en el patrón #2.

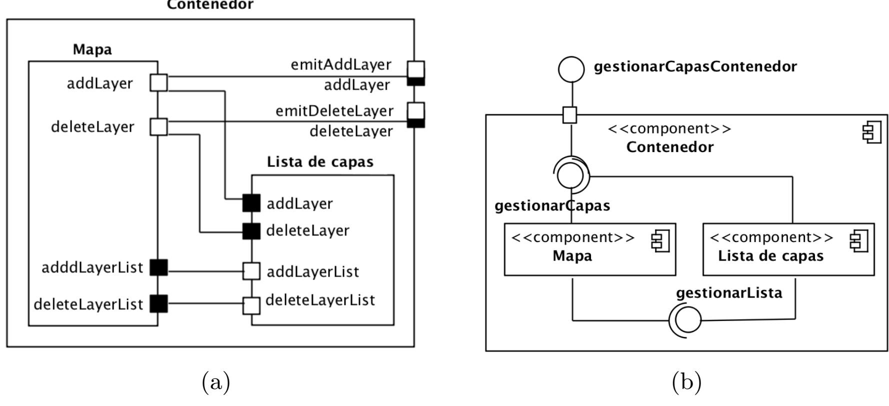


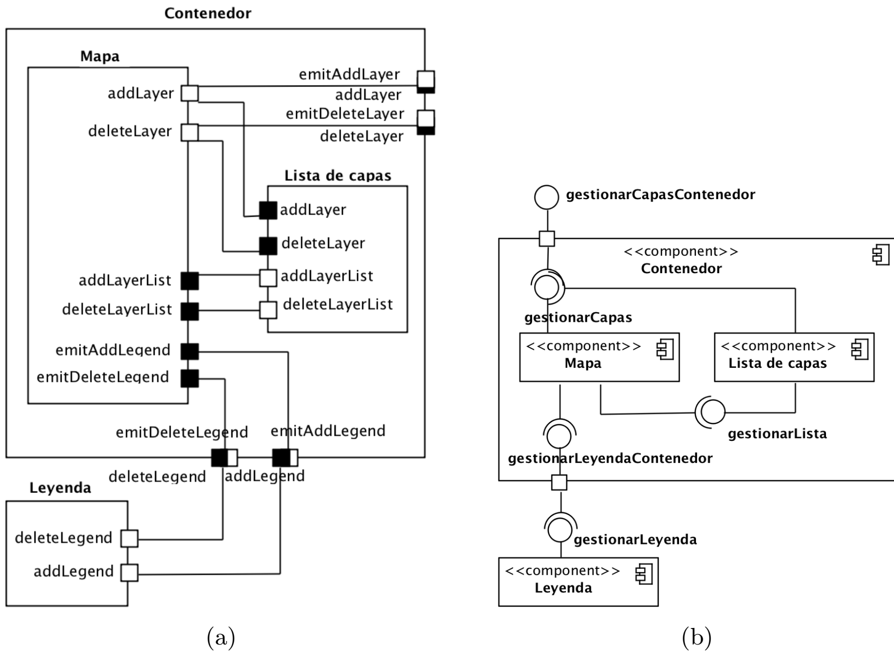
Figura 2.24: Patrón de visualización #6: (a) vista de puertos y (b) vista de interfaces

Patrón de visualización #7: ternario parcialmente dependiente tipo A

En el patrón de visualización ternario parcialmente dependiente de tipo A (Figura 2.25) aparece el componente Lista de capas (C1) junto con el componente Mapa (M5) agrupados en un contenedor y la Leyenda (L) fuera de dicho contenedor (Figura 2.25b). El componente Contenedor (T2) tiene una interfaz requerida llamada `gestionarLe-`

© 2016 Vallecillos, J.

`yendaContenedor` formada por los métodos `emitAddLegend` y `emitDeleteLegend`. Esta interfaz conecta el Mapa con la Leyenda usando como mediador el componente Contenedor. Los puertos que hacen posible el funcionamiento de esta interfaz son el puerto de entrada `emitAddLegend` (vinculado con el puerto de salida `addLegend` de la interfaz requerida `gestionarLeyenda`) y el puerto de entrada `emitDeleteLegend` (vinculado con el puerto de salida `deleteLegend` de la interfaz requerida `gestionarLeyenda`), tal y como muestra la Figura 2.25a. El resto de interfaces han sido descritas en los patrones #5 y #6.



(a) (b)

Figura 2.25: Patrón de visualización #7: (a) vista de puertos y (b) vista de interfaces

Patrón de visualización #8: ternario parcialmente dependiente tipo B

En este patrón (Figura 2.26) hay tres componentes: Mapa (M6), Leyenda (L) y Lista de capas (C2). Las dos primeras se encuentran dentro de un contenedor (T3). Para que se puedan comunicar entre sí los componentes Mapa y Lista de capas, el componente Contenedor tiene una serie de interfaces (Figura 2.26b).

En este patrón aparecen dos nuevas interfaces no tratadas anteriormente, la interfaz `gestionarCapasListaContenedor` y `gestionarListaContenedor`. La primera está formada por el puerto de entrada `emitAddLayerFromList` (vinculado con el puerto de salida `addLayer` de la interfaz requerida `gestionarCapas`) y el puerto de entrada `emitDeleteLayerFromList` (vinculado con el puerto de salida `deleteLayer` de la interfaz requerida `gestionarCapas`). Por otro lado, la interfaz `gestionarListaContenedor`

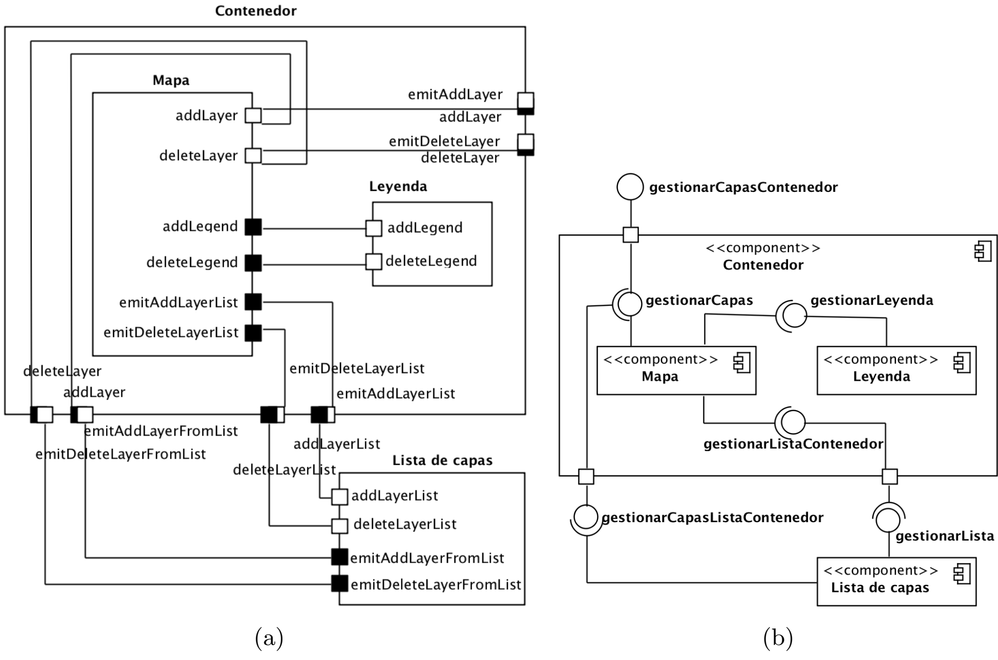


Figura 2.26: Patrón de visualización #8: (a) vista de puertos y (b) vista de interfaces

está formada por los puertos de salida `addLayerList` y `deleteLayerList` (vinculados con los puertos de entrada `emitAddLayerList` y `emitDeleteLayerList`, respectivamente). El resto de interfaces han sido descritas con anterioridad en los patrones #2 y #5.

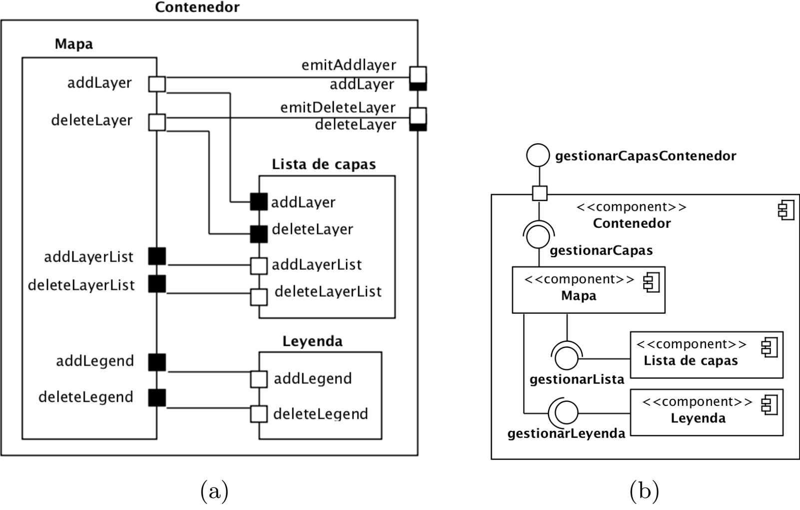
Patrón de visualización #9: ternario dependiente

En este patrón (Figura 2.27) se utilizan los componentes Lista de capas (C1), Leyenda (L) y Mapa (M4) en un componente Contenedor (T1). Las interfaces (Figura 2.27b) `gestionarCapas`, `gestionarLista` y `gestionarLeyenda` han sido descritas en el patrón #4. Por otro lado la interfaz `gestionarCapasContenedor` se describió en el patrón #5. En la Figura 2.27a se pueden ver los puertos de los componentes.

2.6.3. Matriz de regeneración de dependencias

En la subsección anterior se han descrito una serie de patrones que identifican diferentes "combinaciones" de relación entre los componentes del escenario de ejemplo. Siguiendo con dicho escenario, ¿qué ocurre si creamos un W con varios patrones a la vez? ¿pueden presentarse situaciones no deseadas? En esta subsección se va a analizar esta situación. Para facilitar su comprensión se va a establecer una restricción, que el W tenga un máximo de dos patrones a la vez. Teniendo en cuenta esta restricción, se puede presentar 81 W distintos, como se muestra en la Tabla 2.9. El cruce entre el patrón de la fila con el

© 2016 Vallecillos, J.



(a) (b)

Figura 2.27: Patrón de visualización #9: (a) vista de puertos y (b) vista de interfaces

patrón de la columna representa la propuesta de W que un desarrollador podría utilizar en lugar de los patrones iniciales.

Esta propuesta se realiza para obtener un producto que no genere confusión en el usuario. Como ya se indicó anteriormente, esta confusión puede deberse a la existencia de componentes que, por su posición en W, no se tenga claro con qué otros componentes están relacionados, como ocurría en el ejemplo mostrado en la Figura 2.17. Si se tiene en cuenta la propuesta indicada en la tabla, el W a establecer puede no ser el que inicialmente había pensado el desarrollador, aunque el funcionamiento de los componentes implicados no tiene porqué cambiar.

Analicemos un caso concreto de la Tabla 2.9. Supongamos que inicialmente deseamos formar un W con los patrones #4 y #5. La combinación de estos patrones genera una posible confusión dado que no queda claro con qué Mapa está vinculada la Lista de capas. Según la propuesta definida en la tabla, para esos patrones se propone 3 posibles W: (1) un W formado por los patrones #5 y #7; (2) un W formado por los patrones #5 y #9; y (3) un W formado por los patrones #3 y #9. Por ejemplo, el W formado por los patrones #5 y #7 resuelve esta confusión incorporando la Lista de capas dentro del espacio visual (con un Contenedor) de uno de los mapas.

2.7. Relaciones entre componentes mashup

En las secciones previas, se ha descrito la estructura interna de un componente, examinando sus partes e identificando los diferentes tipos de componente. Como hemos visto, entre componentes pueden existir diversos tipos de dependencias, las cuales han dado lugar a un conjunto de patrones de dependencia. A partir de estas dependencias se establecen las relaciones entre componentes, que serán descritas en la presente sección.

	P1	P2	P3	P4	P5	P6	P7	P8	P9
P1	{P1,P1}	{P1,P6}	{P1,P5}	{P1,P9}	{P1,P5}	{P1,P6}	{P1,P7}	{P1,P8}	{P1,P9}
P2	{P1,P6}	{P2,P6} {P6,P6}	{P5,P6}	{P5,P7} {P5,P9} {P2,P9}	{P5,P6}	{P2,P6} {P6,P6}	{P9,P2} {P9,P6} {P8,P6}	{P9,P2} {P9,P6} {P8,P6}	{P5,P6} {P9,P6} {P8,P6} {P4,P6}
P3	{P1,P5}	{P5,P6}	{P3,P5} {P5,P5}	{P3,P9} {P5,P9} {P5,P7}	{P5,P5} {P5,P3}	{P5,P6}	{P9,P3} {P9,P5} {P7,P5}	{P9,P3} {P9,P5} {P7,P5}	{P9,P3} {P9,P5} {P7,P5}
P4	{P1,P9}	{P9,P6} {P8,P6} {P9,P2}	{P9,P5} {P7,P5} {P9,P3}	{P9,P9} {P4,P9} {P8,P9} {P7,P9}	{P7,P5} {P5,P9} {P3,P9}	{P8,P6} {P6,P9} {P2,P9}	{P9,P9} {P7,P9} {P8,P9} {P4,P9}	{P9,P7} {P9,P4} {P8,P9} {P9,P9}	{P9,P9} {P9,P7} {P8,P9} {P4,P9}
P5	{P1,P5}	{P5,P6}	{P3,P5} {P5,P5}	{P5,P7} {P5,P9} {P3,P9}	{P5,P5} {P5,P3}	{P5,P6}	{P7,P5} {P9,P5} {P9,P3}	{P5,P7} {P5,P9} {P3,P9}	{P9,P5} {P9,P3} {P5,P7}
P6	{P1,P6}	{P6,P6}	{P6,P5}	{P8,P6} {P9,P6} {P9,P2}	{P5,P6}	{P6,P6} {P2,P6}	{P9,P6} {P8,P6} {P2,P9}	{P8,P6} {P9,P6} {P9,P2}	{P9,P6} {P9,P2} {P6,P8}
P7	{P1,P9}	{P2,P9} {P6,P8} {P6,P9}	{P3,P9} {P5,P9} {P5,P7}	{P9,P8} {P9,P4} {P9,P7} {P9,P9}	{P5,P7} {P9,P5} {P9,P3}	{P8,P6} {P9,P6} {P9,P2}	{P9,P7} {P9,P4} {P9,P8} {P9,P9}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}
P8	{P1,P9}	{P2,P9} {P6,P9} {P6,P8}	{P3,P9} {P5,P9} {P5,P7}	{P9,P8} {P9,P4} {P9,P7} {P9,P9}	{P5,P9} {P5,P7} {P9,P3}	{P8,P6} {P9,P6} {P9,P2}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}
P9	{P1,P9}	{P2,P9} {P6,P9} {P6,P8}	{P3,P9} {P5,P9} {P5,P7}	{P9,P9} {P9,P8} {P9,P7} {P9,P4}	{P5,P9} {P9,P3} {P5,P7}	{P9,P6} {P8,P6} {P2,P9}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}	{P9,P9} {P9,P7} {P9,P8} {P9,P4}

Tabla 2.9: Matriz de regeneración de dependencias

Como se ha visto con anterioridad, una relación establece un proceso de comunicación entre un conjunto de componentes. Cada relación (concepto de *Relationship* en la Figura 2.11) conecta un componente con otro(s) a través de la entrada y salida de puertos, es decir, si hay una relación entre dos componentes, al menos un conjunto de puertos de salida de un componente debe ser conectado a un conjunto de puertos de entrada del otro componente. Las relaciones entre los componentes pueden ser binarias o n-arias. Para la descripción de ambos de tipos de relaciones, se van a utilizar los dos escenarios presentados anteriormente.

2.7.1. Relaciones binarias

Las relaciones binarias son las relaciones que hay entre dos componentes diferentes. Estas relaciones tienen una propiedad *booleana* llamada *isBidirectional*, para indicar si los componentes involucrados en la conexión se necesitan mutuamente (en cuyo caso, la relación se establece en ambos sentidos). Las relaciones binarias también incluyen el concepto *BinaryType* para identificar la existencia de tipos de relaciones binarias y así diferenciar tipos de comunicación entre componentes. En la Tabla 2.10 se pueden ver las relaciones binarias y los símbolos que las representan.

© 2016 Vallecillos, J.

Nombre	Símbolo
Asociación	$\approx$
Composición	$\sqsupset$
Subordinación	$\gg$
Herencia	$\ni$
Productor-consumidor	$\vdash$

Tabla 2.10: Principales relaciones binarias

Para ayudar a entender el significado de estas relaciones, en la Figura 2.28 se observan cómo están relacionados entre sí los componentes de los escenarios descritos anteriormente. La relación de **Asociación** es el enlace menos restrictivo que existe cuando una relación se establece por defecto entre dos componentes A y B (Figura 2.28a). Esta relación se usa para representar el intercambio regular de información entre dos componentes, donde dicho intercambio no puede ser descrito con otra relación. Un ejemplo relacionado con el escenario de información geográfica podría ser cuando un componente de *Inicio de sesión* se conecta con el componente *Lista de capas*, entonces la lista de capas cargadas cambia su contenido. De esta forma, las capas que se ven están relacionadas con el usuario que inició la sesión. En esta relación, si el componente de inicio de sesión no está disponible o no se encuentra en la arquitectura, el componente de lista de capas mostrará una lista de capas por defecto. Este comportamiento de dependencia *débil* es lo que caracteriza este tipo de relación.

La relación de **Composición** (Figura 2.28b) se presenta cuando todas las interfaces de B están en A. De tal manera que cuando algún componente desea acceder a B necesita pasar a través de A. Esta relación se utiliza para crear un componente a partir de otros componentes. De esta forma, un componente puede contener uno o más componentes, que juntos, desarrollarán una tarea conjunta. Un ejemplo de relación de composición aparece entre el componente *Contenedor* y los componentes *Mapa* y *Leyenda*. El com-

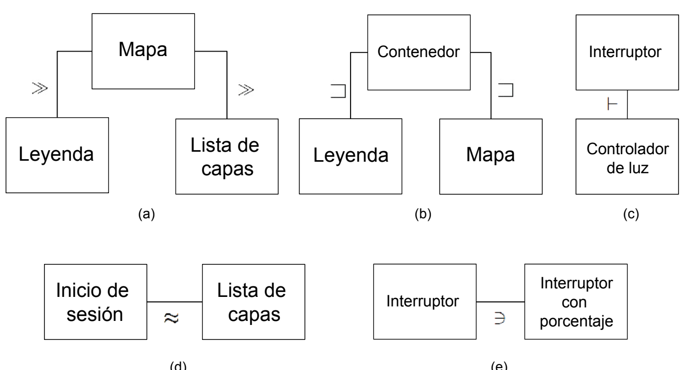


Figura 2.28: Relaciones binarias entre los componentes de los escenarios de ejemplo

ponente *Contenedor* es de tipo *contenedor* y alberga otros dos componentes. Así, el uso que se realice de las interfaces proporcionadas y requeridas (la invocación de sus métodos) de los componentes *Mapa* y *Leyenda* se canaliza a través del *Contenedor*.

La relación de **Subordinación** (Figura 2.28c) aparece cuando un componente A no puede existir sin otro componente B. En nuestro escenario ejemplo, esta relación aparece entre el componente *Mapa* y los componentes *Leyenda* y *Lista de capas*. El componente *Leyenda* y el componente *Lista de capas* están subordinados al *Mapa*. Existe subordinación porque sin la información procedente del mapa es imposible visualizar información relacionada con las capas dentro del componente *Leyenda*. De igual forma, el componente *Lista de capas* no podría cargar las capas para ser visualizadas u ocultadas. Este comportamiento de dependencia *fuerte* caracteriza una relación de subordinación.

La relación de **Herencia** (Figura 2.28d) aparece cuando un componente A incluye todos los puertos que han sido definidos en B, con el objetivo de extender su funcionalidad. Un ejemplo, relacionado con el escenario de domótica, es cuando existe un *Interruptor regulable* (*dimmer*) que hereda toda la funcionalidad de un *Interruptor* básico. La funcionalidad del *Interruptor* básico es apagar y encender, mientras que un *Interruptor regulable* permite, además, encender la bombilla con un nivel de intensidad específico.

La última relación binaria que se describe es la relación **Productor-consumidor** (Figura 2.28e). Esta relación se da cuando existe una relación entre dos componentes A y B, de tal forma que A produce información que es consumida por B. Volviendo a nuestro escenario de domótica, esta relación aparece entre los componentes *Interruptor* y *Controlador de luz*. Cuando a través del componente *Interruptor* un usuario realiza la operación de apagar/encender una bombilla, el componente *Controlador de luz* realizará la operación correspondiente.

2.7.2. Relaciones n-arias

Todas las relaciones que han sido definidas entre dos componentes cubren un amplio rango de posibles escenarios en la construcción de sistemas basados en componentes. Sin embargo, en ocasiones se necesitan otras relaciones más complejas, para lo cual se han definido las relaciones n-arias. Una relación n-aria se define como un conjunto de relaciones binarias. La Tabla 2.11 muestra las principales relaciones n-arias entre componentes junto a su notación gráfica.

Nombre	Símbolo
Secuencia	$\dot{+}\{,...,\}$
Jerarquía	$\overline{\overline{\wedge}}\{,...,\}$
Trading	$\top\{,...,\}$
Observación	$\circledast\{,...,\}$
Control	$\ast\{,...,\}$
Sumidero	$\curlyvee\{,...,\}$

Tabla 2.11: Principales relaciones n-arias

© 2016 Vallecillos, J.

La primera relación que aparece en la tabla es la relación **Secuencia**. Esta relación define que el conjunto de relaciones binarias que la componen llevan a cabo una secuencia de información desde un componente hasta otro. En la Figura 2.29a, se puede ver una representación gráfica de esta relación. Un ejemplo puede ser visto entre los componentes *Lista de capas*, *Mapa* y *Leyenda*, y está compuesta por dos relaciones binarias de *Subordinación*. En este caso, la información va desde la *Lista de capas* hasta la *Leyenda* para decidir que se quiere visualizar u ocultar una capa en la *Lista de capas*.

Otra relación n-aria es la **Jerarquía** que se da cuando un conjunto de componentes tienen una relación de herencia con sus componentes padres. Un ejemplo relacionado con el escenario de domótica (ver Figura 2.29b) es cuando un interruptor que permite encender una bombilla con nivel de intensidad además hereda de un interruptor normal para aportar la funcionalidad de apagar o encender. Adicionalmente, existe otro componente que hereda del interruptor con porcentaje y de otro interruptor central que apaga la corriente de toda la casa.

La relación de **Trading** tiene lugar cuando uno de los componentes realiza una tarea de mediación con respecto a otros, con el objetivo de trasmitir cierta información entre ambos. Por ejemplo, supóngase que en el caso del sistema de información geográfico (ver Figura 2.29c), hay un componente llamado *Conexiones* que se encarga de recibir la lista de componentes con los que está conectado, cuando un componente nuevo es añadido al sistema. Por tanto, cuando el componente *Lista de capas* se añade al sistema, se le envía una notificación al componente *Conexiones* para indicarle con quién puede conectarse. De esta forma, cuando un componente *Mapa* requiera de una lista, lo primero que hace es consultar al componente de *Conexiones* con quién debe hacerlo, el cual le informa que debe comunicarse con la *Lista de capas* previamente registrada que le enviará la información con la lista que el mapa necesita para funcionar.

Las otras relaciones n-arias son las relaciones de **Control**, **Sumidero** y **Observación**. La relación de *Control* (Figura 2.30b) implica que el componente central no sólo observa, sino que además envía órdenes de control a los otros componentes. Estamos ante una relación n-aria más fuerte que la relación de observación. En el escenario de domóti-

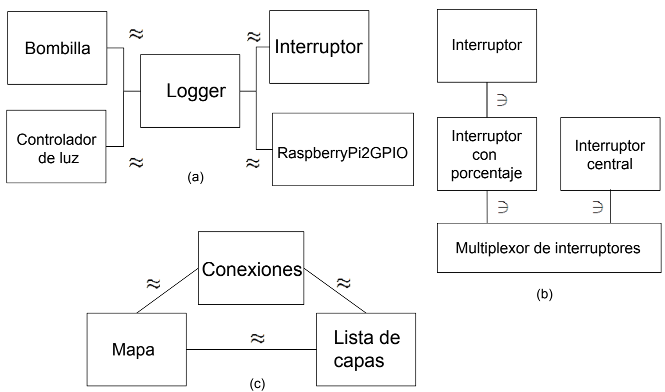


Figura 2.29: Relaciones n-arias: (a) observación, (b) jerarquía y (c) *trading*

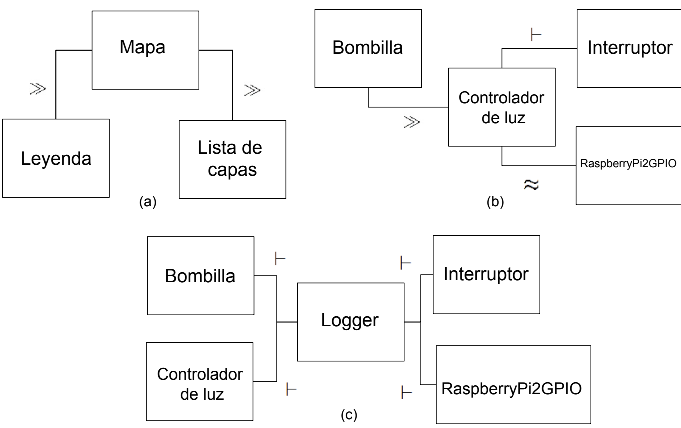


Figura 2.30: Relaciones n-arias: (a) secuencia, (b) control y (c) sumidero

ca, ocurre que el componente *Controlador de luz* se encarga del control de estado de la bombilla, controlando la interacción que se produce en la parte virtual y real. La relación de **Sumidero** describe una relación en la cual el componente central es el que recibe la información generada por los otros componentes involucrados en la relación. Todas las relaciones que forman la relación *Sumidero* son relaciones de tipo *producerconsumer*, en las cuales el consumidor es el componente central (Figura 41c). En el escenario ejemplo de domótica, el componente *Logger* trabaja como consumidor de todos los procesos que tienen lugar en el resto de componentes del entorno, el cual se encarga de registrar en un archivo de texto todo lo ocurrido. Por último, la relación de **Observación** implica que el componente central lleva a cabo tareas de observación en todos los componentes que están vinculados con él, pero sobre los que no influye. Supóngase un caso en el ejemplo de domótica (Figura 2.29a), en el que el componente *Logger* está constantemente observando qué ocurre en el resto de componentes para llevar a cabo un registro de acciones. En este caso, el componente *Logger* no está a la espera de que los demás le informen, sino que es él el que, de forma proactiva, se encarga de obtener las acciones ejecutadas y de registrar lo sucedido.

2.8. Trabajo relacionado

Analicemos algunos trabajos relacionados con este capítulo. En [Abiteboul et al., 2009], los autores muestran un sistema que soporta el desarrollo intuitivo y rápido durante la construcción de aplicaciones *mashup*. Dicho sistema se basa en un mecanismo para autocompletar interfaces de usuario gráficas a partir de características comunes a otros usuarios. También se propone un método para conectar unos componentes con otros y de esta forma ayudar a los usuarios a construir su espacio de trabajo dentro de la aplicación web. De esta forma, los usuarios seleccionan los componentes que serán cargados en la interfaz de usuario, cuando la aplicación es desplegada. Dicho trabajo

© 2016 Vallecillos, J.

se centra sólo en componentes de tipo RSS y no da soporte a otros tipos de componentes. En [Hassan et al., 2014] los autores desarrollan otra propuesta para llevar un *ranking* de componentes de tipo RSS considerados como interesantes para los usuarios. La aplicación web *mashup* se genera a partir de estos componentes. En ambos trabajos la configuración de los componentes que se cargan inicialmente se realiza en tiempo de diseño, y no puede variar en tiempo de ejecución, tal y como se plantea en la propuesta metodológica del presente trabajo de investigación.

En [Gmelch and Pernul, 2012] se presenta una propuesta para el despliegue de una arquitectura basada en componentes que permite la integración de sistemas basados en web. Proponen una arquitectura a través de capas para que la comunicación entre componentes no se realice de forma directa, sino que existe un sistema central que gestiona dicha comunicación. Dicho trabajo se centra en la integración de aplicaciones basadas en *portlets*. A diferencia de esta propuesta, nuestra infraestructura gestiona aplicaciones construidas a partir de componentes terceros de granulidad gruesa que se ejecutan en el lado cliente (aunque la gestión de las arquitecturas y de las comunicaciones se realice en el lado servidor).

En la literatura, se pueden encontrar otros trabajos específicos relacionados con la definición de arquitecturas para la gestión de *widgets*. En [Fan et al., 2012], los autores proponen un nuevo tipo de sistema para plataformas de televisión digital basados en *widgets*. Se centran en componentes que se construyen con tecnología web y *widgets* que siguen el estándar web de W3C. A diferencia de la investigación, donde la comunicación se ha basado en el uso de un servidor *JavaScript*, los autores de dicho trabajo utilizan una forma de comunicar los componentes a través de AJAX, ejecutada en el lado cliente para obtener la información almacenada en un servidor (un recurso en formato XML). Otra plataforma donde se pueden usar arquitecturas basadas en *widgets* son las aplicaciones móviles. En [Pierre et al., 2013] los autores definen una arquitectura específica para la plataforma móvil, construida con componentes que gestionan la interacción y una plataforma software que gestiona su ciclo de vida. En nuestra propuesta, también gestionamos el ciclo de vida de los componentes en el lado servidor, aunque actualmente no se permite el despliegue de aplicaciones en dispositivos móviles.

Existen otros trabajos como [Shirogane et al., 2008] que realizan adaptaciones del código de la interfaz de usuario dinámicamente según las preferencias del usuario y utilizando componentes *widgets*. En dicho trabajo, se definen siete tipos de roles para mejorar la adaptación y recarga de los componentes. Dependiendo del tipo de rol, hay componentes considerados como preferentes para hacer su adaptación. A diferencia de la propuesta que se realiza en nuestra investigación, donde se usa componentes de granularidad gruesa, aquí los autores implementan interfaces con componentes de granularidad fina construidos con la librería *Java Swing*.

En [Wilson et al., 2012] se centran en otra funcionalidad abarcada en este trabajo de tesis doctoral, la comunicación entre componentes. Los autores discuten ciertas características para construir interfaces de usuario *mashups* usando *widgets* W3C y proponen una extensión del modelo de *widgets*. El objetivo de extender este modelo es dar soporte a una variedad de patrones de comunicación entre componentes. En cambio, en nuestra propuesta se ha desarrollado un mecanismo de comunicación que no necesita de una revisión del modelo de componente. Otro trabajo muy similar al tratado anteriormen-

te es [Sire et al., 2009]. La comunicación entre los *widgets* se basa en eventos a través de una API definida, basada en PHP y MySQL. En esta propuesta, se crea un *widget* contenedor, utilizando *JavaScript*, encargado de gestionar el flujo de la información. En cambio, en nuestra propuesta se usa un servidor *JavaScript* que gestiona la comunicación utilizando *WebSockets*.

2.9. Resumen y conclusiones

En este segundo capítulo se han presentado las diferentes capas por las cuales está formada la infraestructura encargada de sustentar el despliegue de aplicaciones *mashup*. De esta forma, se han descrito cada una de las capas que definen el entorno, destacando por qué es interesante llevar a cabo este desglose. Además, para justificar la creación de relaciones entre los componentes que forman las aplicaciones se han tratado una serie de escenarios.

En primer lugar, se ha llevado el foco sobre las aplicaciones *mashup* definidas dentro de la capa cliente. También se ha tratado la capa dependiente de la plataforma. Respecto a esta capa, se han expuesto los servidores que se pueden localizar junto con los diferentes repositorios de componentes que dan soporte a las aplicaciones *mashup*, definiendo para ello un modelo de datos de componentes a dos niveles, uno para la gestión de repositorios internos propios al sistema, y otro para la gestión de repositorios externos. Dentro de estos repositorios se ha realizado una distinción, localizando los repositorios de componentes externos y los repositorios gestionados. Con respecto a los repositorios de componentes externos, ha sido necesario implementar un *wrapper* para que puedan ser empotrados en las aplicaciones *mashup* sobre las que se sustentan.

Para continuar, se ha mostrado la capa independiente de la plataforma, que es la parte principal de la infraestructura y sobre la que se centra la mayor parte de la investigación. En este nivel, se han mostrado los diferentes repositorios que aparecen, definiendo las especificaciones de componentes y las especificaciones de las arquitecturas de las aplicaciones *mashup*. Cada una de estas especificaciones se ha construido en base a un metamodelo, lo cual permite dotar a la capa de un mayor nivel de abstracción para las diferentes aplicaciones a las que da soporte. Seguidamente, se dan detalles sobre dos escenarios de aplicación básicos centrados en el ámbito de los sistemas de información geográficos y las aplicaciones de domótica. Estos escenarios, además de contextualizar la aplicabilidad del sistema COScore, permiten crear posibles casos de dependencias y relaciones entre componentes que forman las aplicaciones *mashup*. Dichos escenarios han permitido, por una lado, realizar un estudio de dependencias entre componentes *mashup* a partir de una lista de patrones de composición de componentes y de una matriz de regeneración de dependencias; y por otro, explicar un conjunto de relaciones entre componentes *mashup* formado por cinco relaciones binarias y seis relaciones n-arias.

© 2016 Vallecillos, J.

Capítulo 3

COScore: modelo de servicios y operaciones

© 2016 Vallecillos, J.

Capítulo 3

COSCORE: MODELO DE SERVICIOS Y OPERACIONES

Contenidos

3.1. Introducción y conceptos relacionados . . . 87
3.2. Soporte de servicios . . . 92
3.2.1. Bases de datos y controladores . . . 92
3.2.2. Módulos . . . 99
3.3. Servicios privados . . . 104
3.3.1. Servicio *User Service* . . . 104
3.3.1.1. Operación *Query User* . . . 104
3.3.1.2. Operación *Update User* . . . 109
3.3.1.3. Operación *Delete User* . . . 113
3.3.1.4. Operación *Create User* . . . 116
3.3.1.5. Operación *Query Profile* . . . 120
3.3.2. Servicio *Manage Architecture Service* . . . 123
3.3.2.1. Operación *Export AAM from String* . . . 124
3.3.2.2. Operación *Export CAM from String* . . . 126
3.3.2.3. Operación *Withdraw CAM* . . . 129
3.3.3. Servicio *Manage Component Service* . . . 131
3.3.3.1. Operación *Export CC from String* . . . 131
3.3.3.2. Operación *Export CC from params* . . . 134
3.3.3.3. Operación *Withdraw CC* . . . 136
3.4. Servicios públicos . . . 139
3.4.1. Servicio *Session Service* . . . 139
3.4.1.1. Operación *Login* . . . 139

3.4.1.2. Operación *Logout* 144
3.4.1.3. Operación *Init User Architecture* 147
3.4.1.4. Operación *Default Init* 153
3.4.2. Servicio *Communication Service* 158
3.4.2.1. Operación *Get Link Components* 158
3.4.3. Servicio *Component Service* 162
3.4.3.1. Operación *Update Architecture* 162
3.4.4. Servicio *Interaction Service* 170
3.4.4.1. Operación *Register Interaction* 170
3.5. Trabajo relacionado 174
3.6. Resumen y conclusiones 176

En el capítulo anterior se han descrito las partes principales de la infraestructura propuesta, constituidas por la capa cliente, la capa dependiente y la capa independiente de la plataforma. En este capítulo se realiza un desglose de la infraestructura de servicios desarrollada en la capa independiente. Este desglose consta de cinco niveles: servicios públicos, servicios privados, módulos, controladores y bases de datos. En el nivel de servicios públicos se localizan las operaciones que son accesibles por las aplicaciones *mashup*. En el nivel de servicios privados se encuentran las operaciones que dan soporte a la gestión interna de las aplicaciones. El tercer nivel se corresponde con los módulos que implementan la funcionalidad de los servicios, tanto de los públicos como de los privados. Muchos de estos módulos hacen uso del nivel donde se encuentran los controladores. Dicho nivel contiene la funcionalidad indispensable para poder acceder a los datos localizados en las bases de datos, las cuales representan el último nivel de la infraestructura de servicios desarrollada.

El capítulo se desglosa en seis secciones. La Sección 3.1 presenta la infraestructura a nivel general, mostrando cuáles son las partes principales. Posteriormente, la Sección 3.2 describe cada uno de los módulos que dotan de funcionalidad a la infraestructura, los controladores y las bases de datos que almacenan la información del entorno. En la Sección 3.3, se describen los servicios privados y cada una de sus operaciones. En la Sección 3.4, se encuentran descritos los servicios públicos, donde se entra en detalles acerca de las operaciones que contienen y su funcionamiento. La Sección 3.5 revisa algunos de los trabajos de la literatura relacionados con arquitecturas e infraestructuras encargadas de dar soporte a las aplicaciones *mashup*. El capítulo finaliza con un breve resumen de los contenidos presentados y con las conclusiones extraídas.

3.1. Introducción y conceptos relacionados

Para dar soporte a las aplicaciones *mashup* se ha creado una infraestructura llamada COScore [Vallecillos et al., 2014], formada por servicios, módulos, controladores y bases de datos, que queda representada por la estructura mostrada en la Figura 3.1. En el primer nivel de servicios (parte superior) se encuentran los *Servicios públicos* (*Public service*), y son utilizados para aportar funcionalidad, persistencia y soporte a las aplicaciones *mashup*. La Tabla 3.1 muestra un resumen de las operaciones ofertadas en este nivel, junto con una breve descripción de las mismas. En el segundo nivel, se encuentran los *Servicios privados* (*Private service*). Estos servicios se utilizan para realizar ciertas tareas de gestión, como las relacionadas con los modelos de arquitectura de las aplicaciones, los modelos de componentes concretos, los usuarios y los componentes disponibles en el sistema. La Tabla 3.2 muestra un resumen de las operaciones disponibles en este nivel, junto con una breve descripción de las mismas.

Para implementar la funcionalidad de los dos niveles de servicios se encuentran los niveles de *Módulos* (*Modules*), *Controladores* (*Controllers*) y *Bases de datos* (*Data bases*). El nivel de *Módulos* es utilizado por los servicios e implementa toda la funcionalidad ofrecida por la infraestructura. La capa *Controlador* realiza la gestión de las diferentes

© 2016 Vallecillos, J.

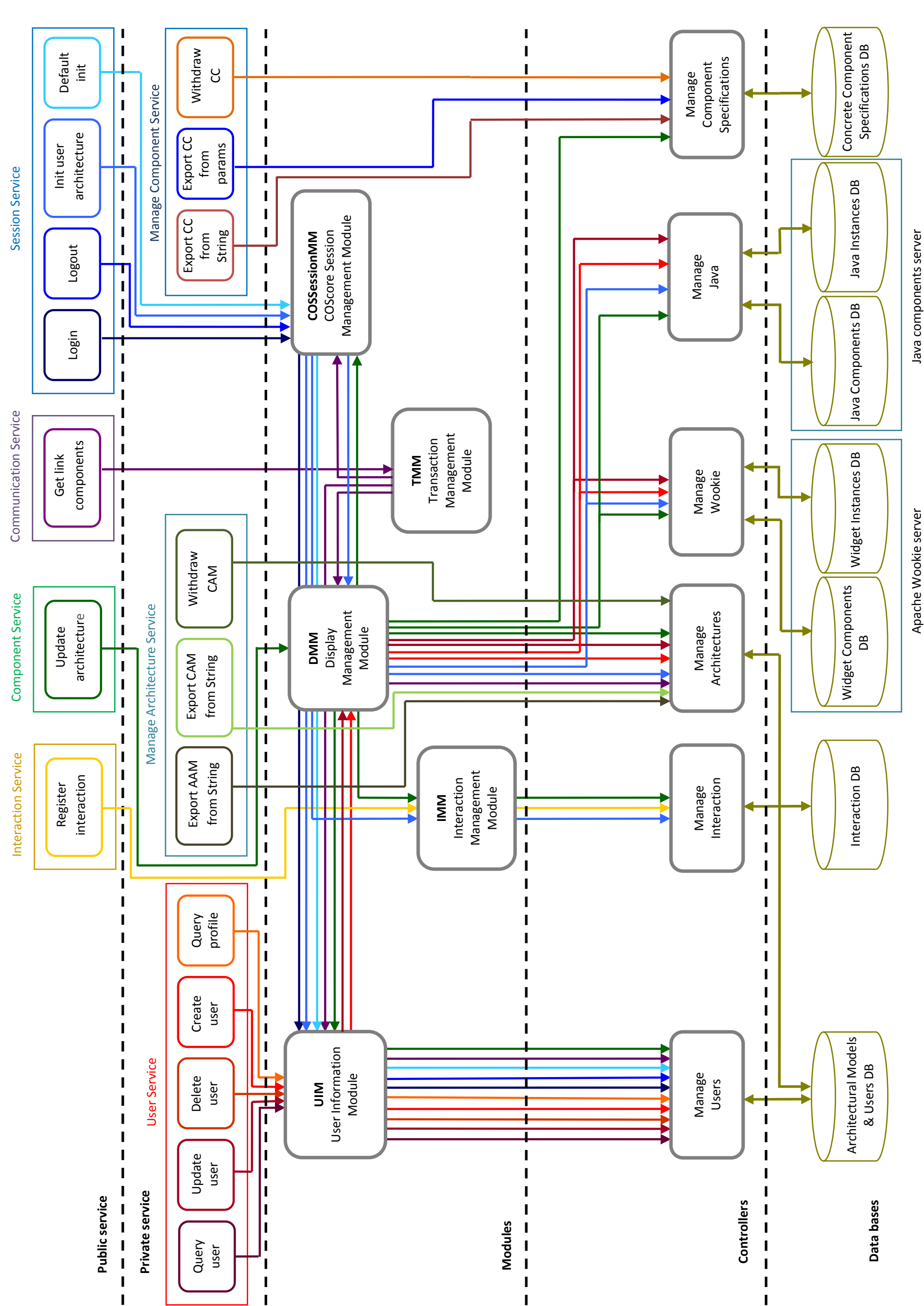


Figura 3.1: Estructura de servicios, módulos, controladores y bases de datos

Servicio	**Operación**	**Descripción**
Interaction Service	*Register interaction*	Servicio encargado de gestionar la interacción que se produce en las aplicaciones *mashup*
Component Service	*Update architecture*	Servicio encargado de gestionar las actualizaciones sobre los componentes que forman la aplicación *mashup*
Communication Service	*Get link components*	Servicio encargado de controlar los procesos de comunicación entre los componentes de la aplicación *mashup*
Session Service	*Login* *Logout* *Init user architecture* *Default init*	Servicio encargado de manejar la inicialización de las aplicaciones *mashup*

Tabla 3.1: Servicios públicos del COScore

Servicio	**Operación**	**Descripción**
User Service	*Query user* *Update user* *Delete user* *Create user* *Query user*	Servicio encargado de gestionar los usuarios de la infraestructura COScore
Manage Architecture Service	*Export AAM from String* *Export CAM from String* *Withdraw CAM*	Servicio encargado de manejar los modelos de arquitectura de las aplicaciones *mashup*
Manage Component Service	*Export CC from String* *Export CC from params* *Withdraw CC*	Servicio encargado de manejar las especificaciones de componentes que constituyen las aplicaciones *mashup*

Tabla 3.2: Servicios privados del COScore

bases de datos que maneja el entorno. Por último, está el nivel *Base de datos*, el cual alberga las diferentes bases de datos utilizadas para almacenar los modelos de arquitectura, los modelos de componentes concretos, los componentes de las aplicaciones de usuario junto a las instancias de los componentes, la interacción y los usuarios de las aplicaciones. En la Tabla 3.3 se muestra el conjunto de módulos, controladores y bases de datos junto con una breve descripción de cada uno de ellos.

La combinación de todos estos elementos (bases de datos, módulos, controladores y servicios) constituyen la capa independiente de la plataforma de la infraestructura propuesta. Como se ha comentado en capítulos anteriores, esta capa, junto con la capa dependiente de la plataforma, constituyen el núcleo de la propuesta desarrollada en este trabajo de tesis doctoral, cuyo objetivo es permitir el despliegue y gestión de aplicaciones *mashup*. Por este motivo, podemos determinar que la infraestructura para el despliegue de interfaces *mashup* (en adelante "infraestructura *mashup*" para simplificar) está formada por un conjunto de elementos arquitectónicos, definida como sigue:

© 2016 Vallecillos, J.

Bases de datos	
Architectural Models and Users DB	Contiene los modelos de las aplicaciones *mashup* y los usuarios que hacen uso de dichos modelos
Interaction DB	Contiene la interacción generada en la aplicación *mashup*
Widget Components DB	Almacena los componentes de tipo *widget*
Widget Instances DB	Almacena las instancias de los componentes de tipo *widget*
Java Components DB	Almacena los componentes de tipo Java
Java Instances DB	Almacena las instancias de los componentes de tipo Java
Concrete Component Specifications DB	Almacena las especificaciones de los componentes concretos
Controladores	
Manage Users	Gestiona la base de datos de usuarios
Manage Interaction	Gestiona la base de datos de interacción
Manage Architectures	Gestiona la base de datos de modelos de arquitecturas de las aplicaciones *mashup*
Manage Wookie	Gestiona la base de datos de componentes e instancias de tipo *widget*
Manage Java	Gestiona la base de datos de componentes e instancias de tipo Java
Manage Component Specifications	Gestiona la base de datos de especificaciones de componentes concretos
Módulos	
User Information Module (*UIM*)	Maneja la funcionalidad relacionada con la gestión de los usuarios
Interaction Management Module (*IMM*)	Maneja la funcionalidad relacionada con la interacción que tiene lugar
Display Management Module (*DMM*)	Controla la funcionalidad relacionada con la visualización de las aplicaciones *mashup*
Transaction Management Module (*TMM*)	Controla la funcionalidad vinculada con la transacción de información entre componentes
COScore Session Management Module (*COSSessionMM*)	Maneja la funcionalidad vinculada con la inicialización de las aplicaciones *mashup*

Tabla 3.3: Descripción de los elementos de la infraestructura COScore

Definición 3.1 (*Infraestructura* mashup) *Una infraestructura* mashup *I se define como una tupla de cuatro elementos base $\mathcal{I}$=($\mathcal{S}$, $\mathcal{M}$, $\mathcal{C}$, $\mathcal{R}$), siendo $\mathcal{S}$ el conjunto de servicios ofrecidos por el sistema; $\mathcal{M}$ un conjunto de módulos que implementan la funcionalidad básica de los servicios, y con los cuales se comunican las operaciones de dichos servicios; $\mathcal{C}$ un conjunto de controladores que median entre los módulos y los repositorios del sistema; y $\mathcal{R}$ el conjunto de repositorios o bases de datos con la información que da soporte a una infraestructura* mashup *$\mathcal{I}$, y siendo:*

a) $\mathcal{S}$: el par (S_-, S_+), donde S_- es el conjunto de servicios privados $S_- = \{S_-^1, S_-^2, S_-^3\}$, y S_+ es el conjunto de servicios públicos $S_+ = \{S_+^1, S_+^2, S_+^3, S_+^4\}$, siendo estos los servicios User Service, Manage Architecture Service, *y* Manage Component Service, *para el caso de los servicios privados, y* Interaction Service, Component Service, Communication Service, *y* Session Service, *repectivamente.*

b) $\mathcal{M}$*: el conjunto de módulos* $\mathcal{M} = \{UIM,\ IMM,\ DMM,\ TMM,\ COSSessionMM\}$*, siendo estos los módulos* User Information Module*,* Interaction Management Module*,* Display Management Module*,* Transaction Management Module*, y* COScore Session Management Module*, respectivamente.*

c) $\mathcal{C}$*: el conjunto de controladores* $\mathcal{C} = \{C_1,\ C_2,\ C_3,\ C_4,\ C_5,\ C_6\}$*, siendo estos los controladores* Manage User*,* Manage Interaction*,* Manage Architectures*,* Manage Wookie*,* Manage Java*, y* Manage Component Specifications*, respectivamente.*

d) $\mathcal{R}$*: el conjunto de bases de datos* $\mathcal{R} = \{R_1,\ R_2,\ R_3,\ R_4,\ R_5,\ R_6,\ R_7\}$*, siendo estas las bases de datos* Architectural Models and User DB*,* Interaction DB*,* Widget Components DB*,* Widget Instances DB*,* Java Components DB*,* Java Instances DB*, y* Concrete Component Specifications DB*, respectivamente.*

Como ha sido mencionado, el objetivo de esta infraestructura *mashup* es permitir el despliegue de un tipo de aplicaciones software que cumplen con una serie de características. Las aplicaciones deben poder construirse a partir de la combinación de distintos servicios (que pueden estar, además, desarrollados por terceros) siempre que dichos servicios hayan sido encapsulados en forma de componentes para que sean gestionados en un repositorio de la infraestructura. De esta manera, las aplicaciones se describen mediante arquitecturas en las que los componentes pueden presentar dependencias entre sí y comunicarse (invocando operaciones entre ellos) para la realización de una tarea determinada. Un ejemplo de este tipo de software, como veremos más adelante en el siguiente capítulo de pruebas y experimentación, es el caso de una aplicación *mashup* de una interfaz gráfica de usuario en la Web, como puede ser la de ENIA[1].

En este tipo de interfaz gráfica, un usuario que accede por primera vez a la aplicación *mashup* (acceso de forma anónima) tiene desplegados unos componentes por defecto que puede utilizar (por ejemplo, un mapa para visualizar información geográfica y un componente de noticias relacionadas con la REDIAM). También tiene disponible un catálogo de servicios para poder añadir componentes a su espacio de trabajo (con otros mapas temáticos, componentes de redes sociales, etc.). El espacio de trabajo consiste en la zona de visualización de los componentes desplegados de la aplicación *mashup* y puede ser modificado y reconfigurado según las necesidades del usuario, por ejemplo, redimensionando, moviendo o eliminando componentes. Si el usuario realiza este tipo de operaciones de reconfiguración sobre su aplicación *mashup* (una vez que ya ha iniciado una sesión en el sistema) todos los cambios son gestionados por la infraestructura de manera que, cuando el usuario se reconecta con la aplicación, pueda disponer de espacio de trabajo tal y como lo reconfiguró la última vez. Además, con el objetivo adaptarse a las necesidades y preferencias de utilización, la interacción realizada por los usuarios (tanto por los anónimos como por los registrados) es almacenada en la infraestructura para permitir su posterior análisis y la modificación de la lógica de adaptación.

[1]ENIA es un prototipo desarrollado para la REDIAM (RED de Información AMbiental de Andalucía) de una interfaz gráfica de usuario que sigue el modelo de aplicación *mashup* desarrollado en esta tesis doctoral, prototipo que ha sido utilizado para realizar las pruebas de validación del modelo de infraestructura COScore, definido en el Capítulo 4 de este documento de tesis doctoral.

© 2016 Vallecillos, J.

En las siguientes secciones, se describe cada uno de los niveles de la infraestructura *mashup* desarrollada, la cual permite realizar el despliegue de aplicaciones, incluyendo el tipo de operaciones que han sido mencionadas para el ejemplo de aplicación anterior. Para facilitar su comprensión, los diferentes niveles se describirán en el siguiente orden: bases de datos con sus respectivos controladores, módulos, servicios privados y públicos.

3.2. Soporte de servicios

Para que los servicios puedan ofrecer un soporte funcional, se ha estructurado el desarrollo en tres niveles: bases de datos, controladores y módulos. Tanto el conjunto de módulos como el conjunto de controladores se han implementado mediante componentes *Enterprise Java Beans*[2] (EJB), ya que permiten un desarrollo modular del software, permiten crear sesiones para cada aplicación *mashup*, además de poder ser desplegados en un servidor de aplicaciones capaz de gestionar grandes volúmenes de peticiones simultáneas. El resto de esta sección describe estos tres niveles. Se comenzará describiendo las bases de datos de la infraestructura junto con los controladores asociados a los mismos, ya que son la base en la que se sustenta toda la infraestructura. Después se describirán los módulos de dicha infraestructura.

3.2.1. Bases de datos y controladores

En los dos niveles más bajos de la infraestructura desarrollada se localizan las bases de datos y sus respectivos controladores. En las bases de datos se almacenan toda la información necesaria para el correcto funcionamiento de las aplicaciones *mashup*. Por otro lado, los controladores se encargan de llevar la gestión de dichas bases de datos. Para comprender mejor el funcionamiento de cada base de datos y de su respectivo controlador, a continuación se realizará una descripción de cada uno de ellos.

Base de datos de usuarios y modelos de arquitectura

La Base de datos de usuarios y modelos de arquitectura (Architectural Models and Users DB) se encarga de almacenar los modelos de arquitectura de las aplicaciones *mashup* y los usuarios que están dados de alta en el entorno. Cada usuario dado de alta en el COScore tiene una aplicación *mashup*, descrita mediante su correspondiente modelo de arquitectura. Para gestionar los modelos de arquitectura de las aplicaciones se ha usado la librería *Hibernate*[3]. Hibernate es una herramienta que permite la asociación *objeto-relacional* (ORM) en Java y facilita la vinculación de atributos entre una base de datos relacional tradicional y un modelo de objetos. Por ese motivo, la librería permite gestionar los modelos como objetos dentro de la infraestructura, facilitando de esta forma su manipulación. La base de datos donde se guardan los modelos de arquitectura ha sido implementada en PostgreSQL[4].

[2] Enterprise Java Beans – `http://www.oracle.com/technetwork/java/javaee/ejb/index.html`
[3] Hibernate – `http://hibernate.org/`
[4] PostgreSQL – `http://www.postgresql.org.es/`

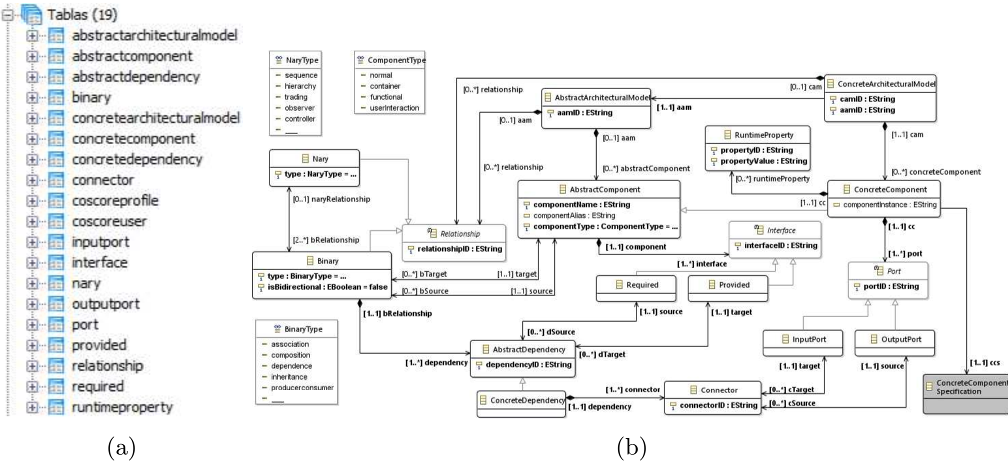


Figura 3.2: (a) Tablas de la base de datos de modelos de arquitectura y usuarios, (b) metamodelo de arquitectura de las aplicaciones *mashup*

En la Figura 3.2a se pueden observar las diferentes tablas que conforman parte de esta base de datos. Para cada una de ellas existe un elemento en el metamodelo de arquitectura de la aplicación mostrada en la Figura 3.2b, puesto que dicha base de datos almacenará los modelos de arquitectura generados a partir de este metamodelo. El metamodelo de arquitectura de las aplicaciones *mashup* fue ampliamente descrito en la Subsección 2.4.1 del Capítulo 2.

La Base de datos de usuarios y modelos de arquitectura (*Architectural Models and Users DB*) es gestionada por dos controladores: el controlador *Manage Users*, utilizado para gestionar los usuarios, y el controlador *Manage Architecture*, utilizado para gestionar los modelos de arquitectura.

El controlador **Manage Users** incluye el conjunto de operaciones encargadas de realizar la gestión de los usuarios en la base de datos. A través de este controlador se pueden añadir, eliminar, modificar o consultar usuarios. En el siguiente listado se puede observar la implementación del controlador junto con los métodos que lo componen.

```
// Implementación del Controlador "Manage Users"
1 public class ManageUsers {
2    public void initialize();
3    public int queryUser(String userName, String userPassword);
4    public String queryCamUser(String userId);
5    public String queryCamProfile(String Profilename);
6    public List<String> queryProfile();
7    public boolean deleteUser(String userId);
8    public void createUser(String userName, String userPassword,
9      String userProfile, String camID);
10   public boolean updateUser(String userId, String userNameNew,
11     String userPassword, String userProfile);
12 }
```

En la línea 2 se muestra la interfaz del método `initialize`, el cual se encarga de inicializar las variables necesarias para que el resto de métodos del controlador puedan realizar operaciones sobre la BD. El método `queryUser` (línea 3) consulta los usuarios de la infraestructura. Para ello, devuelve el identificador del usuario con el que fue dado de alta. Para poder ejecutar este método, se introduce como parámetros el nombre y la contraseña del usuario (parámetros `userName` y `userPassword`). En la línea 4 se encuentra el método `queryCamUser`. Este método devuelve el identificador del modelo de arquitectura de la aplicación *mashup* de un usuario. Para ello, necesita que se le introduzca como parámetro el identificador del usuario (`userId`). El método `queryCamProfile` (línea 5) devuelve una cadena de texto con el identificador del modelo, a partir del perfil de usuario introducido como parámetro. El método `queryProfile` (línea 6), devuelve una lista de perfiles a los que se asocian los diferentes usuarios de la infraestructura. El método `deleteUser` (línea 7), elimina un usuario del entorno a partir del identificador del usuario (`userId`). El método `createUser` se utiliza para crear un nuevo usuario y su correspondiente aplicación *mashup*. Para ello, se le pasa como parámetro el nombre (`userName`), password (`userPassword`), perfil de usuario (`userProfile`) e identificador del modelo de arquitectura concreto (`camID`). Por último aparece el método `updateUser`, utilizado para actualizar cualquier dato de un usuario concreto.

El controlador de gestión de arquitecturas **Manage Architectures** se encarga de realizar la manipulación de los modelos que describen la arquitectura de las aplicaciones *mashup* de los usuarios. En esta gestión se manejan tanto los modelos de arquitectura abstracta como los modelos de arquitectura concreta, ya descritos en el Capítulo 2. Por medio de estos modelos se pueden llevar a cabo procesos de persistencia de las aplicaciones *mashup*, de tal forma que cuando un usuario cierre su aplicación y decida retomar más tarde su actividad, el estado de la aplicación sea el mismo. De esta forma, esta persistencia permite que cuando se añadan o eliminen componentes de la aplicación, el modelo sea actualizado. Además, los modelos de la aplicación *mashup* serán útiles para manejar los procesos de comunicación que tienen lugar entre los componentes que constituyen el entorno *mashup*. Cada usuario debe tener su propio modelo, el cual se obtiene al crearse dicho usuario según su perfil. La implementación del controlador se puede ver en el siguiente listado. En la línea 2 de esta lista aparece el método `initializeDataStore`, el cual está encargado de inicializar las variables que manejan la base de datos *Architectural Models and Users*.

```
// Implementación del Controlador "Manage Architectures"
1 public class ManageArchitectures {
2    private void initializeDataStore();
3    public String exportCAMFromString(String camFileString);
4    private static AbstractArchitecturalModel getAbstractArchitecturalModel(
5      String aamID, Session session);
6    public String exportAAMFromString(String aamFileString);
7    public String withdrawCAM(String camID);
8    public ConcreteArchitecturalModel readModel(String camID);
9    public void saveModel(ConcreteArchitecturalModel cam);
10   public String addComponent(String camID, ConcreteComponent concreteComponent,
11     Port inputPort, List<RuntimeProperty> runtimePropertyList);
12   public void saveComponentInstance(String componentName,
13     String componentAlias, String componentInstance);
```

```
14   public void changeComponentRuntimeProperties(String componentInstance,
15     RuntimeProperty runtimeProperty);
16   public void changeComponentProperties(String componentInstance,
17     String componentNewInstance, String componentNewName,
18     String componentNewAlias);
19   public void changeComponentService(String componentInstance,
20     int numberService, List<RuntimeProperty> listServices);
21   public void deleteComponent(String camID, String componentInstance);
22 }
```

Por otro lado, en la línea 3 se encuentra el método `exportCAMFromString`, donde se realiza la tarea de añadir un modelo de arquitectura concreto para una aplicación *mashup*. Este modelo de arquitectura concreto se añade por medio de un XMI en forma de cadena de texto (`camFileString`). En la línea 4, está el método *getAbstractArchitecturalModel* encargado de devolver el modelo de arquitectura abstracto a partir del identificador del modelo (`aamID`) y de un objeto de tipo `Session`, para poder establecer una sesión con la base de datos de modelos de arquitectura abstracta. En la línea 6, el método `exportAAMFromString` permite añadir un modelo de arquitectura abstracta de una aplicación *mashup*. Para añadir dicho modelo se pasa por parámetro un XMI en forma de cadena de texto (`aamFileString`). En la línea 7, está el método `withdrawCAM` para eliminar modelos de arquitectura concreta de la base de datos, a partir del identificador de dicho modelo (`camID`) que se desea eliminar. El método `readModel` de la línea 8 permite leer un modelo concreto de la base de datos pasándole el identificador del modelo que desea leer. En la línea 9 se encuentra el método `saveModel` encargado de guardar un modelo de arquitectura concreto en la base de datos. A este método se le pasa como parámetro un objeto de tipo `ConcreteArchitecturalModel`. Por otro lado, hay un conjunto de métodos dedicados a trabajar con los componentes que forman los modelos y las propiedades de dichos componentes. En la línea 10 se encuentra el método `addComponent` encargado de añadir un nuevo componente en el modelo de arquitectura concreta. En la línea 12 se añade por medio del método `saveComponentInstance`, el cual permite guardar las instancias de los componentes en los modelos de componentes concretos. En la línea 14 por medio del método `changeComponentRuntimeProperties`, se permite cambiar una propiedad *Runtime* de un componente. En la línea 16 se modifica una propiedad de un componente a través del método `changeComponentRuntimeProperties`. El método `changeComponentService` de la línea 19 permite cambiar los servicios asociados a un componente. Por último, en la línea 21 el método `deleteComponent` permite eliminar un componente del modelo de componentes concretos.

Base de datos de interacción

Esta base de datos almacena la interacción que el usuario realiza con la aplicación en sí. Dado que la infraestructura desarrollada considera a los componentes como "cajas negras", en esta base de datos no se almacena la interacción relacionada con el contenido específico de cada componente que forma parte de la aplicación. Por ello, sólo se guarda información relacionada con la eliminación y agregación de componentes, cambios de tamaño y de posición, así como sobre los procesos de comunicación. La base de datos ha sido construida en PostgreSQL.

© 2016 Vallecillos, J.

La gestión de la BD de interacción es realizada por el controlador de gestión de interacción **Manage Interaction**. Este controlador gestiona todas las funciones que se pueden realizar sobre esta base de datos, y es manipulado por el módulo IMM. En el siguiente listado se muestra la implementación del controlador. El método `initialize` (línea 2) es utilizado para inicializar todas las variables necesarias en la gestión de la base de datos. En la línea 3 se encuentra el método `insertInteraction`, que puede ser utilizado para insertar los eventos de interacción que tienen lugar en la aplicación *mashup*. Para poder registrar un evento, se puede ver que el método define un conjunto de parámetros, como el tipo de dispositivo utilizado (`deviceType`), tipo de interacción (`interactionType`), fecha y hora (`dateTime`), identificador de usuario (`userId`), entre otros. En la línea 8, se encuentra el método `manageRuntimeProperty`, utilizado para cambiar una propiedad del componente sobre el cual ha tenido lugar algún evento.

```
// Implementación del Controlador "Manage Interaction"
1 public class ManageInteraction {
2   public void initialize();
3   public void insertInteraction(String newSession, String deviceType,
4     String interactionType, String dateTime, String userId,
5     String latitude, String longitude, String operationPerformed,
6     String InstaceId, List<String> groupComponent,
7     List<String> ungroupComponent, List<ComponentData> cotsget);
8   public void manageRuntimeProperty(String componentId, String property,
9     String value);
10 }
```

Base de datos de componentes e instancias *widgets*

Las bases de datos utilizadas para almacenar tanto los componentes *widgets* como las instancias de dichos componentes, son la Base de datos de componentes *widgets* (Widgets components DB) y la Base de datos de instancias *widgets* (Widgets instances DB). La primera se utiliza para almacenar todos los componentes de tipo *widget* que pueden ser utilizados por las aplicaciones *mashup* de tipo Web. La segunda BD se utiliza para almacenar las instancias asociadas a cada componente. Ambas bases de datos son gestionadas por un servidor *Apache Wookie* tal y como se puede ver en la Figura 3.1. Este servidor gestiona componentes *widgets* que siguen el estándar de W3C. Para dar de alta un componente en este servidor se necesita acceder a uno de sus servicios REST y registrar el componente. Cuando se desea registrar un componente, el servidor comprueba si dicho componente ha sido construido según la especificación de componente de W3C. En ese proceso el servidor tiene en cuenta la estructura de archivos del componente, además de comprobar el contenido del archivo `config.xml`. El servidor Wookie genera una instancia de un componente *widget* para cada componente que se utiliza en una aplicación *mashup*. Cada instancia de un componente está asociada de forma única a una aplicación. Por supuesto, la infraestructura permite que el mismo componente tenga más de una instancia en la misma o en diferentes aplicaciones. Al generarse distintas instancias del mismo componente, la interacción que se produzca en cada una de ellas, no influye sobre las otras. La instancia es accesible en el servidor Wookie por medio de una URL. Para generar una instancia de un componente concreto, el servidor Woo-

kie recibe una solicitud de generación de instancia por medio de un servicio REST. A continuación el servidor obtiene el componente de la BD de componentes y genera una instancia específica para la solicitud que se almacena en la BD de instancias.

El controlador asociado a las bases de datos de los componentes *widgets* es el controlador de gestión de Wookie **Manage Wookie**. Este controlador permite realizar todo el conjunto de operaciones vinculadas con la gestión de los componentes e instancias de componentes de tipo *widgets* en las bases de datos. La interfaz del controlador se encuentra en el siguiente listado. Como se puede observar, en la línea 2 se identifica el método `initManageWookie`, el cual se encarga de inicializar el controlador. En la línea 3 se encuentra el método `getOrCreateWidgetInstance`, encargado de generar las instancias para un componente dado. Para ello, a este método se le pasa por parámetros el identificador del usuario (`userID`), el nombre del componente (`componentName`) y el alias del componente (`componentAlias`). El resto de funcionalidad asociada a este controlador (eliminación, modificación, etc.) no ha sido implementada debido a que dichas operaciones se realizan directamente utilizando la API del repositorio de componentes Wookie, aunque está previsto incorporar la implementación de dichos métodos en una próxima versión del COScore.

```
// Implementación del Controlador "Manage Wookie"
1 public class ManageWookie {
2  public void initManageWookie();
3  public WidgetData getOrCreateWidgetInstance(String userID,
4    String componentName, String componentAlias);
5 }
```

Base de datos de componentes e instancias Java

Al igual que ocurre para los componentes de tipo *widget*, los componentes de tipo Java son almacenados en dos bases de datos. Por un lado se encuentra la BD de componentes Java (*Java components DB*), la cual almacena todos los componentes que pueden ser incorporados en las aplicaciones *mashup* de tipo Java. Por otro lado se encuentra la BD de instancias Java (Java instances DB), la cual se utiliza para almacenar las instancias de los componentes Java. Ambos repositorios son directorios gestionados por el Servidor de componentes Java (Figura 3.1), en el cual se almacenan los componentes e instancias de componentes respectivamente. Para dar de alta un componente en el repositorio es suficiente con almacenar el archivo `.jar` del componente en el directorio.

Al igual que ocurre con el repositorio de instancias de *widgets*, se necesita una instancia de un componente de tipo Java para cada usuario con el propósito de evitar que la interacción de un usuario sobre su componente afecte al componente de otro usuario. El servidor de componentes Java es el encargado de generar estas instancias para cada usuario. Este servidor gestiona tanto el repositorio de componentes como el de sus instancias. La comunicación con el servidor se realiza por medio de *sockets*. Cuando se solicita una nueva instancia de un componente, el servidor genera dicha instancia a partir del componente ubicado en el repositorio de componentes Java, a continuación, guarda la instancia en el repositorio de instancias y, por último, devuelve la instancia como un objeto Java a través del *socket*.

© 2016 Vallecillos, J.

El controlador de gestión de Java **Manage Java** es el encargado de llevar a cabo las tareas relacionadas con la gestión de componentes de tipo Java en ambas bases de datos. Al igual que ocurre con el controlador *Manage Wookie* con respecto a las bases de datos para los componentes de tipo *widgets*, este controlador permite generar instancias de componentes Java. La implementación del controlador se puede observar en el siguiente listado. En la línea 2 de este código se describe el método `initManageJava`, encargado de manejar el inicio de las variables del controlador. El otro método implementado es `getOrCreateJavaInstance` (línea 3), utilizado para crear las instancias de los componentes Java. Para ello, a este método se le pasa el identificador del usuario (`userID`), el identificador del componente Java (`componentJavaID`) y el nombre del componente Java (`componentJavaName`).

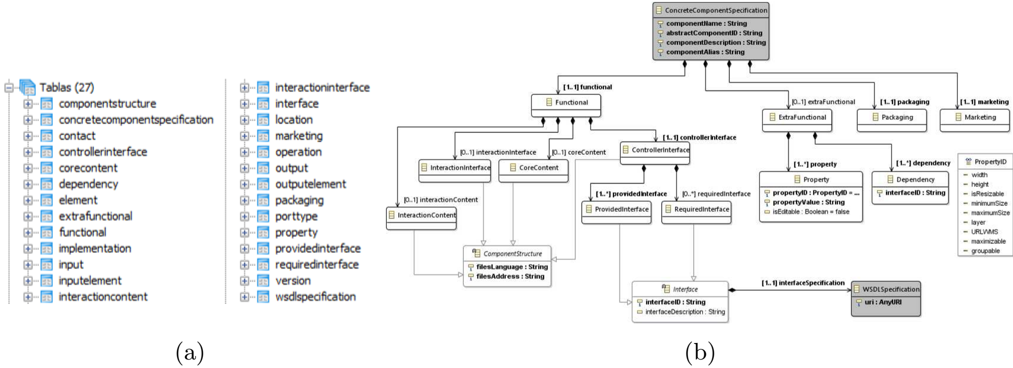
```
// Implementación del Controlador "Manage Java"
1 public class ManageJava {
2   public void initManageJava();
3   public JavaComponentResponse getOrCreateJavaInstance(String userID,
4     String componentJavaID, String componentJavaName);
5 }
```

Base de datos de especificaciones de componentes concretos

Por último, tenemos la Base de datos de especificaciones de componentes concretos (*Concrete Components Specification DB*). Esta base de datos almacena las especificaciones concretas de los componentes. Estas especificaciones son creadas mediante el metamodelo visto en la Sección 2.4.2 y son almacenadas en esta base de datos PostgreSQL. Al igual que ocurre con la base de datos de modelos de arquitecturas, la base de datos de modelos de especificaciones de componentes está formada por un conjunto de tablas asociadas a las diferentes entidades que forman el metamodelo de componente (Figura 3.3a). Al igual que con la base de datos de modelos de arquitectura, también se ha utilizado Hibernate para su gestión.

El controlador utilizado para esta base de datos es la gestión de especificaciones de componentes **Manage Component Specifications**. Este controlador permite llevar a cabo la gestión de los modelos de los componentes concretos que forman parte del entorno. Para cada componente que exista en el sistema habrá un modelo de dicho componente. Cuando se realice un proceso de inicialización de la aplicación a partir del modelo de arquitectura, se llevará a cabo una consulta del modelo del componente concreto para recopilar cierta información de dicho componente y poder reconstruir la aplicación *mashup* al usuario. La implementación del controlador se puede observar en el listado que hay más abajo.

En la línea 2 se muestra la interfaz del método `initializateDataStore`, encargado de inicializar el conjunto de variables necesarias para manejar la base de datos. En la línea 3, el método `exportCCFromString` se encarga de añadir un componente concreto por medio de un XMI en forma de cadena de texto (parámetro `ccFileString`). En la línea 4, el método `exportCCFromParams` se encarga de añadir un componente concreto a través de un conjunto de parámetros, como el nombre (parámetro `componentName`), el alias (`componentAlias`), la descripción (`componentDescription`), etc. El método



(a) (b)

Figura 3.3: (a) Tablas de la base de datos de especificaciones de componentes concretos (b), junto al metamodelo de especificaciones de componentes concretos

`withdrawCC` (línea 14), permite eliminar un componente concreto existente a partir del nombre del componente (`componentName`). El método `queryComponentPlatform` (línea 15) nos permite conocer la plataforma de un componente a partir de su nombre, lo cual puede ser necesario saber en un proceso de inicialización. Por último, el método `readComponentProperty` (línea 16) devuelve el conjunto de propiedades de tipo `RuntimeProperty` asociadas a un componente, a partir del nombre de dicho componente.

```
// Implementación del Controlador "Manage Component Specifications"
1 public class ManageComponentSpecifications {
2    public void initializateDataStores();
3    public String exportCCFromString(String ccFileString);
4    public String exportCCFromParams(String componentName, String componentAlias,
5      String componentDescription, String entityId, String entityName,
6      String entityDescription, String contactDescription, String personName,
7      String email, String phone, String address,
8      String versionId, String versionDate, String programmingLanguage,
9      String platformType, String repositoryId, String repositoryType,
10     String repositoryURI, String componentURI, String[] propertyId,
11     String[] propertyValue, boolean[] isEditable,
12     String dependencyInterfaceId, String[] requiredProvided,
13     String[] interfaceId,  String[] interfaceDescription, String[] anyUri);
14   public String withdrawCC(String componentName);
15   public String queryComponentPlatform(String componentName);
16   public List<RuntimeProperty> readComponentProperty(String componentName);
17 }
```

3.2.2. Módulos

En el tercer nivel de la infraestructura, por encima del nivel de controladores, se encuentra el nivel de *Módulos* (*Modules*), como se puede observar en la Figura 3.1. La funcionalidad ofrecida a través de los servicios, que las aplicaciones *mashup* utilizan, nacen de estos módulos. A continuación, se describe cada uno de los módulos disponibles.

© 2016 Vallecillos, J.

Módulo de información de usuarios

El Módulo de información de usuarios (*User Information Module*, UIM), administra los usuarios que forman parte del entorno. La funcionalidad de este módulo está relacionada con el alta, baja y modificación de usuarios en el COScore. Para ello, recibe la información que procede del servicio y la envía al controlador que guarda dicha información en la base de datos de usuarios. También gestiona la encriptación de la contraseña para el usuario. La interfaz de este módulo puede ser observada en el siguiente listado.

```
// Implementación del Módulo UIM
1 public class UIM {
2   public CreateUserResult createUser(CreateUserParams params);
3   public QueryUserResult queryUser(QueryUserParams params);
4   public QueryProfileResult queryProfile();
5   public InterModulesData queryCamUser(String userID);
6   public InterModulesData queryCamProfile(String profileName);
7   public DeleteUserResult deleteUser(String userID);
8   public UpdateUserResult updateUser(String userID, String userName,
9     String userPassword, String userProfile);
10 }
```

En la línea 2 aparece el método `createUser`, utilizado para crear un usuario en el entorno. Para ello, se le pasa un objeto `CreateUserParams` que contiene un conjunto de parámetros necesarios para dar de alta al usuario. En la línea 3 se encuentra el método `queryUser`, encargado de consultar un usuario del entorno. Este método recibe por parámetro un objeto de tipo `QueryUserParams` que contiene toda la información necesaria para poder ser consultado dicho usuario. En la línea 4 está el método `queryProfile`. Este método permite consultar el conjunto de tipos de perfiles de usuario. El método `queryCamUser` (línea 5), permite obtener el modelo de arquitectura concreta de una aplicación, a partir del identificador del usuario (`userID`) pasado por parámetro. El método `deleteUser` (línea 7) permite eliminar un usuario de la base de datos *Architectural models and users*. Por último, en la línea 8 aparece el método `updateUser`, que puede se utilizado para actualizar un usuario. Para ello, se le pasa por parámetros a dicho método el identificador de usuario (`userID`), el nombre del usuario (`userName`), el password del usuario (`userPassword`) y el perfil del usuario (`userProfile`).

Módulo de gestión de la interacción

El Módulo de gestión de la interacción (*Interaction Management Module*, IMM) se encarga de dar soporte a la interacción del usuario que se produce en la aplicación *mashup*. Los eventos de usuario generados dentro de los propios componentes son inaccesibles por el módulo IMM, como se ha indicado anteriormente. Sin embargo, los eventos producidos en el entorno son guardados con el propósito de ayudar a deducir el comportamiento del usuario y así, poder adaptar la aplicación *mashup* a sus necesidades. Estos eventos están relacionados con cambios de posición, tamaño de los componentes, eliminación y agregación de componentes al entorno y mensajes intercambiados entre los componentes que constituyen la aplicación. En el siguiente listado se encuentra parte de la implementación del módulo, formada por un único método llamado `registerInteraction`.

```
// Implementación del Módulo IMM
1 public class IMM {
2   public InterModulesData registerInteraction(String newSession,
3     String deviceType, String interactionType, String dateTime,
4     String userId, String latitude, String longitude,
5     String operationPerformed, String componentId,
6     List<String> groupComponent, List<String> ungroupComponent,
7     List<ComponentData> cotsget);
8 }
```

Este método almacena los eventos producidos en la aplicación en la base de datos *Interaction*. Para ello, a este método se le pasa por parámetro la sesión (`newSession`) en la cual se produjo la interacción asociada a un usuario, el tipo de dispositivo desde el que se produjo la interacción (`deviceType`), el tipo de interacción que se produjo (`interactionType`), el momento en el que se produjo dicha interacción (`dateTime`), el identificador del usuario (`userId`), la latitud y la longitud (`latitude` y `longitude`) asociada al usuario que realiza el proceso de interacción, la operación que produjo el evento (`operationPerformed`), el identificador del componente (`componentId`), el conjunto de componentes (`groupComponent`) que se agruparon como consecuencia de la interacción (sólo para una aplicación *mashup* de tipo web), el conjunto de componentes (`ungroupComponent`) que pudieron ser desagrupados (al igual que el parámetro anterior sólo es introducido cuando se trata de una aplicación *mashup* de tipo web) y por último, el parámetro `cotsget` indica la lista de componentes que forman la aplicación.

Módulo de visualización de componentes

El módulo *Display Management Module*, DMM se encarga de gestionar los procesos de agregación y eliminación de componentes en el entorno. Este módulo decide la primera configuración de componentes cuando la aplicación se inicia. Para ello, se tiene en cuenta el modelo de arquitectura inicial para el usuario en cuestión. Dependiendo de la plataforma utilizada por el usuario, este módulo generará un código asociado a dicha plataforma, en el cual se incrustarán las instancias de los componentes de dicha aplicación. Los componentes que sean visibles a los usuarios, se adaptarán a la visualización del dispositivo donde se están ejecutando. En próximas versiones, está previsto que este módulo pueda re-dimensionar, cambiar el color, o posición de los componentes según las necesidades del usuario. La implementación del módulo es como sigue:

```
// Implementación del Módulo DMM
1 public class DMM {
2   public InterModulesData getCurrentModelforUser(String userID, String camID);
3   public UpdateArchitectureResult updateArchitectureforUser(String userID,
4     String componentInstance, String actionDone,
5     List<ComponentData> newComponentData,
6     UserInteractionData interaction);
7   public InterModulesData readModelforcamId(String camID);
8   public InterModulesData saveModelforcamId(ConcreteArchitecturalModel cam);
9   public InterModulesData deleteModelforcamId(String camID);
12 }
```

© 2016 Vallecillos, J.

En la línea 2 se encuentra el método `getCurrentModelforUser`, utilizado por otros módulos para obtener el modelo de arquitectura concreta de la aplicación *mashup* de un usuario. Este método acepta por parámetros el identificador del usuario (`userID`) y el identificador del modelo de arquitectura concreta (`camID`). Por otro lado, en la línea 3 está el método `updateArchitectureforUser`, encargado de realizar las actualizaciones sobre el modelo de arquitectura de una aplicación. Este método se ejecuta cuando se han producido cambios en los componentes de una aplicación, como puede ser la eliminación de un componente, para reflejar dichos cambios en el modelo de la arquitectura.

Para ello, el método acepta por parámetros el identificador del usuario (`userID`) asociado a la aplicación *mashup*, la instancia del componente (`componentInstance`) sobre el que se produjo el evento, la acción (`actionDone`) realizada, la lista de componentes (`newComponentData`) por los cuales está formada la aplicación, así como la información asociada al usuario que generó la interacción (`UserInteractionData`), como por ejemplo, el tipo de interacción o la sesión. En la línea 7 está el método `readModelforcamId`, utilizado para devolver un modelo de arquitectura concreto a partir de un identificador de dicho modelo (`camID`). En la línea 8 se encuentra el método `saveModelforcamId`, utilizado para guardar en la base de datos *Architectural models and users* un objeto de tipo `ConcreteArchitecturalModel`. En la línea 9, el método `deleteModelforcamId` permite eliminar un modelo de arquitectura concreto de una aplicación, pasándole como parámetro el identificador del modelo (`camID`).

Módulo de gestión de la transacción

El Módulo de gestión de transacciones (*Transaction Management Module*, TMM) permite controlar el intercambio de mensajes que se produce entre los componentes. Por medio de este intercambio de mensajes se busca la coordinación entre ellos para lograr el correcto funcionamiento de la aplicación. La comunicación entre componentes se realiza de forma asíncrona, es decir, cuando un componente decide emitir un mensaje no espera un retorno del mismo, sino que continúa funcionando independientemente de lo que suceda. Todos los mensajes que se emitan por un puerto de salida son recibidos por su correspondiente puerto de entrada en otro componente. Para poder realizar esta gestión de comunicación entre los componentes, se hace uso del modelo de arquitectura de la aplicación *mashup* (*i.e.*, la arquitectura de componentes). Mediante este modelo se puede conocer qué componente(s) está relacionado con otro para resolver el proceso de comunicación. La interfaz de este módulo TMM es como aparece en el siguiente listado.

```
// Implementación del Módulo TMM
1 public class TMM {
2   public GetLinksResult createRoutingTable(String userID);
3   public GetLinksResult calculateConnectedPorts(String userID,
4     String componentInstance, String portID);
5 }
```

En la línea 2 se encuentra el método `createRoutingTable`, el cual se encarga de construir una estructura de datos en memoria principal a partir del modelo de arquitectura de la aplicación. Esta estructura de datos contiene los componentes del modelo de

arquitectura y las relaciones que existen entre dichos componentes. El método necesita que se le pase por parámetro el identificador del usuario (`userID`) para localizar el modelo de arquitectura correspondiente y poder así construir la estructura de datos. Por otro lado, en la línea 3 aparece el método `calculateConnectedPorts`. Este método se utiliza para obtener una lista de puertos (con sus respectivos componentes) que recibirán una información desde un puerto de otro componente que desea emitir dicha información. Para poder obtener esta lista, al método se le pasa por parámetros el identificador del usuario (`userID`), la instancia del componente (`componentInstance`) y el identificador del puerto del componente que desea emitir la información (`portID`).

Módulo de gestión de sesiones en el COScore

El Módulo de gestión de sesiones en el COScore (*COScore Session Management Module*, COSSessionMM) es un módulo utilizado para el proceso de inicio de una aplicación *mashup*. Para ello, este módulo crea una copia de los módulos IMM, TMM y DMM para poder dar soporte de forma independiente a cada aplicación. Estos módulos permanecerán activos mientras la sesión del usuario esté abierta o expire por inactividad. La interfaz del módulo `COSSessionMM` se muestra en el siguiente listado. El método `initContexts` (línea 2), permite inicializar el conjunto de variables necesarias para el módulo. En la línea 3 aparece el método `initializeModules`, cuya función consiste en inicializar los módulos asociados a una aplicación *mashup* perteneciente a un usuario. Para llevar a cabo esta tarea, a dicho método se le pasa por parámetro un nombre de usuario (`user`) y su password (`password`). En la línea 4 se encuentra el método `destroyModules`, utilizado para eliminar el conjunto de módulos que fueron creados en el inicio de una sesión. El método `initAnonymous` (línea 5), es el encargado de inicializar los módulos necesarios para un usuario anónimo. Este usuario tendrá predefinida la aplicación *mashup* que podrá utilizar en el entorno. Por último, el método `getUserEJB` (línea 6) se encarga de devolver el conjunto de módulos asociados a un usuario, a partir del identificador del usuario (`userID`).

```
// Implementación del Módulo COSSessionMM
1 public class COSSessionMM {
2   public void initContexts();
3   public LoginSessionResult initializeModules(String user, String password);
4   public LogoutSessionResult destroyModules(String userID);
5   public DefaultInitSessionResult initAnonymous();
6   public UserEJBs getUserEJB(String userID);
7 }
```

Una vez descritos los módulos que forman parte de la infraestructura desarrollada, en la siguiente sección se describirán los servicios privados, los cuáles se encuentran disponibles en el cuarto nivel de dicha infraestructura. Recordemos que hasta el momento han sido descritos los niveles inferiores de la infraestructura propuesta, referentes a las bases de datos, los controladores, y los módulos. En las siguientes secciones se describirán los servicios, tanto públicos como privamos. Siguiendo la descripción que se ha venido realizando, de abajo-arriba, de la Figura 3.1, es el turno de los servicios privados.

© 2016 Vallecillos, J.

3.3. Servicios privados

En esta sección se describirá el conjunto de los servicios privados S_{-} definidos en el modelo de infraestructura COScore para el despliegue de aplicaciones *mashup* propuesto en esta tesis doctoral. Como se ha venido comentando hasta el momento, el conjunto de servicios privados de COScore se compone de los servicios *User Service*, *Manage Architecture Service* y *Manage Component Service*. Para describir correctamente el funcionamiento de dichos servicios, en esta sección se utilizará una estructura de presentación similar para cada una de las operaciones que los componen. Dicha estructura constará de una introducción breve de la operación, una definición de las signaturas de la interfaz de la operación, el listado de los parámetros de entrada y salida que admite la operación, la descripción de la operación, la lista de mensajes de error permitida, una explicación de su comportamiento usando un diagrama de flujo de información, una descripción de la implementación junto con un fragmento del código más significativo, un ejemplo de petición de entrada en XML y otro ejemplo de devolución, y para finalizar, un ejemplo de error. Respecto a esto último, hay que destacar que para la implementación de los servicios se ha seguido un desarrollo guiado por pruebas TDD (*Test-Driven Development*) en donde para testar cada una de las operaciones de los servicios de COScore se ha elaborado un *juego de pruebas* y una *herramienta de pruebas en linea*[5].

3.3.1. Servicio *User Service*

El servicio *User Service* permite realizar la gestión de los usuarios y tiene como funciones básicas el alta y baja de usuarios, así como la consulta y modificación de información de dichos usuarios. Este servicio queda constituido por el siguiente conjunto de operaciones, y que se describirán con detalle a continuación:

— *Query user*: se emplea para comprobar que un usuario existe en el sistema.

— *Update user*: se emplea para actualizar la información del usuario en el sistema.

— *Delete user*: se emplea para eliminar un nuevo usuario del sistema.

— *Create user*: se emplea para crear un nuevo usuario en el sistema.

— *Query Profile*: se emplea para obtener la lista de perfiles del sistema.

3.3.1.1. Operación *Query User*

La operación *Query User* se engloba en el servicio *User Service* que da soporte a la gestión de usuarios. Esta operación se emplea con el objetivo de gestionar la existencia de usuarios en el sistema. Este servicio es gestionando por el componente módulo UIM (Módulo de Información de Usuarios) del COScore. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación. Las interfaces de las operaciones de servicio están descritas con etiquetas POJO (*Plain Old Java Object*). Para definir una interfaz se utilizan tres tipos de etiquetas: (a) `@WebMethod` para definir

[5] COScore API – `http://acg.ual.es/projects/enia/ui/webservices/`

el nombre de la operación; (b) `@WebResult` permite definir el nombre del mensaje de respuesta; y (c) `@WebParam` para declarar los parámetros de la operación. En adelante, el resto de las interfaces de las operaciones de servicio definidas en este documento siguen este formato de etiquetado POJO.

Interfaz:

```
1 @WebMethod(operationName="queryUser", action="queryUser")
2 @WebResult(name='result', targetNamespace="http://ws.cos.acg.ual.es/")
3       @XmlElement(required=true)
4 public QueryUserResult queryUser(
5    @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6       @XmlElement(required=true) QueryUserParams params,
7    @WebParam(name="privatekey", targetNamespace="http://ws.cos.acg.ual.es/")
8       @XmlElement(required=true) String privatekey);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un tipo *structure* `QueryUserParams` con los parámetros de entrada y el orden a seguir.		
	string	`userName`	Nombre de usuario a consultar en el sistema. Parámetro obligatorio no nulo.
	string	`userPassword`	Password para ese usuario, se encripta y se comprueba con el de la base de datos. Parámetro obligatorio no nulo
`privatekey`	Una clave para poder acceder al servicio. Parámetro obligatorio no nulo.		
Parámetros de salida			
`result`	Un tipo *structure* `QueryUserResult` con los valores de salida.		
	boolean	`validation`	Variable que muestra si tuvo éxito la acción, y toma un valor *true* si tiene éxito la consulta en la base de datos y la validación del password; *false* en caso contrario.
	int	`iduser`	Identificador del usuario en la base de datos. Devuelve -1 si no encuentra o autentifica al usuario.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: Como se observa en su interfaz, la operación acepta en la entrada una estructura que contiene los valores del nuevo nombre de usuario y el *password*, siendo ambos valores obligatorios y no nulos. La operación encripta el *password* y consulta en la BD si el usuario y su clave existen. Como respuesta, devuelve una estructura, con una variable que indica si el usuario y su *password* existen, otra con su identificador en la BD y un mensaje correspondiente a la operación realizada. Para cada usuario, se le asigna un modelo de arquitectura asociado a su aplicación *mashup*. Por ejemplo, para el caso de una aplicación *mashup* de usuario gráfica basada en componentes *widgets*, como la anunciada antes (ENIA) y que se describirá en detalle en el siguiente capítulo, un usuario registrado en el sistema tiene asociado un estado de la arquitectura de la interfaz, que se corresponde con un modelo abstracto de los componentes COTSgets (componentes *widgets*) de la interfaz de usuario. Como se puede observar en su interfaz, esta operación recibe por parámetro un objeto de tipo `QueryUserParams` y una cadena

© 2016 Vallecillos, J.

de texto (`privatekey`) como clave usada por los usuarios que conocen la operación (de forma similar a una *API key*, normalmente utilizada para este fin en la construcción de servicios). La estructura `QueryUserResult` es devuelta como resultado de la consulta del usuario del sistema. Esta estructura contiene un *booleano* que indica si el usuario forma parte del sistema. Además, contiene un identificador del usuario que se consultó, valor que será "-1" si no se encuentra el usuario en el sistema. Por último, guarda un mensaje en forma de cadena de texto para informar acerca del resultado del proceso de consulta del usuario. En la siguiente tabla se muestra un listado con los tipos de mensajes devueltos por la operación.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al aplicar algoritmos de encriptación o al buscar alguna clase necesaria para la ejecución.
`Private key Error`	Se produce por clave privada incorrecta o se omite este parámetro. El servicio es Privado y se necesita una clave para acceder a él.
`Not found or Empty username Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty userpassword Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor
`Validation Error user password incorrect`	Se produce cuando el usuario no existe en la BD o el password es incorrecto.
`PSQLException`	Se produce por problemas en la conexión a la base de datos.

Comportamiento: En la Figura 3.4 se muestra un diagrama del flujo de información de *Query user*. En esta figura se observa cómo la operación recibe una solicitud (en forma de mensaje SOAP), la cual llega al módulo UIM, que a su vez hace uso del controlador *Manage users*, a través de la tarea *Process to query an user* del módulo UIM. Este módulo se pone en contacto con la base de datos *Architectural models and user* para comprobar si el usuario se encuentra registrado en el sistema y su modelo de arquitectura *mashup* asociado. Hay que recordar que el modelo de arquitectura de una aplicación *mashup* se refiere al estado de la configuración de la arquitectura de componentes que el usuario tiene asociado. Por ejemplo, para el caso de una aplicación *mashup* de una interfaz de usuario gráfica, compuesta por una colección de componentes que el usuario ve y controla en su perfil, el modelo de la arquitectura se referirá al estado de la configuración de componentes en el que se encuentra la sesión del usuario. En el caso de estar el usuario dado de alta, se obtiene su identificador. Por último, el controlador devuelve el "id" del usuario al módulo, que se encarga de devolverlo a través del servicio.

Implementación: Como se observa en el código de la implementación de la operación `queryUser`, mostrado en el listado siguiente, primero se comprueba que los parámetros de entrada son los correctos. Se comprueba que tanto el valor `privatekey`, como el nombre del usuario y el *password* del usuario son correctos. A continuación, se obtiene una instancia del módulo UIM (líneas 11-15). Dicha instancia del módulo hace

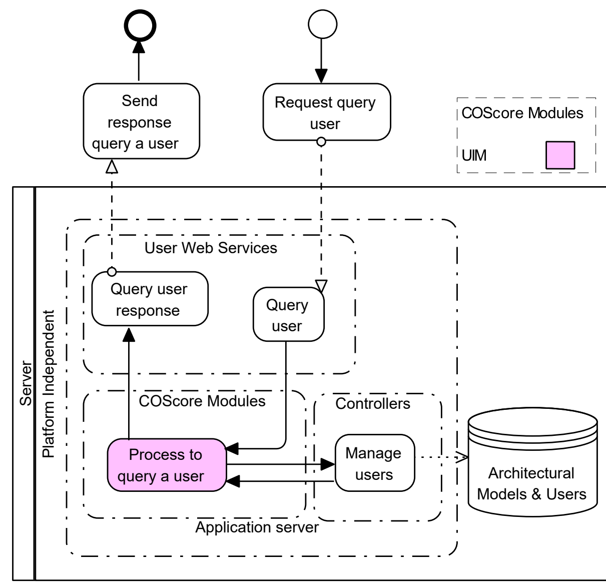


Figura 3.4: Operación *Query user* de *User Service*

uso de la operación `queryUser` al que le pasa los parámetros necesarios para realizar la consulta: `QueryUserParams` y `privatekey`. El parámetro `QueryUserParams` es un objeto que contiene un nombre de usuario y una contraseña, necesarios para conocer si dicho usuario tiene alguna aplicación *mashup* gestionada por la infraestructura. El parámetro `privatekey` permite hacer uso de esta operación privada. Una vez pasados los parámetros al método, este método se pone en contacto con la base de datos de usuarios utilizando el controlador *User Manager* para acceder a la base de datos y comprobar que dicho usuario existe.

```
 1 public QueryUserResult queryUser(QueryUserParams params, String privatekey){
 2
 3 QueryUserResult queryUserResult = new QueryUserResult();
 4 //First: check the private key
 5 if(privatekey!=null && this.privatekey.compareTo(privatekey) == 0) {
 6     //Second: check the params.username
 7     if(params.getUserName()!=null && params.getUserName().compareTo("") != 0){
 8        //Third: check the params.userpassword
 9        if(params.getUserPassword() != null && params.getUserPassword().
10                                                       compareTo("") != 0){
11           Context initialContext;
12           try {
13              initialContext = new InitialContext();
14              UIM userInformation = (UIM)initialContext.
15                                                lookup("java:app/cos/UIM");
16              queryUserResult = userInformation.queryUser(params);
17           }        catch (NamingException e) {
18              LOGGER.error(e);
```

© 2016 Vallecillos, J.

```
19              queryUserResult.setValidation(false);
20              queryUserResult.setIduser(-1);
21              queryUserResult.setMessage("> Internal Server Error");
22            }
23        } else {
24           queryUserResult.setValidation(false);
25           queryUserResult.setIduser(-1);
26           queryUserResult.setMessage("> Not found or Empty userpassword Error");
27           LOGGER.error("Not found or Empty userpassword Error");
28        }
29     } else {
30        queryUserResult.setValidation(false);
31        queryUserResult.setIduser(-1);
32        queryUserResult.setMessage("> Not found or Empty username Error");
33        LOGGER.error("Not found or Empty username Error");
34       }
35 } else {
36    queryUserResult.setValidation(false);
37    queryUserResult.setIduser(-1);
38    queryUserResult.setMessage("> Private key Error");
39    LOGGER.error("Private key error");
40 }
41 return queryUserResult;
42 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ws="http://ws.cos.acg.ual.es/">
   <soapenv:Header/>
   <soapenv:Body>
     <ws:queryUser>
       <params>
         <userName> ejemplo nombre </userName>
         <userPassword> ejemplo password </userPassword>
       </params>
       <privatekey> ejemplo clave </privatekey>
     </ws:queryUser>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:queryUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <validation> ejemplo true o false </validation>
        <iduser> ejemplo número con el ID o -1 </iduser>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:queryUserResponse>
 </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:queryUserResponse xmlns:ns2=‘http://ws.cos.acg.ual.es/’>
      <result>
        <validation> false </validation>
        <iduser> -1 </iduser>
        <message> Validation Error user o password incorrect </message>
      </result>
    </ns2:queryUserResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.1.2. Operación *Update User*

La operación *Update User* es una función perteneciente al servicio *User Service* que da soporte a la gestión de usuarios. Esta operación se emplea para actualizar la información de usuario en el sistema. Este servicio es controlado por el componente UIM (Modulo de Información de Usuarios) y hace uso del componente COSSessionMM (Modulo de Gestión de Sesiones del COSCore) para poder obtener el modelo concreto de arquitectura del perfil seleccionado, modificarlo y asignarlo al usuario actualizado. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="updateUser", action="updateUser")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3        @XmlElement(required=true)
4 public UpdateUserResult updateUser(
5     @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6        @XmlElement(required=true) UpdateUserParams params,
7     @WebParam(name="privatekey", targetNamespace = "http://ws.cos.acg.ual.es/")
8        @XmlElement(required=true) String privatekey);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un *structure* `UpdateUserParams` con los parámetros:		
	string	`userId`	Identificador del usuario, se requiere que exista en el sistema. Parámetro obligatorio no nulo.
	string	`newUserName`	Nuevo nombre del usuario. Parámetro obligatorio no nulo.
	string	`newUserPassword`	Nuevo Password para el usuario. Parámetro obligatorio no nulo.
	string	`newUserProfile`	Nuevo Perfil para el usuario. Parámetro obligatorio no nulo.
`privatekey`	Un *string* con la clave de acceso al servicio. Parámetro obligatorio no nulo.		
Parámetros de salida			
`result`	Un tipo *structure* `UpdateUserResult` con los valores de salida:		
	boolean	`updated`	Toma el valor *true* si tiene éxito la actualización en la BD, *false* si no.
	string	`message`	Mensaje de éxito o de error y su tipo.

© 2016 Vallecillos, J.

Descripción: El método acepta como entrada la estructura `UpdateUserParams` con los valores identificador de usuario, nuevo nombre de usuario, nuevo *password* y nuevo perfil; estos valores son obligatorios y además no nulos. La operación encripta la clave e introduce los valores en la BD para el usuario al que pertenece el identificador. Devuelve como respuesta una estructura con una variable que indica si la actualización ha tenido éxito o no, y su mensaje correspondiente. Como se puede ver en su interfaz, la operación recibe como parámetro la estructura `UpdateUserParams` y una cadena `privatekey` como clave usada por los usuarios que conocen la operación. La estructura `UpdateUserResult` es devuelta como resultado del proceso de actualizar el usuario del sistema. Dicha estructura está compuesta por un *booleano* para indicar si el usuario pudo ser actualizado, y de un mensaje con una breve descripción del resultado de la operación. En la siguiente tabla se resume la lista de los posibles mensajes devueltos por la operación.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al aplicar algoritmos de encriptación o al buscar alguna clase necesaria para la ejecución.
`Private key Error`	Se produce por clave privada incorrecta o se omite este parámetro. El servicio es Privado y se necesita una clave para acceder a él.
`Not found or Empty userid Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty new username Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty new userpassword Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty new userprofile Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Error in Architectural Models BD`	Se produce por problemas en la consulta a la base de datos de modelos de arquitectura.
`Error in Wookie`	Se produce por problemas en la consulta al repositorio Wookie.
`PSQLException`	Se produce por problemas en la conexión a la base de datos. El error más común se produce por intentar registrar un usuario nuevo con un nombre ya existente en la BD, siendo este el mensaje de error: **`>org.postgresql.util.PSQLException: ERROR: llave duplicada viola restricción de unicidad «unique_username»`**.

Comportamiento: En la Figura 3.5 se muestra un diagrama del flujo de información de *Update user*. En la figura se observa cómo la operación se encarga de actualizar un usuario que fue registrado en la base de datos de usuarios. Esta operación se comunica con la tarea *Process to update an user* del módulo *UIM* del COScore para pasarle la información del usuario que debe ser actualizada. A su vez, el módulo se comunica con el controlador *Manage users* que se pone en contacto con la base de datos *Architectural models and users* para guardar la nueva información del usuario.

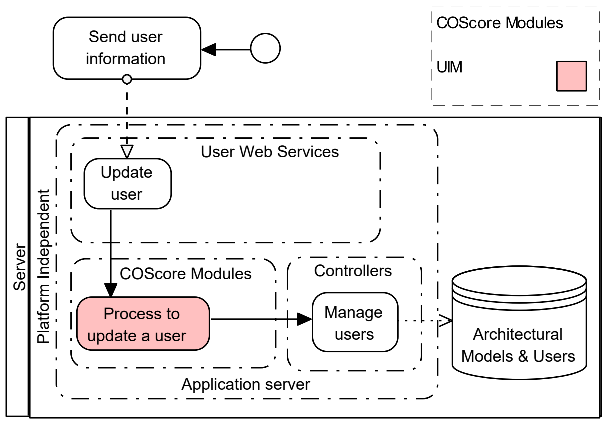


Figura 3.5: Operación *Update user* de *User Service*

Implementación: Como se observa en el código de la operación, la cual se muestra a continuación, primero se comprueban la validez de los parámetros de entrada, estos es, se comprueba que el valor `privatekey` es correcto, que el identificador del usuario no esté vacío, y que el nombre del usuario y la clave están bien escritas y son correctas. Después, se inicializa un contexto para obtener una instancia del módulo UIM. Dicha instancia del módulo hace uso de la operación `updateUser` pasándole el parámetro `UpdateUserParams` que contiene todos los valores nuevos para el usuario, como son el identificador del usuario, nombre del usuario, la clave y el perfil. Cuando se introducen estos parámetros, el método actualiza todos los campos del usuario modificándolos en la base de datos.

```
 1 public UpdateUserResult updateUser(UpdateUserParams params, String privatekey){
 2
 3 UpdateUserResult updateUserResult = new UpdateUserResult();
 4 //First: check the private key
 5 if(privatekey != null && this.privatekey.compareTo(privatekey) == 0) {
 6    //Second: check the params.userId
 7    if(params.getUserId() != null && params.getUserId().compareTo("") != 0){
 8       //Third: check the params.userName
 9       if(params.getNewUserName() != null && ...){
10          if(params.getNewUserPassword() != null && ...){
11             if(params.getNewUserProfile() != null && ...){
12                Context initialContext;
13                try {
14                   initialContext = new InitialContext();
15                   UIM userInformation = (UIM)initialContext.
16                                                   lookup("java:app/cos/UIM");
17                   updateUserResult = userInformation.updateUser(...);
18                } catch (NamingException e) {
19                   LOGGER.error(e);
20                   updateUserResult.setUpdated(false);
21                   updateUserResult.setMessage("> Internal Server Error");
22                }
23             } else {
```

```
24               updateUserResult.setUpdated(false);
25               updateUserResult.setMessage("> Not found or Empty new userprofile");
26               LOGGER.error("Not found or Empty userprofile Error");   }
27         } else {
28            updateUserResult.setUpdated(false);
29            updateUserResult.setMessage("> Not found or Empty new userpassword");
30            LOGGER.error("Not found or Empty userpassword Error");   }
31      } else {
32         updateUserResult.setUpdated(false);
33         updateUserResult.setMessage("> Not found or Empty new username");
34         LOGGER.error("Not found or Empty username Error");  }
35   } else {
36      updateUserResult.setUpdated(false);
37      updateUserResult.setMessage("> Not found or Empty userid Error");
38      LOGGER.error("Not found or Empty username Error"); }
39 } else {
40   updateUserResult.setUpdated(false);
41   updateUserResult.setMessage("> Private key Error");
42   LOGGER.error("Private key error");
43 }
44 return updateUserResult;
45 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                 xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:updateUser>
      <params>
        <userId> ejemplo identificador </userId>
        <newUserName> ejemplo nombre </newUserName>
        <newUserPassword> ejemplo password </newUserPassword>
        <newUserProfile> ejemplo perfil </newUserProfile>
      </params>
      <privatekey> ejemplo clave </privatekey>
    </ws:updateUser>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:updateUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <updated> ejemplo true o false</updated>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
      </ns2:updateUserResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:updateUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <updated> false </updated>
        <message> > Not found o Empty new userprofile Error </message>
      </result>
    </ns2:updateUserResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.1.3. Operación *Delete User*

La operación *Delete User* se engloba en el servicio *User Service* que da soporte a la gestión de usuarios. Esta operación se emplea con el objetivo de borrar un usuario del sistema que tiene acceso a una aplicación *mashup*. Este servicio es controlado por el componente UIM (Módulo de Información de Usuarios). La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="deleteUser", action="deleteUser")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3        @XmlElement(required=true)
4 public DeleteUserResult deleteUser(
5    @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6        @XmlElement(required=true) DeleteUserParams params,
7    @WebParam(name="privatekey", targetNamespace="http://ws.cos.acg.ual.es/")
8        @XmlElement(required=true) String privatekey);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un valor *structure* `DeleteUserParams` con los parámetros:		
	string	`userId`	Identificador de usuario, se requiere que exista en el sistema. Parámetro obligatorio no nulo.
	privatekey	`string`	Una clave para poder acceder al servicio. Parámetro obligatorio no nulo.
Parámetros de salida			
`result`	Un valor *structure* `DeleteUserResult` con los valores de salida:		
	boolean	`deleted`	*true* si tiene éxito el borrado en la BD, false si no.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: Esta operación elimina un usuario de la base de datos *Architectural models and users*. El método acepta como entrada la estructura `DeleteUserParams` con el valor del ID de usuario, este valor es obligatorio y además no nulo. La operación realiza el borrado del usuario en la BD. Devuelve como respuesta una estructura con una variable que indica si el borrado ha tenido éxito o no, y su mensaje correspondiente.

© 2016 Vallecillos, J.

Como se puede ver en la interfaz, esta operación requiere por parámetro la estructura DeleteUserParams y una cadena de texto privatekey como clave usada por los usuarios que conocen la operación. La estructura DeleteUserResult es devuelta por la operación como resultado del proceso de borrado del usuario en el sistema. Esta clase está compuesta por un *booleano* que indica si el usuario pudo ser eliminado del sistema, y de un mensaje en forma de cadena texto para informar del resultado de la operación. En la siguiente tabla se muestra un listado con los mensajes que puede devolver la operación.

Mensajes de error:

error	descripción
Internal Server Error	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.
Private key Error	Se produce por clave privada incorrecta o se omite este parámetro. El servicio es Privado y se necesita una clave para acceder a él.
Not found or Empty userid Error	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
PSQLException	Se produce por problemas en la conexión a la BD. El error más común se produce por intentar registrar un usuario nuevo con un nombre ya existente en la BD, siendo este el mensaje de error: >org.postgresql.util.PSQLException: ERROR: llave duplicada viola restricción de unicidad «unique_username».

Comportamiento: Esta operación también hace uso del módulo *User Information Module* (UIM) para su funcionamiento. En la Figura 3.6 se muestra un diagrama del flujo de información de *Delete user*. Cuando se produce un proceso de eliminación de un usuario, se envía una solicitud de eliminación por medio de la operación *Delete user*. La operación envía la solicitud al módulo UIM a través de la tarea *Process to delete an user*. Después, el módulo se pone en contacto con el controlador *Manage users* que se comunica con la base de datos *Architectural models and user* para indicar que usuario debe ser eliminado. Como se ha comentado para las otras operaciones, cada usuario registrado en el sistema tiene asociado un estado del modelo de la arquitectura de la aplicación *mashup* que gestiona. Como ejemplo, en el caso de una aplicación *mashup* de una interfaz gráfica basada en componentes *widgets*, como la de ENIA, la operación de eliminación de un usuario en el sistema implica que el estado de la sesión, asociada a la configuración de los componentes de la interfaz, debe ser también borrado de la base de datos, además de la información del perfil de usuario.

Implementación: Como se observa en el código de la operación (abajo), primero se comprueba que los parámetros de entrada son los correctos. Para ello, se comprueba que el valor privatekey es válido y el identificador del usuario no está vacío. Después, se inicializa un contexto para obtener una instancia del módulo UIM, el cual gestiona el funcionamiento del servicio. La operación deleteUser es accesible desde el módulo pasándole el parámetro DeleteUserParams que hace uso del identificador del usuario, a partir del cual el controlador *Manage Users* accede a la base de datos de usuarios y eliminar el usuario y su modelo de arquitectura *mashup* asociado.

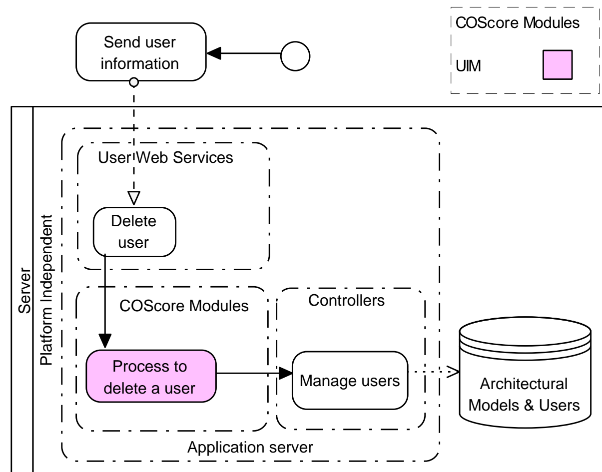


Figura 3.6: Operación *Delete user* de *User Service*

```
 1 public DeleteUserResult deleteUser(DeleteUserParams params, String privatekey){
 2
 3 DeleteUserResult deleteUserResult = new DeleteUserResult();
 4 if(privatekey!=null && this.privatekey.compareTo(privatekey) == 0) {
 5    if(params.getUserId()!=null && params.getUserId().compareTo("") != 0){
 6       Context initialContext;
 7       try {  initialContext = new InitialContext();
 8               UIM userInformation = (UIM)initialContext.lookup("java:app/cos/UIM");
 9               deleteUserResult = userInformation.deleteUser(params.getUserId());
10       } catch (NamingException e) {
11          LOGGER.error(e); deleteUserResult.setDeleted(false);
12          deleteUserResult.setMessage("> Internal Server Error"); }
13    } else {
14       deleteUserResult.setDeleted(false);
15       deleteUserResult.setMessage("> Not found o Empty userid Error");
16       LOGGER.error("Not found o Empty userid Error");  }
17 } else { deleteUserResult.setDeleted(false);
18    deleteUserResult.setMessage("> Private key Error");
19    LOGGER.error("Private key error");  }
20 return deleteUserResult;
21 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                 xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:deleteUser>
      <params>
        <userId> ejemplo identificador </userId >
```

© 2016 Vallecillos, J.

```
      </params>
      <privatekey> ejemplo clave </privatekey>
    </ws: deleteUser >
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:deleteUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <deleted> ejemplo true o false </ deleted >
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2: deleteUserResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:deleteUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <deleted> false </ deleted >
        <message> > org.postgresql.util.PSQLException: ERROR: llave duplicada
              viola restricción de unicidad «unique_username» </message>
       </result>
     </ns2: deleteUserResponse >
  </soap:Body>
</soap:Envelope>
```

3.3.1.4. Operación *Create User*

La operación *Create User* es otro de los métodos del servicio *User Service* que da soporte a la gestión de usuarios. Esta operación se emplea con el objetivo de crear un nuevo usuario en el sistema. Este servicio es controlado por el componente UIM (Módulo de Información de Usuarios) y hace uso del componente COSSessionMM (Módulo de Gestión de Sesiones del COSCore) para poder obtener el modelo concreto de arquitectura del perfil seleccionado, modificarlo y asignarlo a este nuevo usuario. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="createUser", action="createUser")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3       @XmlElement(required=true)
4 public CreateUserResult createUser(
5    @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6        @XmlElement(required=true) CreateUserParams params,
7     @WebParam(name="privatekey", targetNamespace="http://ws.cos.acg.ual.es/")
8        @XmlElement(required=true) String privatekey);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un valor *structure* `CreateUserParams` con los parámetros:		
	string	`userName`	Nombre del nuevo usuario, se requiere que no exista en el sistema. Parámetro obligatorio no nulo.
	string	`userPassword`	Password para ese nuevo usuario, se encriptará y se guardará en la BD. Obligatorio no nulo.
	string	`userProfile`	Perfil asignado a ese nuevo usuario. Parámetro obligatorio no nulo.
`privatekey`	Un *string* con la clave para acceder al servicio. Obligatorio no nulo.		
Parámetros de salida			
`result`	Un valor *structure* `CreateUserResult` con los valores de salida:		
	boolean	`created`	*true* si éxito la inserción en la BD, false si no.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: El método acepta en su entrada la estructura `CreateUserParams` con los valores del nuevo nombre de usuario, su clave y perfil. Estos valores son obligatorios y además no nulos. La operación encripta la clave e introduce los valores en la BD si el usuario no existe. Devuelve como respuesta una estructura con una variable que indica si la creación ha tenido éxito o no, y su mensaje correspondiente. Como se puede observar por tanto en la interfaz de la operación mostrada arriba, se recibe por parámetro un objeto de tipo `CreateUserParams` y una cadena de texto `privatekey` como clave usada por los usuarios que conocen la operación. La estructura `CreateUserResult` es devuelta como resultado del proceso de alta del usuario en el sistema. Esta clase contiene un valor *booleano* que indica si el usuario se pudo crear. Además, contiene un mensaje en forma de cadena de texto para informar sobre el resultado del proceso de creación del usuario. En la siguiente tabla se muestra el listado de mensajes de la operación.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al aplicar algoritmos de encriptación o al buscar alguna clase necesaria para la ejecución.
`Private key Error`	Se produce por clave privada incorrecta o se omite este parámetro. El servicio es Privado y se necesita una clave para acceder a él.
`Not found or Empty username Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty userpassword Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty userprofile Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Error in Architectural Models BD`	Se produce por problemas en la consulta a la base de datos de modelos de arquitectura.
`Error in Wookie`	Se produce por problemas en la consulta al repositorio Wookie.
`PSQLException`	Se produce por problemas en la conexión a la base de datos. Véase tabla de mensajes de la operación anterior, para más información.

© 2016 Vallecillos, J.

Comportamiento: En la Figura 3.7 se muestra un diagrama del flujo de información de la operación *Create user*. Cuando se crea un nuevo usuario en el sistema, se realiza una solicitud con la operación *Create user*. Esta operación del servicio hace uso del módulo UIM desde donde se ejecuta la tarea *Process to create a user* para comunicarse con el controlador *Manage users* que finalmente da de alta el usuario en la base de datos *Architectural models and users*. Como se ha venido comentando para las otras operaciones del servicio, el proceso de alta de usuario en el sistema implica también el alta de una arquitectura de componentes *mashup* para el usuario, la cual será una, establecida previamente por omisión en el sistema para todos los usuarios del mismo perfil. Así por ejemplo, para el caso de la interfaz de usuario *mashup* ENIA que se está usando como caso de aplicación *mashup* en esta tesis, un usuario nuevo que se da de alta en el sistema con un perfil de "agricultor", tendrá una interfaz gráfica de usuario asignada de partida, con una configuración de componentes *widgets* por omisión para ese perfil, y que el usuario luego podrá configurar. Para este caso, se almacena en la base de datos la información del nuevo usuario, los datos del perfil, y un modelo con la configuración base de la arquitectura *mashup*.

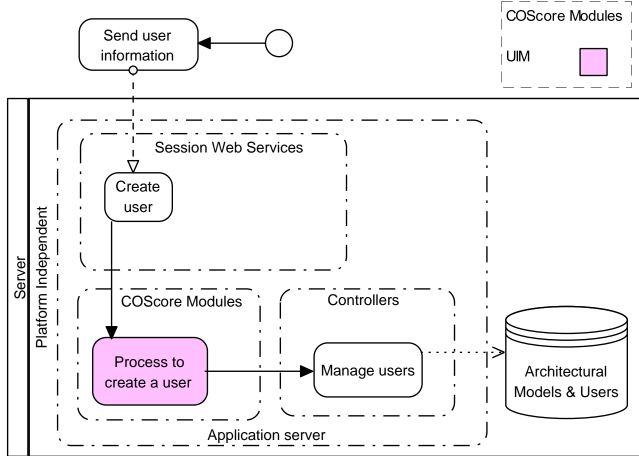


Figura 3.7: Operación *Create user* de *User Service*

Implementación: Como se puede observar en el código de la implementación que se muestra a continuación, primero se comprueban que los parámetros de entrada son los correctos. Para ello, se comprueba que el valor `privatekey` es correcto, que el nombre del usuario no está vacío, y que el identificador del usuario introducido es el correcto y el perfil de usuario no es nulo. Tras ello, se inicializa un contexto para crear una instancia del módulo UIM, el cual usa la operación `createUser` pasándole como parámetro el objeto *CreateUserParams*. Dicho objeto contiene el nombre del usuario que se da de alta, la contraseña del usuario y el perfil de usuario. Finalmente, este usuario se guarda en la base de datos de usuarios utilizando como mediador el controlador de gestión de usuarios *Manage Users*.

```
 1 public CreateUserResult createUser(CreateUserParams params, String privatekey){
 2
 3 CreateUserResult createUserResult = new CreateUserResult();
 4 if(privatekey!=null && this.privatekey.compareTo(privatekey) == 0) {
 5    if(params.getUserName()!=null && params.getUserName().compareTo("") != 0){
 6       if(params.getUserPassword()!=null &&
 7           params.getUserPassword().compareTo("") != 0){
 8          if(params.getUserProfile()!=null && params.getUserProfile().
 9                                                         compareTo("") != 0){
10             Context initialContext;
11             try {
12                initialContext = new InitialContext();
13                UIM userInformation = (UIM)initialContext.
14                                                  lookup("java:app/cos/UIM");
15                createUserResult = userInformation.createUser(params);
16             } catch (NamingException e) {
17                LOGGER.error(e);
18                createUserResult.setCreated(false);
19                createUserResult.setMessage("> Internal Server Error"); }
20          } else {
21             createUserResult.setCreated(false);
22             createUserResult.setMessage("> Not found or Empty userprofile");
23             LOGGER.error("Not found or Empty userprofile Error"); }
24       } else {
25          createUserResult.setCreated(false);
26          createUserResult.setMessage("> Not found or Empty userpassword");
27          LOGGER.error("Not found or Empty userpassword Error"); }
28    } else {
29       createUserResult.setCreated(false);
30       createUserResult.setMessage("> Not found or Empty username");
31       LOGGER.error("Not found or Empty username"); }
32 } else {
33    createUserResult.setCreated(false);
34    createUserResult.setMessage("> Private key Error");
35    LOGGER.error("Private key error"); }
36 return createUserResult;
37 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                 xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:createUser>
      <params>
        <userName> ejemplo nombre </userName>
        <userPassword> ejemplo password </userPassword>
        <userProfile> ejemplo perfil </userProfile>
      </params>
      <privatekey> ejemplo clave </ privatekey>
    </ws:createUser>
  </soapenv:Body>
</soapenv:Envelope>
```

© 2016 Vallecillos, J.

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:createUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <created> ejemplo true o false </created>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:createUserResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:createUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <created> false </created>
        <message> > org.postgresql.util.PSQLException: ERROR: llave duplicada
              viola restricción de unicidad «unique_username» </message>
      </result>
    </ns2:createUserResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.1.5. Operación *Query Profile*

La operación *Query Profile* es el último método del servicio *User Service* que da soporte a la gestión de usuarios. Esta operación se emplea con el objetivo de obtener la lista de perfiles del sistema. Este servicio es controlado por el componente UIM (Módulo de Información de Usuarios). La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="queryProfile", action="queryProfile")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3       @XmlElement(required=true)
4 public QueryProfileResult queryProfile(
5   @WebParam(name="privatekey", targetNamespace="http://ws.cos.acg.ual.es/")
6       @XmlElement(required=true) String privatekey);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`privatekey`	Un *string* de la clave para acceder al servicio. Obligatorio no nulo.		
Parámetros de salida			
`result`	Un valor *structure* `QueryProfileResult` con los valores de salida:		
	boolean	`validation`	*true* si éxito consulta en la BD, false si no.
	list string	`profiles`	Lista de perfiles del sistema.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: Como se observa en la interfaz, la operación acepta como entrada una clave privada `privatekey` de acceso al servicio. Realiza una consulta en la BD para obtener la lista completa de perfiles. Devuelve como respuesta una estructura con una variable que indica si la consulta ha tenido éxito o no, una lista de perfiles y un mensaje correspondiente a la operación realizada. La estructura `QueryProfileResult` es devuelta como resultado del proceso de consulta del conjunto de perfiles que hay dados de alta en el sistema. Dicha clase contiene un valor *booleano* que indica el éxito o no de la consulta en la base de datos; también contiene la lista de perfiles disponibles en el sistema, y un mensaje en forma de cadena de texto para informar del éxito de la consulta. En la siguiente tabla se muestran los posibles mensajes de la operación.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.
`Private key Error`	Se produce por clave privada incorrecta o se omite. El servicio es Privado y se necesita una clave para acceder a él.
`PSQLException`	Se produce por problemas en la conexión a la base de datos.

Comportamiento: En la Figura 3.8 se muestra un diagrama del flujo de información de la operación *Query profile*. El proceso se inicia con una petición para la obtención de la lista de perfiles de usuario con una llamada a la operación *Query profile* del servicio. Dicho servicio hace uso del módulo UIM el cual implementa la funcionalidad asociada a la operación. Para ello, el módulo ejecuta la tarea *Process to query an user profile* que se encarga de comunicar con el controlador *Manage architectures* de la base datos para realizar la consulta pertinente. Dicho controlador consulta en la BD *Architectural models and users* para obtener la lista de perfiles de usuarios disponibles en el sistema.

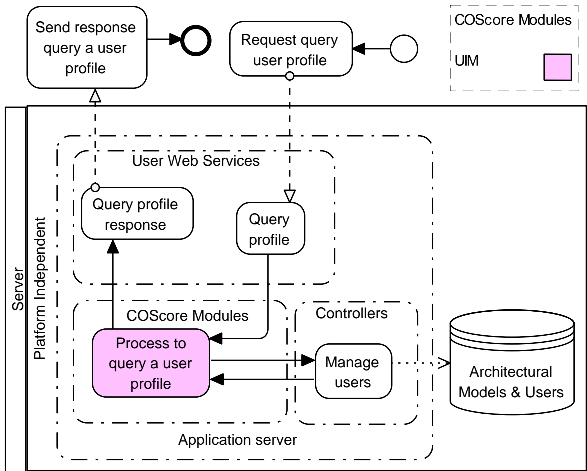


Figura 3.8: Operación *Query profile* de *User Service*

© 2016 Vallecillos, J.

Implementación: Como se observa en el código de la implementación de la operación, que se muestra seguidamente, primero se comprueba que la clave `privatekey` es correcta. Tras ello, se inicializa un contexto para crear una instancia del módulo UIM el cual se encarga de la gestión básica de la operación, como se ha dicho. Una vez obtenido el módulo, hace uso de la operación `queryProfile` la cual no necesita pasarle ningún parámetro para que devuelva el conjunto de perfiles. Dentro de este método, se accede a la base de datos de usuarios por medio de una consulta a todos los perfiles almacenados por medio del controlador *Manage Users*.

```
 1 public QueryProfileResult queryProfile(String privatekey)       {
 2
 3 QueryProfileResult queryProfileResult = new QueryProfileResult();
 4 if(privatekey!=null && this.privatekey.compareTo(privatekey) == 0) {
 5    Context initialContext;
 6    try {
 7       initialContext = new InitialContext();
 8       UIM profileInformation = (UIM)initialContext.lookup("java:app/cos/UIM");
 9       queryProfileResult = profileInformation.queryProfile();
10    } catch (NamingException e) {
11       LOGGER.error(e);
12       queryProfileResult.setValidation(false);
13       queryProfileResult.setProfiles(null);
14       queryProfileResult.setMessage("> Internal Server Error");}
15 } else {
16    queryProfileResult.setValidation(false);
17    queryProfileResult.setProfiles(null);
18    queryProfileResult.setMessage("> Private key Error");
19    LOGGER.error("Private key error");  }
20 return queryProfileResult;
21 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                 xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:queryProfile>
      <privatekey> ejemplo clave </ privatekey>
    </ws: queryProfile >
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2: queryProfileResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <validation> ejemplo true o false </validation>
          <profiles> ejemplo perfil </profiles>
              ...
```

```
        <profiles> ejemplo perfil </profiles>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2: queryProfileResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:updateUserResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <validation> false </validation>
        <message> > > Private key Error </message>
      </result>
    </ns2:updateUserResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.2. Servicio *Manage Architecture Service*

El servicio *Manage Architecture* permite realizar la gestión de las arquitecturas de componentes que describen las aplicaciones *mashup*. Para la gestión de dichas arquitecturas, el sistema utiliza dos modelos de arquitecturas distintos, los modelos de arquitectura abstracta y los modelos de arquitectura concreta. Estos modelos se utilizan para definir las aplicaciones *mashup* en distintos niveles de abstracción, gracias al uso de técnicas MDA (*Model Driven Architecture*). Los modelos de arquitectura abstracta permiten definir una aplicación de usuario independientemente de la plataforma donde será desplegada. Por tanto, dichas arquitecturas se describen en términos de los tipos de componentes existentes en la aplicación así como de las relaciones que existan entre dichos componentes. Estos modelos se corresponden con el nivel de PIM (*Platform Independent Model*) de MDA. Por otro lado, los modelos de arquitectura concreta permiten definir una aplicación en base a los componentes concretos (que tiene una correspondencia con un componente real, implementado y disponible para ser desplegado por la infraestructura) que se utilizan en una determinada plataforma. Por tanto, estos modelos de arquitectura se corresponden con el nivel PSM (*Platform Specification Model*) de MDA. Este servicio está compuesto por el siguiente conjunto de operaciones:

— *Export AAM from String*: se utiliza para insertar una nueva definición de arquitectura abstracta en el repositorio de especificaciones del sistema.

— *Export CAM from String*: se emplea para insertar una nueva definición de arquitectura concreta en el repositorio de especificaciones del sistema.

— *Withdraw CAM*: se utiliza para eliminar especificaciones de arquitecturas concretas.

© 2016 Vallecillos, J.

3.3.2.1. Operación *Export AAM from String*

La operación *Export AAM from String* forma parte del servicio *Manage Architecture*, el cual da soporte a la gestión de especificaciones de arquitecturas de las aplicaciones *mashup*. Esta operación se utiliza para insertar nuevas definiciones de arquitecturas abstractas en el repositorio de especificaciones (recordemos que este repositorio, o base de datos, como se ha nombrado en alguna ocasión anteriormente, contiene modelos de arquitecturas). En esta operación no intervienen módulos, sino que se comunica directamente con el controlador *Manage Architectures*. A continuación, en el siguiente listado se muestra la definición de la interfaz de la operación, así como los parámetros de entrada y salida. Más adelante se realiza una descripción.

Interfaz:

```
1 @WebMethod(operationName="exportAAMFromString", action="exportAAMFromString")
2 public String exportAAMFromString(
3   @WebParam(name="aamFileString", targetNamespace="http://ws.cos.acg.ual.es/")
4     String aamFileString);
```

Parámetros de Entrada/Salida:

Parametros de entrada	
`aamFileString`	Un valor *string* con el contenido del archivo que describe el modelo de arquitectura abstracta. Parámetro obligatorio no nulo.
Valores de salida	
`result`	Un valor *string* con un mensaje informando del éxito o del error de la operación de inserción.

Descripción: Como se observa en la interfaz, la operación acepta como entrada una cadena de texto que contiene la descripción del modelo de arquitectura abstracta en formato XMI. Como resultado, esta operación devuelve un mensaje en forma de cadena de texto para informar del éxito o fracaso de la actualización de la base de datos. Sólo hay dos mensajes de salida posibles: un mensaje con el resultado de la ejecución de la operación de inserción, o un mensaje de error informando de que el modelo no ha podido ser insertado debido a un problema interno del servidor.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: En la Figura 3.9 se muestra un diagrama del flujo de información de la operación *Export AAM from String* del servicio. No se utilizan llamadas a ningún módulo para la ejecución de esta operación sino que, directamente, se realiza una comunicación con el controlador *Manage Architectures*. Dicho controlador realiza una actualización de la base de datos llevando a cabo una inserción del modelo de arquitectura abstracta introducido como parámetro de la operación. El resultado obtenido por el controlador es enviado como respuesta de la operación.

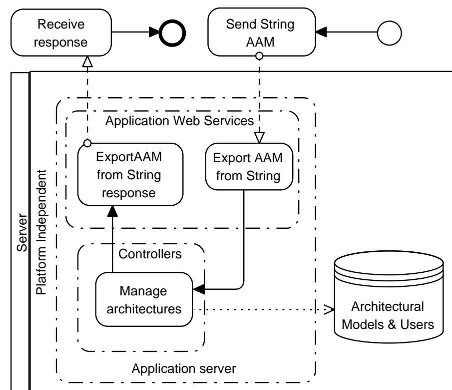


Figura 3.9: Operación *Export AAM from String* de *Manage Architecture Service*

Implementación: Como se puede observar en el código de la operación que se muestra a continuación, la implementación de esta operación consiste, básicamente, en obtener el controlador *Manage Architectures* y ejecutar el método `exportAAMFromString`, tal y como se muestra en las líneas 7–9. En este método se accede a la base de datos *Architectural Models and Users* para almacenar el modelo de arquitectura abstracta (pasado como parámetro en forma de cadena de texto).

```
1 public String exportAAMFromString(String aamFileString){
2 String result = "No results obtained";
3
4 ManageArchitectures mngArch = null;
5 try {
6    Context initialContext = new InitialContext();
7    mngArch = (ManageArchitectures)initialContext.
8                                   lookup("java:app/cos/ManageArchitectures");
9    result = mngArch.exportAAMFromString(aamFileString);
10 } catch (NamingException e) {
11    LOGGER.error(e); return "> Internal Server Error"; }
12 return result;
13 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:exportAAMFromString>
      <ws:aamFileString><![CDATA[<?xml version="1.0" encoding="ASCII"?>
        <architectural_metamodel:AbstractArchitecturalModel xmi:version="2.0"
          xmlns:xmi="http://www.omg.org/XMI"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:architectural_metamodel="http://architectural_metamodel/1.9"
          xsi:schemaLocation="http://architectural_metamodel/1.9
```

© 2016 Vallecillos, J.

```
          architectural_metamodel1.9.ecore" aamID="aam23"> ...
        </architectural_metamodel:AbstractArchitecturalModel>
      </ws:aamFileString>
    </ws:exportAAMFromString>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportAAMFromStringResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return>ID exist --> AAM Specification is not inserted</return>
    </ns2:exportAAMFromStringResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportAAMFromStringResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return> > Internal Server Error</return>
    </ns2:exportAAMFromStringResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.2.2. Operación *Export CAM from String*

La operación *Export CAM from String* también forma parte del servicio *Manage Architecture*, y es necesaria para la gestión del nivel concreto de las especificaciones de arquitecturas. Esta operación se utiliza para insertar nuevas definiciones de arquitecturas concretas. Al igual que para el resto de operaciones de este servicio, en su ejecución no se utiliza ningún módulo de la infraestructura, sino que el servicio se comunica directamente con el controlador *Manage Architectures*. A continuación, se muestra la definición de su interfaz, incluyendo los parámetros de entrada y los datos obtenidos como salida.

Interfaz:

```
1 @WebMethod(operationName="exportCAMFromString", action="exportCAMFromString")
2 public String exportCAMFromString(
3   @WebParam(name="camFileString", targetNamespace="http://ws.cos.acg.ual.es/")
4     String camFileString);
```

Parámetros de Entrada/Salida:

Parametros de entrada	
`camFileString`	Un valor *string* con el contenido del archivo que describe el modelo de arquitectura concreta. Parámetro obligatorio no nulo.
Valores de salida	
`result`	Un valor *string* con un mensaje informando del éxito o del error de la operación de inserción.

Descripción: Como establece la interfaz, la operación tiene un parámetro de entrada que contiene la descripción del modelo de arquitectura concreta en formato XMI (como una cadena de texto). Como resultado, la operación devuelve una cadena de texto para informar del éxito o fracaso de la actualización de la base de datos. Al igual que para la operación anterior, los mensajes de salida posibles son un mensaje con el resultado de la ejecución de la operación de inserción, o un mensaje de error informando que el modelo no ha podido ser insertado debido a un problema interno del servidor.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: La Figura 3.10 muestra un diagrama que representa el comportamiento de la operación *Export CAM from String*. No se utilizan llamadas a ningún módulo para la ejecución de esta operación, sino que se realiza una invocación de la operación correspondiente del controlador *Manage Architectures*. Dicho controlador realiza una actualización de la base de datos, llevando a cabo una inserción del modelo de arquitectura concreta, introducido como parámetro de la operación. El resultado obtenido por el controlador es enviado como respuesta de la operación.

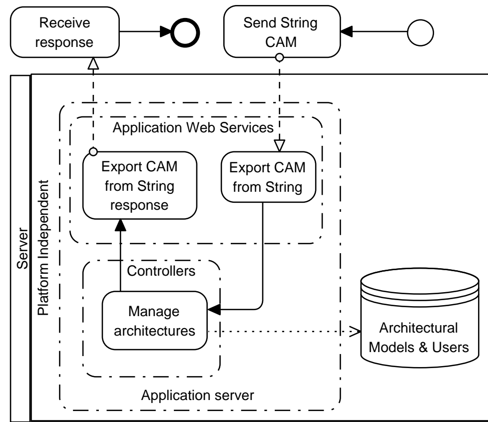


Figura 3.10: Operación *Export CAM from String* de *Manage Architecture Service*

Implementación: Como se observa en el código de la operación (abajo), se obtiene el controlador *Manage Architectures* para poder ejecutar el método `exportCAMFromString` (ver líneas 7–9). Este método actualiza la base de datos *Architectural Models and Users* insertando el modelo de arquitectura concreto que se introduce como parámetro.

```
1 public String exportCAMFromString(String camFileString){
2 String result = "No results obtained";
3
4 ManageArchitectures mngArch = null;
```

© 2016 Vallecillos, J.

```
5 try {
6    Context initialContext = new InitialContext();
7    mngArch = (ManageArchitectures)initialContext.
8                                  lookup("java:app/cos/ManageArchitectures");
9    result = mngArch.exportCAMFromString(camFileString);
10 } catch (NamingException e) {
11    LOGGER.error(e);
12    return "> Internal Server Error";
13 }
14
15 return result;
16 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:exportCAMFromString>
      <ws:camFileString><![CDATA[<?xml version="1.0" encoding="ASCII"?>
        <architectural_metamodel:ConcreteArchitecturalModel xmi:version="2.0"
          xmlns:xmi="http://www.omg.org/XMI"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:architectural_metamodel="http://architectural_metamodel/1.9"
          xsi:schemaLocation="http://architectural_metamodel/1.9
            architectural_metamodel1.9.ecore" aamID="aam23" camID="cam51">
        ...
        </architectural_metamodel:ConcreteArchitecturalModel>]]>
      </ws:camFileString>
    <ws:exportCAMFromString>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportCAMFromStringResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return>cam51 ID does not exist --> Insert CAM Specification</return>
    </ns2:exportCAMFromStringResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportCAMFromStringResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return> > Internal Server Error</return>
    </ns2:exportCAMFromStringResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.2.3. Operación *Withdraw CAM*

La operación *Withdraw CAM* es la tercera operación que forma parte del servicio *Manage Architecture*. Esta operación se emplea para eliminar especificaciones de arquitecturas concretas que existen en el repositorio de la infraestructura. Para ello, esta operación se comunica con el controlador *Manage Architectures*. La interfaz de la operación, y los parámetros de entrada y salida se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="withdrawCAM", action="withdrawCAM")
2 public String withdrawCAM(
3   @WebParam(name="camID", targetNamespace="http://ws.cos.acg.ual.es/")
4     String camID);
```

Parámetros de Entrada/Salida:

Parametros de entrada	
`camID`	Un valor *string* con el identificador del modelo de arquitectura concreta que debe eliminarse. Parámetro obligatorio no nulo.
Valores de salida	
`result`	Un valor *string* con un mensaje informando del éxito o del error de la operación de eliminación.

Descripción: La operación tiene como entrada una cadena de texto con el identificador del modelo de arquitectura concreta que debe ser eliminado del repositorio de especificaciones que forma parte de la infraestructura. Como resultado, esta operación devuelve un mensaje en forma de texto para informar del éxito o fracaso de la actualización de la BD. Hay dos mensajes de salida que pueden ser obtenidos, un mensaje con el resultado de la ejecución de la operación de eliminación, o un mensaje de error informando de que el modelo no ha podido ser eliminado debido a un problema interno del servidor.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: En la Figura 3.11 se muestra un diagrama que representa el flujo de información de la operación *Withdraw CAM*. No se utilizan llamadas a ningún módulo para la ejecución de esta operación, sino que se realiza una comunicación directamente con el controlador *Manage Architectures*. Dicho controlador realiza una actualización de la base de datos llevando a cabo una eliminación del modelo de arquitectura concreta introducido como parámetro de la operación. El resultado obtenido por el controlador es enviado como respuesta de la operación.

Implementación: Se obtiene el controlador *Manage Architectures* para ejecutar el método `withdrawCAM` (líneas 7–9). Este método actualiza la BD *Architectural Models and Users* eliminando el modelo de arquitectura concreta que se introduce como parámetro.

© 2016 Vallecillos, J.

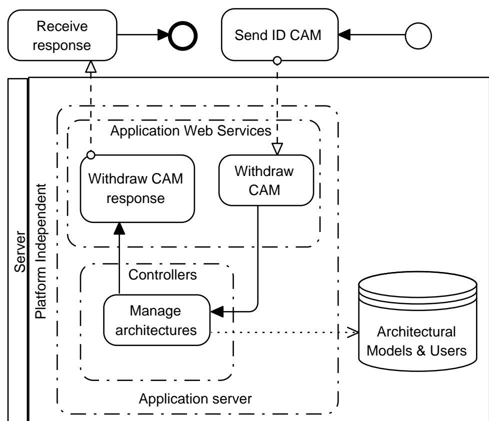


Figura 3.11: Operación *Withdraw CAM* de *Manage Architecture Service*

```
1 public String withdrawCAM(String camID){
2 String result = "No results obtained";
3
4 ManageArchitectures mngArch = null;
5 try {
6    Context initialContext = new InitialContext();
7    mngArch = (ManageArchitectures)initialContext.
8                                  lookup("java:app/cos/ManageArchitectures");
9    result = mngArch.withdrawCAM(camID);
10 } catch (NamingException e) {
11    LOGGER.error(e); return "> Internal Server Error"; }
12 return result;
13 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:withdrawCAM> <ws:camID>cam55</ws:camID> </ws:withdrawCAM>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:withdrawCAMResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return>cam55 ID does not exist --> Cannot delete CAM Specification</return>
    </ns2:withdrawCAMResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:withdrawCAMResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return> > Internal Server Error</return>
    </ns2:withdrawCAMResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.3. Servicio *Manage Component Service*

El servicio privado*Manage Component Service* permite realizar la gestión de las especificaciones de componentes que intervienen en la descripción de las aplicaciones *mashup* que son manejadas por la infraestructura. Para ello, el servicio está constituido por las operaciones que se describen a continuación:

— *Export CC from String*: se utiliza para insertar una nueva definición de componente concreto en el repositorio de especificaciones del sistema (a partir de una cadena de texto).

— *Export CC from params*: se emplea para insertar una nueva definición de componente concreto en el repositorio de especificaciones del sistema (a partir de una serie de parámetros).

— *Withdraw CC*: se utiliza para eliminar especificaciones de componentes concretas.

3.3.3.1. Operación *Export CC from String*

La operación *Export CC from String* forma parte del servicio *Manage Component*, que da soporte a la gestión de las especificaciones de componentes. Esta operación se utiliza para añadir nuevas especificaciones de componentes, de manera que dichos componentes sean tenidos en cuenta por la infraestructura en las operaciones de despliegue y reconfiguración de las aplicaciones *mashup*. La interfaz y los parámetros de entrada y salida se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="exportCCFromString", action="exportCCFromString")
2 public String exportCCFromString(
3   @WebParam(name="ccFileString", targetNamespace="http://ws.cos.acg.ual.es/")
4     String ccFileString);
```

Parámetros de Entrada/Salida:

Parametros de entrada	
`ccFileString`	Un valor *string* con el contenido del archivo que describe el modelo de componente concreto. Parámetro obligatorio no nulo.
Valores de salida	
`result`	Un valor *string* con un mensaje informando del éxito o del error de la operación de inserción.

© 2016 Vallecillos, J.

Descripción: Como se puede ver en la definición de la interfaz de la operación (que se muestra en el listado más abajo), un parámetro de entrada contiene la descripción del modelo de componente concreto definido como una cadena de texto. Esta cadena de texto está constituida por el contenido del archivo XMI que describe dicho modelo. La operación devuelve una cadena de texto para informar del éxito o fracaso de la actualización de la base de datos. Hay dos tipos de mensajes de respuesta posibles: un mensaje con el resultado de la ejecución de la operación de inserción de componente concreto, o un mensaje de error informando de que el modelo no ha podido ser insertado debido a un problema interno del servidor.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: En la Figura 3.12 se muestra un diagrama con el flujo de la información de tareas que sucede internamente en la operación *Export CC from String*. En esta operación se realiza una llamada al método correspondiente del controlador *Manage Component Specifications*. Este controlador actualiza la base de datos *Concrete Component Specifications* mediante la inserción del modelo de componente concreto introducido como parámetro. El resultado obtenido por el controlador es enviado como respuesta de la operación.

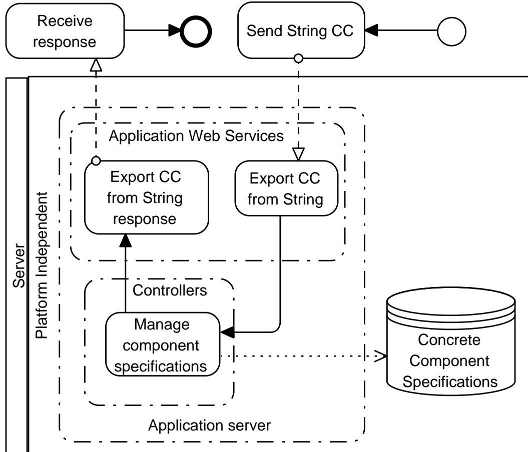


Figura 3.12: Operación *Export CC from Srting* de *Manage Component Service*

Implementación: Como se puede observar en el código (que se muestra en el listado de abajo), en primer lugar se obtiene el controlador *Manage Architectures* para poder ejecutar el método `exportCAMFromString` (ver líneas 7–9). Este método actualiza la base de datos *Architectural Models and Users* insertando el modelo de arquitectura concreto que se introduce como parámetro.

```
 1 public String exportCCFromString(String ccFileString){
 2 String result = "No results obtained";
 3
 4 ManageComponentSpecifications mngComp = null;
 5 try { Context initialContext = new InitialContext();
 6       mngComp = (ManageComponentSpecifications)initialContext.
 7       lookup("java:app/cos/ManageComponentSpecifications");
 8       result = mngComp.exportCCFromString(ccFileString);
 9 } catch (NamingException e) {
10    LOGGER.error(e); return "> Internal Server Error"; }
11 return result;
12 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:exportCCFromString>
      <ws:ccFileString><![CDATA[<?xml version="1.0" encoding="ASCII"?>
        <component_metamodel:ConcreteComponentSpecification xmi:version="2.0"
         xmlns:xmi="http://www.omg.org/XMI"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:component_metamodel="http://component_metamodel/1.9"
         xsi:schemaLocation="http://component_metamodel/1.9component_metamodel1.9.ecore"
         componentID="http://acg.ual.es/wookie/widgets/map01" abstractComponentID="map"
         componentDescription="Map" componentName="map01">
        ...
        </component_metamodel:ConcreteComponentSpecification>]]>
      </ws:ccFileString>
    </ws:exportCCFromString>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportCCFromString xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return>http://acg.ual.es/wookie/widgets/map01 ID does not exist
        --> Insert CC Specification</return>
    </ns2:exportCCFromString>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportCCFromString xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return> > Internal Server Error</return>
    </ns2:exportCCFromString>
  </soap:Body>
</soap:Envelope>
```

© 2016 Vallecillos, J.

3.3.3.2. Operación *Export CC from params*

Esta operación forma parte del servicio *Manage Component*, el cual da soporte a la gestión de especificaciones de componentes. Se utiliza para insertar nuevas definiciones de componentes concretos en el repositorio de especificaciones, a través de la introducción de valores específicos para los atributos que constituyen una especificación de componente concreto. Esta operación no hace uso de módulos, sino que se comunica directamente con el controlador *Manage Component Specifications*. A continuación, se muestra la definición de la interfaz de la operación, así como los parámetros de entrada y salida.

Interfaz:

```
 1 @WebMethod(operationName="exportCCFromParams", action="exportCCFromParams")
 2 public String exportCCFromParams(
 3   @WebParam(name="componentName", targetNamespace="http://ws.cos.acg.ual.es/")
 4     String componentName,
 5   @WebParam(name="componentAlias",targetNamespace="http://ws.cos.acg.ual.es/")
 6     String componentAlias,
 7   @WebParam(name="componentDescription",targetNamespace="http://ws.cos.acg.ual.es/")
 8     String componentDescription,
 9   @WebParam(name="entityId",targetNamespace="http://ws.cos.acg.ual.es/")
10     String entityId,
11   @WebParam(name="entityName",targetNamespace="http://ws.cos.acg.ual.es/")
12    String entityName,
13   ...
14 );
```

Parámetros de Entrada/Salida:

Parametros de entrada	
`params`	Los valores en forma de cadena para cada uno de los atributos que forman parte del metamodelo de componente, definido en el Capítulo 2. La obligatoriedad de cada atributo queda establecida a través de la cardinalidad de dicho metamodelo.
Valores de salida	
`result`	Un *string* informando del éxito o error de la operación de inserción.

Descripción: Como se observa en la interfaz, la operación acepta como entrada una secuencia de valores que se corresponden con cada uno de los atributos de la especificación utilizada para describir los componentes concretos. Por tanto, estos atributos pertenecen a las cuatros partes de la especificación: funcional, extra-funcional, de empaquetamiento y de mercado. Como resultado, esta operación devuelve un mensaje en forma de cadena de texto para informar del éxito o fracaso de la actualización de la base de datos. Sólo hay dos mensajes de salida posibles: un mensaje con el resultado de la ejecución de la operación de inserción, o un mensaje de error informando de que el modelo no ha podido ser insertado debido a un problema interno del servidor.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: La Figura 3.13 muestra un diagrama del flujo de tareas de la operación *Export CC from params*. No se utilizan llamadas a ningún módulo para la ejecución de esta operación, sino que se realiza una comunicación directamente con el controlador *Manage Component Specifications*. Dicho controlador realiza una actualización de la base de datos llevando a cabo una inserción de un modelo de componente concreto que se construye a partir de los valores introducidos como parámetros. El resultado obtenido por el controlador es enviado como respuesta de la operación.

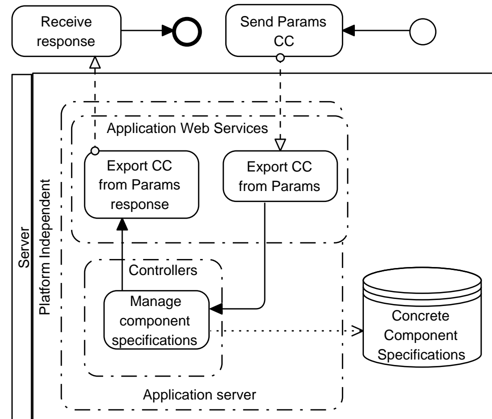


Figura 3.13: Operación *Export CC from params* de *Manage Component Service*

Implementación: Como se puede observar en el código de la operación que se muestra a continuación, la implementación de esta operación consiste, básicamente, en obtener el controlador *Manage Component Specifications* y ejecutar el método `exportCCFromParams`, tal y como se muestra en las líneas 8–10. Este método accede a la base de datos *Concrete Component Specifications* para almacenar el modelo de componente concreto que se construye a partir de los parámetros introducidos en forma de cadena de texto.

```
1  public String exportCCFromParams(String componentName, ...
2       , componentURI){
3  String result = "No results obtained";
4
5  ManageComponentSpecifications mngComp = null;
6  try {
7    Context initialContext = new InitialContext();
8    mngComp = (ManageComponentSpecifications)initialContext.
9      lookup("java:app/cos/ManageComponentSpecifications");
10    result = mngComp.exportCCFromParams(componentName, ... , componentURI);
11 } catch (NamingException e) {
12    LOGGER.error(e);
13    return "> Internal Server Error";
14 }
15 return result;
16 }
```

© 2016 Vallecillos, J.

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:exportCCFromParams>
      <ws:componentName>http://acg.ual.es/wookie/widgets/map65</ws:componentName>
      <ws:componentAlias> map65</ws:componentAlias>
      <ws:componentDescription> This component shows...</ws:componentDescription>
      <ws:entityId> TIC211</ws:entityId>
      <ws:entityName> Applied Computing Group </ws:entityName>
      ...
    </ws:exportCCFromParams>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportCCFromParamsResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return>http://acg.ual.es/wookie/widgets/map65 ID does not exist
      --> Insert CC Specification</return>
    </ns2:exportCCFromParamsResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:exportCCFromParamsResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return> > Internal Server Error</return>
    </ns2:exportCCFromParamsResponse>
  </soap:Body>
</soap:Envelope>
```

3.3.3.3. Operación *Withdraw CC*

La operación *Withdraw CC* forma parte del servicio *Manage Component*. Esta operación se utiliza para eliminar especificaciones de componentes concretos que existen en el repositorio de la infraestructura. Para ello, esta operación se comunica con el controlador *Manage Component Specifications*. La interfaz de la operación y los parámetros de entrada y salida, se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="withdrawCC", action="withdrawCC")
2 public String withdrawCC(
3   @WebParam(name="ccID", targetNamespace="http://ws.cos.acg.ual.es/")
4     String ccID
5 );
```

Parámetros de Entrada/Salida:

Parámetros de entrada	
`ccID`	Un valor *string* con el identificador del modelo de componente concreto que debe eliminarse. Parámetro obligatorio no nulo.
Valores de salida	
`result`	Un *string* con un mensaje informando del éxito o del error de la operación de eliminación.

Descripción: La operación tiene como entrada una cadena de texto con el identificador del modelo de componente concreto que debe ser eliminado del repositorio de especificaciones. Como resultado, esta operación devuelve un mensaje en forma de cadena de texto para informar del éxito o fracaso de la actualización de la base de datos. Hay dos mensajes de salida que pueden ser obtenidos, un mensaje con el resultado de la ejecución de la operación de eliminación, o un mensaje de error informando de que el modelo no ha podido ser eliminado debido a un problema interno del servidor.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: En la Figura 3.14 se muestra un diagrama que representa flujo de información de la operación *Withdraw CC*. No se utilizan llamadas a ningún módulo para la ejecución de esta operación, sino que se realiza una comunicación directamente con el controlador *Manage Component Specifications*. Dicho controlador realiza una actualización de la base de datos llevando a cabo una eliminación del componente concreto cuyo identificador es igual al introducido como parámetro. El resultado obtenido por el controlador es enviado como respuesta de la operación.

Implementación: Se obtiene el controlador *Manage Component Specifications* para ejecutar el método `withdrawCC` (ver líneas 7–9). Este método actualiza la base de datos *Concrete Component Specifications* eliminando el modelo de componente concreto que se introduce como parámetro.

```
1 public String withdrawCC(String ccID){
2 String result = "No results obtained";
3
4 ManageComponentSpecifications mngComp = null;
5 try {
6    Context initialContext = new InitialContext();
7    mngComp = (ManageComponentSpecifications)initialContext.
8      lookup("java:app/cos/ManageComponentSpecifications");
9    result = mngComp.withdrawCC(ccID);
10 } catch (NamingException e) {
11    LOGGER.error(e); return "> Internal Server Error"; }
12 return result;
13 }
```

© 2016 Vallecillos, J.

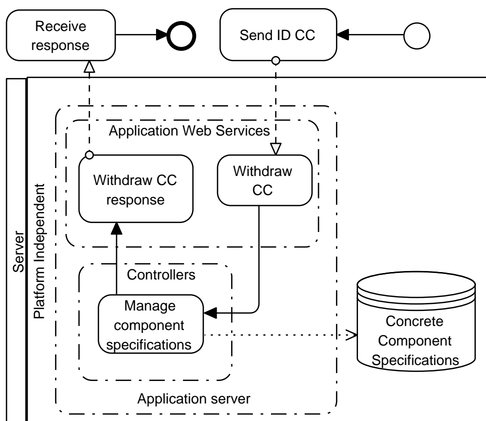


Figura 3.14: Operación *Withdraw CC* de *Manage Component Service*

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:withdrawCC>
      <ws:ccID>map47</ws:ccID>
    </ws:withdrawCC>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:withdrawCCResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return>map47 ID does not exist --> Cannot delete CC Specification</return>
    </ns2:withdrawCCResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:withdrawCCResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <return> > Internal Server Error</return>
    </ns2:withdrawCCResponse>
  </soap:Body>
</soap:Envelope>
```

3.4. Servicios públicos

En esta sección se describe el conjunto de servicios públicos S_+ definidos en el modelo de infraestructura COScore propuesto para el despliegue de aplicaciones *mashup*. Como se ha comentando anteriormente, el conjunto de servicios públicos de COScore consta de los servicios *Session Service*, *Communication Service*, *Component Service* e *Interaction Service*. Para describir correctamente el funcionamiento de dichos servicios, en esta sección se utilizará la misma estructura que la usada en la sección anterior para los servicios privados. Igualmente, para la implementación de estos servicios se ha seguido un desarrollo guiado por pruebas TDD (*Test-Driven Development*) donde para testar cada una de las operaciones de los servicios públicos de COScore se ha elaborado un juego de pruebas y una herramienta de pruebas en linea[6]. En el Anexo A de este documento de tesis se incluye además el juego de pruebas realizado para los servicios públicos. Como se ha mencionado, para el desarrollo de los servicios privados también se ha seguido TDD, pero sólo se ha dejado en el Anexo A el juego de pruebas de los servicios públicos dado que son aquellos servicios que pueden ser accesibles por terceros. Veamos a continuación cada una de las operaciones de los servicios públicos de COScore.

3.4.1. Servicio *Session Service*

El servicio *Session Service* se encarga de dar soporte a la sesión de los usuarios del sistema. Este servicio queda constituido por el siguiente conjunto de operaciones, que se describirán con detalle a continuación:

— *Login*: operación de acceso de un usuario en el sistema.

— *Logout*: se emplea con el objetivo de eliminar los módulos de sesión inicializados por el usuario.

— *Init user architecture*: se emplea con el objetivo de obtener la lista de componentes de la arquitectura asociada a un usuario.

— *Default Init*: por defecto se emplea para dar un contexto inicial a los usuarios no registrados, y por tanto asignarle un modelo de arquitectura por defecto.

3.4.1.1. Operación *Login*

La operación *Login* se engloba como parte de la funcionalidad que presta el servicio *Session Service* que da soporte a la gestión de sesiones de los usuarios. Esta operación se emplea con el objetivo de validar el usuario en el sistema e inicializar los módulos de sesión de ese usuario COSSessionMM (Módulo de Gestión de Sesiones del COSCore) y UIM (Módulo de Información de Usuarios), al igual que el servicio privado *Query User*. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación en el siguiente listado.

[6]COScore API – `http://acg.ual.es/projects/enia/ui/webservices/`

© 2016 Vallecillos, J.

Interfaz:

```
1 @WebMethod(operationName="login", action="login")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3       @XmlElement(required=true)
4   public LoginSessionResult login(
5     @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6     @XmlElement(required=true) LoginSessionParams params);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un *structure* `LoginSessionParams` con los parametros de entrada y el orden a seguir:		
	string	`userName`	Nombre de usuario a consultar en el sistema. Parámetro obligatorio no nulo.
	string	`userPassword`	Password para ese usuario, se encriptará y se comprueba con el de la BD. Parámetro obligatorio no nulo.
Parámetros de salida			
`result`	Un *structure* `LoginSessionResult` con los valores de salida:		
	boolean	`validation`	Variable que muestra si tuvo éxito la acción, *true* si tiene éxito en la consulta en la BD y la validación del password, *false* si no.
	string	`userID`	Texto con el identificador de usuario en la BD, devuelve la cadena -1 si no encuentra o autentifica al usuario.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: Como se observa en la definición de la interfaz, la operación acepta en su entrada la estructura `LoginSessionParams` con los valores para el nuevo nombre de usuario y la clave, necesarios para iniciar una sesión en el COScore; estos valores son obligatorios y además no nulos. La operación solicita al sistema la validación del usuario, donde se comprueba si existe en la base de datos y si su clave es correcta. Si es válido, se inician los módulos asociados a la sesión de ese usuario. Devuelve como respuesta una estructura con una variable que indica si el usuario y su clave han sido validados, otra con una cadena de texto con su identificador en la BD y un mensaje correspondiente a la operación realizada. La estructura `LoginSessionResult` es devuelta por la operación. Dicha estructura devuelve tres valores, el resultado de la operación, que será *true* si se ha podido realizar la operación con éxito, el identificador del usuario, que será "-1" si la operación no se pudo realizar con éxito, y un mensaje de texto indicando qué ocurrió durante el proceso de inicio de sesión. En la siguiente tabla se muestra un listado con los tipos de mensajes devueltos por la operación.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al aplicar algoritmos de encriptación o al buscar alguna clase necesaria para la ejecución.

`Not found or Empty username Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty userpassword Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Validation Error user password incorrect`	Se produce cuando el usuario no existe en la BD o el password es incorrecto.
`Error Modules previously initialized`	Se produce cuando se realiza un *login* habiendo realizado un *login* anteriormente y por tanto encontrándose los módulos ya inicializados. Este error no devuelve error de validación ya que la validación sigue siendo correcta, solamente informa que ya se inicializó el sistema.
`PSQLException`	Se produce por problemas en la conexión a la base de datos.

Comportamiento: En la Figura 3.15 se muestra un diagrama de flujo de procesos que ocurren en la llamada de la operación. Como se puede apreciar en la figura, la operación presenta dos puntos de acceso: *Web application* y *Java application*. Como se adelantó en el anterior capítulo, el modelo de infraestructura COScore ha sido implementado para dos tipos de arquitecturas *mashup* posibles: (a) componentes *widgets*, para permitir el despliegue de interfaces gráficas Web de usuario basados en componentes *COTSgets*, y (b) componentes Java, que permiten el despliegue de otros escenarios distintos como el de arquitecturas domóticas, presentado en el capítulo anterior. Para el caso de aplicaciones *mashup* de interfaz gráfica Web de usuario, la operación accede al servidor de aplicaciones web *Applicacion web server* para obtener previamente la aplicación. La aplicación obtenida no contiene ningún componente, sino que consiste en el código necesario para conectarse con el servidor *JavaScript* y para poder realizar la carga de los componentes iniciales. Posteriormente, se accede al servidor *JavaScript* para iniciar la secuencia de identificación. En cambio, para aplicaciones basadas en componentes Java la operación directamente se comunica con el servidor de *JavaScript*. En ambos casos, a continuación se estable conexión con la operación *login* de la capa independiente. Después se envía la petición al módulo UIM donde se ejecuta la tarea *Query user in data base*. Este módulo comprueba si el usuario fue dado de alta en la base de datos. Para ello, el módulo hace uso del controlador *Manage users* para acceder a la base de datos *Architecture models and users* y realizar la consulta. En caso de estar dado de alta, se inicializan los módulos para el usuario por medio de la tarea *Initialize modules*. Una vez inicializados los módulos, se responde a la aplicación cliente con el identificador de usuario con el que fue dado de alta el cliente en la base de datos.

Implementación: En el siguiente listado se muestra el código de la implementación de la operación *Login*. Como se observa, en las líneas 6 y 8 se comprueba la validez de los valores de entrada para el usuario y la clave. En el caso de no haber ningún tipo de error se responde con un mensaje de "*Not found or Empty userpassword Error*" o "*Not found or Empty username Error*", y se continúa con el uso del módulo *COSSessionMM* (línea

© 2016 Vallecillos, J.

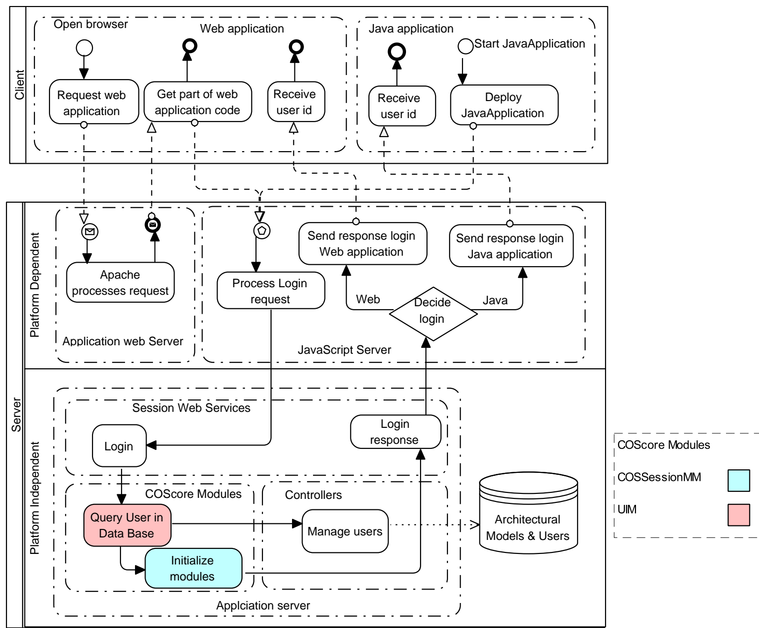


Figura 3.15: Operación *Login* de *Session Service*

14). Por medio de dicho módulo se inicializa el conjunto de módulos necesarios para el usuario en la línea 16. La inicialización del conjunto de módulos permitirá dar soporte a la funcionalidad de la aplicación *mashup*. Los módulos que se inicializan para ese usuario son *Display Management Module* (DMM), *Transaction Management Module* (TMM) e *Interaction Management Module* (IMM). Este conjunto de módulos están dedicados a dar soporte a aplicaciones *mashup* para un usuario en concreto, respondiendo a las solicitudes de la aplicación que ésta vaya realizando a lo largo del tiempo. Como se puede observar en el código para implementar el módulo *COSSessionMM*, se ha hecho uso de *Enterprise JavaBeans* (EJB). Por ello, para consumir el componente EJB se ha creado una variable de contexto en la línea 13 donde se despliega el componente.

```
 1 public LoginSessionResult login(LoginSessionParams params) {
 2
 3 LoginSessionResult result = new LoginSessionResult();
 4
 5 // First: check the params.username
 6 if (params.getUserName() != null && params.getUserName().compareTo("") != 0) {
 7    // Second: check the params.userpassword
 8   if (params.getUserPassword() != null && params.getUserPassword().
 9                                                        compareTo("") != 0) {
10      COSSessionMM cossmng = null;
11      Context initialContext;
12      try {
13         initialContext = new InitialContext();
14         cossmng = (COSSessionMM) initialContext.
```

```
15                                            lookup("java:app/cos/COSSessionMM");
16          result = cossmng.initializeModules(params.getUserName(),
17                                                      params.getUserPassword());
18       } catch (NamingException e) {
19          LOGGER.error(e);
20          result.setValidation(false);
21          result.setUserId("-1");
22          result.setMessage("> Internal Server Error");
23       }
24    } else {
25       LOGGER.error("Not found or Empty userpassword Error");
26       result.setValidation(false);
27       result.setUserId("-1");
28       result.setMessage("> Not found or Empty userpassword Error");
29    }
30 } else {
31    LOGGER.error("Not found or Empty username Error");
32    result.setValidation(false);
33    result.setUserId("-1");
34    result.setMessage("> Not found or Empty username Error");
35 }
36 return result;
37 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:login>
      <params>
        <userName> ejemplo nombre </userName>
        <userPassword> ejemplo password </userPassword>
      </params>
    </ws:login>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:loginResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <validation> ejemplo true o false </validation>
        <userID> ejemplo ID o -1 </userID>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:loginResponse>
  </soap:Body>
</soap:Envelope>
```

© 2016 Vallecillos, J.

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:loginResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <validation> true </validation>
        <userID> 113 </userID>
        <message> Error Modules previously initialized </message>
      </result>
    </ns2:loginResponse>
  </soap:Body>
</soap:Envelope>
```

3.4.1.2. Operación *Logout*

Es una funcionalidad del servicio *Session Service* que da soporte a la gestión de sesiones. Esta operación se emplea con el objetivo de eliminar las variables de sesión y los módulos de sesión inicializados por el usuario. Este servicio es controlado por el componente COSSessionMM (Modulo de Gestión de Sesiones del COSCore). La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="logout", action="logout")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3    @XmlElement(required=true)
4 public LogoutSessionResult logout(
5    @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6    @XmlElement(required=true)
7     LogoutSessionParams params);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un *structure* `LogoutSessionParams` con los parámetros:		
	string	`userId`	Id de usuario con una sesión iniciada en el sistema. Parámetro obligatorio no nulo.
Parámetros de salida			
`result`	Un valor *structure* `LogoutSessionResult` con los valores de salida:		
	boolean	`deleted`	*true* si tiene éxito el borrado de los módulos de sesión, *false* si no.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: El método acepta de entrada una estructura con el identificador del usuario que tiene la sesión iniciada, este valor es obligatorio y además no nulo. La operación elimina los módulos asociados a la sesión de ese usuario y devuelve como respuesta una estructura `LogoutSessionResult` con una variable que indica si el borrado ha tenido éxito o no y un mensaje correspondiente a la operación realizada. En la siguiente tabla se muestran los posibles mensajes de error devueltos por la operación `logout`.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.
`Not found or Empty userId Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Error Delete Modules`	Se produce cuando se intenta realizar un *logout* sobre una sesión no inicializada, al no estar la sesión iniciada no se encuentran los módulos asociados a ese usuario.

Comportamiento: La operación de `logout` se encarga de cerrar la sesión para una aplicación *mashup* de un usuario. Su comportamiento se observa en el diagrama de flujo que se muestra en la Figura 3.16. Es una operación simple; en primer lugar se recibe la petición por parte del usuario desde el lado cliente. Esta petición está programada para comunicarse con el servidor *JavaScript*, el cual se encarga de reenviar la solicitud llamando a la operación *logout* del servicio *Session Service*. Una vez recibida la solicitud, el servicio se comunica con el módulo *COSSessionMM* para eliminar los módulos del usuario localizados en la infraestructura. Al eliminar estos módulos (que existen a partir del inicio de sesión de un usuario o de la inicialización de un usuario anónimo) se eliminan también las estructuras de datos auxiliares que fueron creadas para un usuario. Dichas estructuras de datos tienen sentido mientras el usuario mantiene una sesión abierta en la infraestructura, puesto que suponen una mejora del rendimiento en la ejecución de las operaciones, pero deben ser eliminadas una vez que el usuario finaliza su conexión (para volver a ser creadas nuevamente en el siguiente establecimiento de sesión).

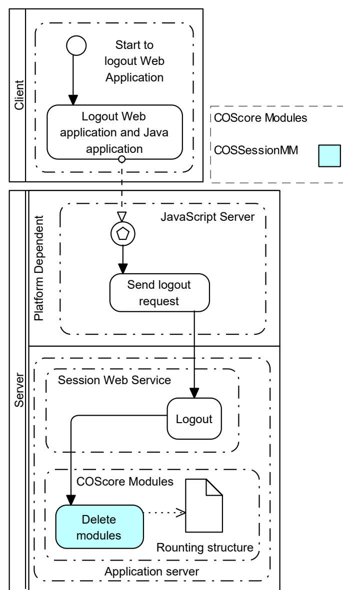


Figura 3.16: Operación *Logout* de *Session Service*

© 2016 Vallecillos, J.

Implementación: En el código mostrado a continuación se detalla cómo ha sido construida la operación *logout*. En la línea 4 se comprueba que no hay errores con respecto al identificador del usuario. Si el identificador es correcto, se obtiene el módulo *COSSessionMM* en la línea 9, para posteriormente en la línea 10 utilizar el método `destroyModules` para eliminar los módulos del COScore. Dentro de esta operación se eliminan todos los módulos inicializados para la aplicación *mashup* de un usuario que inició una sesión en la infraestructura. Eliminar los módulos quiere decir que habrá que destruir los componentes EJB relacionados con los módulos que fueron "activados" en el sistema para dar soporte a la arquitectura de la aplicación *mashup* asociada a ese usuario. El proceso de eliminación de módulos del usuario (línea 10) implica el borrado de los módulos *Display Management Module* (DMM), *Transaction Management Module* (TMM) e *Interaction Management Module* (IMM) asociados al usuario. Por otro lado, también puede ocurrir que exista un error con el identificador del usuario, para lo cual la operación avisa con un error indicando "*Not found or Empty userId Error*". Se puede presentar otro tipo de error dentro de esta operación que sucede cuando se quieren eliminar los módulos del COScore de forma indebida, mostrado en la línea 14 con el tipo de mensaje "*Internal Server Error*".

```
 1 public LogoutSessionResult logout(LogoutSessionParams params) {
 2
 3 LogoutSessionResult result = new LogoutSessionResult();
 4 if(params.getUserId()!=null && params.getUserId().compareTo("") != 0){
 5    COSSessionMM cossmng = null;
 6    Context initialContext;
 7    try {
 8       initialContext = new InitialContext();
 9       cossmng = (COSSessionMM)initialContext.lookup("java:app/cos/COSSessionMM");
10       result = cossmng.destroyModules(params.getUserId());
11    } catch (NamingException e) {
12       LOGGER.error(e);
13       result.setDeleted(false);
14       result.setMessage("> Internal Server Error"); }
15 } else {
16    LOGGER.error("Not found or Empty userId Error");
17    result.setDeleted(false);
18    result.setMessage("> Not found or Empty userId Error"); }
19 return result;
20 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                 xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:logout>
      <params> <userId> ejemplo Id </userName> </params>
    </ws:logout>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:logoutResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <deleted> ejemplo true o false </deleted >
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:logoutResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:logoutResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <deleted> false </deleted>
        <message> > Error Delete Modules </message>
      </result>
    </ns2:logoutResponse>
  </soap:Body>
</soap:Envelope>
```

3.4.1.3. Operación *Init User Architecture*

La operación *Init User Architecture* se engloba en el servicio *Session Service* que da soporte a la gestión de sesiones de los usuarios. Esta operación se emplea con el objetivo de obtener la lista de componentes que forman parte de la arquitectura de componentes de la aplicación *mashup* asociada al usuario. Este servicio es controlado por el componente COSSessionMM (Módulo de Gestión de Sesiones del COSCore) y hace uso de otros tres módulos: (a) el módulo UIM (Módulo de Información de Usuarios) para obtener, a partir del identificador de usuario, su identificador de modelo de arquitectura *concreta*, (b) el módulo DMM (Módulo de Visualización de Componentes) para obtener el propio modelo del usuario, y (c) el módulo IMM (Módulo de Gestión de la interacción) para registrar en la base de datos de interacción el inicio de sesión del usuario. Como veremos a continuación, se trata de una operación compleja, que afecta a varios módulos, procesos y tablas de las bases de datos. La operación hace uso de diversas estructuras de datos para el manejo de las arquitecturas de componentes de las aplicaciones *mashup* asociadas a los usuarios. Concretamente, y como se podrá observar en las siguientes tablas, las estructuras contienen la implementación de aplicaciones *mashups* para el caso de interfaces gráficas Web de usuario cuyas arquitecturas de componentes están relacionadas con componentes *COTSgets* implementados como servicios OGC para el prototipo *mashup* ENIA, anteriormente adelantado y que se verá con más detalle en el capítulo siguiente. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación en el siguiente listado. Más adelante se realiza una explicación para dicha interfaz y sus estructuras de datos.

© 2016 Vallecillos, J.

Interfaz:

```
1 @WebMethod(operationName="initUserArchitecture", action="initUserArchitecture")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3    @XmlElement(required=true)
4 public InitUserArchitectureSessionResult initUserArchitecture(
5    @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6    @XmlElement(required=true) InitUserArchitectureSessionParams params);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un valor *structure* `InitUserArchitectureSessionParams` con los parámetros de entrada y el orden a seguir. La estructura es la siguiente:		
	string	`userId`	Identificador de usuario con una sesión iniciada en el sistema. Parámetro obligatorio no nulo.
	structure	`interaction`	Un *structure* con los datos de interacción que puede proporcionar el usuario (ver estructura abajo).
Parámetros de salida			
`result`	Un valor *structure* `InitUserArchitectureSessionResult` con los valores de salida. La estructura es la siguiente:		
	boolean	`init`	*true* si tiene éxito la obtención de la lista de componentes para ese usuario, *false* si no.
	list structure	`componentData`	Lista de componentes del modelo concreto para ese usuario (ver estructura abajo).
	string	`message`	Mensaje de éxito o de error y su tipo.
Estructura de datos adicionales			
`Interaction`			
	string	`deviceType`	Tipo de dispositivo desde el que se accede (Browser, Phone, Tablet, TV). Parámetro obligatorio.
	string	`interactionType`	Tipo de interaction de entrada (MouseKeyboard, Voice, Gestural, Touch). Parámetro obligatorio.
	string	`latitude`	Latitud, información geográfica que pueda proporcionar el dispositivo. Parámetro obligatorio.
	string	`longitude`	Longitud, información geográfica que pueda proporcionar el dispositivo. Parámetro obligatorio.
`componentData`			
	string	`platform`	Plataforma a la que está sirviendo: web, java, ...
	string	`componentname`	Nombre del componente.
	string	`componentAlias`	Alias de la instancia componente.
	string	`instanceId`	Instancia del componente.
	string	`codeHTML`	Código *html* que incluye el *iframe*.
	string	`objectJava`	Objeto serializado.
	string	`jarJava`	Nombre del archivo *jar* que contiene el objeto java.
	string	`idHtml`	Identificador del elemento *html* en el DOM.
	int	`posx`	Posición del componente en la columna.
	int	`posy`	Posición del componente en la fila.
	int	`tamanox`	Tamaño que ocupa a lo ancho.
	int	`tamanoy`	Tamaño que ocupa a lo alto.
	boolean	`servicio_maximizable`	Si el componente puede verse a pantalla completa.
	boolean	`servicio_agrupable`	Si ese componente puede agrupar servicios.
	int	`numero_servicios`	Número de servicios en este componente.
	list structure	`servicios`	Lista de servicios en este componente.
`servicios`			
	string	`componentname`	Nombre del componente. Si es el servicio base, coincide con el `componentData.componentname`
	string	`componentAlias`	Alias de la instancia componente. Si es el servicio base, coincide con el `componentData.componentAlias`
	string	`instanceId`	Instancia del componente. Si es el servicio base, coincide con el `componentData.InstanceId`
	string	`mapaKML`	Mapa del servicio OGC cargado.
	string	`capa`	Capa del servicio OGC cargado.

Descripción: Veamos algunas consideraciones del entorno, antes realizar una descripción de la operación, y de su comportamiento. Como se ha indicado anteriormente, esta operación maneja estructuras de datos relacionadas con aplicaciones *mashup* de interfaces gráficas Web de usuario (en adelante, interfaz de usuario o IU, para simplificar). En el lado cliente, la IU es manejada como una colección de componentes (modelo o arquitectura de componentes para la infraestructura) llamados componentes *COTSgets* que estarán ubicados en un "*espacio de trabajo*" gestionado como una rejilla, donde los componentes estarán localizados, dentro de la misma, en una posición o celda (x,y) que el usuario luego podrá manejar para redimensionar el tamaño del componente, maximizar, cerrar o agrupar/desagrupar uno o más componentes.

En el lado servidor, la IU es gestionada como una colección de componentes implementados como *Widgets* W3C almacenados y disponibles en un repositorio Wookie. En el caso del prototipo de aplicación *mashup* como ENIA (que veremos con más detalle en el siguiente capítulo), por su naturaleza (*i.e.*, dominio de información medioambiental), muchos de los componentes gestionados por el sistema son servicios OGC[7]. Son componentes externos (desarrollados por terceros) que han sido incluidos en el sistema como componentes con código envoltorio. Internamente existe una única instancia concreta para cada componente, pero cada vez que un usuario decide consumir una instancia de ese componente, internamente en la infraestructura se crea una instancia específica del componente, una para cada usuario que lo consuma. Por tanto, en el sistema existirá una referencia para el componente concreto, y una o varias referencias de sus instancias. Con todas estas consideraciones de contexto, veamos algunas características de la operación y de su comportamiento.

Como se puede apreciar en la interfaz de la operación, ésta acepta como entrada una estructura `InitUserArchitectureSessionParams` con el identificador del usuario que tiene la sesión iniciada, y cierta información de interacción relacionada/proporcionada con/por el dispositivo del usuario. Dicha información de dispositivo se refiere al tipo de dispositivo desde el cual se está accediendo al sistema (*e.g.*, Browser, Phone, Tablet, TV), el tipo de interacción de entrada que se está utilizando (MouseKeyboard, Voice, Gestural, Touch), y la localización geográfica (en coordenadas) del dispositivo.

La operación obtiene el identificador del modelo de arquitectura *concreta* asociada a la aplicación *mashup* del usuario (*i.e.*, la interfaz gráfica Web de usuario de componentes *COTSgets*), y a partir de ese identificador la operación obtiene el modelo *concreto*, el cual representa la lista de componentes de la arquitectura *mashup* (*i.e.*, componentes *COTSgets* para el caso de las interfaces gráficas). La lista de componentes es gestionada como una estructura `componentData` que almacena un conjunto de propiedades para cada componente de la arquitectura *mashup*. Dichas propiedades se refieren: al nombre del componente *concreto*; el alias de la instancia; el identificador de la instancia; un código en HTML que se incluye en el *widget* dentro del *iframe* cuando se despliega en el lado cliente; un objeto Java serializado y el objeto JAR, para el caso de aplicaciones *mashup* basada en JAVA, como por ejemplo el caso del escenario domótico, introducido en el Capítulo 2, como un escenario de arquitectura de componentes alternativo al de las IU; la posición del componente en la rejilla; sus dimensiones; y propiedades de visibilidad

[7]Componentes que para su implementación siguen el estándar OGC (Open Geospatial Consortium, `http://www.opengeospatial.org`).

© 2016 Vallecillos, J.

y de agrupación del componente. La estructura también tiene una lista de los servicios externos en OGC que encapsula. La operación también devuelve un mensaje con el resultado de la operación. En la siguiente tabla se muestra el listado de posibles mensajes devueltos por la operación.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos al buscar alguna clase necesaria para la ejecución.
`Not found or Empty userId Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found Interaction Information Error`	Se produce cuando se omite este parámetro en la llamada al servicio.
`PSQLException`	Se produce por problemas en la conexión a la base de datos.
`Error in Architectural Models BD`	Se produce por problemas en la consulta a la base de datos de modelos de arquitectura.
`Error in Wookie`	Se produce por problemas en la consulta al repositorio Wookie.

Comportamiento: Para analizar el comportamiento de la operación *Init User Architecture* haremos uso de la Figura 3.17 que muestra el flujo de tareas que suceden a partir de su llamada. La operación se encarga de inicializar una aplicación *mashup* para un usuario, esto es, despliega y activa todos los módulos y componentes necesarios que dan soporte al modelo cliente/servidor de la aplicación del usuario. Cuando se inicia una petición, en primer lugar la aplicación cliente establece una comunicación con el servidor *JavaScript* solicitando la iniciación. Posteriormente, la operación *InitUserArchitecture* emite la solicitud al módulo UIM. Dentro de la plataforma independiente, la primera tarea que tiene lugar es la de consultar el usuario en la base de datos (*Query User in Data Base*). A continuación, el módulo *COSSessionMM* inicializa el modelo de arquitectura de la aplicación *mashup* por medio de la tarea *Initialize user architecture*. Este proceso de inicio conlleva leer los componentes de la base de datos de componentes tanto si son del tipo Java (*i.e.*, aplicaciones *mashup* escenario tipo domótica) como si son del tipo *widget* (*i.e.*, una interfaz gráfica Web como la de ENIA). Previamente, por mediación del controlador *Manage component specifications*, se obtienen las propiedades de los componentes de la arquitectura de la aplicación *mashup*, como el tamaño inicial que deban tener los componentes, su posición inicial, y otras propiedades (vistas anteriormente), con el propósito de realizar el despliegue de dicha arquitectura en el lado cliente. Cuando ya se conocen las propiedades de los componentes, se procede a leer los componentes concretos de los respectivos repositorios. Para esta lectura, tanto el controlador *Manage Java* como el controlador *Manage Wookie* obtienen los componentes de los respectivos y si no existen instancias ya creadas para dichos componentes, las crean. Estas instancias son las que se utilizarán en la aplicación *mashup* cuando se despliegue en la capa cliente. Después de obtener las instancias o de crearlas (en el caso de que dichas instancias no estuvieran creadas previamente), el módulo DMM (encargado de

leer el modelo de arquitectura desde la base de datos, para posteriormente construir una estructura de componentes) se comunica con el módulo COSSessionMM para que éste devuelva el conjunto de componentes de la aplicación Java (para el caso de aplicaciones Java como los escenarios de domótica) o el código HTML *widget* de una arquitectura Web (para el caso de las aplicaciones IU basadas en *COTSgets* como la de ENIA). Como resultado de la operación, se devuelve una lista con el código necesario para llevar a cabo el despliegue de los componentes que forman parte de las aplicaciones *mashup*.

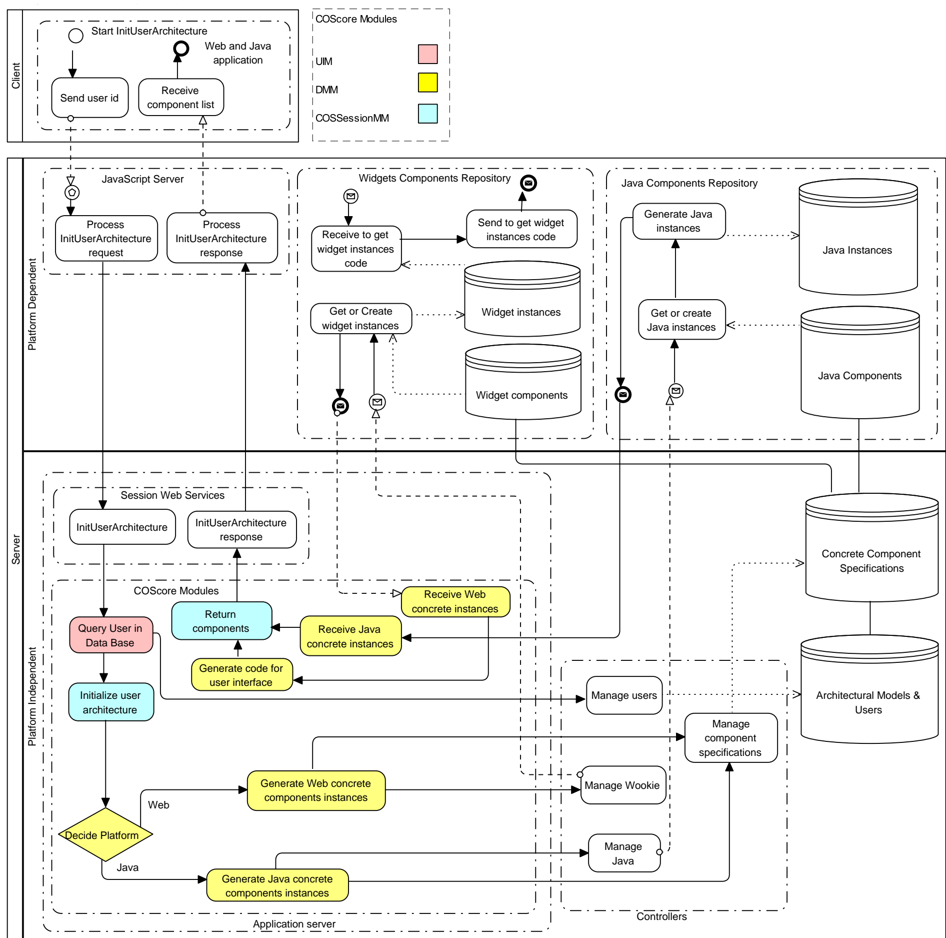


Figura 3.17: Operación *Init user architecture* de *Session Service*

Implementación: En el siguiente listado se muestra el código de la implementación de la operación *Init user*. En esta implementación, se realizan dos comprobaciones antes de comenzar con la inicialización de la arquitectura de la aplicación *mashup*. Como parámetros, a esta operación se le pasa un objeto `InitUserArchitectureParams`. Para este objeto, en primer lugar se comprueba que el identificador del usuario es correcto. En

el caso de que el identificador del usuario no sea correcto la operación devuelve un mensaje `Not found or Empty UserId Error`. En segundo lugar se comprueba que no ha habido ningún problema con la estructura `UserInteractionData` que viene por parámetros. En caso de error, se devuelve un mensaje `Not found Interaction Information Error`. A continuación, si todo está correcto, se crea un módulo *COSSessionMM*. A partir de ese módulo, y utilizando el identificador del usuario asociado, se accede al módulo *DMM* para inicializar la arquitectura de la aplicación del usuario. Para ello, al método `initModelforUsers` (línea 15) se le pasa el identificador del usuario y el conjunto de parámetros necesarios para inicializar el modelo de arquitectura de la aplicación *mashup* por medio de la clase `UserInteractionData`. Este método busca en la base de datos de modelos de arquitectura el modelo de la aplicación para el usuario que solicita la inicialización. Luego se localizan las instancias de los componentes de la aplicación, y si estas no estuvieran creadas con anterioridad, se crean en ese instante. Por último, la operación devuelve el conjunto de componentes necesarios para iniciar y desplegar la aplicación en el lado cliente.

```
 1 public InitUserArchitectureSessionResult initUserArchitecture(
 2                         InitUserArchitectureSessionParams params) {
 3 InitUserArchitectureSessionResult result =
 4                         new InitUserArchitectureSessionResult();
 5
 6 if (params.getUserId() != null && params.getUserId().compareTo("") != 0) {
 7    if (params.getInteraction().getDeviceType() != null &&
 8                        params.getInteraction().getInteractionType() != null &&
 9                        params.getInteraction().getLatitude() != null &&
10                        params.getInteraction().getLongitude() != null) {
11       try {
12          Context initialContext = new InitialContext();
13          COSSessionMM cossmng = (COSSessionMM)initialContext.
14                                 lookup("java:app/cos/COSSessionMM");
15          result = cossmng.initModelforUsers(
16                                 params.getUserId(),params.getInteraction());
17       } catch (NamingException e) {
18          LOGGER.error(e);
19          result.setInit(false);
20          result.setComponentData(null);
21          result.setMessage("> Internal Server Error");
22       }
23    } else {
24       LOGGER.error("Not found Interaction Information Error");
25       result.setInit(false);
26       result.setComponentData(null);
27       result.setMessage("> Not found Interaction Information Error");
28    }
29 } else {
30    LOGGER.error("Not found or Empty UserId Error");
31    result.setInit(false);
32    result.setComponentData(null);
33    result.setMessage("> Not found or Empty UserId Error");
34 }
35 return result;
36 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:InitUserArchitecture>
      <params>
        <userId> ejemplo Id </ userId >
        <interaction>
          <deviceType> ejemplo de dispositivo o vacío </deviceType>
          <interactionType> ejemplo de tipo de interacción o vacío </interactionType>
          <latitude> ejemplo de latitud o vacío </latitude>
          <longitude> ejemplo de longitud o vacío </longitude>
        </interaction>
      </params>
    </ws: InitUserArchitecture >
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:InitUserArchitectureResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <init> ejemplo true o false </ init >
        <componentData> ... </componentData>   ...
        <componentData> ... </componentData>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:InitUserArchitectureResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:InitUserArchitectureResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <init> false </init>
        <message> > org.postgresql.util.PSQLException:
                 ERROR: no existe la relación «coscoreuser» Position: 19 </message>
      </result>
    </ns2:InitUserArchitectureResponse>
  </soap:Body>
</soap:Envelope>
```

3.4.1.4. Operación *Default Init*

Esta operación es la última dentro del servicio *Session Service* que da soporte a la gestión de inicio de los usuarios en el sistema. Esta operación se emplea con el objetivo dar un contexto inicial a los usuarios que no están registrados en el sistema para asignarles un

© 2016 Vallecillos, J.

modelo de arquitectura por defecto, es decir una configuración para la arquitectura de inicio. Este servicio es controlado por el componente COSSessionMM (Módulo de Gestión de Sesiones del COSCore) y hace uso de dos módulos UIM (Módulo de Información de Usuarios), para crear un usuario anónimo y asignarles un modelo de arquitectura concreto (CAM), y DMM (Módulo de Visualización de Componentes) para obtener el modelo CAM anónimo. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación. Más adelante se realiza una explicación para dicha interfaz y sus estructuras de datos.

Interfaz:

```
1 @WebMethod(operationName="defaultInit", action="defaultInit")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3        @XmlElement(required=true)
4 public DefaultInitSessionResult defaultInit();
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
	La operación no tiene parámetros de entrada		
Parámetros de salida			
`result`	Un valor *structure* `DefaultInitSessionResult` con los valores de salida:		
	boolean	`init`	Variable que muestra si tuvo éxito la acción, *true* si tiene éxito la creación del usuario anónimo y la obtención de su modelo por defecto.
	string	`anonymousId`	Texto con el identificador de usuario anónimo en la BD, devuelve la cadena -1 si hay algún error. Este identificador es único y está formado por una cadena de texto que contiene la cadena `anonymous`, más la fecha y hora del sistema con una precisión de milisegundos.
	list structure	`componentData`	Lista de componentes del modelo concreto para ese usuario (véase la estructura en la tabla de la operación *Init user*).
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: Esta operación hace uso de las operaciones *CreateUser* (servicio privado *User service*), *Login* e *Init user architecture* (ambas del servicio *Session service*). Por tanto, la operación *Default Init* crea una sesión de usuario anónimo, creando un tipo de usuario especial anónimo, iniciando la sesión del usuario y asignando un modelo de arquitectura *mashup* establecido por omisión para usuarios anónimos. Este usuario anónimo será identificado del resto por la fecha de conexión a la aplicación, hora, minutos y segundos. Hay que tener en cuenta que en un momento dado ese usuario puede solicitar darse de alta en el entorno, por lo que sería necesario almacenar el estado de su modelo de arquitectura de componentes para la aplicación, siempre y cuando no haya seleccionado un perfil de usuario determinado durante el proceso de registro en el sistema; en caso contrario se le asignaría el modelo de arquitectura de componente por omisión establecido para el perfil elegido (hay que recordar que para cada perfil de usuario, en el sistema existe un modelo de arquitectura de componentes por omisión). La operación

devuelve como respuesta la estructura `DefaultInitSessionResult` compuesta por una variable lógica que indica el éxito de la operación, el identificador creado para ese usuario anónimo, una la lista de componentes por defecto para el usuario anónimo y un mensaje correspondiente a la operación realizada.

Mensajes de error:

error	descripción
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.
`PSQLException`	Se produce por problemas en la conexión a la base de datos, al crear o consultar el usuario anónimo.
`Error in Architectural Models BD`	Se produce por problemas en la consulta a la base de datos de modelos de arquitectura.
`Error in Wookie`	Se produce por problemas en la consulta al repositorio Wookie.

Comportamiento: El flujo de tareas del comportamiento de la operación mostrada en la Figura 3.18 es muy parecido al de la operación *Init user architecture* mostrado anteriormente, salvo que en este caso, no se lee el usuario de la base de datos, sino que se crea un usuario por defecto. Observando el flujo de tareas de comportamiento, en primer lugar, cuando se accede directamente a la aplicación *mashup*, desde el lado cliente se solicita el establecimiento de la operación *InitUserArchitecture* con el servidor *JavaScript* que recibe la petición y emite la solicitud de inicialización al módulo *COSSessionMM*. Dicho módulo, por medio de la tarea *Initialize user architecture*, inicializa el modelo de arquitectura de la aplicación *mashup* que hay por defecto en el sistema, preparado para el usuario anónimo. Cuando ya se tiene el modelo de arquitectura de la aplicación *mashup* para el usuario anónimo, el modelo se carga en memoria. Este proceso de inicio conlleva leer los componentes de la base de datos de componentes dependiendo del tipo de aplicación, esto es, si la aplicación es de tipo Java o de tipo Web. Antes de leer los componentes de sus respectivos repositorios, se consultan las propiedades de los componentes por medio del controlador *Manage component specifications*. Tras ello, bien el controlador *Manage Java* o el controlador *Manage Wookie* (dependiendo del tipo de aplicación) acceden al repositorio de componentes, y si no existen instancias asociadas a dichos componentes, las crean. Estas instancias se usan por la aplicación *mashup* del lado cliente. De esta forma, se reciben las instancias en el módulo DMM, que se pone en contacto con el módulo COSSessionMM para devolver la lista de componentes para hacer el despliegue correcto de la aplicación *mashup* ubicada en el lado cliente.

Implementación: En el siguiente listado se muestra parte del código de la implementación de la operación *Default init*. En primer lugar, se crea el módulo *COSSessionMM* (línea 8) que se encarga de construir el conjunto de componentes pertenecientes a la aplicación *mashup*. Una vez obtenido dicho módulo, se llama al método `initAnonymous`. Este método es el núcleo de la operación, encargado de crear un usuario anónimo en la base de datos de usuarios, y al cual se le asigna un modelo de aplicación por defecto para este tipo de usuarios. A continuación, se crean todos lo módulos necesarios

© 2016 Vallecillos, J.

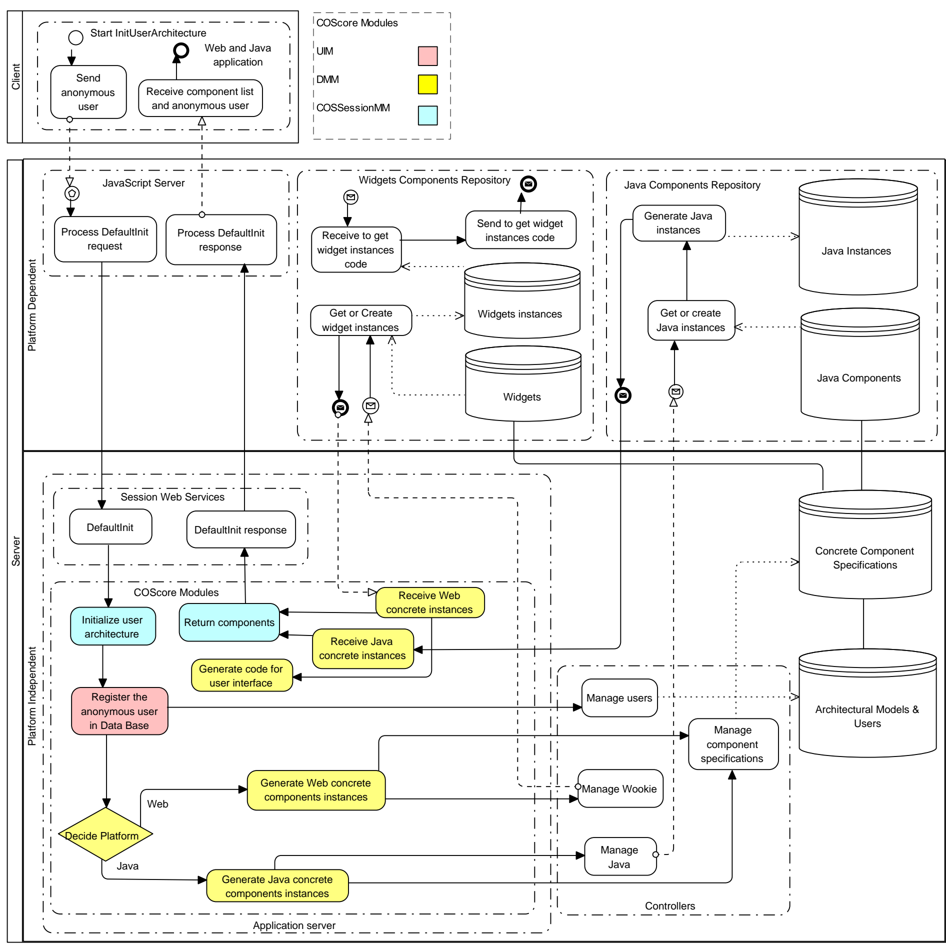


Figura 3.18: Operación *Default Init* de *Session Service*

en la infraestructura para dar soporte a la aplicación *mashup*. Después se generan todas las instancias necesarias para la aplicación *mashup* de dicho usuario. Este conjunto de instancias son devueltas a través de la variable `result`. Esta variable es de tipo `DefaultInitSessionResult`. En caso de producirse algún error en el proceso de inicialización de la aplicación, ocurrirá un error gestionado con el bloque `catch`, informando con un mensaje de "*Internal Server Error*".

```
1 public DefaultInitSessionResult defaultInit(){
2
3 DefaultInitSessionResult result = new DefaultInitSessionResult();
4 COSSessionMM cossmng = null;
5 Context initialContext;
6 try {
```

```
 7     initialContext = new InitialContext();
 8     cossmng = (COSSessionMM) initialContext.lookup("java:app/cos/COSSessionMM");
 9     result = cossmng.initAnonymous();
10 } catch (NamingException e) {
11     LOGGER.error(e);
12     result.setInit(false);
13     result.setAnonymousId("-1");
14     result.setComponentData(null);
15     result.setMessage("> Internal Server Error");
16 }
17 return result;
18 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:defaultInit/>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:DefaultInitResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <init> ejemplo true o false </ init >
        <anonymousId> anonymous-fecha-hora </anonymousId>
        <componentData> ... </componentData>
          ...
        <componentData> ... </componentData>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2: DefaultInitResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2: DefaultInitResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <init> false </init>
        <anonymousId> -1 </anonymousId>
        <message> > org.postgresql.util.PSQLException:
                   ERROR: no existe la relación «coscoreuser» Position: 13 </message>
      </result>
    </ns2: DefaultInitResponse >
  </soap:Body>
</soap:Envelope>
```

© 2016 Vallecillos, J.

3.4.2. Servicio *Communication Service*

El servicio *Communication Service* se encarga de dar soporte a la sesión de los usuarios en el sistema, y está constituido por una única operación denominada *Get links components*, que se describe a continuación.

3.4.2.1. Operación *Get Link Components*

La operación *Get Link Components* se emplea para que el servidor `nodejs` pueda obtener las conexiones para un determinado puerto de un componente, y de esta forma saber con qué otros componentes se comunica y qué puertos intervienen. Este servicio es controlado por el componente TMM (Módulo de Gestión de Transacción). La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="getLinksComponents", action="getLinksComponents")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3     @XmlElement(required=true)
4 public GetLinksResult getLinksComponents(@WebParam(name="params",
5         targetNamespace="http://ws.cos.acg.ual.es/")@XmlElement(required=true)
6         GetLinksParams params);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un valor *structure* `GetLinksParams` con los parámetros:		
	string	`userId`	Identificador de usuario con una sesión iniciada en el sistema. Obligatorio no nulo.
	string	`componentInstance`	Instancia del componente. Obligatorio no nulo.
	string	`PortId`	Puerto del componente. Obligatorio no nulo.
Parámetros de salida			
`result`	Un valor *structure* `GetLinksResult` con los valores de salida:		
	boolean	`gotten`	*true* si tiene éxito la obtención de la lista de componentes para ese usuario, *false* si no.
	string	`portList`	Cadena con lista concatenada de puertos que conectan con el puerto consultado.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: El propósito de esta operación es resolver caminos de comunicación de las dependencias entre componentes de la aplicación *mashup*. Por ejemplo, para el caso de una interfaz gráfica *mashup* como la de ENIA pueden existir en el lado cliente dos componentes mapa A y B relacionados entre sí, de forma que la interacción del usuario sobre uno de ellos afecte al otro. La comunicación entre dichos componentes no sucede en el lado cliente, sino que a través de esta operación, se establece un canal de comunicación hacia el lado servidor donde se produce el intercambio de información de A hacia B y que finalmente se despliega en el cliente. La operación acepta en su entrada una estructura de datos `GetLinksParams` con el identificador del usuario que tiene la sesión iniciada, el identificador de instancia del componente a consultar y el puerto en cuestión; estos

valores son obligatorios y además no nulos. La operación calcula la tabla de enrutamiento de los componentes y busca la ruta para el componente y puerto proporcionado. Como resultado, la operación devuelve la estructura `GetLinksResult` que contiene un valor booleano para indicar si pudo o no realizar el proceso de encaminamiento de información entre los componentes, una cadena con los puertos a los que se puede conectar y un mensaje correspondiente a la operación realizada. En la siguiente tabla se resume los tipos de mensajes que la operación puede devolver.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.
`Not found or Empty userId Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty Component Instance Error`	Se produce cuando se omite este parámetro en la llamada al servicio.
`Not found or Empty Port Error`	Se produce cuando se omite este parámetro en la llamada al servicio o sin valor. También cuando el puerto consultado no existe en la BD.
`PSQLException`	Se produce por problemas en conexión en BD.
`Error in Architectural Models BD`	Se produce por problemas en la consulta a la base de datos de modelos de arquitectura.

Comportamiento: En la Figura 3.19 se muestra el flujo de tareas que sucede internamente a la operación. Como se ha adelantado, esta operación establece un canal de comunicación para el paso de información entre dos componentes relacionados en la aplicación *mashup* desplegada en el lado cliente. Para comunicar componentes entre sí la aplicación cliente emite hacia el servidor *JavaScript* el identificador del componente encargado de realizar la emisión y la información que quiere enviar de ese componente. A continuación, el servidor *JavaScript* reenvía la solicitud a la operación *Get link components*. Esta operación realiza un registro del proceso de comunicación que está teniendo lugar entre los componentes y para ello utiliza el módulo *Store interaction*. Almacenar esta interacción implica tener que utilizar el controlador *Manage interaction* para acceder a la base de datos *Interaction*. En paralelo, se resuelve el envío de la información al otro componente mediante la tarea *Calculate routing* del módulo *TMM*. Esta tarea utiliza una estructura de datos localizada en memoria que es dependiente para cada aplicación *mashup* de un usuario. Esta estructura de datos se construye cuando se lee por primera vez el modelo de arquitectura de la aplicación *mashup* del usuario y contiene qué componentes están relacionados entre sí. A continuación, el módulo *TMM* devuelve al servidor *JavaScript* el identificador de componente al cual se debe enviar la información. Una vez llega la información al servidor *JavaScript*, éste decide si debe comunicarse con la aplicación Java o Web.

Implementación: En el siguiente listado se muestra parte del código de la implementación realizada para la operación *Get link components*. Observando el código, primero se comprueba si ha habido algún problema en la recepción de los parámetros en la operación del servicio. Si todo ha llegado de forma correcta, se obtiene el módulo *TMM*

© 2016 Vallecillos, J.

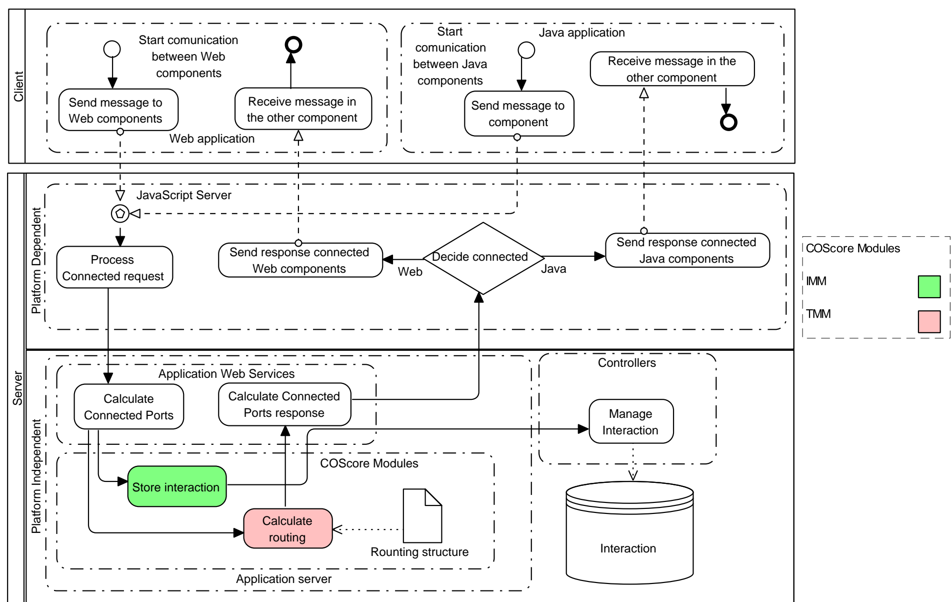


Figura 3.19: Operación *Get link components* de *Communication Service*

perteneciente al usuario en la línea 15. En la línea 16 se localiza con qué componente se debe comunicar el primero, encargado de emitir la información a través del método `calculateConnectedPorts`. En este método se hace uso de una estructura de datos auxiliar y cargada en memoria. Esta estructura de datos se construye a partir del modelo de la aplicación *mashup* en el momento de inicializar la aplicación. Esta estructura de datos contiene la arquitectura de componentes y sus relaciones, donde se establece qué componentes están conectados entre sí, permitiendo resolver de forma rápida la consulta. Una vez se conoce a que componente o componentes se debe informar, se construye un objeto de tipo `GetLinksResult`.

```
 1 public GetLinksResult getLinksComponents(GetLinksParams params) {
 2
 3 GetLinksResult result = new GetLinksResult();
 4 result.setGotten(false);
 5 result.setPortList(null);
 6 if (params.getUserId() != null && params.getUserId().compareTo("") != 0) {
 7    if (params.getComponentInstance() != null
 8       && params.getComponentInstance().compareTo("") != 0) {
 9       if (params.getPortId() != null
10           && params.getPortId().compareTo("") != 0) {
11           TMM tmm = null; Context initialContext;
13           try { initialContext = new InitialContext();
15                 tmm = (TMM) initialContext.lookup("java:app/cos/TMM");
16                 result = tmm.calculateConnectedPorts(params.getUserId(),
17                 params.getComponentInstance(), params.getPortId());
18           } catch (Exception e) {
```

```
19              LOGGER.error(e); result.setMessage("> Internal Server Error"); }
20          } else {
21              LOGGER.error("Not found o Empty Port Error");
22              result.setMessage("> Not found o Empty Port Error"); }
23      } else {
24          LOGGER.error("Not found o Empty Component Instance Error");
25          result.setMessage("> Not found o Empty Component Instance Error"); }
26 } else {
27    LOGGER.error("Not found o Empty userid Error");
28    result.setMessage("> Not found o Empty username Error"); }
29 return result;
30 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:getLinksComponents>
      <params>
        <userId> ejemplo Id </userName>
        <componentInstance> ejemplo de identificador de componente </componentInstance>
        <portId> ejemplo de puerto </portId>
      </params>
    </ws: getLinksComponents>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:getLinksComponentsResponse xmlns:ns2="http://ws.céxitoos.acg.ual.es/">
      <result>
        <gotten> ejemplo true o false </gotten>
        <portList> ejemplo lista de puertos </portList>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:getLinksComponentsResponse >
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2: getLinksComponentsResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <gotten> false </gotten>
        <portList> No connected ports </portList>
        <message> > Not found o Empty Port Error </message>
      </result>
    </ns2: getLinksComponentsResponse>
  </soap:Body>
</soap:Envelope>
```

© 2016 Vallecillos, J.

3.4.3. Servicio *Component Service*

El servicio *Component Service* se encarga de dar soporte a los cambios producidos sobre los componentes de la aplicación *mashup*. El servicio contiene una única operación, denominada como *Update Architecture* y que se describe a continuación.

3.4.3.1. Operación *Update Architecture*

La operación *Update Architecture* tiene una doble funcionalidad. Por un lado, se encarga de gestionar las distintas acciones que se pueden producir sobre los componentes de la aplicación *mashup*. Como se ha venido comentando, el modelo de infraestructura COScore propuesto se ha llevado a cabo sobre la implementación de una interfaz gráfica Web de usuario basada en componentes *widgets*, como escenario de aplicación *mashup*. Dicha interfaz está compuesta por una colección de componentes con un tamaño prestablecido, dispuestos y ordenados en una rejilla (no visible). Los usuarios pueden directamente interaccionar sobre los componentes para realizar ciertas acciones sobre ellos. Para la implementación, se han desarrollado estas cinco **acciones**:

(a) `add`: añade un nuevo componente al modelo de la aplicación *mashup* (*i.e.*, a la rejilla, en el caso de interfaces gráficas *mashup*) previamente seleccionado por el usuario desde un catálogo de componentes.

(b) `delete`: elimina un componente del modelo de la aplicación *mashup*, esto es, elimina de la rejilla un componente seleccionado por el usuario.

(c) `changeproperty`: cambia alguna de las propiedades de algún componente perteneciente al modelo de la aplicación *mashup*.

(d) `groupfromdesktop`: agrupa algún componente del escritorio de componentes perteneciente al modelo de la aplicación *mashup*. Esta acción tiene lugar sólo para las aplicaciones *mashup* de tipo web.

(e) `ungroupfornew`: desagrupa un conjunto de componentes que forman parte de la aplicación *mashup*. Esta acción sólo tiene lugar para las aplicaciones *mashup* web.

Las acciones de agrupación y desagrupación de componentes son gestionadas mediante contenedores, de forma parecida a como han sido descritos previamente en el Capítulo 2 en los patrones de dependencias. En el capítulo siguiente se describe un caso estudio completo de aplicación *mashup* de una interfaz gráfica basada en componentes *widgets* denominada ENIA, donde se describirán en detalle las acciones permitidas sobre los componentes de la interfaz, y sobre las cuales se ha realizado la implementación de la operación *Update Architecture*.

Por otro lado, esta operación también se encarga de almacenar todas estas acciones de interacción en una base de datos de interacción, con el propósito de llevar un registro "*Log*" de la interacción[8]. Como veremos más abajo, este servicio es controlado por el

[8] En la actualidad, esta información de registro de interacción almacenada está siendo usada por el Grupo de Investigación TIC-211 en otra tesis doctoral en curso para inferencia de información.

componente DMM (Módulo de Visualización de Componentes) y hace uso del módulo IMM (Módulo de Gestión de la interacción) para registrar en la base de datos de interacción los cambios realizados. La interfaz y los parámetros de entrada y de salida de la operación se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="updateArchitecture", action="updateArchitecture")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3      @XmlElement(required=true)
4 public UpdateArchitectureResult updateArchitecture(
5    @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6    @XmlElement(required=true) UpdateArchitectureParams params);
```

Parámetros de Entrada/Salida:

Parametros de entrada			
`params`	Un valor *structure* `UpdateArchitectureParams` con los parámetros:		
	string	`userId`	Id de usuario con una sesión iniciada en el sistema. Parámetro obligatorio no nulo.
	string	`actionDone`	Tipo acción: `add`, `delete`, `changeproperty`, `groupfromdesktop`, o `ungroupfornew`. Parámetro obligatorio no nulo.
	string	`componentInstance`	Vacío para este caso.
	liststructure	`newComponentData`	Lista de componentes del modelo concreto para ese usuario. Véase estructura `ComponentData` definida en la operación *Init User Architecture*.
	structure	`interaction`	Un valor *structure* con los datos de interacción que puede proporcionar el usuario. Véase estructura `Interaction` definida en la operación *Init User Architecture*.
Parámetros de salida			
`result`	Un *structure* `UpdateArchitectureResult` con los valores de salida:		
	boolean	`allowed`	Variable que muestra si tuvo éxito la acción, *true* si tiene éxito la obtención de la lista de componentes para ese usuario, *false* si no.
	liststructure	`oldComponentData`	Lista antigua de componentes del modelo concreto para ese usuario. Véase estructura `ComponentData` definida en la operación *Init User Architecture*.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: La operación *Update architecture* es la operación más compleja de todas las operaciones pertenecientes a los servicios públicos. Como se ha dicho, el objetivo de esta operación es hacer cambios sobre el modelo de arquitectura de componentes de las aplicaciones *mashup*. Eso implica que debe gestionar todos los cambios de tamaño de los componentes, cambios de posición de los componentes dentro del modelo de arquitectura de la aplicación *mashup*, debe controlar la eliminación y agregación de componentes dentro del modelo, además de otros eventos relacionados con los escenarios de experimentación que aquí se tratan. La operación en primer lugar identifica la acción a realizar,

© 2016 Vallecillos, J.

acto seguido comprueba que la operación sea posible y se permita para ese modelo, desarrolla la acción haciendo los cambios necesarios en el modelo de componente concreto y en las bases de datos y por último se encarga de mandar registrar la interacción realizada. Para ello, la operación recibe un objeto de tipo `UpdateArchitectureParams` el cual contiene una cadena de texto con el identificador de usuario con el que se identifica la aplicación *mashup* que sufrirá algún cambio. También, en caso de tratarse de una operación relacionada con un componente en concreto, se almacena en este objeto el identificador del componente que se ve afectado. Además, contiene una lista del resto de componentes que el modelo de arquitectura de la aplicación contiene. Por último, guarda la acción que se realizará sobre el modelo para tenerla en cuenta cuando se actúen sobre los cambios. Esta operación está soportada por el módulo *Display Management Module* (*DMM*). Este módulo está dedicado a gestionar todos los procesos de visualización de la aplicación *mashup*. Como respuesta, la operación devuelve el objeto de estructura `UpdateArchitectureResult` con una variable que indica si se ha permitido la acción, un mensaje correspondiente a la operación realizada, en caso de error, la lista de componentes antigua (*i.e.*, el estado anterior previo a la acción realizada).

Mensajes de error:

error	descripción
`Not found or Empty UserId Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty ComponentsInstance Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty Action Done Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.Posx or Posy List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.idHtml List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.Alias List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.componentname List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.instanceID List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.Servicios.componentalias List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.Servicios.componentname List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.Servicios.instanceId List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty News Components.Numero_servicios List Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found Interaction Information Error`	Se produce cuando se omite este parámetro en la llamada al servicio.
`Not found Component`	Se produce cuando el componente proporcionado que sufre la acción no se localiza en el sistema.
`PSQLException`	Se produce por problemas en la conexión a la base de datos.
`Error in Architectural Models BD`	Se produce por problemas en la consulta a la base de datos de modelos de arquitectura.
`Error in Wookie`	Se produce por problemas en la consulta al repositorio Wookie.
`Error in Register Interaction`	Se produce por problemas en la consulta a la base de datos de interacción.
`Error in Component specification BD`	Se produce por problemas en la consulta a la base de datos de especificación de componentes.
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.

Comportamiento: En la Figura 3.20 se muestra el flujo de información interno de la operación *Update Architecture*. Desde la aplicación *mashup* se envía a esta operación del servicio del COScore el componente sobre el que se realizó alguna acción de cambio (*i.e.*, `add`, `delete`, `changeproperty`, `groupfromdesktop`, `ungroupfornew`) junto a la lista de componentes que conforman la arquitectura de la aplicación *mashup*. El servidor *JavaScript*, situado en la capa dependiente de la plataforma, reenvía la interacción a la operación *Update architecture* del servicio *Component Service*. Una vez dentro del servicio se lee el identificador del modelo de arquitectura asociada a la aplicación *mashup* del usuario, para posteriormente acceder a dicho modelo mediante el módulo DMM. A continuación, el módulo DMM decide qué acción se lleva a cabo sobre el modelo de la aplicación *mashup* en función de la acción introducida por parámetros en la operación. Cuando se llega a este punto, se modifica el modelo de arquitectura de la aplicación *mashup*, cambiando alguna propiedad del componente, eliminando un componente o agregándolo. La acción de añadir un componente implica previamente leer las propiedades del componente pasando por el controlador *Manage component specifications* para poder interactuar con la base de datos *Concrete component specifications*. A continuación, se guarda el modelo de arquitectura de la aplicación *mashup* y se registra la interacción en la base de datos *Interaction*, invocando al controlador *Manage interaction*. Para finalizar, se reconstruye la nueva lista de componentes que conforman la aplicación *mashup*. Esta lista se devuelve a través del servicio, para que la aplicación *mashup* pueda reconstruirse (desplegarse en el lado cliente).

Implementación: En el listado siguiente se muestra parte del código de la operación *Update architecture*. Primero, se comprueban la validez de los parámetros de entrada (línea 9) y que el identificador del usuario sea el correcto. También se comprueba que el parámetro *actionDone* de la estructura `UpdateArchitectureParams` y la lista de componentes no sean nulos (líneas 10 y 11 respectivamente). En caso de no haberse producido ningún error, se comienzan a realizar los cambios necesarios sobre el modelo. Debido a que la implementación de esta operación es demasiado extensa, en este documento se ha incluido sólo una pequeña parte del mismo. La implementación completa está disponible en el portal web de COScore API elaborado para esta tesis doctoral[9]. Todas las acciones de la operación son manejadas por el módulo *DMM*. Por último, se lanza un mensaje de error en caso de no poder realizarse ninguna de las acciones mencionadas.

```
 1 public UpdateArchitectureResult updateArchitecture(UpdateArchitectureParams params){
 2 UpdateArchitectureResult result = new UpdateArchitectureResult();
 3 InterModulesData resultDMMforError = new InterModulesData();
 4 try {  Context initialContext = new InitialContext();
 5    DMM dmm = (DMM)initialContext.lookup("java:app/cos/DMM");
 6    resultDMMforError = dmm.getCurrentModelforUser(params.getUserId(),null);
 7    result.setOldComponentData(resultDMMforError.getModel());
 8   result.setAllowed(false);
 9    if (params.getUserId() != null && params.getUserId().compareTo("") != 0) {
10     if (params.getActionDone() != null){
11      if (params.getNewsComponentData() != null){
12       if (...) {
13        switch (params.getActionDone()){
```

[9]COScore API — `http://acg.ual.es/projects/enia/ui/webservices/`

© 2016 Vallecillos, J.

```
14          case "add": ... result = dmm.updateArchitectureforUser(...); ... break;
15          case "delete": ... result = dmm.updateArchitectureforUser(...); ... break;
16          case "changeproperty": ... result = dmm.updateArchitectureforUser(...); ...
17             break;
18          case "groupfromdesktop": ... result = dmm.updateArchitectureforUser(...); ...
19             break;
20          case "ungroupfornew": ... result = dmm.updateArchitectureforUser(...); ...
21             break;
22          default:
23              LOGGER.error("Not found or Empty Action Done Error");
24              result.setMessage("> Not found or Empty Action Done Error");
25              break;
26        }
27       // Errors ...
28 } catch (NamingException e) {  ...  } return result;
29 }
```

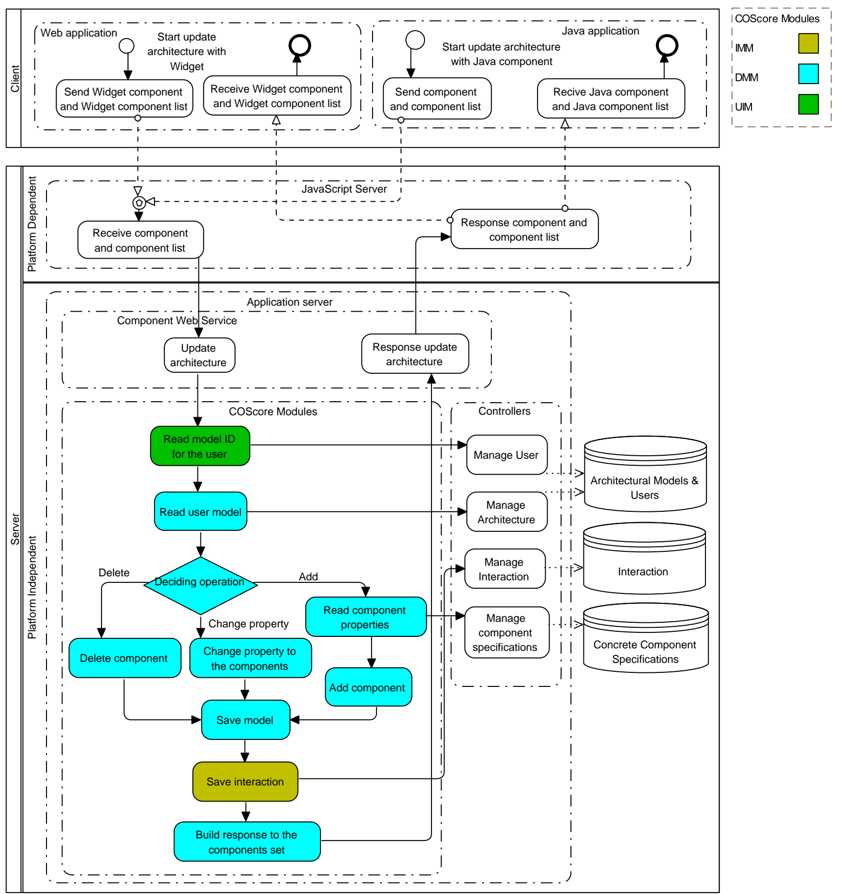


Figura 3.20: Operación *Update architecture* de *Component Service*

Ejemplo Petición XML (add):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:UpdateArchitecture>
      <params>
        <userId> ejemplo de Id </userId>
        <actionDone> add </actionDone>
        <newComponentData>
          <componentname> ejemplo de component Name </componentname>
          <componentAlias> ejemplo de component Alias </componentAlias>
          <instanceId> ejemplo de Instance Id o vacío si es el nuevo </instanceId>
          <idHtml> ejemplo de IdHtml </idHtml>
          <posx> ejemplo de posx </posx>
          <posy> ejemplo de posy </posy>
        <newComponentData>
                  ...
        <newComponentData> ... </newComponentData>
        <interaction>
          <deviceType> ejemplo de dispositivo o vacío </deviceType>
          <interactionType> ejemplo de tipo de interacción o vacío </interactionType>
          <latitude> ejemplo de latitud o vacío </latitude>
          <longitude> ejemplo de longitud o vacío </longitude>
        </interaction>
      </params>
    </ws:UpdateArchitecture>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Petición XML (delete):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:UpdateArchitecture>
      <params>
        <userId> ejemplo de Id </userId>
        <actionDone> delete </actionDone>
        <componentInstance> Id del componente borrado </componentInstance>
        <newComponentData>
          <instanceId> ejemplo de Instance Id </instanceId>
          <posx> ejemplo de posx </posx>
          <posy> ejemplo de posy </posy>
        </newComponentData>
          ...
        <newComponentData> ... </newComponentData>
        <interaction> ... </interaction>
      </params>
    </ws:UpdateArchitecture>
  </soapenv:Body>
</soapenv:Envelope>
```

© 2016 Vallecillos, J.

Ejemplo Petición XML (changeproperty):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:UpdateArchitecture>
      <params>
        <userId> ejemplo de Id </userId>
        <actionDone> changeproperty </actionDone>
        <componentInstance> Id del componente </componentInstance>
        <newComponentData>
          <instanceId> ejemplo de Instance Id </instanceId>
          <numero_servicios> total servicios cargados en componente </numero_servicios>
          <servicios> ... </servicios>
        </newComponentData>
           ...
        <newComponentData> ... </newComponentData>
        <interaction> ... </interaction>
      </params>
    </ws:UpdateArchitecture>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Petición XML (groupfromdesktop):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:UpdateArchitecture>
      <params>
        <userId> ejemplo de Id </userId>
        <actionDone> groupfromdesktop </actionDone>
        <componentInstance> Id del COTsget eliminado </componentInstance>
        <newComponentData>
          <instanceId> ejemplo de Instance Id </instanceId>
          <posx> ejemplo de posx </posx>
          <posy> ejemplo de posy </posy>
          <numero_servicios> total servicios cargados en componente </numero_servicios>
          <servicios> ... </servicios>
        </newComponentData>
            ...
        <newComponentData> ... </newComponentData>
        <interaction> ... </interaction>
      </params>
    </ws:UpdateArchitecture>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Petición XML (ungroupfornew):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ws="http://ws.cos.acg.ual.es/">
```

```
  <soapenv:Header/>
  <soapenv:Body>
    <ws:UpdateArchitecture>
      <params>
        <userId> ejemplo de Id </userId>
        <actionDone> ungroupfornew </actionDone>
        <componentInstance> Id del componente creado </componentInstance>
        <newComponentData>
          <instanceId> ejemplo de Instance Id </instanceId>
          <idHtml> ejemplo de IdHtml, debe incluir el nuevo identificador
                del nuevo COTSget que se ha creado </idHtml>
          <posx> ejemplo de posx </posx>
          <posy> ejemplo de posy </posy>
          <numero_servicios> ejemplo de numero de servicios cargado
                en ese componente </numero_servicios>
          <servicios> ... </servicios>
        </newComponentData>
           ...
        <newComponentData> ... </newComponentData>
        <interaction> ... </interaction>
      </params>
    </ws:UpdateArchitecture>
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:initUserArchitectureResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <init> ejemplo true o false </init>
        <componentData> ... </componentData>
         ...
        <componentData> ... </componentData>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:initUserArchitectureResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:InitUserArchitectureResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <init> false </init>
        <message> > org.postgresql.util.PSQLException: ERROR: no existe la relación
             «coscoreuser» Position: 19 </message>
      </result>
    </ns2:InitUserArchitectureResponse>
  </soap:Body>
</soap:Envelope>
```

© 2016 Vallecillos, J.

3.4.4. Servicio *Interaction Service*

El servicio *Interaction Service* representa el último servicio público de COScore. Este servicio se encarga de dar soporte al registro manual de la interacción de usuarios. Al igual que los otros dos servicios públicos, éste también está compuesto de una única operación denominada *Register Interaction* la cual se describe a continuación.

3.4.4.1. Operación *Register Interaction*

La operación *Register Interaction* está preparada para almacenar otra interacción del usuario con la aplicación *mashup* que pueda surgir, distinta a la tenida en cuenta en el sistema (tal y como se ha descrito en la operación *Update Architecture*). Esta operación se emplea con el objetivo de que el usuario pueda registrar información como crea conveniente en la BD, más allá de la que se registra al interactuar con el sistema de forma automática. Por ejemplo, cuando se realiza un cierre de sesión, esta interacción no se captura a través de la operación *Update Architecture* puesto que no realiza ninguna modificación de la aplicación *mashup*. No obstante, el usuario y/o la aplicación (actuando como clientes de la infraestructura) pueden llevar a cabo el registro de dicha interacción. Esta operación se realiza a través del módulo IMM (Módulo de Gestión de Interacción). La interfaz y los parámetros de entrada y de salida se muestran a continuación.

Interfaz:

```
1 @WebMethod(operationName="registerInteraction", action="registerInteraction")
2 @WebResult(name="result", targetNamespace="http://ws.cos.acg.ual.es/")
3         @XmlElement(required=true)
4 public RegisterInteractionResult registerInteraction(
5     @WebParam(name="params", targetNamespace="http://ws.cos.acg.ual.es/")
6     @XmlElement(required=true) RegisterInteractionParams params);
```

Parámetros de Entrada/Salida:

Parámetros de entrada			
`params`	Un valor *structure* `RegisterInteractionParams` con los parámetros:		
	string	`userId`	Id de usuario con una sesión iniciada en el sistema. Obligatorio no nulo.
	string	`newSession`	Nueva sesión iniciada en el sistema, “0” y “1” cuando se registra la interacción por el sistema.
	structure	`interaction`	Un *structure* con los datos de interacción que proporciona el usuario. Estructura `Interaction`, definida en la operación *Init User Architecture*.
	string	`operationPerformed`	La operación que se ha llevado a cabo y que se va a registrar. Obligatorio no nulo.
	string	`componentId`	Instancia del componente.
	list string	`groupComponent`	Lista de servicios agrupados. Se utiliza para expresar el conjunto de servicios de un componente que ha sufrido una agrupación.
	list string	`ungroupComponent`	Lista de servicios agrupados. Se utiliza para expresar el conjunto de servicios de un componente que ha sufrido una desagrupación.

	liststructure	`cotsget`	Lista de componentes del modelo concreto para ese usuario. Es una estructura `Interaction`, definida en la operación *Init User Architecture*.
Parámetros de salida			
`result`	Un *structure* `RegisterInteractionResult` con los valores de salida:		
	boolean	`registered`	*true* si tiene éxito la inserción en la BD de interacción, *false* si no.
	string	`message`	Mensaje de éxito o de error y su tipo.

Descripción: Esta operación no pretende guardar interacción que tiene lugar dentro de los componentes, sino almacenar interacción que tiene lugar en la aplicación en sí. Esta interacción puede estar relacionada con cambios de tamaño de los componentes, cambios de posición, eliminación o agregación de componentes dentro de la aplicación *mashup*, así como procesos de comunicación entre los componentes que constituyen la aplicación. Como se puede comprobar en la interfaz, mostrada anteriormente, la operación acepta en su entrada una estructura `RegisterInteractionParams` que contiene el identificador del usuario con la sesión iniciada y el tipo de acción de interacción realizada sobre la aplicación *mashup*; estos valores son obligatorios y además no nulos. Opcionalmente se puede enviar el resto de información de interacción y el estado del modelo (*i.e.*, el estado de la arquitectura de la aplicación *mashup*). La operación obtiene la hora de sistema e inserta la interacción en la base de datos de interacción. Devuelve como respuesta una estructura `RegisterInteractionResult` con una variable que indica el éxito o no de la operación, y un mensaje correspondiente a la operación realizada. En la siguiente tabla se muestra el listado de tipos de mensaje que la operación puede devolver.

Mensajes de error:

error	**descripción**
`Internal Server Error`	Se produce por fallos en el servidor al buscar alguna clase necesaria para la ejecución.
`Not found or Empty userid Error`	Se produce cuando se omite este parámetro en la llamada al servicio o no se proporciona un valor.
`Not found or Empty Operation Performed`	Se produce cuando se omite este parámetro en la llamada al servicio.
`Error in Register Interaction`	Se produce por problemas en la consulta a la base de datos de interacción.
`PSQLException`	Se produce por problemas en la conexión a la base de datos.

Comportamiento: En la Figura 3.21 se muestra un diagrama con el flujo de tareas que suceden al llamar la operación *Register Interaction*. Como se ha dicho, esta operación se utiliza cada vez que se pretende almacenar información sobre la interacción que se produce en la aplicación *mashup*. Como se puede observar, esta comunicación conlleva realizar el envío de interacción a través del servidor *JavaScript*. Dicho servidor se pone en contacto con el servicio *Register interaction* para informar sobre el suceso. La operación del servicio informa al módulo IMM (encargado de la gestión de la interacción procedente de las aplicaciones) sobre lo sucedido, enviándole la información necesaria

© 2016 Vallecillos, J.

para el registro de dicha interacción. Por último, el módulo informa al controlador *Manage Interaction* para guardar la interacción en la base de datos *Interaction*. Para los procesos de almacenamiento de información no sólo se registran tareas de comunicación entre componentes, si no también procesos de redimensión de componentes, cambios de posición de los componentes de la aplicación *mashup* junto a procesos de eliminación y agregación de componentes en la aplicación.

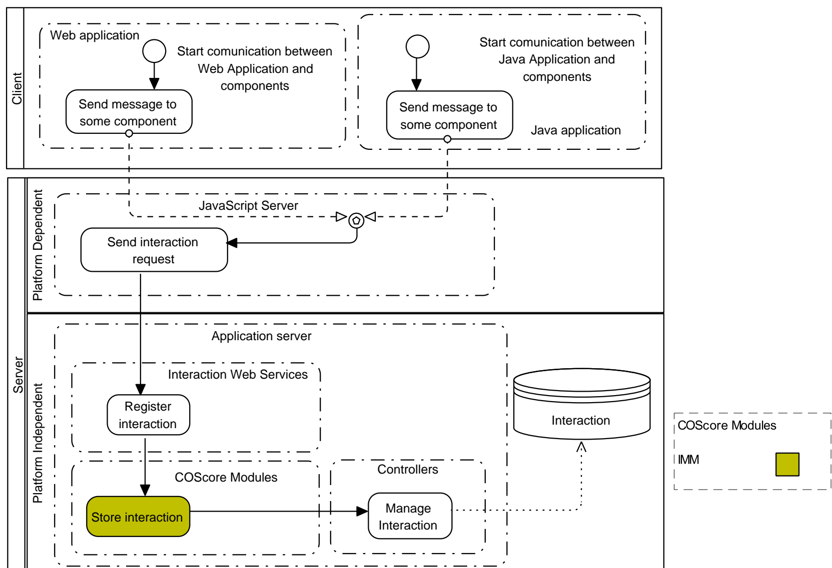


Figura 3.21: Operación *Register interaction* de *Interaction Service*

Implementación: Respecto a la implementación, en el siguiente listado se muestra parte del código de la operación *Register Interaction*. En primer lugar, se obtiene el módulo *IMM* perteneciente al usuario de la aplicación (líneas 6-7). Después se comprueba que los valores pertenecientes al parámetro de entrada son los correctos, primero verificando que el identificador del usuario es correcto, y luego si el parámetro *operationPerformed* del objeto `RegisterInteractionsParams`, reservado para definir el tipo de interacción que se desea almacenar, no es nulo o no está vacío. Si está libre de errores, se procede a almacenar la información de la interacción (línea 11). Aquí, se llama al método `registerinteraction` del módulo *IMM* al cual se le pasa la información necesaria para ser almacenada en la base de datos de interacción. Almacenar la información de la interacción implica guardar en la base de datos de interacción todas las variables existentes de la clase *RegisterInteractionParams*. Estas variables son: el identificador del usuario que produjo la interacción, el tipo de interacción que se produjo en la aplicación, y los identificadores del componente y del modelo (*i.e.*, arquitectura de aplicación *mashup*) sobre los cuales se produjeron la interacción. Si no se produce ningún error durante el proceso de registro, se crea el objeto *RegisterInteractionResult* que será devuelto a través del servicio.

```
1 public RegisterInteractionResult registerInteraction(
2       RegisterInteractionParams params){
3 RegisterInteractionResult result = new RegisterInteractionResult();
4 InterModulesData resultIMMforError = new InterModulesData();
5 try{ Context initialContext = new InitialContext();
6    IMM imm = (es.ual.acg.cos.modules.IMM)initialContext.
7                                                lookup("java:app/cos/IMM");
8    if (params.getUserId() != null && params.getUserId().compareTo("") != 0) {
9       if (params.getOperationPerformed() != null &&
10                        params.getOperationPerformed().compareTo("") != 0) {
11         resultIMMforError = imm.registerinteraction(...);
12               if(resultIMMforError.getValue().equalsIgnoreCase("-1")){
13            result.setRegistered(false);
14            result.setMessage(resultIMMforError.getMessage());
15         }else{
16            result.setRegistered(true);
17            result.setMessage(resultIMMforError.getMessage()); }
18      } else {
19         LOGGER.error("Not found or Empty Operation Performed");
20         result.setMessage("> Not found or Empty Operation Performed");
21         result.setRegistered(false); }
22          } else {
23        LOGGER.error("Not found or Empty userId Error");
24        result.setMessage("> Not found or Empty userId Error");
25        result.setRegistered(false); }
26       } catch (Exception e) {
27     LOGGER.error(e);
28     result.setRegistered(false);
29     result.setMessage("> Error in Register Interaction " + e); }
30   return result;
31 }
```

Ejemplo Petición XML:

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:ws="http://ws.cos.acg.ual.es/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:registerInteraction>
      <params>
        <userId> ejemplo de Id </userID>
        <newSession> ejemplo de valor de sesión o vacío </newSession>
        <interaction>
          <deviceType> ejemplo de dispositivo o vacío </deviceType>
          <interactionType> ejemplo de tipo de interacción o vacío </interactionType>
          <latitude> ejemplo de latitud o vacío </latitude>
          <longitude> ejemplo de longitud o vacío </longitude>
        </interaction>
        <operationPerformed> ejemplo de operación </operationPerformed>
        <componentId> ejemplo de identificador de componente o vacío </componentId>
        <groupComponent>
          ejemplo de identificador de servicios en el componente agrupado o vacío
        </groupComponent>
                 ...
        <ungroupComponent>
          ejemplo de identificador de servicios en el componente desagrupado o vacío
```

© 2016 Vallecillos, J.

```
        </ungroupComponent>
                  ...
        <cotsget> ... </cotsget>
                  ...
      </params>
    </ws:registerInteraction >
  </soapenv:Body>
</soapenv:Envelope>
```

Ejemplo Respuesta XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:registerInteractionResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <registered> ejemplo true o false </registered>
        <message> ejemplo mensaje de éxito o de error </message>
      </result>
    </ns2:registerInteractionResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo Error Response XML

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:registerInteractionResponse xmlns:ns2="http://ws.cos.acg.ual.es/">
      <result>
        <registered> false </registered>
        <message> > Not found o Empty Operation Performed </message>
      </result>
    </ns2:registerInteractionResponse>
  </soap:Body>
</soap:Envelope>
```

3.5. Trabajo relacionado

Este capítulo se centra en el modelo de servicios y operaciones que ha sido desarrollado para la implementación de la infraestructura propuesta en el presente trabajo de tesis doctoral, que tiene el objetivo principal de permitir el despliegue y gestión de aplicaciones *mashup*. Por esta razón, este apartado revisa otras infraestructuras parecidas construidas para gestionar este tipo de aplicaciones. Un ejemplo es el proyecto OMELETTE [Chudnovskyy et al., 2012], que está basado en las tecnologías de las aplicaciones *mashup* para permitir a los usuarios crear sus propias plataformas de colaboración. Esto se consigue proporcionando un conjunto de herramientas y componentes (basados en *widgets* de W3C) que soportan el desarrollo de *mashup* Telco [Gebhardt et al., 2012] (aplicaciones *mashup* que permite la comunicación a través de diferentes canales). También hacen uso de modelos para gestionar los espacios de trabajo de los usuarios. Algunas diferencias que se pueden encontrar con respecto al trabajo de tesis doctoral es que la

infraestructura propuesta se centra en aplicaciones individuales y no se comparten escritorios para realizar tareas colaborativas, como ocurre en OMELETTE. Por otro lado, en OMELETTE, las comunicaciones entre componentes se realizan a través de Apache Rave, lo cual limita que puedan existir procesos de comunicación entre otros tipos de componentes que no sean *widgets*. En la infraestructura desarrollada a lo largo de esta tesis doctoral, se usa un servidor *JavaScript* para interconectar entre sí componentes de diferentes tipos a través de *WebSockets*. Otros proyectos similares a OMELETTE son DashMash [Cappiello et al., 2011] y ServFace [Nestler et al., 2010].

En [Cappiello et al., 2015] se propone un *framework* para que los usuarios puedan construir interfaces *mashup* basadas en componentes (de tipo *widget*) según sus necesidades. También hacen uso, como en nuestro caso, de MDE para la representación del entorno, aunque sólo se centran en dar soporte a plataformas de tipo web. En este trabajo de tesis doctoral en cambio se da soporte a múltiples plataformas.

En [Aghaee et al., 2013] han creado un entorno, llamado *NaturalMash*, que permite la construcción de interfaces de usuario *mashup* a través de componentes de tipo *widgets* seleccionados de una paleta de componentes. Estos componentes pueden ser arrastrados y colocados en el espacio de trabajo para que el usuario pueda diseñar su propio entorno. El sistema incluye una forma de seleccionar los componentes, implementada mediante lenguaje natural. Pero su propuesta tiene algunas limitaciones como la posibilidad de comunicar entre sí los componentes que forman el entorno, lo cual limita la interoperatividad entre dichos componentes.

Con respecto al uso de servicios web para manejar interfaces de usuario *mashup*, hay trabajos como [Ardito et al., 2014] donde se utilizan servicios web para proporcionar a las aplicaciones la oportunidad de compartir espacios de trabajo construidos a partir de interfaces de usuario *mashup* sobre diferentes dispositivos. Para realizar este proceso de compartición de espacios de trabajo se utilizan diferentes tipos de modelos. Así, aparece un modelo para componer los patrones de la interfaz de usuario, definiendo los componentes que deben de formar el espacio de trabajo, otro modelo para describir el estado actual de la interfaz de usuario y un modelo de plantilla visual que integra datos representativos sobre elementos gráficos. Estos modelos son proporcionados a través de servicios web que utilizan las aplicaciones *mashup*. A diferencia de la infraestructura desarrollada en el trabajo de tesis doctoral, en este trabajo, los autores no dividen los servicios como públicos y privados para controlar el entorno, ni tampoco se permite que dichos servicios puedan manejar otros elementos diferentes a los modelos, como son los procesos de interacción o gestión de usuarios.

Existen otros trabajos como [Hoyer et al., 2009] centrados en el uso de componentes e interfaces de usuario *mashup* para la construcción de aplicaciones adecuadas a las necesidades del usuario. Para ello, disponen de una colección de componentes a partir de la cual los usuarios crean sus propios entornos. Asimismo, para realizar los procesos de adaptación del entorno de trabajo de los usuarios y manejar la comunicación entre los *widgets*, hacen uso de arquitecturas SOA. Por tanto, al igual que la infraestructura construida en este trabajo de tesis doctoral, se hace uso de servicios para controlar los procesos de gestión de las interfaces de usuario y la comunicación entre los componentes. Sin embargo, su propuesta no está centrada en manejar otros entornos que no sean de tipo web, lo que supone una limitación para ampliar su utilidad a otras aplicaciones.

© 2016 Vallecillos, J.

3.6. Resumen y conclusiones

En este capítulo se ha descrito de forma detallada la capa independiente de la infraestructura encargada de dar soporte a las aplicaciones *mashup*. Para ello, se ha realizado un desglose de todos los elementos por los cuales está constituido el modelo de servicios y operaciones que da soporte a esta parte de la infraestructura.

En primer lugar se ha realizado una introducción del capítulo centrada en la descripción de la estructura general del modelo de servicios y operaciones que forma parte de la capa independiente del COScore de infraestructura. Posteriormente, se han detallado todos los niveles por los que está formada dicha capa. Para ello, se han descrito las partes principales en las que se sustentan los servicios y operaciones desarrollados, *i.e.*, las bases de datos y los controladores. Existe una base de datos para los modelos de arquitectura de las aplicaciones *mashup* (que también contiene la información relativa a los usuarios), una base de datos para la interacción que se produce en las aplicaciones, una base de datos que maneja las especificaciones de los componentes concretos por los que se forman las aplicaciones *mashup*, y cuatro bases de datos adicionales para gestionar los componentes de tipo web (*widgets*) y los componentes de tipo Java.

Las bases de datos del modelo propuesto son manejadas por los controladores y, por lo tanto, se ha descrito cuál es la funcionalidad de cada uno de ellos. Por último, en el nivel más próximo a los servicios se encuentran los módulos, que dotan de toda la funcionalidad necesaria a las operaciones de los servicios web. Es decir, contienen y se encargan de ejecutar la lógica de negocio de la capa independiente de la plataforma. Además, cada módulo agrupa cierto tipo de funcionalidad de la infraestructura, como por ejemplo, el módulo de usuarios, que implementa toda la funcionalidad relacionada con la gestión de los usuarios que interactúan con las aplicaciones *mashup* desplegadas por la infraestructura.

Como parte principal del capítulo, se ha descrito cada una de las operaciones que forman parte de los servicios de la infraestructura desarrollada. Se han establecido dos tipos de acceso para dichos servicios, uno privado y otro público. Por un lado, los servicios privados se utilizan de forma interna (por los desarrolladores y por la propia infraestructura) para la gestión de los datos principales que dan soporte a las aplicaciones *mashup* (arquitecturas, componentes y usuarios). Por otro lado, los servicios públicos son aquellos que son accesibles por las aplicaciones *mashup* y se utilizan para darles soporte, incluyendo la gestión de las sesiones de los usuarios, la comunicación entre componentes, la actualización y reconfiguración de la arquitectura y el registro de la interacción. Para cada servicio se ha mostrado cuáles son las operaciones por las cuales está constituido y, para cada operación se muestra la siguiente información: una introducción breve de la operación, una definición de las signaturas de su interfaz, el listado de los parámetros de entrada y salida, una descripción de la operación, la lista de mensajes de error permitida, una explicación de su comportamiento usando un diagrama de flujo de información, una descripción de la implementación junto con un fragmento del código más significativo, un ejemplo de invocación de la operación con su correspondiente devolución, y para finalizar, un ejemplo de error.

Capítulo 4

Escenario y experimentación

© 2016 Vallecillos, J.

Capítulo 4

Escenario y experimentación

Contenidos

4.1. Introducción y conceptos relacionados . . . 180
4.2. Interfaz *mashup* ENIA . . . 182
4.2.1. Componente COTSget . . . 183
4.2.2. Escritorio de ENIA . . . 184
4.2.3. Menú de ENIA . . . 186
4.2.4. Panel de componentes de ENIA . . . 187
4.2.5. Acciones sobre la interfaz ENIA . . . 188
4.3. Escenarios de prueba . . . 190
4.3.1. Escenario 1: Usuario anónimo . . . 190
4.3.2. Escenario 2: Registro de usuario . . . 192
4.3.3. Escenario 3: Usuario registrado . . . 194
4.3.4. Escenario 4: Reconfiguración de la interfaz . . . 197
4.3.5. Escenario 5: Cierre de sesión . . . 199
4.4. Experimentación y pruebas . . . 200
4.5. Resumen y conclusiones . . . 206

Una vez presentado en los capítulos anteriores el modelo de infraestructura de COScore que da soporte a las aplicaciones *mashup*, en este capítulo describiremos algunas pruebas de experimentación realizadas, para la validación de la propuesta desarrollada en este trabajo de investigación. Para ello, se ha implementado un prototipo de aplicación *mashup* llamado ENIA, un tipo de interfaz gráfica de usuario Web basada en componentes *widgets* del W3C, desarrollado para la REDIAM (Red de Información Ambiental de Andalucía) de la Junta de Andalucía. Dicho prototipo ha sido elaborado y usado en este trabajo de tesis doctoral para justificar, por un lado, la validez de un modelo *mashup* para interfaces gráficas de usuario Web, y por otro, la validez del modelo de infraestructura COScore aquí propuesto.

Como se ha visto en los anteriores capítulos, para la implementación del modelo de infraestructura COScore se ha seguido un paradigma de desarrollo basado en componentes y servicios, con una propuesta de arquitectura de despliegue en capas, donde se ubican las bases de datos, controladores, módulos y servicios (públicos y privados). Asimismo, la implementación ha estado guiada por pruebas TDD (*Test-Driven Development*) donde, para testar cada una de las operaciones de los servicios públicos de COScore, se ha elaborado un *juego de pruebas* y una herramienta de pruebas en línea[1]. En el Anexo A se incluye parte del juego de pruebas realizado para los servicios públicos, dado que son aquellos que pueden ser accesibles por terceros. También se han llevado a cabo unas pruebas de estrés para medir el rendimiento de los servicios en tiempo de ejecución. Para ello, se ha desarrollado otro juego de pruebas diferente donde se miden tasas de rendimiento principalmente en tres parámetros: (a) el tamaño de la aplicación *mashup* que se despliega en el cliente, (b) el grado de acoplamiento de la arquitectura de la aplicación (*i.e.*, número de conexiones entre los componentes) y (c) el número de accesos concurrentes que se producen por parte de los usuarios.

El presente capítulo consta de cinco secciones principales, y queda estructurado de la siguiente forma. La Sección 4.1 contiene una breve descripción del dominio de aplicación de la investigación y una justificación del prototipo desarrollado en el ámbito de esta tesis doctoral. La Sección 4.2 presenta una descripción de la interfaz gráfica *mashup* ENIA (un sistema de información medioambiental basado en interfaces de componentes *mashup*), que ha servido como marco experimental para realizar las pruebas de validación y evaluación. Seguidamente, la Sección 4.3 describe varios escenarios de prueba realizados sobre ENIA, y donde se detallan la secuencia de tareas que suceden en la infraestructura COScore para llevar a cabo el despliegue de la aplicación *mashup* ante diferentes situaciones de partida. A continuación, en la Sección 4.4 se explican cuáles han sido los juegos de prueba realizados para medir el rendimiento del sistema y se analizan algunas consideraciones de los resultados obtenidos como consecuencia de la experimentación. El capítulo finaliza con un resumen y conclusiones generales en la Sección 4.5.

[1]COScore API – `http://acg.ual.es/projects/enia/ui/webservices/`

Como aclaración, el juego de herramientas de prueba disponibles en linea en el sitio web indicado arriba, ha sido implementado de forma aislada para cada operación de servicio público, con el propósito de poder testar cada una de ellas por separado.

© 2016 Vallecillos, J.

4.1. Introducción y conceptos relacionados

En el presente capítulo se describe el funcionamiento de la interfaz *mashup* ENIA. Como se ha venido comentando hasta el momento, ENIA (cuyas siglas proceden de su término en inglés *ENvironmental Information Agent*) es un prototipo de sistema de gestión medioambiental que implementa el modelo de aplicación *mashup* definido por el Grupo de Investigación de Informática Aplicada (TIC-211) de la Universidad de Almería, en el marco de un proyecto de excelencia de la Junta de Andalucía P10-TIC-6114[2] titulado "Desarrollo de un agente web inteligente de información medioambiental" financiado por la Consejería de Economía, Innovación, Ciencia y Empleo.

Dicho modelo de aplicación *mashup* permite y facilita que la arquitectura interna de componentes que subyace detrás de la misma (en este caso de la interfaz gráfica de usuario) pueda cambiar en el tiempo por sí misma. Esto se produce gracias a que las partes que conforman la aplicación se encuentran débilmente acopladas entre sí, permitiendo con esto, por un lado, que los usuarios del sistema puedan configurar la apariencia de sus interfaces, diseñándolas según sus preferencias a partir de un catálogo de componentes, y pudiendo realizar un determinado conjunto de acciones sobre ellos, como añadir nuevos componentes a la apariencia, quitar, mover, redimensionar, entre otras posibles acciones que el sistema permita. Pero, por otro lado, las "bondades" o ventajas de una aplicación *mashup* van algo más allá del mero hecho de que estas puedan ser modificadas por los propios usuarios *ad-hoc* adaptándolas a su gusto en el mismo instante; Por la propia naturaleza del modelo *mashup* (*i.e.*, modular, escalable, interoperable, adaptable, etc.), contar con aplicaciones basadas en *mashup* facilitan el desarrollo (y propician la existencia) de infraestructuras dotadas y soportadas por mecanismos proactivos inteligentes, que provoquen los cambios de la arquitectura subyacente a la aplicación, para adaptarla a las necesidades del usuario que la consume, sin la intervención directa del propio usuario en dichos cambios, o que el usuario sea consciente de ello. Por lo que el sistema aprende de los comportamientos (del uso cotidiano de la aplicación por parte del usuario) o al menos, el sistema es capaz de reaccionar ante las acciones que éste realiza sobre la aplicación, siendo consideradas por el sistema como estímulos o eventos que activan ciertas reglas que manejan la apariencia de la arquitectura, sobre la cual se basa la aplicación, y sobre la cual el usuario interactúa.

Es por ello, que ha sido necesario estudiar y desarrollar un modelo de infraestructura que proporcione el soporte a toda la gestión que involucra el mantenimiento de aplicaciones *mashup* en la red. Y como se ha venido comentando, en el presente trabajo de investigación de tesis doctoral que culmina una parte de la investigación realizada en el marco del citado proyecto de excelencia P10-TIC-6114, se ha implementado un prototipo de interfaz gráfica de usuario que respeta el modelo de aplicación *mashup* definido por el citado grupo de investigación, y que se presenta como resultado y aporte de la presente tesis doctoral. Como veremos en la siguiente sección, la interfaz ENIA consta de un repertorio de componentes tipo, denominados *COTSgets* que son una clase de componente COTS implementada mediante *widgets*. Como se ha visto en el capítulo anterior, estos componentes *widgets* son desplegados en el cliente mediante código incrus-

[2]Portal web del Proyecto ENIA – `http://acg.ual.es/enia/`

tado en HTML con etiquetas `<iframe>`. En el lado servidor, la infraestructura mantiene una implementación real (concreta) de dichos componentes, alojados en un repositorio o base de datos de componentes *concretos*. También mantiene el estado actual de la arquitectura de componentes asociada a la interfaz de usuario, esto es, el conjunto de *widgets* mostrados en la interfaz del usuario.

Por otro lado, para poder ofrecer una infraestructura de servicios capaz de dar soporte a las aplicaciones *mashup* como la de ENIA, ha sido necesario contar con un repertorio de operaciones accesibles a través de estos servicios, encargados de implementar la funcionalidad específica de cada servicio (tanto privado como público). Como se ha visto en los capítulos anteriores, las operaciones de servicio se comunican directamente con los módulos, implementados como componentes que proporcionan la funcionalidad de la infraestructura. Además, en caso de que un módulo necesite manejar alguna información a la cual no tiene acceso, este se comunicará con el controlador encargado de tratar las bases de datos del entorno. El hecho de haber realizado un desarrollo basado en servicios implica que se puede hacer uso de la infraestructura desde cualquier punto y en diferentes situaciones durante la ejecución de la aplicación *mashup*. Algunas de estas operaciones pueden implicar acciones de complejidad media o alta, mientras que otras pueden consistir en acciones sencillas, con una complejidad pequeña. Ejemplos de operaciones del primer tipo son la invocación de la operación para generar una aplicación *mashup* durante el proceso de inicio de una sesión de usuario, o la modificación de algún componente perteneciente a una aplicación *mashup*, entre otros ejemplos, las cuales involucran la participación de varios módulos y accesos a distintas bases de datos de la infraestructura. Para el caso de operaciones que implican una complejidad más sencilla está la operación de dar de alta un modelo de arquitectura concreta, o eliminar un componente concreto del repositorio de componentes concretos del entorno. En este capítulo se describen cinco escenarios base que ponen de manifiesto, a partir del caso estudio de la interfaz ENIA, el funcionamiento interno de algunas de las operaciones de servicio más relevantes, y la secuencia de tareas que sucede detrás de la aplicación ENIA, desde el cliente hacia el servidor y en ambos sentidos.

Un tema relacionado con la propuesta del modelo de infraestructura para el despliegue de aplicaciones *mashup* aquí realizado, es su portabilidad a diferentes plataformas. En este sentido, debido al tipo de solución tecnológica propuesto para el desarrollo de la infraestructura (*i.e.*, una solución basada en el manejo de modelos y metamodelos que mantienen una visión abstracta de las arquitecturas de componentes asociadas a las aplicaciones *mashup*, y con un desacople funcional en su implementación, centrada en componentes y servicios web públicos accesibles), el modelo de infraestructura propuesto para *mashups* está preparado para que, en el caso de las interfaces gráficas de usuario, pueda ser adaptado con facilidad para poder funcionar en diferentes dispositivos y plataformas. En este sentido, por ejemplo, son posibles adaptaciones del modelo para entornos de interfaces para TDT (televisión digital) y dispositivos móviles. Asimismo, la propuesta está preparada para manejar otros entornos o sistemas que tengan claramente una arquitectura de componentes subyacente, como por ejemplo, sistemas de hogar digital, domótica inmersiva, o para escenarios *Smart-Cities*, donde los controladores, actuadores o sensores (dependiendo del escenario) conforman una red de componentes (arquitectura) que el modelo de infraestructura propuesto podría manejar.

© 2016 Vallecillos, J.

4.2. Interfaz *mashup* ENIA

En la presente sección se describe la estructura y funcionalidad de la interfaz *mashup* ENIA. Como se ha comentado anteriormente, ENIA es el resultado de la investigación que ha venido desarrollando el grupo de investigación de Informática Aplicada TIC-211 de la Universidad de Almería en el marco de un proyecto de excelencia de la Junta de Andalucía referencia P10-TIC-6114.

En el transcurso de la investigación se ha contado con la colaboración de miembros de la línea de SIG y desarrollo de la REDIAM de la Subdirección de Tecnologías de la Información. La REDIAM[3] (Red de Información Ambiental de Andalucía) es una estructura organizativa dependiente de la Consejería de Medio Ambiente y Ordenación del Territorio, que tiene como propósito la integración, normalización y difusión de información medioambiental de Andalucía.

La colaboración con el citado equipo de REDIAM ha consistido en su participación en el estudio, clasificación y catalogación de cierta información medioambiental disponible en el catálogo de servicios OGC[4] de la REDIAM[5]. Como resultado, se ha seleccionado un subconjunto de los servicios OGC ofrecidos por la REDIAM desde su catálogo público de servicios, y a partir de estos se han confeccionado los repositorios de componentes OGC propios incluidos en la infraestructura COScore. Asimismo, el equipo de trabajo de la REDIAM también ha colaborado en recomendaciones de diseño y de usabilidad para algunas partes de la interfaz *mashup* de ENIA.

ENIA está pensado como un caso estudio procedente de la investigación desarrollada en el marco del citado proyecto, siendo ENIA un prototipo constituido y construido a partir del modelo de aplicaciones *mashup* propuesto, y aplicado en este caso para interfaces gráficas de usuario que soportan componentes, cuya funcionalidad está relacionada con la gestión de información ambiental. Veamos a continuación, en el siguiente apartado, cómo se estructura la interfaz de ENIA, y luego, algunas de las acciones más relevantes que se pueden hacer sobre ella, al final de esta sección.

La interfaz de ENIA básicamente consta de tres zonas visibles, sobre las cuales el usuario puede interactuar. Como se puede ver en la Figura 4.1, que muestra una vista *Wireframe* de una interfaz ENIA, estas zonas son: (a) una barra de *Menú* general: ubicada en la parte superior de la vista; (b) un *Escritorio*: que ocupa la mayor parte de la zona de trabajo, y donde se ubican los componentes (denominados en la propuesta como *COTSgets*) sobre los que el usuario puede trabajar; y (c) un *Panel* de servicios: situado en la parte izquierda de la vista, desde donde se ofrece al usuario una lista de servicios OGC y *Apps* que este pueda consumir (manejar) en el Escritorio. Un poco más adelante veremos algunas características para cada una de estas zonas principales de la interfaz; pero antes veremos algunos de los aspectos de un componente COTSget.

[3]REDIAM – `http://www.juntadeandalucia.es/medioambiente/site/rediam`

[4]OGC (*Open Geospatial Consortium*) es una organización internacional creada en 1994 que trabaja en la generación de estándares abiertos e interoperables para Sistemas de Información Geográfica y en la Web (`http://www.opengeospatial.org`). La REDIAM desarrolla todos sus servicios en base a estándares de la OGC, y respeta los estándares de datos abiertos en la red, dejando disponible de forma pública estos servicios, los cuales son accesibles en línea a través de un enlace.

[5]Catálogo de servicios OGC en línea de la REDIAM:
`http://www.juntadeandalucia.es/medioambiente/geoinspire/servicios/srv/es/main.home`

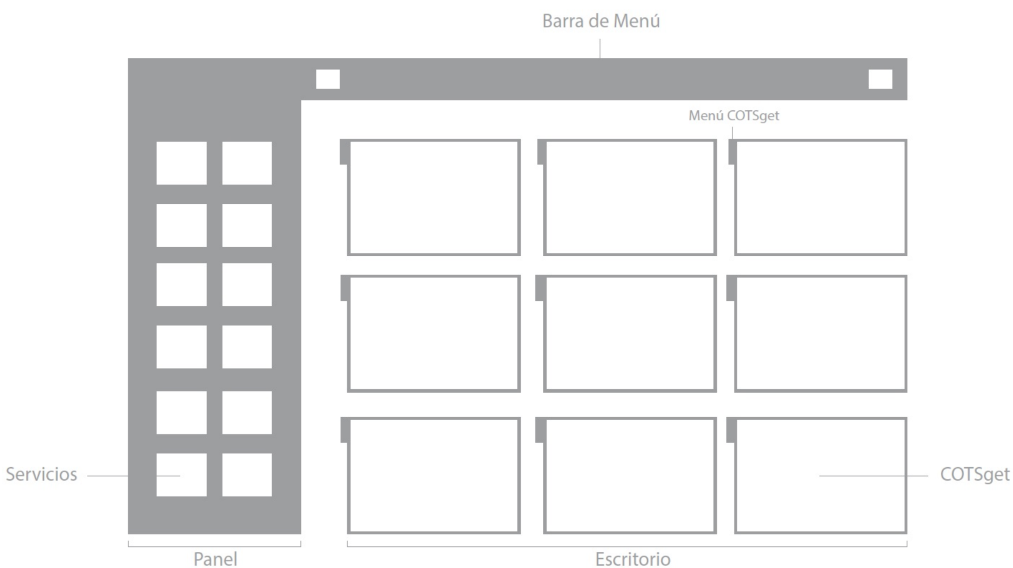


Figura 4.1: Vista *Wireframe* de la interfaz *mashup* ENIA

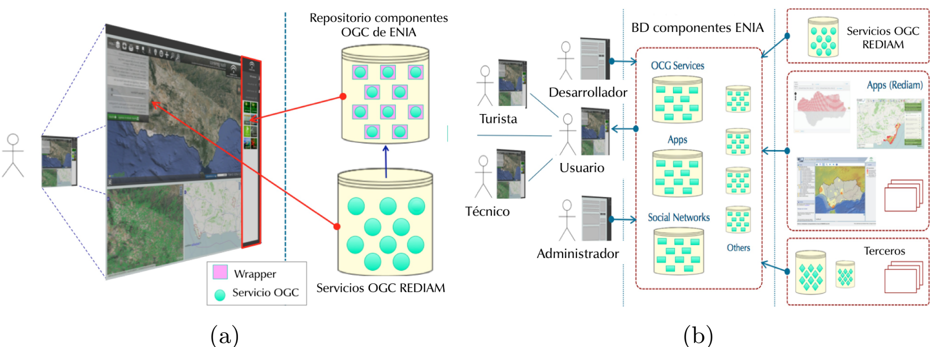
4.2.1. Componente COTSget

El elemento principal en la propuesta es el *COTSget*[6], el cual (desde el punto de vista visual) es como un visor que encapsula la funcionalidad de un componente externo, que en nuestro caso (en el caso de ENIA) es un servicio OGC o una *App*. Como se observa en la Figura 4.2a-(a) existen repositorios externos pertenecientes a desarrolladores de terceros que dejan sus implementaciones particulares para esos componentes en sus repositorios, normalmente accesibles de forma pública. En el caso de ENIA, los componentes externos son los servicios OGC implementados por el equipo de desarrollo de REDIAM. Estos servicios OCG representan mapas temáticos de información medioambiental.

Cada servicio OCG externo es elaborado a partir de un mapa base de OpenStreetMap sobre el cual se superpone una capa temática relativa a información espacial. Así, para el caso de ENIA, se han tenido en cuenta sólo las capas temáticas OCG relacionadas con "usos del suelo". Para adoptar dichos servicios OGC en el sistema, se crea código envoltorio *wrapper* para cada servicio externo importado, creando así un nuevo componente adaptado que se almacena en repositorios propios de la estructura. En nuestro caso ENIA, dichos componentes *wrapper* han sido desarrollados en Wookie, como componentes *Widgets*. Cuando un componente *widget* (que embebe al servicio) es desplegado en el cliente (en el Escritorio de la interfaz de usuario) pasa a denominarse componente COTSget. Esta misma forma de adoptar servicios (*i.e.*, integración con Wookie/*widgets*)

[6]Como se ha comentado en más de una ocasión a lo largo de este documento de tesis, el término COTSget procede del acrónimo de COTS (*Commercial Off-The-Shelf*, el cual se refiere a componentes desarrollados por terceras partes) y *Widget* del W3C, dado que la implementación realizada para el modelo de aplicación *mashup* de esta tesis doctoral está basada en el manejo de componentes de caja negra usando *Widgets* para ser incrustados en la interfaz.

© 2016 Vallecillos, J.



(a) (b)

Figura 4.2: Estructura de la organización de las fuentes de ENIA

también sucede para las *Apps* externas. En el caso de ENIA, se han incluido *Apps* embebidas para algunas de las aplicaciones desarrolladas por la REDIAM (como veremos más adelante), todas ellas disponibles y accesibles por la red a través de protocolo HTTP. Por tanto, como se observa en la Figura 4.2b-(b), la cual muestra la estructura organizativa de las fuentes de datos de ENIA, la interfaz se nutre de componentes procedentes de repositorios externos de REDIAM que son importados a repositorios propios internos del sistema. Además, como veremos, en ENIA también existen diferentes perfiles de usuario para el acceso al sistema, como por ejemplo, perfiles de Agricultor, Turista, o Técnico, entre otros. Un usuario se puede registrar en el sistema seleccionando uno de estos perfiles, y una vez dentro, cada uno de ellos puede acceder a un conjunto de componentes particular para cada perfil de usuario, accesible y ofrecido desde el panel de ENIA (parte izquierda de la interfaz).

4.2.2. Escritorio de ENIA

En la Figura 4.3 se muestra una vista ejemplo concreta de la interfaz ENIA que usaremos como guía para explicar algunas características de la interfaz. La zona más importante de la interfaz de ENIA es el **Escritorio**[7]. En el caso de la interfaz de la figura ejemplo, en el Escritorio se puede observar que hay seis componentes COTSgets, que alojan cuatro servicios OGC y dos *Apps*. De la figura se desprende que el usuario está visualizando una misma vista geográfica sobre cuatro visores COTSgets. Concretamente se trata de un mapa del sureste de España (costa de Almería[8]) sobre el que el usuario está observando cuatro capas temáticas de usos de suelo, correspondientes (de izquierda a derecha, y de arriba a abajo) a las capas de espacios naturales, líneas base, montes públicos, y vías pecuarias. Además de estos cuatro visores de mapas, en el Escritorio hay otros dos componentes COTSgets de tipo *App* situados a la derecha de la vista de la interfaz, que se corresponden con una aplicación de Twitter (conectado en el ejemplo a la cuenta de la REDIAM) y una aplicación del tiempo externa.

[7]En partes de este documento, al Escritorio se le ha denominado Espacio de Trabajo (W) o Rejilla
[8]En el mapa se observa la bahía de Almería, con la ciudad de Almería en el centro; El Cabo de Gata a la derecha; y a la izquierda se pueden ver los invernaderos de El Ejido.

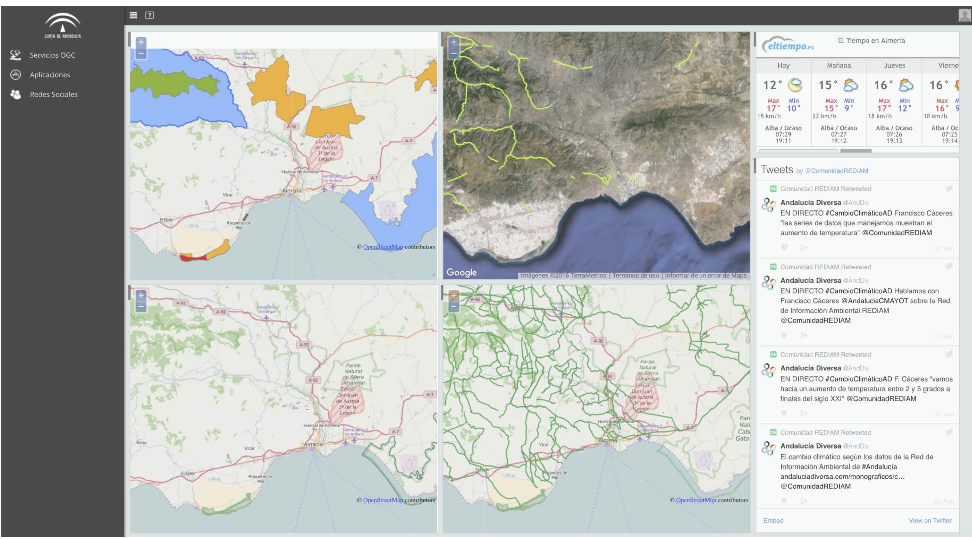


Figura 4.3: Una vista de la interfaz *mashup* ENIA

El Escritorio realmente representa lo que hemos venido llamando a lo largo de este documento de tesis doctoral como "arquitectura de componentes" de una aplicación *mashup*. Por tanto, en nuestro caso, una aplicación *mashup*, trasladado al caso de una interfaz gráfica Web como la de ENIA, sería el conjunto de componentes COTSgets, residentes en el Escritorio y sobre los que el usuario puede interactuar. Por tanto, dicho conjunto de componentes COTSgets del Escritorio es lo que se ha denominando como "interfaz gráfica de usuario *mashup*", y que internamente representa una *arquitectura de componentes* que el sistema gestiona, pudiendo existir relaciones (dependencias) entre ellos, de tal forma que, para algunos casos, la interacción del usuario sobre uno de ellos podría tener consecuencias sobre otro u otros componentes del Escritorio (*i.e.*, de la arquitectura de componentes subyacente).

Además, el Escritorio es manejado en el lado cliente como una *rejilla* (oculta) donde se pueden situar los COTSgets, identificados por una posición (x,y) dentro de esta. La rejilla ayuda internamente al sistema a desplegar la arquitectura de componentes de una interfaz ENIA de un usuario identificado cuando este inicia una nueva sesión en el sistema, recuperando su estado desde la base de datos (*i.e.*, la estructura de componentes, y las dependencias entre ellos, si los hubiese), y regenerando y situando los componentes COTSgets en la misma posición de rejilla para cada uno de ellos dentro del Escritorio. En la propuesta, en la mayoría de los casos, no se almacena el estado interno de los componentes externos (como los procedentes de los servicios OGC) ya que, como se ha comentado, estos son componentes de terceros y de caja negra a los que no se tiene acceso a su funcionalidad. Sólo en los casos en los que los desarrolladores de los componentes hayan dejado formas explícitas de hacer introspección sobre sus componentes, entonces se podrá llevar control del estado interno de un componente en ENIA.

© 2016 Vallecillos, J.

4.2.3. Menú de ENIA

En el caso del *Menú* general, que (como hemos dicho) se encuentra situado en la parte superior, en su versión actual, está diseñado como una barra de herramientas con tres opciones: (a) un botón para mostrar u ocultar el panel de componentes en la parte izquierda; (b) un botón con la ayuda rápida para el manejo de la interfaz[9]; y (c) un botón de perfil de usuario. En este último caso, inicialmente, cuando se accede a la aplicación ENIA, se hace con un perfil de "Usuario Anónimo". La operación ligada a este botón de perfil tiene una funcionalidad ligada parecida a como lo hace cualquier sistema de identificación en la Web, la cual permite el inicio de sesión en el sistema para un usuario registrado, o bien su registro, en el caso de que el usuario no lo esté aún. Se contempla también una operación para el recordatorio de claves, con validación por e-mail. En la Figura 4.4a se muestra el panel de inicio y registro en el sistema. En la Figura 4.4b se muestra los tipos de perfiles de usuario permitidos en ENIA. Para finalizar con la descripción de la zona del Menú, hay que mencionar que, como hemos visto, sólo contiene tres operaciones básicas, pero éste está preparado para poder albergar nuevas funciones en posibles ampliaciones del sistema.

(a) Principal

(b) Perfiles de usuario

Figura 4.4: Panel de inicio/registro

[9]En el Anexo B del presente documento se ofrece la guía rápida de ENIA la cual se muestra desde la opción de ayuda del menú.

4.2.4. Panel de componentes de ENIA

Respecto a la tercera parte destacable de la interfaz ENIA, el **Panel** contiene la colección de componentes que el usuario puede usar en el Escritorio. En la Figura 4.5 se muestra un desglose de todos los paneles que hay en ENIA. El panel principal (que se muestra en la parte superior de la figura) se ha organizado en tres catálogos, para albergar la lista de servicios OGC, la colección de *Apps*, y varios servicios aglutinados como redes sociales. En cualquier caso, la estructura del panel no es dependiente de la capa cliente, pudiendo ser reorganizada en un futuro con cierta facilidad. La estructura que presenta actualmente dicho panel ha sido la consensuada con los miembros del equipo de REDIAM. Por otro lado la apariencia (que también ha sido consensuada) está basada en el uso de paneles plegables, que permiten mostrar con cierta facilidad las colecciones de componentes, siguiendo una estructura organizativa. Desde este panel, un usuario puede buscar, seleccionar y arrastrar al Escritorio aquellos componentes con los que desea trabajar. Veamos a continuación las acciones más relevantes que se pueden hacer sobre los componentes de las zonas de Panel y Escritorio de la interfaz ENIA.

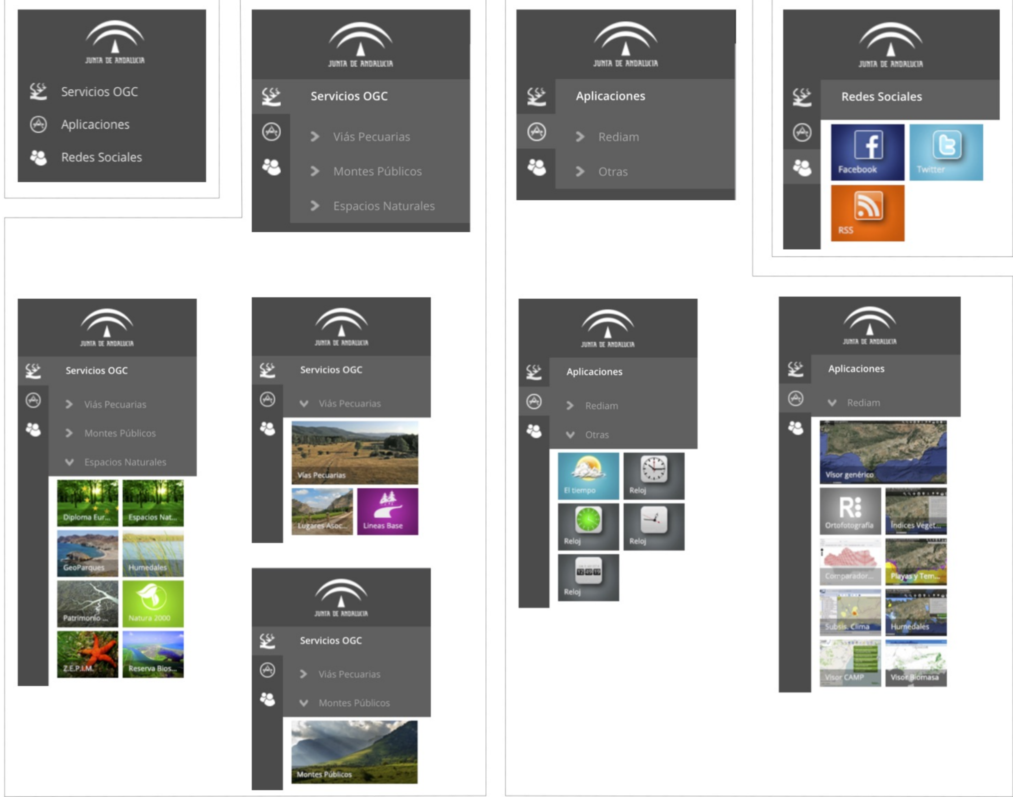


Figura 4.5: Paneles de menú de componentes de ENIA

© 2016 Vallecillos, J.

4.2.5. Acciones sobre la interfaz ENIA

Sobre la interfaz ENIA se pueden llevar a cabo diferentes acciones resumidas en la Tabla 4.1, la mayoría de ellas realizadas sobre el Escritorio[10]. Las acciones relativas al menú principal ya se han comentado, y que se corresponden con las de inicio o registro de un usuario en el sistema, y las acciones para mostrar y ocultar el panel de servicios (que se ofrece a la izquierda) y mostrar ayuda, acciones #9, #12 y #5 respectivamente.

Núm.	**Acción**	**Zona**	**OGC**	**App**
#1	Agrupar componentes OGC desde Escritorio	Escritorio	Si	No
#2	Agrupar componentes OGC desde Panel	Panel	Si	No
#3	Agrupar componentes OGC desagrupando	Escritorio	Si	No
#4	Añadir componentes al escritorio	Panel	Si	Si
#5	Ayuda	Menú	-	-
#6	Borrar componente via papelera	Escritorio	Si	Si
#7	Borrar componente via menú	COTSget	Si	Si
#8	Desagrupar componentes OGC	Escritorio	Si	No
#9	Iniciar sesión/registro	Menú	-	-
#10	Maximizar componente	Escritorio	Si	Si
#11	Restaurar componente	Escritorio	Si	Si
#12	Mostrar/ocultar panel	Menú	-	-
#13	Mostrar/ocultar menú	COTSget	Si	Si
#14	Mover componente	Escritorio	Si	Si
#15	Redimensionar componente	Escritorio	Si	Si

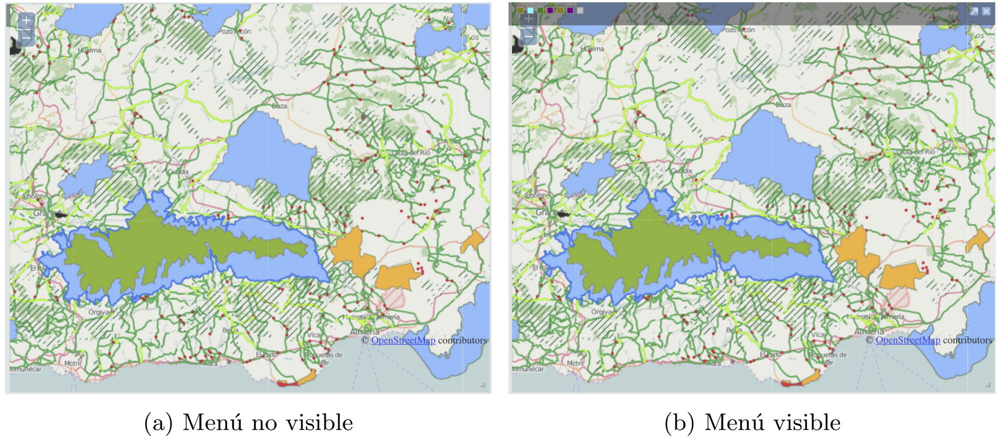
Tabla 4.1: Acciones que se pueden realizar sobre ENIA

Como se ha visto, el Escritorio de la interfaz ENIA contiene el conjunto de componentes que conforma la arquitectura de la interfaz manejada internamente por el sistema. Dichos componentes aparecen ubicados en el Escritorio, bien porque el propio sistema los carga ahí, tras recuperar una sesión anterior del usuario al iniciarse en el sistema, o bien porque el usuario los ha ido incluyendo (añadiendo) en el Escritorio, seleccionándolos desde el Panel, acción #4. Los usuarios pueden mover componentes (ya sean App como OGC) desde Panel al Escritorio, y con la ayuda de orientación de la rejilla se busca la mejor posición donde dejar el componente que desea añadir.

Una vez que los componentes están en el Escritorio[11], sobre ellos se pueden realizar diversas acciones. La acción más común es la de mostrar u ocultar su menú (acción #13), situado en su parte superior de un COTSget. En la Figura 4.6, a la izquierda, se observa un componente COTSget de tipo OGC con el menú cerrado, y a la derecha, el mismo componente, con su menú abierto. Desde ese menú se puede eliminar (borrar) el componente del Escritorio, usando el icono de aspa (acción #7) ubicado en la parte superior derecha del menú de componente. También se puede eliminar un componente del Escritorio arrastrándolo hacia una papelera que aparece en la parte superior del Escritorio en el momento de realizar la acción de borrado (acción #6).

[10]En el Anexo B del presente documento se muestra el resumen de todas las acciones, donde se ilustra de forma gráfica el estilo de interacción que se puede realizar sobre la interfaz ENIA para llevar a cabo cada una de las acciones permitidas.

[11]Los componentes que se encuentran sobre el Escritorio pasan a denominarse *COTSgets*.



(a) Menú no visible (b) Menú visible

Figura 4.6: Componentes COTSgets OGC

Otras operaciones que se pueden realizar sobre un COTSget son las de redimensionar, maximizar y restaurar el tamaño de un componente. En el caso de la redimensión (acción #15) esta acción se puede llevar a cabo sobre la esquina inferior derecha. Las acciones para maximizar y restaurar (#10 y #11 respectivamente) se realizan sobre la barra de menú del COTSgets, no estando disponibles para todas las App. La acción de maximizar amplía el tamaño del componente a las dimensiones máximas posibles del navegador Web sobre el cual se esté manejando ENIA; y la acción para restaurar, devuelve un componente maximizado a su tamaño previo.

El usuario también puede mover un COTSget de sitio en el Escritorio (acción #14), ayudándose para ello de la barra de menú, pinchando con el cursor del ratón sobre ella, y sin soltar, desplazarse a la nueva posición donde se desea mover o reubicar el componente. Esta acción de reubicación, activa un comportamiento interno en el gestor de la rejilla para desplazar COTSget entre sí, en el caso de que un usuario desee colocar el COTSget sobre una posición ya ocupada por otro.

Por último, quedan unas de las acciones más importantes que se pueden hacer sobre los COTSgets de tipo OGC de ENIA, que son las acciones de agrupar y desagrupar capas OGC en un mismo COTSget o en varios. Por ejemplo en la Figura 4.6b (aunque puede que no se aprecie del todo bien) se muestra un COTSget OGC con siete capas temáticas activas, indicadas como pequeñas leyendas de color, situadas en la parte superior izquierda de su barra de menú. En una situación de partida, se dispone de un COTSget OGC base. El usuario luego puede seleccionar nuevas capas como servicios OGC desde el Panel y arrastrarlas directamente sobre el Escritorio (acción #4, ya vista), creando así nuevos componentes COTSget OGC en el Escritorio, o bien dejarlas caer sobre un COTSget base previamente existente en el Escritorio (acción #2), el cual puede ya tener otras capas previamente agrupadas. Esto último hace que el nuevo servicio OGC se superponga sobre el que había, pudiendo tener COTSget con varias capas a la vista (como el ejemplo de la figura). Otra forma permitida de agrupar capas OGC es a través de

© 2016 Vallecillos, J.

agrupación de COTSgets OGC base ya existentes en el Escritorio. Así, el usuario puede arrastrar un COTSget OGC base (esto es, con una solo capa, sin tener otras agrupadas) sobre otro COTSget OGC que hará de contenedor de capas (acción #1). Existe una tercera forma de agrupar capas OGC, y es a través de la acción de desagrupación. Para llevar a cabo una acción de desagrupación de capas, el usuario debe seleccionar una de las capas sobre la barra de menú del COTSget que contiene las capas agrupadas, y sacarla fuera, hacia el Escritorio (acción #8). El usuario también puede llevar esa capa que está desagrupando hacia otro componente COTSget que actúa como contenedor de capas (acción #3) el cual puede ser uno base o ya con otras capas agrupadas.

4.3. Escenarios de prueba

Una vez realizada la descripción de la aplicación ENIA y de cómo se lleva a cabo la gestión de este tipo de interfaz de usuario *mashup* construida a partir de *widgets*, es necesario mostrar ejemplos de su comportamiento cuando se realiza una interacción real sobre ella. La finalidad es, por lo tanto, ofrecer al lector una serie situaciones que justifican el uso de las aplicaciones *mashup* a las que la infraestructura propuesta da soporte, y que, además, suponen una parte importante del proceso de experimentación para validar y evaluar el trabajo de investigación desarrollado.

Para conseguir ese objetivo, esta sección presenta una serie de escenarios que podrían suceder (en un orden normal de ejecución) de manera secuencial. El primer escenario consiste en un usuario anónimo que accede a la aplicación; el segundo escenario describe la forma en la que un usuario anónimo pasa a formar parte del sistema, a partir de la operación de registro. En el tercer escenario, el usuario registrado accede al sistema y obtiene una interfaz cuya estructura viene determinada por su perfil. El cuarto escenario tiene como objetivo mostrar cómo un usuario (tanto registrado como anónimo) puede reconfigurar su interfaz. Por último, en el quinto escenario se describen las acciones que se llevan a cabo cuando un usuario registrado finaliza su sesión con la aplicación.

4.3.1. Escenario 1: Usuario anónimo

De manera inicial, cuando un usuario se conecta a la aplicación ENIA (accesible en la dirección `http://acg.ual.es/projects/enia/ui/`), el sistema muestra una interfaz por defecto que está asociada a los usuarios anónimos. Dicha interfaz contiene dos componentes a modo de ejemplo, (1) un componente mapa que visualiza una capa de información geográfica obtenida a partir de un servicio de la REDIAM que ofrece la localización de las vías pecuarias de Andalucía, y (2) un componente de redes sociales que permite ver los mensajes que existen en el muro del usuario de Twitter de la REDIAM. Estos componentes no representan necesariamente los dos componentes que podrían resultar más útiles para un primer acceso a la aplicación o para la interacción de un usuario anónimo, sino que han sido escogidos como dos ejemplos de componentes de entre todos los servicios disponibles y ofrecidos por la aplicación (en el panel de componentes que se encuentra en la parte izquierda de la interfaz). Además, dichos componentes de ejemplo resumen la capacidad de la infraestructura, la cual permite ofrecer

componentes de terceros sin ninguna modificación (como en el caso del componente de Twitter) o construyendo componentes que integran distintos servicios (como es el caso del componente mapa que integra servicios de OpenLayers con un servicio OGC ofrecido por la REDIAM). En la Figura 4.7 se muestra el aspecto de la interfaz gráfica ofrecida a un usuario anónimo cuando accede a la aplicación. Tal y como se ha mencionado anteriormente, la parte superior derecha de la interfaz nos ofrece información sobre el tipo de acceso, representado en nuestro caso con una imagen de usuario sin identificar.

En lo que respecta a la ejecución específica que se realiza en la infraestructura, cuando un usuario anónimo accede a ENIA, suceden los pasos que se describen a continuación. En primer lugar, la aplicación que se carga de inicio contiene únicamente dos componentes, un componente menú para gestionar la sesión de usuario y un componente panel que contiene el catálogo de servicios proporcionados por la aplicación ENIA y que pueden ser visualizados en forma de componente en la interfaz gráfica ofrecida. Además, la página inicial de la aplicación también contiene el código necesario para poder establecer una conexión entre la capa cliente a la capa dependiente de la plataforma de la infraestructura. Esta conexión se mantiene durante la sesión del usuario anónimo. El código ejecutado consiste en (a) crear un WebSocket para poder recibir mensajes de la capa dependiente y (b) informar al servidor JavaScript de la conexión de un nuevo usuario anónimo. Para ello, en dicho servidor existe un WebSocket que se encuentra a la escucha para poder establecer nuevas conexiones de usuario. Como resultado de esta conexión, en el servidor se crea un nuevo canal de comunicación para poder emitir mensajes al WebSocket creado en la aplicación *mashup*.

En segundo lugar, una vez creada la conexión entre la aplicación y el servidor JavaScript, desde el segundo se llama a la operación *Default init* del servicio público *Session*

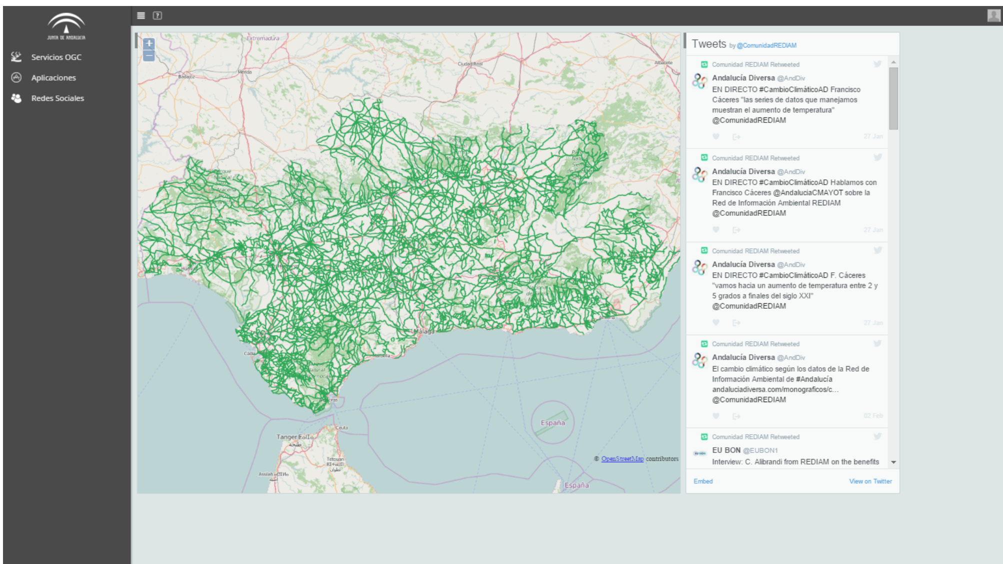


Figura 4.7: Interfaz *mashup* ENIA en el acceso de un usuario anónimo

© 2016 Vallecillos, J.

Service. La implementación de dicha operación traslada la invocación al módulo *COSSessionMM* el cual, por medio de la tarea *Initialize user architecture* realiza la inicialización de un modelo de arquitectura que existe y que representa la interfaz gráfica de usuario que se muestra por defecto. Esta inicialización consiste en crear un duplicado del modelo por defecto, el cual será vinculado con el nuevo usuario anónimo, asociándole un identificador derivado de la sesión temporal de dicho usuario. Además de la creación de este modelo, se inserta un nuevo usuario en la base de datos *Architectural Models and Users* cuyo identificador es el mencionado anteriormente. Para ello, el módulo *COSSessionMM* se comunica con el módulo *UIM* el cual, a su vez, se comunica con el controlador *Manage Users*, que se encarga de realizar los cambios en la base de datos. Como último paso, el módulo *COSSessionMM* hace uso del módulo *DMM* para llevar a cabo la creación de las instancias de los componentes que forman parte de la arquitectura de la aplicación. En el caso del modelo por defecto de ENIA, se proporcionan de inicio los componentes de mapa y de Twitter. Para la creación de dichas instancias, se tiene en cuenta el tipo de plataforma en el que se realiza el despliegue de la aplicación, en este caso la plataforma Web. Consecuentemente, las acciones relacionadas con la tarea de creación de instancias utilizan el controlador *Manage Wookie* de la infraestructura *mashup*. Dicho controlador hace uso de las bases de datos de *Widgets* y de *Widget instances* para obtener las definiciones de los dos componentes mencionados y crear las instancias correspondientes.

En tercer lugar, una vez que se han creado las instancias de los componentes, el código correspondiente es devuelto como resultado de la ejecución de la tarea *Initialize user architecture* del módulo *COSSessionMM*. Adicionalmente, también se devuelve el identificador del usuario anónimo y un mensaje con el éxito o error de la ejecución. El código de las instancias, el identificador y el mensaje son devueltos como resultado de la operación *Default init*. El resultado se obtiene en el servidor JavaScript de la capa dependiente, que se encarga de enviar la información a la capa cliente (*i.e.*, a la aplicación *mashup* desplegada de inicio). Si el mensaje obtenido es de error, se muestra en la interfaz gráfica una notificación para informar al usuario de la inicialización incorrecta. Si el mensaje obtenido es de éxito, la aplicación almacena su identificador para posteriores procesos de comunicación con el COScore de la infraestructra, y realiza el despliegue del código obtenido de las instancias de los componentes. Para llevar a cabo dicho despliegue, se inserta el código HTML en la parte que identifica el escritorio (dentro del espacio de trabajo) de la interfaz gráfica de ENIA. Cuando el código se despliega (recordemos que consiste en elementos de tipo `<iframe>` que hacen referencia al componente alojado en el repositorio de componentes Wookie), la capa cliente accede a los recursos que han sido creados en la base de datos *Widget instances* y se construye la interfaz gráfica.

4.3.2. Escenario 2: Registro de usuario

Una vez que se ha accedido a la aplicación ENIA como usuario anónimo y que se ha realizado la carga de la interfaz gráfica por defecto, es posible realizar tres tipos de operaciones. La primera opción que tiene el usuario es interactuar con su escritorio, utilizando los componentes que han sido cargados por defecto en su interfaz, añadiendo nuevos componentes a partir del catálogo de servicios (panel), redimensionando los COTSgets, etc. Es necesario recordar que toda esta interacción no se puede utilizar para

guardar el estado de la interfaz del usuario (puesto que se trata de un usuario anónimo), pero sí puede ser almacenada en la infraestructura para su posterior análisis de los comportamientos de los usuarios con la aplicación. La segunda opción es que el usuario inicie sesión en la aplicación. La tercera opción de la que dispone el usuario es llevar a cabo su registro en la aplicación mediante la introducción de una serie de datos de identificación. En esta subsección se describe la tercera de las opciones.

Para poder realizar el registro en ENIA, es necesario utilizar el componente menú de la interfaz gráfica, que es responsable, entre otras funcionalidades, de la gestión de la sesión de los usuarios. En la Figura 4.8 se muestra el formulario de inicio/registro que se visualiza como resultado de la interacción con el icono que representa al usuario cuya sesión se encuentra abierta y que se muestra en la parte superior derecha de la interfaz. En la parte inferior del formulario se muestran los campos que se deben rellenar para poder registrarse como nuevo usuario de la aplicación: nombre, apellidos, fecha de nacimiento, email, dirección, cuidad, país, perfil y contraseña. Cuando se rellenan dichos campos con la información correspondiente, el usuario debe presionar el botón *Registrarse*, dando así inicio al proceso. En primer lugar, se realiza una validación de los datos introducidos en el formulario. Este tipo de validación ocurre en la capa cliente de la infraestructura y se lleva a cabo mediante la ejecución de código que existe en la interfaz *mashup* que se carga inicialmente. Esta validación incluye comprobaciones típicas de un formulario de registro, como por ejemplo, que todos los campos estén rellenos, que el formato de la dirección de correo electrónico sea correcto, o que la contraseña se haya introducido (y repetido) correctamente. Una vez que los datos han sido validados, se utiliza la conexión existente con el servidor JavaScript de la capa dependiente para ejecutar el resto de la funcionalidad asociada al proceso de registro. Dicha conexión fue creada previamente cuando el usuario anónimo accede a la interfaz *mashup* de ENIA (ver Subsección 4.3.1).

La interfaz *mashup* (desplegada en la capa cliente) envía una petición a la capa dependiente para llevar a cabo el proceso de registro. En esta petición, se incluye como parámetro de entrada la información cumplimentada en el formulario. El servidor JavaScript recibe dicha petición e invoca la operación *Create user* del servicio privado *User*

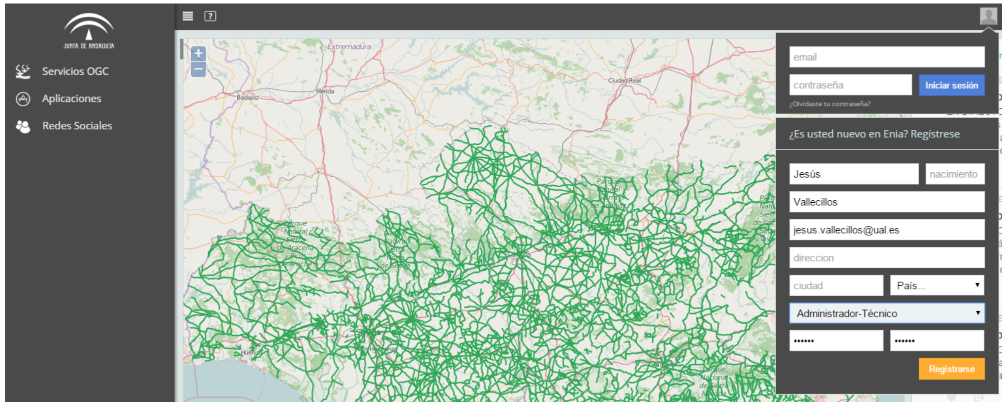


Figura 4.8: Interfaz *mashup* ENIA para el registro de nuevos usuarios

© 2016 Vallecillos, J.

Service que ofrece la capa independiente. El motivo de hacer uso de una operación que se encuentra en un servicio privado se debe a que este proceso de registro sólo puede ejecutarse si se conoce la clave privada del COScore que permite hacer uso de este tipo de operaciones. De esta forma, no se permite que cualquier aplicación que utilice la infraestructura propuesta pueda añadir nuevos usuarios a la base de datos *Architectural Models and Users*. Además, el servicio *User Service* (que incluye también operaciones de modificación, eliminación, etc.) se utiliza para la gestión de usuarios que se lleva a cabo de forma interna en el COScore. Si llegado el caso fuera necesario disponer de un registro de usuarios de carácter público, se podría añadir de manera sencilla una operación análoga en un servicio de tipo público para que cualquier aplicación *mashup* que interactúe con la infraestructura pueda realizar dicho proceso.

En la invocación que se realiza desde la capa dependiente a la operación *Create user*, se añade el parámetro de la clave privada al resto de información del usuario. La implementación de la operación comprueba si la clave privada introducida es correcta y en caso de éxito, se traslada la ejecución al módulo *UIM*. En este módulo, la tarea *Process to create a user* se comunica con el controlador *Manage Users*, el cual se encarga de dar el alta al usuario en la base de datos *Architectural Models and Users*. De manera adicional al alta de usuarios, en la base de datos también se realiza la inserción de un nuevo modelo de arquitectura. Dicho modelo consiste en un duplicado del modelo por defecto que existe para el perfil con el cual se crea al nuevo usuario. Por ejemplo, en el registro de usuario mostrado en la Figura 4.8 el perfil escogido es `Administrador-Técnico` y su modelo de arquitectura asociado está constituido por seis componentes: cuatro componentes de mapas temáticos, un componente de Twitter y un componente que muestra información meteorológica. El nuevo modelo de arquitectura creado se vincula al usuario para que todos los cambios y reconfiguraciones que se realicen se apliquen en él. Cuando el usuario acceda al sistema con sus credenciales, la interfaz gráfica que se le muestre se construirá a partir de los componentes definidos en dicho modelo.

4.3.3. Escenario 3: Usuario registrado

Un usuario que se ha registrado en la aplicación ENIA puede acceder a un espacio de trabajo particular en el cual tiene la capacidad de personalizar los componentes que se muestran en su escritorio. Para ello, la interfaz gráfica dispone de un menú que incluye un botón de perfil de usuario (parte superior derecha de la interfaz). Cuando un usuario anónimo interactúa con este botón, se muestra el formulario de inicio/registro, tal y como se puede observar en la Figura 4.9. En la parte superior de dicho formulario, se muestran los dos campos que permiten acceder al sistema (identificador y contraseña). Cuando se presiona el botón *Iniciar sesión* de dicho formulario, se utiliza la conexión existente con el servidor JavaScript de la capa dependiente (ver Subsección 4.3.1) para ejecutar el resto de la funcionalidad asociada al proceso de inicio de sesión.

La interfaz *mashup* envía una petición a la capa dependiente, incluyendo como parámetro de entrada la información de acceso introducida en el formulario. El servidor Node.js recibe la petición y, como consecuencia, invoca la operación *Login* del servicio público *Session Service* proporcionado por la capa independiente. La implementación de esta operación traslada la ejecución al módulo *UIM* que ejecuta la tarea *Query user*

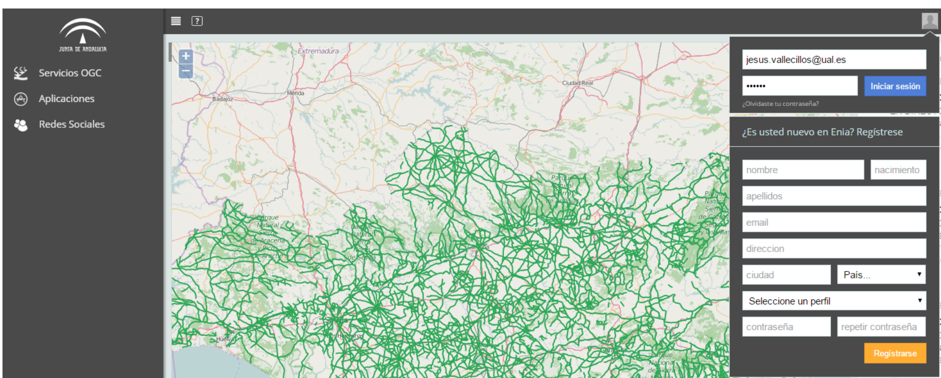


Figura 4.9: Interfaz *mashup* ENIA para la identificación de usuarios

in database para comprobar si el usuario se encuentra dado de alta en la base de datos *Architectural Models and Users* y si la contraseña es correcta. Si la comprobación de los datos del usuario es correcta, el módulo *UIM* se comunica con el módulo *COSSessionMM* para, a través de la tarea *Initialize modules*, llevar a cabo la inicialización del resto de los módulos (*IMM*, *DMM* y *TMM*) asociados a un usuario. Esta acción tiene como objetivo reducir el tiempo de respuesta en posteriores llamadas a las operaciones proporcionadas por la infraestructura.

Una vez inicializados los módulos, la operación de *Login* devuelve como respuesta el resultado obtenido en la tarea *Query user in database*. Dicho resultado consiste en (a) un valor booleano que indica si la comprobación del identificador y la contraseña ha sido correcta, (b) el índice de dicho usuario en la base de datos y (c) un mensaje con el resultado de la operación (de éxito o de error y su tipo). La respuesta de la operación es enviada a la capa dependiente que, a su vez, reenvía los datos a la capa cliente. Cuando la interfaz *mashup* recibe el resultado obtenido, pueden ocurrir las dos alternativas que se describen a continuación. Si se ha producido un error en el inicio de sesión, se muestra al usuario el mensaje correspondiente y se termina el proceso. Si, por el contrario, la operación de *Login* se realiza de forma correcta, desde la interfaz *mashup* se solicita la ejecución de la operación *Init user architecture* con el objetivo de construir el escritorio de la interfaz en base al último estado del modelo de arquitectura asociado al usuario.

Para poder llevar a cabo dicha solicitud, la capa cliente no puede utilizar la conexión previamente establecida con la capa dependiente, puesto que está asociada al usuario anónimo antes de iniciar la sesión en la aplicación ENIA. Por este motivo, se crea una nueva conexión con el servidor JavaScript en la cual se hace uso del identificador del usuario registrado para la creación de los WebSockets. Haciendo uso de dicha conexión, la capa cliente envía una solicitud a la capa dependiente para ejecutar la operación *Init user architecture*, que pertenece al servicio *Session Service* de la capa independiente. La implementación de dicha operación es similar a la operación *Defualt init*, descrita en la Subsección 4.3.1. En este caso, el módulo *COSSessionMM* utiliza el módulo *UIM* para obtener el modelo de arquitectura asociado al usuario. Esta acción la realiza la tarea *Query user in database* mediante el controlador *Manage Users*, el cual accede a la base de

© 2016 Vallecillos, J.

datos *Architectural Models and Users*. Cuando se obtiene dicho modelo de arquitectura, el módulo *COSSessionMM* inicializa los componentes. Como último paso, el módulo *COSSessionMM* se comunica con el módulo *DMM* para llevar a cabo la creación de las instancias de los componentes que forman parte de la arquitectura de la aplicación.

El modelo de arquitectura asociado al perfil `Administrador-Técnico` está formado por cuatro componentes de mapas temáticos, un componente de Twitter y un componente con información meteorológica asociada a la localización en la que se despliega la interfaz *mashup* ENIA. Por lo tanto, cuando el usuario inicia sesión por primera vez y se realiza la inicialización de su arquitectura, son estos seis componentes los que se utilizan para la construcción del escritorio de la interfaz gráfica. En este sentido, el controlador *Manage Wookie* hace uso de las bases de datos de *Widgets* y de *Widget instances* para obtener las definiciones de los componentes y crear las instancias correspondientes.

Una vez que se han creado las instancias de los componentes, el código se devuelve como resultado de la operación *Init user architecture*. De forma adicional, también se devuelve un mensaje indicando el éxito o error de la operación. El resultado se obtiene en el servidor JavaScript de la capa dependiente, que se encarga de enviar la información a la capa cliente. Si el mensaje obtenido es de error, se muestra en la interfaz una notificación para informar de la inicialización incorrecta. Si el mensaje obtenido es de éxito, la aplicación realiza el despliegue del código obtenido de las instancias de los componentes. Para llevar a cabo dicho despliegue, se inserta el código HTML en la parte que identifica el escritorio de la interfaz gráfica. Cuando el código ha sido desplegado, la capa cliente accede a los recursos creados en la base de datos *Widget instances* y se construye la interfaz gráfica. La Figura 4.10 muestra la interfaz *mashup* que se construye para el usuario registrado que ha sido utilizado de ejemplo en este escenario.

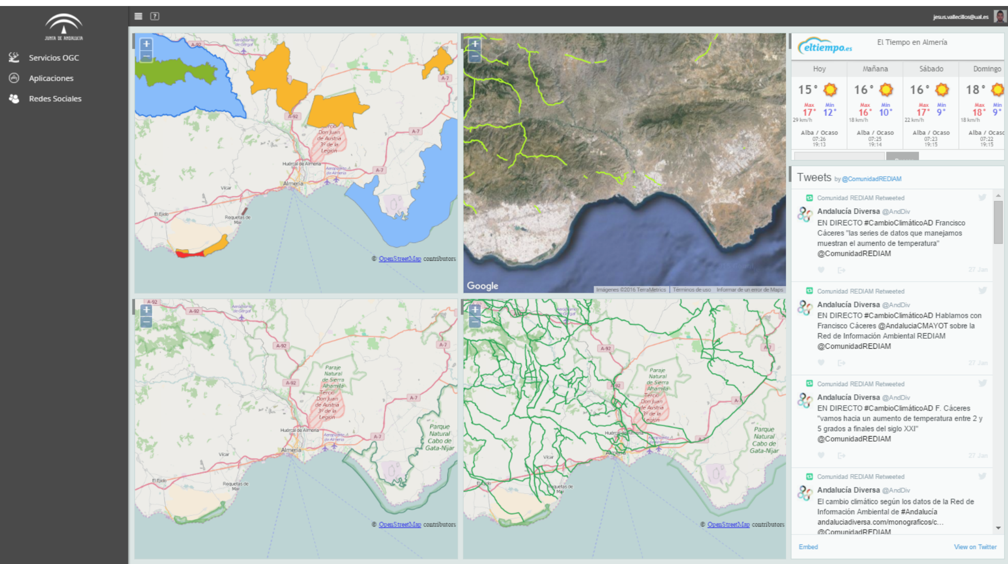


Figura 4.10: Interfaz *mashup* ENIA de un usuario registrado

4.3.4. Escenario 4: Reconfiguración de la interfaz

Este escenario parte de una interfaz *mashup* de ENIA en la que los tres escenarios anteriores ya han sido ejecutados previamente. Es decir, se ha realizado una carga inicial de la interfaz por defecto para los usuarios anónimos y, además, se ha iniciado con éxito la sesión de un usuario registrado. Por lo tanto, el objetivo de esta subsección es ilustrar cómo se lleva a cabo la modificación de una interfaz *mashup* asociada a un usuario, y cómo estas operaciones de reconfiguración se ejecutan en la infraestructura propuesta. Tal y como se ha mencionado previamente, la interfaz *mashup* del usuario registrado de ejemplo se corresponde con la arquitectura inicial del perfil `Administrador-Técnico`, y está constituida por seis componentes: cuatro mapas con capas geográficas, un Twitter y un componente con información meteorológica (Figura 4.10).

De entre todas las posibles interacciones que puede llevar a cabo el usuario, supongamos que decide realizar las que se describen a continuación. En primer lugar, elimina del escritorio los dos mapas temáticos que se encuentran en la parte inferior. A continuación, modifica la posición de los dos mapas restantes, pasando a ocupar el lugar de los dos mapas eliminados. Seguidamente, añade un componente visor de humedales al escritorio (servicio que se encuentra en el panel de ENIA, dentro de la categoría de aplicaciones de la REDIAM). Para finalizar, el usuario redimensiona el visor de humedales para que su tamaño ocupe el hueco que surge en el escritorio como resultado de haber modificado la localización de los dos mapas temáticos. Al ejecutar estas acciones, la interfaz *mashup* que se origina como resultado es la mostrada en la Figura 4.11.

La ejecución de cada de una estas interacciones tiene como consecuencia la invocación de la operación *Update architecture* del servicio *Component Service*, cuya finalidad

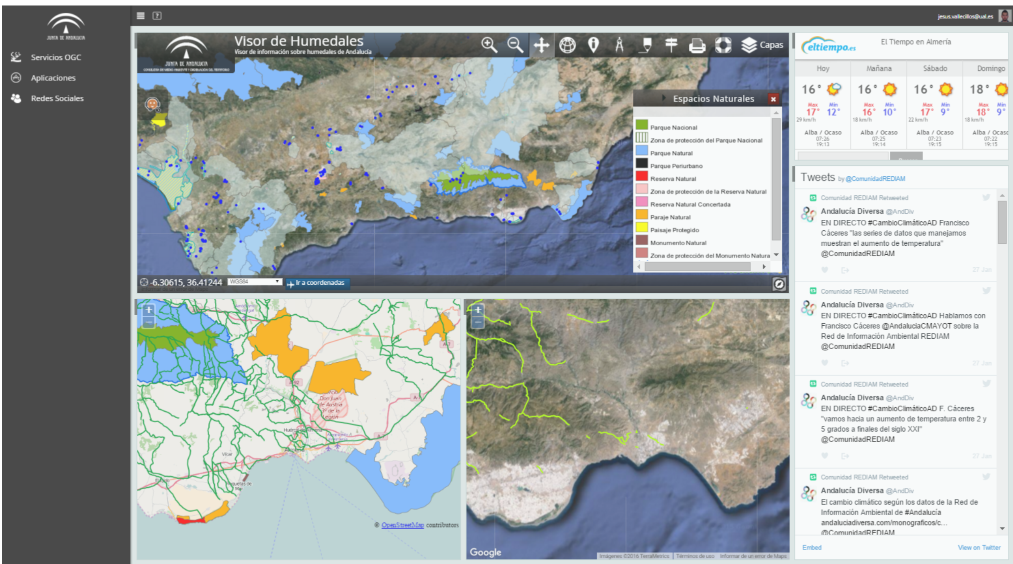


Figura 4.11: Interfaz *mashup* ENIA del usuario una vez reconfigurado el escritorio

© 2016 Vallecillos, J.

es realizar las modificaciones correspondientes en el modelo de arquitectura. Los cambios que dicha operación produce en el modelo incluyen acciones tales como las de inserción de nuevos componentes, eliminación de aquellos que no sean necesarios, o modificación de alguna propiedad de los componentes existentes. Teniendo en cuenta que todas las interacciones de ejemplo del escenario tienen un comportamiento similar en la infraestructura, a continuación se describen cuáles son los pasos que se llevan a cabo en la ejecución de uno de los procesos mencionados anteriormente, *i.e.*, la incorporación del nuevo componente visor en el escritorio de la interfaz *mashup*.

Para añadir un nuevo componente desde el panel de ENIA, es necesario arrastrar el icono que representa al componente hacia el escritorio de la interfaz, tal y como se muestra en la Figura 4.12. En el momento en el que el usuario suelta el componente en el escritorio, la interfaz *mashup* desplegada en el cliente hace uso de la comunicación establecida previamente con el servidor JavaScript (ver Subsección 4.3.4) para solicitar la ejecución de la operación encargada de añadir el componente. Cuando la capa cliente recibe dicha petición, realiza la invocación de la operación *Update architecture* proporcionada por la capa independiente. La implementación de dicha operación traslada la ejecución al módulo *DMM*. El primer paso que se lleva a cabo es obtener el modelo de arquitectura correspondiente al usuario que está interactuando con la interfaz. Para ello, el módulo *DMM* se comunica con el módulo *UIM* que, a su vez, hace uso del controlador *Manage Users* para consultar la base de datos *Architectural Models and Users*.

Una vez obtenido el modelo, el módulo *DMM* se encarga de determinar qué tipo de acción (insertar, eliminar, cambiar el valor de una propiedad, agrupar o desagrupar de servicios) se debe ejecutar. En este caso, se trata de la inserción de un nuevo componente (acción `add`). Posteriormente, el módulo *DMM* hace uso del controlador *Manage Component Specifications* para obtener las características del componente (a partir de la especificación almacenada en la base de datos *Concrete Component Specifications*) que se va a incorporar en el modelo. Tras ejecutar dicha tarea, el módulo *DMM* utili-

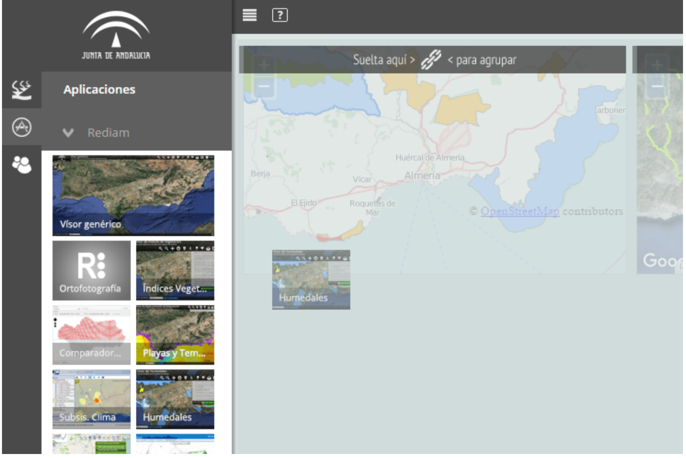


Figura 4.12: Inserción de un nuevo componente en el escritorio de la interfaz *mashup*

za el controlador *Manage Architectures* para actualizar el modelo, insertando el nuevo componente y conectándolo con otros elementos de la arquitectura en el caso de que sea necesario. Por último, el módulo *DMM* se comunica con el módulo *IMM* para guardar la interacción realizada por el usuario. Para ello, *IMM* hace uso del controlador *Manage Interaction*, que se encarga de añadir un nuevo registro en la base de datos *Interation* incluyendo información acerca del usuario, del tipo de acción realizada, o de los componentes implicados, entre otros posibles ejemplos. Como resultado de la operación *Update architecture*, se reconstruye la nueva lista de componentes (y su código correspondiente) que conforman la interfaz. Dicho resultado es enviado desde la capa dependiente de la plataforma a la capa cliente, que se encarga de reconfigurar la interfaz a partir de la nueva lista de componentes obtenida. En este caso, el nuevo componente insertado en el modelo de arquitectura debe ser también añadido en la interfaz (mediante la incorporación de un nuevo elemento `<iframe>` en la parte de la aplicación web que describe el escritorio de la interfaz *mashup*).

4.3.5. Escenario 5: Cierre de sesión

Cuando un usuario registrado finaliza su trabajo con la aplicación ENIA tiene la opción de cerrar la sesión que fue establecida con el sistema. La ejecución de este proceso es recomendable, puesto que tiene como resultado la liberación de recursos utilizados por la infraestructura, debido a que se eliminan estructuras de datos auxiliares y se finaliza la ejecución de los módulos que están asociados a un usuario. No obstante, es posible que el usuario se desconecte de la aplicación sin realizar este cierre de sesión, por ejemplo, cerrando el navegador web donde está desplegada la interfaz *mashup*. Para estos casos, cuando la inactividad de un usuario con la aplicación es superior a treinta minutos, el módulo *COSSessionMM* de la infraestructura de servicios ejecuta esta operación aunque no se haya recibido una petición desde el usuario.

En un escenario de ejecución normal de cierre de sesión, el usuario debe utilizar el botón de perfil de usuario del menú de ENIA que se encuentra en la parte superior derecha de la interfaz. Presionando dicho botón, aparece una pequeña ventana para finalizar la sesión del usuario, tal y como se muestra en la Figura 4.13. Cuando se pulsa sobre la opción *Cerrar sesión*, se llevan a cabo las siguientes acciones en el COScore. En primer lugar, se utiliza la conexión que existe entre la capa cliente y la capa dependiente de la plataforma (y que fue creada durante la ejecución del escenario 3 de la Subsección 4.3.3) para llevar a cabo de invocación de la operación *Logout* proporcionada por el servicio *Session Service* de la capa independiente.

La implementación de la operación traslada la ejecución al módulo *COSSessionMM*, el cual, haciendo uso de la tarea *Delete modules*, se encarga de eliminar cada uno de los módulos que han sido creados para el usuario registrado que solicita el cierre de sesión (*UIM*, *IMM*, *DMM* y *TMM*). Como consecuencia de la eliminación de dichos módulos, también se eliminan las estructuras de datos auxiliares que fueron creadas en sus respectivas inicializaciones para mejorar el rendimiento de las operaciones ofrecidas por la infraestructura. Por ejemplo, al eliminar el módulo *TMM* se elimina la tabla que contiene todas las relaciones que existen entre los componentes de la arquitectura. Dicha tabla se crea para reducir el tiempo de respuesta cuando se ejecuta la operación

© 2016 Vallecillos, J.

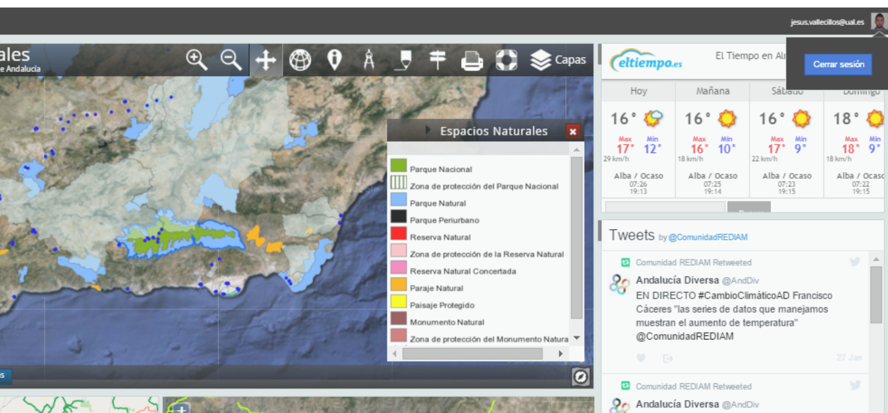


Figura 4.13: Cierre de sesión en la interfaz *mashup* ENIA

encargada de resolver los caminos de comunicación entre los componentes de la interfaz *mashup* (es decir, la operación *Get link components* del servicio *Communication Service*).

Como resultado de la operación *Logout*, se devuelve a la capa dependiente si la operación ha sido ejecutada correctamente (es decir, si los módulos han sido eliminados de manera satisfactoria) y un mensaje indicando el éxito o el error y, en este segundo caso, su tipo. Tal y como se ha descrito, la operación no se encarga de llevar a cabo ningún tipo de acción relacionada con la persistencia o con la actualización del modelo de arquitectura que define la interfaz *mashup* con la que interactúa el usuario. En este sentido, cada uno de los cambios que se llevan a cabo con la interacción del usuario, y que suponen una modificación del modelo de arquitectura, se gestionan en el COScore mediante la invocación de la operación *Update architecture* del servicio *Component Service* (ver Subsección 4.3.4). Como consecuencia, cuando un usuario registrado vuelve a acceder de forma identificada a la aplicación ENIA (4.3.3) tras su cierre de sesión, la interfaz *mashup* que se muestra presenta exactamente la misma apariencia que la que tenía tras ejecutar las operaciones de reconfiguración.

4.4. Experimentación y pruebas

Se ha realizado un estudio para validar y evaluar los servicios en la gestión de las aplicaciones *mashup* [Vallecillos et al., 2015]. Hasta ahora se ha mostrado la infraestructura a partir de la cual se da soporte a las aplicaciones *mashup*, detallando cómo ha sido desplegada y aportando detalles de implementación. Además, es interesante aportar un estudio de evaluación para validar la infraestructura propuesta. Por eso, se miden tiempos para los procesos de ejecución y se controlan tiempos de respuesta para dos operaciones de ejemplo de los servicios web públicos, dado que han sido las de mayor complejidad. Estas dos operaciones son: (1) inicialización de la arquitectura del usuario (*initUserArchitecture*) del *Session Service*, y (2) obtención de las conexiones de los componentes (*getLinkComponents*) del *Communication Service*.

Para ello, se han desarrollado varias pruebas diferentes con el objetivo de analizar el comportamiento de la infraestructura donde se obtienen tres parámetros que pueden influir sobre el rendimiento. Estos parámetros son: (a) el tamaño de la aplicación *mashup* construida en un proceso inicial y mostrada posteriormente a los usuarios, (b) el grado de acoplamiento de la arquitectura de la aplicación, es decir, el número de conexiones entre los componentes, y (c) el número de accesos concurrentes que se producen por parte de los usuarios. En este trabajo de investigación desarrollado, somos conscientes de que existen otros parámetros de entrada que afectan a los tiempos de respuesta, tales como la latencia de red o el uso del navegador web en la parte cliente, entre otros. Sin embargo, para asegurar que las pruebas realizadas fueran objetivas y funcionaran correctamente, sólo se ha experimentado con características de la aplicación que son manejables y controlables. Para ejecutar estos experimentos y medir los tiempos de respuesta, se ha usado un ordenador con un Intel(R) Core(TM) i5 CPU 660 3.33 CHz, con 4 GB de memoria principal que se ejecuta bajo el sistema operativo Windows 8.1 Profesional de 64 bits. Esta máquina incluye los servidores localizados en la capa dependiente e independiente de la plataforma. Para las pruebas propuestas, se desarrolló una aplicación web que realiza la función de cliente para la invocación de los servicios ofrecidos por la infraestructura. Cada tiempo de respuesta se calcula como la media de 100 repeticiones de la misma prueba unitaria.

Tal y como se muestra en la Figura 4.14, la infraestructura que da soporte a las aplicaciones *mashup* tiene tres capas (capa cliente o aplicación *mashup* (C), capa dependiente de la plataforma (B) y capa independiente de la plataforma (A)). El tiempo obtenido en (A) es el tiempo de ejecución de las funciones implementadas en el servidor de la capa independiente (COScore). Como consecuencia, el tiempo en (B) contiene el tiempo de (A) pero además incluye el tiempo que tiene el comportamiento implementado en la capa dependiente de la plataforma (servidor JavaScript). Finalmente, (C) representa el tiempo transcurrido entre el momento de la llamada del cliente en la aplicación *mashup* y el momento de la respuesta que es recibida y mostrada al usuario en su aplicación. La Figura 4.15a muestra el tiempo de respuesta para el proceso de inicialización (operación *initUserArchitecture*) cuando varia el número de componentes que forma la aplicación. Las diferencias entre los tiempos de medida en (A), (B) y (C) son muy pequeños. Por eso, los siguientes tiempos que se muestran se centran en (C) ya que son los tiempos más grandes (de hecho, es igual al tiempo de procesado total) y corresponden a los tiempos reales que observa un usuario mientras espera la respuesta del servicio.

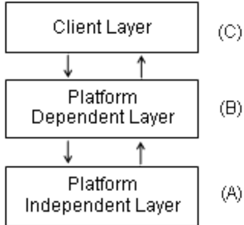
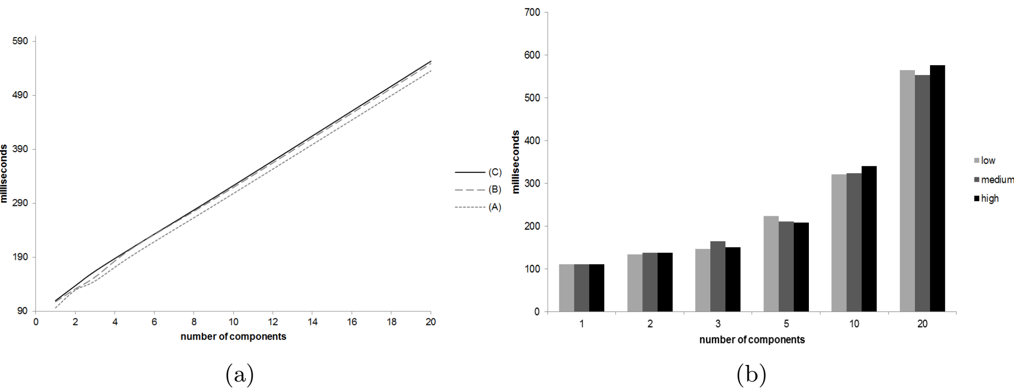


Figura 4.14: Puntos de medición en el proceso de evaluación

© 2016 Vallecillos, J.



(a) (b)

Figura 4.15: (a) Tiempos de respuesta en las capas A, B y C; (b) Inicialización de la aplicación *mashup* con tamaños de modelos y acoplamientos diferentes

El primer conjunto de pruebas se creó para evaluar la inicialización de la aplicación *mashup*. Se evaluaron los tiempos obtenidos para diferentes modelos de interfaz gráfica iniciales con 1, 2, 3, 5, 10 y 20 componentes. Este número de componentes seleccionado es el que suele manejarse en las interfaces de usuario *mashup* gestionadas por la infraestructura, donde suelen estar compuestas por menos de 20 componentes. Por ejemplo, en el escenario de ejemplo seleccionado, una interfaz de usuario con un único componente es una interfaz gráfica compuesta por un mapa que muestra información geoespacial. Además del mapa, la interfaz de usuario podría contener un componente de leyenda y/o un componente con la lista de capas mostrada en el mapa, lo cual representa un ejemplo típico de interfaz gráfica con dos y tres componentes. Debido a las limitaciones visuales y a las preferencias habituales del usuario, el escenario normal es manejar interfaces gráficas con un máximo de cuatro, cinco o seis componentes. Para mayor completitud del experimento, se incluyeron pruebas con diez y veinte componentes para evaluar el rendimiento bajo condiciones menos favorables.

También se han tomado tiempos con un acoplamiento bajo (*low*), medio (*medium*) y alto (*high*) en las aplicaciones. Si n es el número de componentes de una aplicación *mashup*, un acoplamiento bajo entre los componentes significa que habrá $n-1$ comunicaciones entre los componentes. Un alto acoplamiento entre los componentes significa que el número de comunicaciones entre los componentes es cercano al valor $n * n(n-1)/2$, y el acoplamiento medio es un número intermedio. La Figura 4.15b muestra los resultados de esta prueba. Es importante mencionar que los tiempos mostrados en la Figura 4.15a están muy relacionados con los tiempos mostrados en la Figura 4.15b.

A partir de las Figura 4.15a y 4.15b se pueden obtener tres conclusiones: (1) que los tiempos de respuesta crecen en proporción al número de componentes; (2) que el grado de acoplamiento no afecta al funcionamiento (o tiene una influencia insignificante), y (3) que el tiempo transcurrido para llevar a cabo el proceso de inicialización de la aplicación *mashup* es adecuado, ya que el tiempo de respuesta es inferior a 600 ms, lo cual hace que la experiencia del usuario al utilizar una aplicación *mashup* sea la esperada.

También se ha desarrollado otro conjunto de pruebas para comprobar los tiempos de inicialización de la aplicación *mashup* cuando varios usuarios pretenden iniciar sesión de forma concurrente, con el objetivo de probar cómo se comportaría la infraestructura en una situación real. Un resumen de los resultados de estos experimentos se muestra en la Tabla 4.2, donde cada fila (U_i) muestra el número de usuarios concurrentes que pretenden acceder a la inicialización. Por otro lado, cada columna (C_i) representa el número de componentes que contiene el modelo de la aplicación *mashup*. También se ve reflejado el grado de acoplamiento de los componentes quedando definido como bajo (l, *low*), medio (m, *medium*) y alto (h, *high*). La unidad de tiempos que se observa en la Tabla 4.2 son milisegundos. Cuando múltiples usuarios pretenden acceder de manera concurrente a la inicialización de la aplicación, el lado servidor no puede responder a todos los usuarios al mismo tiempo. En la Tabla 4.2 dentro de la columna *min* se muestra el tiempo invertido por el primer usuario que recibe la respuesta de la infraestructura y lleva a cabo el proceso de inicialización. La columna *max* representa el tiempo que tuvo lugar hasta que el usuario recibió la última respuesta y completó el proceso. La Figura 4.16 muestra una representación gráfica de los resultados.

U_i \ C_i	2						5					
	l		m		h		l		m		h	
	min	max	min	max	min	max	min	max	min	max	min	max
1	132.50		137.50		137.67		223.33		210.67		208.00	
10	162.00	381.67	131.33	353.00	157.67	393.33	264.00	565.00	225.67	574.67	244.00	561.33
50	120.67	1784.33	133.33	1815.33	115.67	1766.67	202.67	2747.33	196.67	3017.33	216.67	2759.33
100	152.67	3709.67	116.00	3651.33	153.00	3682.67	251.67	5806.67	224.67	5730.33	225.67	5704.00

U_i \ C_i	10						20					
	l		m		h		l		m		h	
	min	max	min	max	min	max	min	max	min	max	min	max
1	321.00		323.00		340.33		565.00		553.33		576.33	
10	414.33	937.67	440.67	915.67	412.00	923.00	653.00	1452.33	681.33	1593.67	684.67	1466.00
50	399.33	4311.33	476.00	4500.67	369.67	4307.33	670.67	7433.33	736.00	7502.67	715.67	7390.33
100	375.67	9083.33	464.33	9252.33	402.67	9152.00	677.00	14898.67	676.00	15043.00	695.67	16071.67

Tabla 4.2: Inicialización de la aplicación variando el número de usuarios concurrentes

A partir de los resultados mostrados en la Tabla 4.2 y los diferentes gráficos de la Figura 4.16, es posible determinar las siguientes conclusiones: (1) el tiempo de respuesta *min* crece de forma constante y no se ve afectado por el número de accesos concurrentes; (2) el tiempo de respuesta *max* aumenta en proporción al número de usuarios concurrentes; (3) cuanto mayor sea el modelo de aplicación, mayor será el incremento en el valor *max* para la inicialización de la aplicación; y (4) el acoplamiento de la arquitectura no influye en el rendimiento cuando se incrementa el número de usuarios que pretenden acceder de forma concurrente a la inicialización de la aplicación (como se mencionó antes para un usuario, ver Figura 4.15b). Los resultados en la Figura 4.16 son resumidos en los gráficos mostrados en la Figura 4.17a y la Figura 4.17b, los cuales representan los tiempos más bajos y más altos obtenidos, respectivamente.

El comportamiento observado en estos experimentos es relativamente bueno si se compara con los tiempos de respuesta obtenidos en la inicialización de la aplicación, cuando diferente número de usuarios acceden de forma concurrente. A pesar de eso, para modelos con 10 componentes, los tiempos de respuesta obtenidos (3.5 segundos) por los últimos usuarios (valor *max*) sobrepasan valores aceptables cuando en el entorno hay

© 2016 Vallecillos, J.

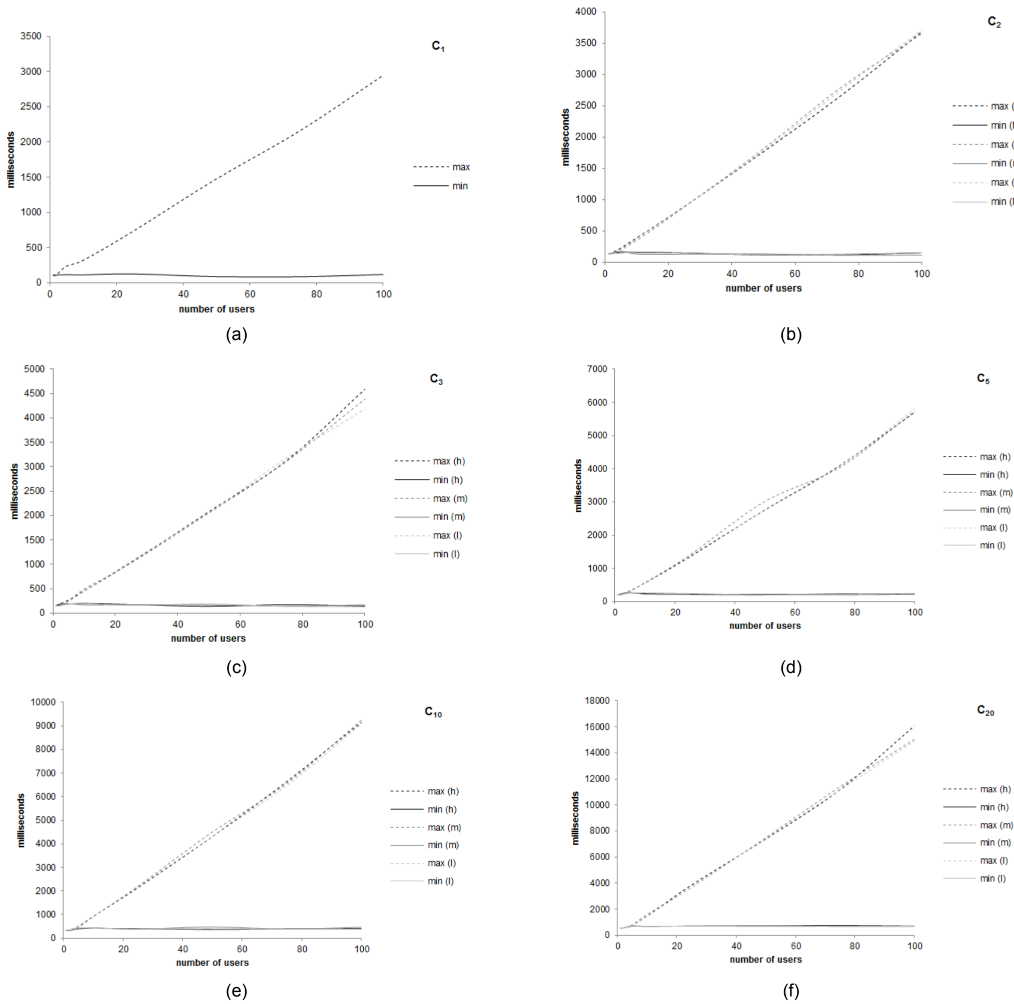


Figura 4.16: Inicialización de la aplicación variando el tamaño y el acoplamiento

cuarenta solicitudes accediendo de forma concurrente. Para modelos con 20 componentes, el tiempo de respuesta más alto está entorno a los 3.6 segundos, cuando se realizan alrededor de veinticinco inicializaciones concurrentes, tiempo que es excesivo. La razón por la cual el número de usuarios concurrentes influye en los tiempos de respuesta es porque el servidor de aplicaciones (servidor independiente de la plataforma) comparte el mismo punto de acceso para todos los usuarios (comparten EJBs) lo que implica un cuello de botella de la aplicación. El componente EJB encargado de gestionar las sesiones de los usuarios (módulo *COSSessionMM*) es uno de los componentes que dan pie a este cuello de botella. No obstante, con el propósito de mejorar el rendimiento, se realizaron experimentos adicionales en los cuales se consiguieron mejores tiempos de respuesta. En

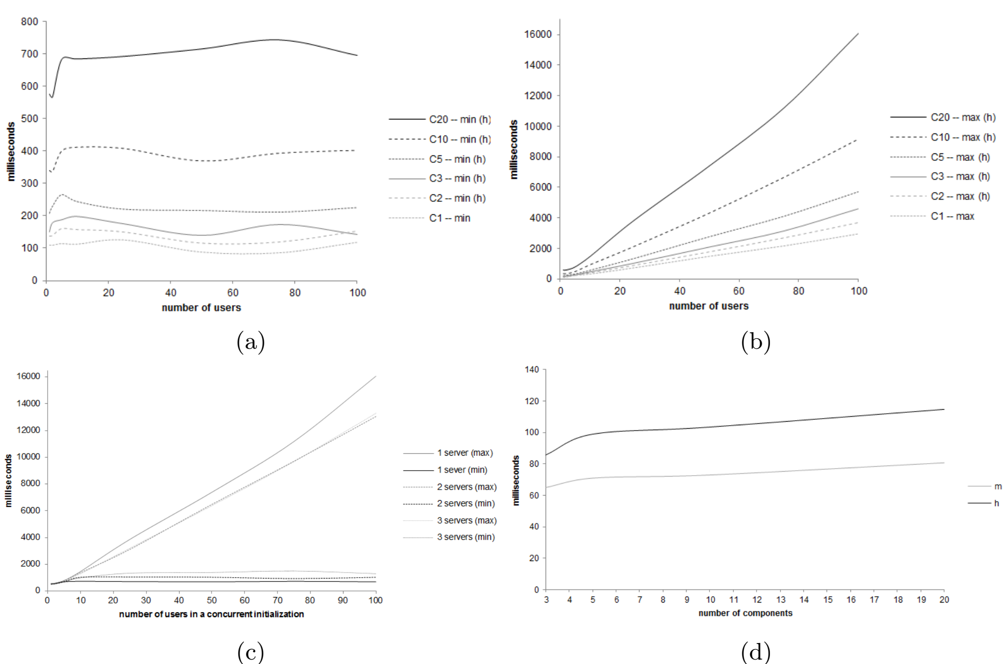


Figura 4.17: (a) Tiempos más bajos (primer usuario que recibe la respuesta); (b) Tiempos más altos (último que recibe la respuesta); (c) Usando 1, 2 y 3 servidores; (d) Evaluación de la comunicación con diferentes tamaños de la interfaz *mashup*

estas pruebas, se establecen dos y tres servidores independientes de la plataforma y se balanceó la carga de trabajo para distribuir las solicitudes entre ellos (Figura 4.17c).

Debido a los resultados de los experimentos se decidió finalmente desplegar la infraestructura de servicios con dos servidores independientes de la plataforma en lugar de uno o tres. Esta decisión se basó en dos factores: (a) el uso de más de un servidor es útil para evitar un cuello de botella en el sistema, y (b) usando más de dos servidores se invierte más tiempo en elegir el servidor destino, lo cual afecta a los tiempos de respuesta. La Figura 4.17c muestra que no hay mejora al usar tres servidores para el tiempo *max* pero, en cambio, los resultados son peores para el tiempo *min*.

Se realizaron las mediciones con diferente número de componentes y acoplamientos. Por razones prácticas, no se realizó la medición de tiempos con niveles de acoplamiento bajos, ya que hay muy pocas conexiones establecidas de comunicación. Se han desarrollado estas últimas pruebas para aplicaciones que contienen de tres a veinte componentes, puesto que este escenario proporciona suficientes conexiones para poder evaluar el proceso. La Figura 4.17d muestra los resultados de los experimentos. Los tiempos de respuesta para la comunicación permanecen por debajo de 100 ms para acoplamiento medio, y por debajo de 140 ms para el caso de acoplamiento alto. Estos tiempos crecen en proporción al número de componentes. Por lo tanto, se puede afirmar que la comunicación entre los componentes de las aplicaciones *mashup* se realiza en un periodo de tiempo adecuado.

© 2016 Vallecillos, J.

4.5. Resumen y conclusiones

En este capítulo se ha llevado a cabo la descripción del escenario principal que ha sido utilizado en este trabajo de tesis doctoral para la validación de la infraestructura de servicios propuesta. Se trata de un prototipo de aplicación *mashup* que está vinculado con el proyecto de investigación P10-TIC-6114 y que recibe el nombre de ENIA (ENvironmental Information Agent). Esta aplicación *mashup* consiste en una interfaz gráfica de usuario de tipo web que se construye a partir de componentes *widgets* de granularidad gruesa. El objetivo de esta aplicación es permitir la consulta de datos relacionados con el medioambiente (además de otra información relevante) de la REDIAM (Red de Información Ambiental de Andalucía). Para ello, el prototipo de interfaz gráfica desarrollado proporciona diferentes tipos de componentes, como son los mapas temáticos, los visores de información ambiental, o los componentes de redes sociales, entre otros posibles ejemplos. De esta manera, este prototipo permite validar y evaluar el correcto funcionamiento de los distintos servicios implementados, además de justificar una aplicación práctica del modelo de infraestructura propuesto.

La primera sección del capítulo ha descrito el contexto en el que se enmarca el prototipo de interfaz *mashup* ENIA, recordando las ventajas principales de utilizar este tipo de aplicaciones. Uno de los beneficios más importantes de hacer uso de aplicaciones *mashup* es que permiten que el usuario pueda modificarlas y reconfigurarlas para adaptar su estructura a sus preferencias. Además, la naturaleza modular, escalable e interoperable de estas aplicaciones facilita el desarrollo de componentes de terceros y la construcción de interfaces complejas que permitan realizar tareas de cierta envergadura. Por otro lado, también es posible analizar las interacciones de los usuarios con la aplicación para poder construir procesos de adaptación automáticos que reconfiguren la interfaz sin la necesidad de que el usuario así lo solicite de manera pro-activa.

Posteriormente se ha descrito detalladamente tanto la estructura de la interfaz *mashup* desarrollada, como su funcionamiento. Para ello, la segunda sección del capítulo ha presentado las partes principales de la interfaz, entre las que se encuentra el panel de componentes, el menú y el escritorio. Estas tres partes son constantes en todas las opciones de reconfiguración de la interfaz y es el contenido del escritorio el único que puede ser modificado (añadiendo nuevos elementos, eliminando componentes, etc.). De esta manera, la arquitectura de componentes propuesta (como parte del trabajo de investigación) se utiliza para representar el estado y la estructura de dicho escritorio. El panel de ENIA contiene el conjunto de componentes que el usuario puede añadir al escritorio y que incluye diferentes tipos de COTSgets. Algunos ejemplos de estos componentes son los mapas temáticos que hacen uso de servicios OGC o las aplicaciones de la REDIAM, entre las que se incluyen los visores de información ambiental o los comparadores de capas geográficas. El menú de ENIA proporciona la funcionalidad necesaria para la gestión de las sesiones de los usuarios. Además, este menú también se utiliza para mostrar y ocultar el panel de componentes, y para ofrecer una ayuda rápida en forma de guía para el manejo de la interfaz. Una vez descritas las partes principales de la interfaz *mashup*, se han resumido las distintas acciones que pueden llevarse a cabo sobre la interfaz ENIA.

A partir de la presentación del prototipo de interfaz desarrollado, la tercera sección del capítulo ha presentado cinco escenarios que permiten analizar el comportamiento de

ENIA y de la infraestructura propuesta cuando se realizan diferentes tipos de interacción. En el primer escenario, un usuario anónimo que accede a la aplicación y se construye un escritorio por defecto. El segundo escenario describe los pasos que se ejecutan cuando un usuario anónimo se registra en el sistema. En el tercer escenario, un usuario registrado accede a ENIA y construye la interfaz con la estructura asociada a su perfil. El cuarto escenario describe la ejecución cuando se realizan operaciones de reconfiguración. Por último, en el quinto escenario se describen las acciones que se llevan a cabo cuando un usuario registrado cierra su sesión en la aplicación.

Como parte del proceso de validación y evaluación de la infraestructura desarrollada, se han llevado a cabo distintas pruebas de estrés cuya finalidad ha sido medir el rendimiento de los servicios proporcionados (en términos de tiempos de respuesta). Con este objetivo, se han realizado distintas mediciones en base a tres parámetros: el tamaño de la arquitectura que define la aplicación *mashup* que se despliega en el cliente, el grado de acoplamiento de dicha arquitectura y el número de accesos concurrentes que llevan a cabo los usuarios de la infraestructura. El resultado obtenido nos permite validar que los tiempos de respuesta son correctos para el despliegue de interfaces *mashup* y para su posterior gestión en tiempo de ejecución.

© 2016 Vallecillos, J.

Capítulo 5

Resultados y conclusiones finales

© 2016 Vallecillos, J.

Capítulo 5

Resultados y conclusiones finales

Contenidos

5.1. Aportaciones a la comunidad científica 213
5.2. Limitaciones de la infraestructura desarrollada 216
5.3. Líneas de investigación abiertas 217
5.4. Publicaciones derivadas de la tesis doctoral 218

En este trabajo de investigación se ha desarrollado una infraestructura para ofrecer un conjunto de servicios a aplicaciones *mashup* construidas a partir de componentes COTSgets. La funcionalidad actual de esta infraestructura se ha obtenido a partir de diferentes versiones de este producto que se ha denominado **COScore**, permitiendo un desarrollo incremental del mismo. En la Figura 5.1 se muestran las diferentes versiones desarrolladas para esta solución, hasta llegar a la versión actual (*COScore 2.0.0*), descrita en varios capítulos a lo largo de esta memoria.

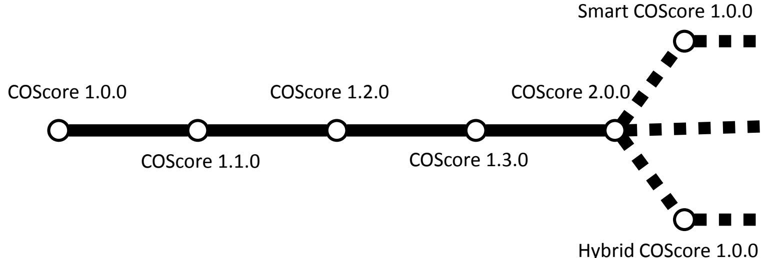


Figura 5.1: Versiones de COScore

Para poder diferenciar cada versión entre sí, se ha seguido una nomenclatura que identifica de manera única cada versión. La nomenclatura utilizada está formada por dos partes; por un lado el nombre del producto software (en este caso es *COScore*); y por el otro, un conjunto de tres valores numéricos: el primer valor se utiliza para identificar los cambios importantes desde el punto de vista funcional del software (un ejemplo podría ser la incorporación de una nueva capa dependiente de la infraestructura para dar servicios a componentes construidos para la plataforma tecnológica Android); el segundo valor numérico es utilizado para describir pequeñas modificaciones funcionales (como puede ser añadir un nuevo método a un servicio web público); y el tercer valor numérico se utiliza para identificar una infraestructura para la cual se ha resuelto algún error de programación. En una versión donde sólo se ha modificado el último número, no se incluye nueva funcionalidad. A continuación se describe brevemente cada una de las versiones implementadas en el COScore:

- *Versión COScore 1.0.0*: en esta primera versión se implementó la gestión del modelo de arquitectura de las aplicaciones, añadir/eliminar componentes de la aplicación, inicio/cierre de sesión, así como la persistencia para los usuarios registrados.
- *Versión COScore 1.1.0*: en esta versión se implementó la capacidad de comunicación entre los componentes de una aplicación *mashup*, a partir de un conjunto de relaciones binarias y n-arias.
- *Versión COScore 1.2.0*: esta versión incorporó la gestión de usuarios anónimos así como el cierre de sesión automático cuando se registra un tiempo de inactividad en el COScore por parte de una aplicación.

© 2016 Vallecillos, J.

- *Versión COScore 1.3.0*: en esta versión se implementó el registro de la interacción que se produce en una aplicación *mashup*. Este registro implica almacenar en la base de datos de interacción determinados eventos que se producen en la aplicación.

- *Versión COScore 1.4.0*: esta es la versión actual e incorporó, a partir de la versión anterior, la integración de dos plataformas. Así, la infraestructura está preparada para gestionar aplicaciones *mashup* construidas tanto con componentes widgets (siguiendo las especificaciones de W3C), como componentes construidos en Java.

A partir de la versión 2.0.0 del COScore, se está trabajando en tres líneas de desarrollo distintas, tal y como muestra la Figura 5.1. Cada una de estas versiones se asocia con los diferentes modos de ejecución que tendrán las aplicaciones *mashup*. La versión actual del COScore da soporte a aplicaciones construidas a partir de los componentes incluidos en el modelo de arquitectura inicial de un usuario. Este modelo cambia sólo cuando el propio usuario añade o elimina algún componente de su aplicación. De esta forma, el COScore nos ofrece un modo de funcionamiento “determinista” en su soporte a las aplicaciones. La línea discontinua de tiempo central identifica el conjunto de versiones del COScore que permitirá la extensión funcional de la infraestructura para dar soporte a aplicaciones, bajo esta forma de funcionamiento “determinista”.

La línea de tiempo superior (representada por *Smart COScore 1.0.0*), identifica el conjunto de versiones ligado a una vertiente del producto COScore en la cual el Grupo de Investigación de Informática Aplicada de la Universidad de Almería, está investigando un modo de funcionamiento “inteligente” de la infraestructura. En este modo, la infraestructura detectará automáticamente la necesidad de realizar cambios sobre la aplicación a partir de un proceso de inferencia utilizando la información proporcionada de la interacción obtenida del usuario, así como de otra información obtenida del contexto (como puede ser el ancho de banda de la red durante una sesión de un usuario).

Por último, la línea inferior de tiempo representa un modo de funcionamiento “híbrido” de los dos anteriores. En este modo, la infraestructura permite una cierta evolución de las aplicaciones *mashup*. Para conseguir esta evolución, la infraestructura incorpora un proceso de transformación dinámico basado en modelos y mediación que se encarga de mediar en la creación de nuevas versiones (nuevos modelos de arquitectura) de las aplicaciones añadiendo o eliminando componentes de la aplicación. Aquí el servicio de mediación hace uso de repositorios de reglas que conducen el proceso de transformación de la aplicación [Criado, 2015].

Teniendo en cuenta lo indicado anteriormente, en las siguientes secciones se detallan los resultados y conclusiones finales de este trabajo de investigación de tesis doctoral. Para ello, el resto del capítulo se estructura en cuatro secciones. En primer lugar, la Sección 5.1 describe las principales aportaciones del trabajo de investigación al ámbito de la comunidad científica de Ingeniería del Software, y en especial en el de la Ingeniería basada en componentes y servicios. La Sección 5.2 identifica ciertas limitaciones que presenta la propuesta que aquí se realiza en esta tesis doctoral. La sección 5.3 presenta las líneas de investigación que quedan abiertas tras la finalización de la tesis. Para finalizar, la Sección 5.4 ofrece un listado de las publicaciones derivadas del trabajo de investigación realizado en el desarrollo de esta tesis doctoral.

5.1. Aportaciones a la comunidad científica

El trabajo de investigación desarrollado en la presente tesis doctoral tenía como objetivo principal la elaboración de un modelo de infraestructura como solución para el despliegue y gestión de aplicaciones *mashup* pertenecientes a distintos dominios e, incluso, en dispositivos de plataformas diferentes. Con esta finalidad, el modelo de infraestructura finalmente desarrollado se basa en un proceso de abstracción que nos permite definir las aplicaciones *mashup* como arquitecturas software, en las que se representan cada uno de los componentes que forman parte de ellas, así como sus relaciones. De esta manera, las distintas piezas de las aplicaciones se gestionan de forma similar, independientemente del dominio y de la plataforma, y únicamente se ejecutan operaciones específicas de una plataforma cuando hay que generar el código necesario para el despliegue de las aplicaciones. Cada una de estas piezas ha sido nombrada como componente COTSget, de la combinación del concepto de componente COTS (*Commercial Off-The-Shelf*) para hacer referencia a componentes que pueden haber sido desarrollados por terceras partes, y del término *gadget*, que se refiere a un artefacto software que encapsula cierta funcionalidad y que permite llevar a cabo una tarea.

El modelo de **infraestructura** construido está constituido por tres capas, como ya se ha visto. La capa que se encuentra en el nivel inferior es la capa independiente de la plataforma, y contiene la funcionalidad común a todos los dispositivos para los que se pueden construir aplicaciones *mashup*. La capa superior está formada por las propias aplicaciones *mashup* y por los clientes que hacen uso de ellas, entre los que se encuentran, por ejemplo, los navegadores (en el caso de que las aplicaciones de tipo web) o los contenedores de aplicaciones (en el caso de las aplicaciones de tipo Java). Entre ambas capas se encuentra la capa dependiente de la plataforma, encargada de conectar la capa cliente con la capa independiente de la plataforma. El objetivo de esta capa intermedia es permitir la comunicación entre ambas capas, de manera que las aplicaciones que se encuentran desplegadas en la capa cliente puedan ser gestionadas por la lógica de las operaciones de la capa independiente (teniendo en cuenta la información que proporciona la primera) y que las decisiones tomadas en la capa inferior puedan modificar las aplicaciones en tiempo de ejecución y de forma dinámica. Esta flexibilidad se consigue, por ejemplo, sin llevar a cabo una recarga de toda la aplicación y actualizando únicamente aquellas partes de la arquitectura de la aplicación que hayan sido modificadas.

La capa dependiente de la plataforma también contiene los distintos repositorios de componentes que pueden formar parte de las aplicaciones *mashup* y, dependiendo del tipo de plataforma utilizada en cada momento, se hará uso de una colección u otra. Teniendo en cuenta que la capa cliente no es una aportación propia del trabajo de investigación, sino que únicamente se utiliza como soporte para el despliegue de las aplicaciones, las otras dos capas de la infraestructura (dependiente e independiente de la plataforma) constituye el núcleo de la propuesta de esta tesis doctoral, y juntas reciben el nombre de COScore (*COTSget-based architecture Operating Support core*). Uno de los aspectos que caracterizan este núcleo de la infraestructura es que está desarrollado como un conjunto de servicios. Cada servicio, a su vez, agrupa un conjunto de operaciones relacionadas entre sí. Por ejemplo, existe un servicio que contiene todas las operaciones relacionadas con la gestión de usuarios (creación, eliminación, modificación, entre otras operaciones,

© 2016 Vallecillos, J.

como se ha visto). De esta forma, el desarrollo de la infraestructura ha generado una API[1] que ofrece toda la funcionalidad necesaria para poder llevar a cabo el despliegue de aplicaciones *mashup* y que, además, permite realizar todas las acciones de gestión relacionadas con el uso de dichas aplicaciones.

La aportación del trabajo de investigación realizado se puede resumir en las contribuciones que se listan a continuación:

— Se ha construido una infraestructura estructurada en tres capas para dar soporte a aplicaciones *mashup* de distintos dominios y plataformas. Para que sea posible comunicarse con la funcionalidad que implementa esta infraestructura, se ha desarrollado un conjunto de servicios (públicos y privados). Haciendo uso de estos servicios, las aplicaciones pueden conectarse con la infraestructura para permitir su despliegue y su posterior gestión. Las capas por las cuales está construida la infraestructura son la capa cliente, la capa dependiente de la plataforma y la capa independiente de la plataforma. En la capa cliente se encuentran las aplicaciones *mashup*, en la capa dependiente de la plataforma se localiza un servidor JavaScript encargado de hacer de mediador para gestionar los procesos de comunicación entre la capa cliente y la capa independiente de la plataforma. Además, en la capa dependiente se encuentran los repositorios de componentes que son utilizados para el despliegue en las aplicaciones *mashup*. Por último, la capa independiente de la plataforma contiene el núcleo funcional de la infraestructura.

— Se ha gestionado una serie de repositorios indispensables dentro de la infraestructura. Se trata de repositorios de componentes, tanto propios como de terceras partes. Con respecto a los repositorios de terceras partes, se ha implementado diverso código de envolvente (*wrapper*) para los componentes localizados en bases de datos que no pertenecen a nuestro desarrollo. A partir de estos componentes adaptados, los componentes desarrollados por terceras partes pueden ser gestionados para ser integrados en las aplicaciones *mashup*. Por otro lado, los repositorios de componentes propios contienen los componentes construidos íntegramente siguiendo la especificación de componente aquí desarrollada.

— Se ha definido una especificación de componente por medio del uso de técnicas de metamodelado. Este metamodelo de componente se utiliza para poder construir las arquitecturas de componentes que definen las aplicaciones *mashup*. Esta especificación contiene todos los elementos necesarios para que un componente pueda ser integrado en la arquitectura de componentes de una aplicación. Además, se ha construido una especificación para describir formalmente las aplicaciones *mashup*, en forma de arquitecturas software constituidas por componentes y relaciones.

— Se han definido conjuntos de relaciones que pueden existir entre los componentes de una arquitectura. Para realizar este desglose de tipos, en primer lugar se ha creado una serie de escenarios simples de ejemplo: un escenario de domótica y otro relacionado con un sistema de información geográfica. A partir de estos escenarios, se han descrito dos grupos de relaciones, las binarias y las n-arias. Las relaciones binarias

[1] COScore API – `http://acg.ual.es/projects/enia/ui/webservices/`

relacionan dos componentes de la arquitectura entre sí y las n-arias relacionan tres o más componentes entre sí. Asimismo, se ha realizado un estudio de dependencias entre componentes, definiendo un conjunto de patrones y una matriz de regeneración de dependencias. Este estudio permite detectar situaciones alternativas en la configuración de una arquitectura. El estudio se ha realizado sólo sobre la combinación de tres componentes base, en el entorno de visor de mapas temáticos, con un componente de mapa, un componente de leyenda y otro de lista de capas.

— Con el objetivo de que la infraestructura pueda dar soporte a las aplicaciones *mashup*, se ha creado un conjunto de servicios públicos por medio de los cuales las aplicaciones se conectan y acceden a la funcionalidad ofrecida. Dichos servicios públicos son: *Register interaction* (encargado de gestionar procesos de registro de la interacción producida en la aplicación), *Update architecture* (el cual gestiona la actualización de las arquitecturas de componentes de las aplicaciones), *Get link components* (para gestionar los procesos de comunicación entre los componentes) y *Session* (encargado de gestionar los procesos de inicio de sesión para los usuarios de las aplicaciones). Cada servicio, a su vez, está compuesto por un conjunto de operaciones.

— También se necesitan operaciones de carácter privado para gestionar la infraestructura y poder dar así funcionalidad a las aplicaciones *mashup*. Para ello, se ha definido un conjunto de servicios privados: *User* (encargado de gestionar los usuarios que hacen uso de aplicaciones *mashup*), *Manage Architecture* (que gestiona las especificaciones de las arquitecturas utilizadas para describir las aplicaciones *mashup*) y *Manage Component* (el cual gestiona las especificaciones de los componentes que pueden ser utilizados por la infraestructura). De forma similar a los servicios públicos, cada servicio privado está compuesto por un conjunto de operaciones.

— Se han ofrecido detalles acerca de cómo se han implementado los servicios pertenecientes a la infraestructura. Para ello, se han descrito cada una de las operaciones que constituyen los servicios y, además se han mostrado ejemplos para ilustrar de qué manera se deben consumir dichos servicios.

— Se han realizado distintas pruebas y experimentos con el objetivo de poder analizar los tiempos de respuesta de la infraestructura. Para realizar estas pruebas, se han tenido en cuenta diferentes escenarios de ejemplo en los cuales se ha evaluado el impacto de diferentes factores que afectan a los resultados obtenidos, como por ejemplo, el número de componentes que forman parte de la aplicación *mashup* o el número de usuarios que acceden en el mismo instante de tiempo a un recurso.

— Finalmente, se ha analizado el principal caso de estudio que ha sido desarrollado para la validación del trabajo de investigación realizado. Se trata de una aplicación *mashup* de tipo web para la explotación de datos pertenecientes a un sistema de información geográfica. Dicho sistema recibe el nombre de ENIA (*ENvironmental Information Agent*) y se enmarca dentro de un proyecto de investigación regional, cuyo objetivo principal es la explotación de información geográfica basada en mapas y obtenida a partir de servicios OGC (*Open Geospatial Consortium*) ofrecidos por la REDIAM (*Red de Información Ambiental de Andalucía*).

© 2016 Vallecillos, J.

5.2. Limitaciones de la metodología desarrollada

La infraestructura desarrollada tiene diversas limitaciones que afectan a las capacidades de las aplicaciones *mashup* que pueden ser desplegadas en ella. A continuación se indican las más importantes:

— Los componentes que forman parte de la infraestructura, que serán integrados en las aplicaciones *mashup*, son componentes cerrados o de caja "negra". Esto significa que la infraestructura no tiene acceso a los eventos que tienen lugar en su interior. Esto tiene varias implicaciones, por un lado cuando se produzca un error en su interior la infraestructura no será consciente de lo ocurrido. Además la interacción que se produce dentro de los componentes no puede ser registrada por la infraestructura, lo cuál implica que esa interacción no podrá ser utilizada para realizar tomas de decisiones futuras basadas en dicha interacción.

— No es trivial realizar una nueva integración de una tecnología en la infraestructura. Esto significa que cada vez que sea necesario añadir un nuevo tipo de aplicación *mashup*, por ejemplo una aplicación *mashup* basada en tecnología python, sería necesario realizar numerosos cambios en la infraestructura. Por un lado, se necesita integrar un nuevo tipo de repositorio de componentes a los que se accede para manejar componentes de este tipo. Pero también, hay que realizar modificaciones para distinguir el nuevo tipo de aplicación del resto de aplicaciones para que de este modo, se le pueda dar soporte.

— Para añadir un nuevo tipo de aplicación *mashup* a la infraestructura en primer lugar deben de existir los repositorios de componentes correspondientes. Estos repositorios deben dar soporte a componentes que han sido construidos bajo la especificación de componentes aquí definida.

— En este momento la infraestructura sólo da soporte a dos tipos de aplicaciones *mashup*, aplicaciones de tipo web (basadas en Widgets) o aplicaciones de tipo Java. Esto significa que la infraestructura se encuentra algo limitada al no poder abarcar otros tipos de aplicaciones.

— Para poder gestionar una aplicación *mashup* en primer lugar se debe dar de alta el modelo de arquitectura de la aplicación en la infraestructura. Eso significa que se debe de crear el modelo de la aplicación previamente. La construcción de los modelos de la arquitectura de las aplicaciones se realiza de forma manual a través del editor reflexivo que EMF a partir del metamodelo del lenguaje propuesto. Sin embargo, sería deseable disponer de un editor gráfico que permita definir dichas arquitecturas, facilitando así las tareas de diseño y desarrollo de las arquitecturas de las aplicaciones. Lo mismo ocurre para la construcción de las especificaciones de los componentes concretos.

— Dentro de la infraestructura, para lograr comunicar los componentes entre sí se necesita que en el modelo de arquitectura de la aplicación *mashup* se haya definido una relación entre los componentes. Sin embargo, para un componente nuevo, no se podrán comunicar los componentes entre sí, puesto que se desconoce su relación.

5.3. Líneas de investigación abiertas

A lo largo del trabajo de investigación desarrollado en esta tesis doctoral, y partiendo también de las limitaciones descritas en la sección anterior, han aparecido nuevas líneas de investigación que quedan abiertas y que pueden ser abordadas como trabajos de investigación futuro. Estas líneas han sido:

— Definir y desarrollar tipos de componentes para *mashup* que estén preparados con propiedades de reflexión, que puedan ser accesibles y consultar su estado u obtener otra información de interés, como la vinculada con la interacción del usuario con dichos componentes.

— Modificar la infraestructura para facilitar la incorporación de nuevos tipos de aplicaciones *mashup*. El estado actual de la implementación realizada, sólo ofrece soporte a dos tipos de aplicaciones *mashup*, aplicaciones de tipo web (basadas en componentes *Widgets*) y aplicaciones de tipo Java. La extensión a nuevos tipos de aplicaciones, puede ayudar a potenciar la utilidad de la infraestructura COScore a otros tipos de aplicaciones, con el requisito de que estas deben estar formadas por una arquitectura de componentes subyacente.

— Desarrollar, como herramienta de diseño, un editor gráfico que permita definir la arquitectura de las aplicaciones *mashup*. Disponer de una herramienta como ésta puede facilitar el diseño y desarrollo de las arquitecturas de aplicaciones *mashup*. Este editor gráfico se podría utilizar también para construir las especificaciones de los componentes concretos, que luego se integran como parte de una arquitectura que conforma una aplicación *mashup*.

— Desarrollar una nueva variante de los modos de ejecución "Smart" e "Hybrid" para permitir la evolución de las aplicaciones *mashup* de forma automática o semiautomática, basada en información del contexto e información de la interacción del usuario con la aplicación.

— Desarrollar un mecanismo para la creación de nuevos componentes complejos (en tiempo de ejecución) a partir de otros más sencillos. Esto requiere que dichos componentes nuevos deban estar dados de alta en el repositorio de especificaciones de componentes concretos como la combinación de los componentes simples que los constituyen (definidos también a nivel de especificación).

En la actualidad, algunas de estas líneas de investigación abiertas están ya siendo abordadas como parte de otras tesis doctorales y proyectos de investigación, en el seno del grupo de investigación de *Informática aplicada* de la *Universidad de Almería*. Igualmente, la tecnología desarrollada en este trabajo de investigación está siendo estudiada para ser transferida e implantada de forma progresiva en el marco de un contrato I+D establecido con el Ayuntamiento de Almería para el desarrollo de una aplicación SIG de uso corporativo, como actividad dentro de un convenio de colaboración para el fomento de las tecnologías en la administración local entre el Ayuntamiento de Almería, el grupo de investigación de Informática Aplicada y la Universidad de Almería.

© 2016 Vallecillos, J.

5.4. Publicaciones derivadas de la tesis doctoral

Para finalizar, en esta sección se presentan los artículos y aportaciones a actas de congresos que han sido publicados como resultado del trabajo de investigación llevado a cabo durante el desarrollo de la presente tesis doctoral. La lista de publicaciones aparece en orden cronológico:

— **Vallecillos, J.**, Fernández, A.J., Criado, J., Iribarne, L. (2012). TvCSL: An XML-based language for the specification of TV-component applications. In *Communications in Computer and Information Science*, CCIS Vol. 278, pp. 574–580. *Springer.* doi:10.1007/978-3-642-35879-1_73

— A.J. Fernandez-Garcia, L. Iribarne, J. Criado, **J. Vallecillos** (2012). An approach to a pattern for business process management and deployment of software engineering for small companies in a crossplatform era. IEEE CS, 252-256. In 2nd Int. Conf. on Advances in Computational Tools for Engineering Applications (ACTEA), 12-15 December 2012 in Zouk-Mosbeh, Lebanon. *IEEE.* doi:10.1109/ICTEA.2012.6462877. doi:10.5220/0004257903970402

— Sobrino, J.F., Criado, J., **Vallecillos, J.**, Padilla, N., Iribarne, L. (2013). An Interface Agent for the Management of COTS-based User Interfaces. *5th International Conference on Agents and Artificial Intelligence (ICAART'2013)*, pp. 397-402, Barcelona, Spain. *INSTICC.* doi:10.5220/0004257903970402

— **Vallecillos, J.**, Criado, J., Padilla, N., Iribarne, L. (2014). A component-based user interface approach for Smart TV. *9th Int. Conf. on Software Engineering and Applications (ICSOFT-EA)*, pp. 455–463. *IEEE.* doi:10.5220/0004999304550463

— **Vallecillos, J.**, Criado, J., Iribarne, L., Padilla, N. (2014). Dynamic Mashup Interfaces for Information Systems Using Widgets-as-a-Service. *On the Move to Meaningful Internet Systems: OTM 2014 Workshops*, LNCS 8842, pp. 438–447. *Springer.* doi:10.1007/978-3-662-45550-0_44

— **Vallecillos, J.**, Criado, J., Iribarne, L., Padilla, N. (2014). Embedding Widgets-as-a-Service into Dynamic GUI. In *Proc. of the 10th Jornadas de Ciencia e Ingeniería de Servicios*, pp. 78–87.

— **Vallecillos, J.**, Criado, J., Iribarne, L., Padilla, N., (2015). A cloud service for COTS component-based architectures. *Computer Standards & Interfaces. Elsevier.* doi:10.1016/j.csi.2015.11.008

— **Vallecillos, J.**, Criado, J., Fernández, A.J., Padilla, N., Iribarne, L. (2015). A Web Services Infraestructure for the Management of Mashup Interfaces. *11th Int. Workshop on Engineering Service-Oriented Applications (WESOA'15)* in ICSOC (Int. Conf. on Service-Oriented Computing). 16 Nov. 2015, Goa University, Goa, India. *Springer.*

Anexo A

Pruebas de los servicios públicos

© 2016 Vallecillos, J.

Anexo A

Pruebas de los servicios públicos

Contenidos

A.1. Pruebas de Session Service A-3
A.1.1. Operación Login A-3
A.1.2. Operación Logout A-4
A.1.3. Operación Init user architecture A-4
A.1.4. Operación Default init A-6
A.2. Pruebas de Communication Service A-6
A.2.1. Operación Get link components A-6
A.3. Pruebas de Component Service A-8
A.3.1. Operación Update architecture A-8
A.4. Pruebas de Interaction Service A-23
A.4.1. Operación Register interaction A-23

Este anexo contiene las pruebas de error realizadas sobre las operaciones pertenecientes a los servicios públicos del COScore.

A.1. Pruebas de Session Service

Para el servicio público *Session Service*, a continuación se observa cuáles son las pruebas y errores obtenidos para cada operación.

A.1.1. Operación Login

Descripción: Omisión el parámetro username	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userPassword>aaa</userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Omisión el parámetro userpassword	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName>aaa</userName> </params>	<result> <validation>false</validation><params> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Parámetro username vacío	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName></userName> <userPassword>aaa</userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Parámetro password vacío	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName>aaa</userName> <userPassword></userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Parámetro username incorrecto	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName>aa2</userName> <userPassword>aa</userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Parámetro userpassword incorrecto	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName>aa</userName> <userPassword>aa2</userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Volver a hacer un Login sobre un usuario que ya se ha hecho	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName>aa</userName> <userPassword>aa</userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

Descripción Fallo en la conexión a la Base de Datos, se ha cambiado los parámetros de conexión	Resultado	ok
Parámetros de entrada	**Valores de salida**	
<params> <userName>aa</userName> <userPassword>aa</userPassword> </params>	<result> <validation>false</validation> <iduser>-1</iduser> <message>Not found or Empty userpassword Error</message> </result>	

© 2016 Vallecillos, J.

Descripción	Fallo interno de código, se ha provocado un error en la excepciones de encriptación del password	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userName>aa</userName>
  <userPassword>aa</userPassword>
</params>
```

```
<result>
  <validation>false</validation>
  <iduser>-1</iduser>
  <message>Not found or Empty userpassword Error</message>
</result>
```

A.1.2. Operación Logout

Descripción	Omisión del parámetro userId	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
</params>
```

```
<result>
  <deleted>false</deleted>
  <message>Not found or Empty userpassword Error</message>
</result>
```

Descripción	Parámetro userId vacío	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId></userId>
</params>
```

```
<result>
  <deleted>false</deleted>
  <message>Not found or Empty userpassword Error</message>
</result>
```

Descripción	Parámetro userId incorrecto no existe como sesión iniciada	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId>1234</userId>
</params>
```

```
<result>
  <deleted>false</deleted>
  <message>Error Delete Modules</message>
</result>
```

Descripción	Parámetro userId al cual ya se le ha hecho un logout	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId>4321</userId>
</params>
```

```
<result>
  <deleted>false</deleted>
  <message>Error Delete Modules</message>
</result>
```

Descripción	Fallo interno de código, se ha provocado un error en el nombre del módulo al cual se llama	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId>4321</userId>
</params>
```

```
<result>
  <deleted>false</deleted>
  <message>Internal Server Error</message>
</result>
```

A.1.3. Operación Init user architecture

Descripción	Omisión del parámetro userId	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <interaction>
      <deviceType>Browser</deviceType>
      <interactionType>MouseKeyboard
          </interactionType>
      <latitude>36.8276781 </latitude>
      <longitude>-2.4060609 </longitude>
  </interaction>
</params>
```

```
<result>
  <init>false</init>
  <message>Not found or Empty userId Error</message>
</result>
```

Descripción	Parámetro userId vacío	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId></userId>
  <interaction>
      <deviceType>Browser</deviceType>
      <interactionType>MouseKeyboard
          </interactionType>
      <latitude>36.8276781 </latitude>
      <longitude>-2.4060609 </longitude>
  </interaction>
</params>
```

```
<result>
  <init>false</init>
  <message>Not found or Empty userId Error</message>
</result>
```

Descripción	Omisión del parámetro deviceType de interation	Resultado	ok
Parámetros de entrada		Valores de salida	

<params> <userId>48</userId> <interaction> <deviceType>Browser</deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.8276781 </latitude> <longitude>-2.4060609 </longitude> </interaction> </params>	<result> <init>false</init> <message>Not found or Empty userId Error</message> </result>

Descripción	**Parámetro userId incorrecto, no existe como sesión iniciada**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>488</userId> <interaction> <deviceType>Browser</deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.8276781 </latitude> <longitude>-2.4060609 </longitude> </interaction> </params>		<result> <init>false</init> <message>Internal Server Error</message> </result>	

Descripción	**Parámetro userId al cual no se le ha hecho un login**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>48</userId> <interaction> <deviceType>Browser</deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.8276781 </latitude> <longitude>-2.4060609 </longitude> </interaction> </params>		<result> <init>false</init> <message>Internal Server Error</message> </result>	

Descripción	**Fallo interno de código, se ha provocado un error en el nombre del módulo al cual se llama**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>48</userId> <interaction> <deviceType>Browser</deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.8276781 </latitude> <longitude>-2.4060609 </longitude> </interaction> </params>		<result> <init>false</init> <message>Internal Server Error</message> </result>	

Descripción	**Fallo interno de código, se ha provocado un error en el nombre del módulo al cual se llama**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>48</userId> <interaction> <deviceType>Browser</deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.8276781 </latitude> <longitude>-2.4060609 </longitude> </interaction> </params>		<result> <init>false</init> <message>Internal Server Error</message> </result>	

Descripción	**Fallo en la conexión a la Base de Datos de usuarios, se han cambiado los parámetros de conexión**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>48</userId> <interaction> <deviceType>Browser</deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.8276781 </latitude> <longitude>-2.4060609 </longitude> </interaction> </params>		<result> <init>false</init> <message>java.sql.SQLException: org.postgresql.util.PSQLException: Conexión rechazada. Verifique que el nombre del Host y el puerto sean correctos y que postmaster este aceptando conexiones TCP/IP </message> </result>	

Descripción	**Fallo en la conexión a la Base de Datos de modelos de arquitectura concretas**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>48</userId> </params>		<result> <init>false</init> <message>Error in Architectural Models BD javax.ejb.EJBTransactionRolledbackException: ERROR: no existe la relación «concretearchitecturalmodel» Position:182 </message> </result>	

© 2016 Vallecillos, J.

Descripción	**Fallo en la conexión Wookie**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>48</userId> </params>		<result> <init>false</init> <message>Error in Wookie javax.ejb.EJBTransactionRolledbackException: java.net.ConnectException: Connection refused: connect </message> </result>	

A.1.4. Operación Default init

Descripción	**Fallo interno de código, se ha provocado un error en el nombre del módulo al cual se llama**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
		<result> <init>false</init> <anonimousIs>-1</anonimousIs> <message>Internal Server Error</message> </result>	

Descripción	**Fallo en la conexión a la Base de Datos de usuarios, se ha cambiado los parámetros de conexión**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
		<result> <init>false</init> <anonimousIs>-1</anonimousIs> <message>java.sql.SQLException: org.postgresql.util.PSQLException: Conexión rechazada. Verifique que el nombre del Host y el puerto sean correctos y que postmaster este aceptando conexiones TCP/IP </message> </result>	

Descripción	**Fallo en la conexión a la Base de Datos de modelos de arquitecturas concretas**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
		<result> <init>false</init> <anonimousIs>-1</anonimousIs> <message>Error in Architectural Models BD javax.ejb.EJBTransactionRolledbackException: ERROR: no existe la relación «concretearchitecturalmodel» Position:182 </message> </result>	

Descripción	**Fallo en la conexión Wookie**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
		<result> <init>false</init> <anonimousIs>-1</anonimousIs> <message>Error in Wookie javax.ejb.EJBTransactionRolledbackException: java.net.ConnectException: Connection refused: connect </message> </result>	

A.2. Pruebas de Communication Service

Para el servicio público *Session Service*, a continuación se observa cuáles son las pruebas y errores obtenidos para cada operación.

A.2.1. Operación Get link components

Descripción	**Omisión del parámetro userId**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <componentInstance> aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty username Error</message> </result>	

Descripción	**Parámetro userId vacío**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId></userId> <componentInstance> aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty username Error</message> </result>	

Descripción	**Parámetro userId incorrecto, no existe**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>33</userId> <componentInstance> aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Internal Server Error</message> </result>	

Descripción	**Parámetro userId al cual no se le ha hecho un login**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <componentInstance> aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Internal Server Error</message> </result>	

Descripción	**Omisión del parámetro componentInstance**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>	

Descripción	**Parámetro componentInstance vacío**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <componentInstance></componentInstance> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>	

Descripción	**Parámetro componentInstance incorrecto, no existe**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <componentInstance>eee</componentInstance> <portId>presas-port3</portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>	

Descripción	**Omisión del parámetro portId**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <componentInstance>aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> </params>		<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>	

Descripción	**Parámetro portId vacío**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <componentInstance>aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId></portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>	

Descripción	**Parámetro portId incorrecto, no existe**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	
<params> <userId>1</userId> <componentInstance>aOKFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>gg</portId> </params>		<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>	

Descripción	**Fallo interno de código, se ha provocado un error en el nombre del módulo al cual se llama**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	

© 2016 Vallecillos, J.

<params> <userId>1</userId> <componentInstance>a0KFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>presas-port3</portId> </params>	<result> <gotten>false</gotten> <message>Not found or Empty Component Instance Error</message> </result>

Descripción Fallo interno de código, se ha provocado un error en el nombre del módulo al cual se llama	**Resultado** ok
Parámetros de entrada	**Valores de salida**
<params> <userId>1</userId> <componentInstance>a0KFUSWtLIfHApPYnZRZjB2KHnE.eq. </componentInstance> <portId>presas-port3</portId> </params>	<result> <gotten>false</gotten> <message>Error in Architectural Models BD org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: OutputPort.cSource, no session or session was closed </message> </result>

A.3. Pruebas de Component Service

Para el servicio público *Session Service*, a continuación se observa cuáles son las pruebas y errores obtenidos para cada operación.

A.3.1. Operación Update architecture

Descripción Omisión del parámetro userId	**Resultado** ok
Parámetros de entrada	**Valores de salida**
<params> <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq. </componentInstance> <actionDone>changeproperty</actionDone> <newComponentData> <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq. </instanceId> <posx>4</posx> <posy>4</posy> </newComponentData> <newComponentData> <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq. </instanceId> <posx>5</posx> <posy>5</posy> </newComponentData> <newComponentData> <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq. </instanceId> <posx>6</posx> <posy>6</posy> </newComponentData> <interaction> <deviceType>Browser </deviceType> <interactionType>MouseKeyboard </interactionType> <latitude>36.827087399999996 </latitude> <longitude>-2.435696 </longitude> </interaction> </params>	<result> <allowed>false</allowed> <oldComponentData>...</oldComponentData> ... <oldComponentData>...</oldComponentData> <message>Not found or Empty UserId Error</message> </result>

Descripción Parámetro userId vacío	**Resultado** ok
Parámetros de entrada	**Valores de salida**
<params> <userId></userId> <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq. </componentInstance> <actionDone>changeproperty</actionDone> <newComponentData> <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq. </instanceId> <posx>4</posx> <posy>4</posy> </newComponentData> <newComponentData> <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq. </instanceId> <posx>5</posx> <posy>5</posy> </newComponentData> <newComponentData> <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq. </instanceId> <posx>6</posx> <posy>6</posy> </newComponentData>	<result> <allowed>false</allowed> <oldComponentData>...</oldComponentData> ... <oldComponentData>...</oldComponentData> <message>Not found or Empty UserId Error</message> </result>

```
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Descripción	Parámetro userId incorrecto, no existe	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>99</userId>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <actionDone>changeproperty</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <message>Internal Server Error</message>
</result>
```

Descripción	Parámetro userId al cual no se le ha hecho un login	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <actionDone>changeproperty</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <message>Internal Server Error</message>
</result>
```

Descripción	Omisión del parámetro componentInstance	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty ComponentsInstance
Error</message>
</result>
```

© 2016 Vallecillos, J.

```
    <newComponentData>
        <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
        </instanceId>
        <posx>5</posx>
        <posy>5</posy>
    </newComponentData>
    <newComponentData>
        <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
        </instanceId>
        <posx>6</posx>
        <posy>6</posy>
    </newComponentData>
    <interaction>
        <deviceType>Browser </deviceType>
        <interactionType>MouseKeyboard
        </interactionType>
        <latitude>36.827087399999996 </latitude>
        <longitude>-2.435696 </longitude>
    </interaction>
</params>
```

Descripción	**Parámetro componentInstance incorrecto**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
    <userId>1</userId>
    <componentInstance>dhmgBVWypAXutmmprCBkis9Jg.eq.
    </componentInstance>
    <actionDone>changeproperty</actionDone>
    <newComponentData>
        <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
        </instanceId>
        <posx>4</posx>
        <posy>4</posy>
    </newComponentData>
    <newComponentData>
        <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
        </instanceId>
        <posx>5</posx>
        <posy>5</posy>
    </newComponentData>
    <newComponentData>
        <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
        </instanceId>
        <posx>6</posx>
        <posy>6</posy>
    </newComponentData>
    <interaction>
        <deviceType>Browser </deviceType>
        <interactionType>MouseKeyboard
        </interactionType>
        <latitude>36.827087399999996 </latitude>
        <longitude>-2.435696 </longitude>
    </interaction>
</params>
```

Valores de salida:

```
<result>
    <allowed>false</allowed>
    <oldComponentData>...</oldComponentData>
        ...
    <oldComponentData>...</oldComponentData>
    <message>Error in Architectural Models DB
        javax.ejb.EJBTransactionRolledbackException:
        Index: 0, Size: 0
    </message>
</result>
```

Descripción	**Parámetro componentInstance vacío para una opción obligatoria**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
    <userId>1</userId>
    <componentInstance></componentInstance>
    <actionDone>delete</actionDone>
    <newComponentData>
        <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
        </instanceId>
        <posx>4</posx>
        <posy>4</posy>
    </newComponentData>
    <newComponentData>
        <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
        </instanceId>
        <posx>5</posx>
        <posy>5</posy>
    </newComponentData>
    <newComponentData>
        <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
        </instanceId>
        <posx>6</posx>
        <posy>6</posy>
    </newComponentData>
    <interaction>
        <deviceType>Browser </deviceType>
        <interactionType>MouseKeyboard
        </interactionType>
        <latitude>36.827087399999996 </latitude>
        <longitude>-2.435696 </longitude>
    </interaction>
</params>
```

Valores de salida:

```
<result>
    <allowed>false</allowed>
    <oldComponentData>...</oldComponentData>
        ...
    <oldComponentData>...</oldComponentData>
    <message>Not found or Empty ComponentsInstance Error
    </message>
</result>
```

Descripción	**Omisión del parámetro actionDone**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	

```
<params>
  <userId>1</userId>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty Action Done Error
  </message>
</result>
```

Descripción	Parámetro actionDone vacío	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId>1</userId>
  <actionDone></actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty Action Done Error
  </message>
</result>
```

Descripción	Parámetro actionDone incorrecto	Resultado	ok
Parámetros de entrada		Valores de salida	

```
<params>
  <userId>1</userId>
  <actionDone>change</actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty Action Done Error
  </message>
</result>
```

© 2016 Vallecillos, J.

Descripción	Omisión del parámetro newComponentData	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found Component
  </message>
</result>
```

Descripción	Parámetro newComponentData incorrecto	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    fallo
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Internal Server Error</message>
</result>
```

Descripción	Parámetro newComponentData vacío	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.instanceID List Error</message>
</result>
```

Descripción	Lista de newComponentData incorrecta, componente inexistente	Resultado	ok

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Error in Architectural Models DB
    javax.ejb.EJBTransactionRolledbackException:
    Index: 0, Size: 0
  </message>
</result>
```

```
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>dKsW9YUEyrRnDDWwl4T763ipRKc.eq.
    </instanceId>
    <posx>8</posx>
    <posy>8</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Descripción	**Omisión del parámetro newComponentData.componentInstance**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.instanceID List Error
  </message>
</result>
```

Descripción	**Parámetro newComponentData.componentInstance incorrecto**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>BVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Error in Architectural Models DB
    javax.ejb.EJBTransactionRolledbackException:
    Index: 0, Size: 0
  </message>
</result>
```

© 2016 Vallecillos, J.

Descripción	**Parámetro newComponentData.componentInstance vacío**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>
  </componentInstance>
  <newComponentData>
    <instanceId></instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.instanceID List Error
  </message>
</result>
```

Descripción	**Omisión del parámetro newComponentData.posx o posy**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Internal Server Error
  </message>
</result>
```

Descripción	**Parámetro newComponentData.posx o posy vacío con la información de la operación incorrecta**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId></instanceId>
    <posx></posx>
    <posy>4</posy>
  </newComponentData>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Internal Server Error
  </message>
</result>
```

```
    <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>5</posx>
      <posy>5</posy>
    </newComponentData>
    <newComponentData>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <posx>6</posx>
      <posy>6</posy>
    </newComponentData>
    <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
    </interaction>
  </params>
```


Descripción	Omisión del parámetro newComponentData.idHtml	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:


```
<params>
  <userId>1</userId>
  <actionDone>add</actionDone>
  <newComponentData>
    <componentname>http://acg.ual.es/wookie/widgets/
    OGC-ViasPecuarias-InventarioVVPP</componentname>
    <componentAlias>InventarioVVPP</componentAlias>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <idHtml>iniwidget3</idHtml>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <componentname>http://acg.ual.es/enia/widgets/
    RedesSociales-REDIAM-Twitter</componentname>
    <componentAlias>Twitter</componentAlias>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <idHtml>iniwidget7</idHtml>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <componentname>http://acg.ual.es/wookie/widgets/
    OGC-RENPA-DiplomaEuropeo</componentname>
    <componentAlias>Europeo13</componentAlias>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <idHtml>widget</idHtml>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <newComponentData>
    <componentname>http://acg.ual.es/wookie/widgets/
    OGC-RENPA-DiplomaEuropeo</componentname>
    <componentAlias>EEEE</componentAlias>
    <instanceId></instanceId>
    <posx>7</posx>
    <posy>7</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```


Valores de salida:


```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.idHtml List Error
  </message>
</result>
```


Descripción	Parámetro newComponentData.idHtml vacío	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:


```
<params>
  <userId>1</userId>
  <actionDone>add</actionDone>
  <newComponentData>
    <componentname>http://acg.ual.es/wookie/widgets/
    OGC-ViasPecuarias-InventarioVVPP</componentname>
    <componentAlias>InventarioVVPP</componentAlias>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <idHtml>iniwidget3</idHtml>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
```


Valores de salida:


```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News Components.idHtml
    List Error
  </message>
</result>
```


© 2016 Vallecillos, J.

```
    <newComponentData>
      <componentname>http://acg.ual.es/enia/widgets/
      RedesSociales-REDIAM-Twitter</componentname>
      <componentAlias>Twitter</componentAlias>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <idHtml>iniwidget7</idHtml>
      <posx>5</posx>
      <posy>5</posy>
    </newComponentData>
    <newComponentData>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-RENPA-DiplomaEuropeo</componentname>
      <componentAlias>Europeo13</componentAlias>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <idHtml>widget</idHtml>
      <posx>6</posx>
      <posy>6</posy>
    </newComponentData>
    <newComponentData>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-RENPA-DiplomaEuropeo</componentname>
      <componentAlias>EEEE</componentAlias>
      <instanceId></instanceId>
      <idHtml></idHtml>
      <posx>7</posx>
      <posy>7</posy>
    </newComponentData>
    <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
    </interaction>
  </params>
```

Descripción	**Omisión del parámetro newComponentData.numero servicios**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
  </componentInstance>
  <actionDone>groupfromdesktop</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>1</posx>
    <posy>1</posy>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-InventarioVVPP</componentname>
      <componentalias>InventarioVVPP</componentalias>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>InventarioVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-LugaresVVPP</componentname>
      <componentalias>LugaresVVPP</componentalias>
      <instanceId>0H2Y40tUByKUPvgkVITSFT9CvMY.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>LugaresVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-RENPA-DiplomaEuropeo</componentname>
      <componentalias>Europeo13</componentalias>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMDiplomaEuropeo?
      </mapaKML>
      <capa>DiplomaEuropeo</capa>
    </servicios>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>2</posx>
    <posy>2</posy>
    <numero servicios>1</numero servicios>
    <servicios>
      <componentname>http://acg.ual.es/enia/widgets/
      RedesSociales-REDIAM-Twitter</componentname>
      <componentalias>Twitter</componentalias>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <mapaKML/>
      <capa/>
    </servicios>
  </newComponentData>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.numero servicios List Error
  </message>
</result>
```

```
    <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
    </interaction>
  </params>
```

Descripción	**Parámetro newComponentData.numero servicios vacío**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
  </componentInstance>
  <actionDone>groupfromdesktop</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>1</posx>
    <posy>1</posy>
    <numero servicios></numero servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-InventarioVVPP</componentname>
      <componentalias>InventarioVVPP</componentalias>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>InventarioVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-LugaresVVPP</componentname>
      <componentalias>LugaresVVPP</componentalias>
      <instanceId>0H2Y4OtUByKUPvgkVITSFT9CvMY.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>LugaresVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-RENPA-DiplomaEuropeo</componentname>
      <componentalias>Europeo13</componentalias>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMDiplomaEuropeo?
      </mapaKML>
      <capa>DiplomaEuropeo</capa>
    </servicios>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>2</posx>
    <posy>2</posy>
    <numero servicios>1</numero servicios>
    <servicios>
      <componentname>http://acg.ual.es/enia/widgets/
      RedesSociales-REDIAM-Twitter</componentname>
      <componentalias>Twitter</componentalias>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <mapaKML/>
      <capa/>
    </servicios>
  </newComponentData>
   <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.numero servicios List Error
  </message>
</result>
```

Descripción	**Omisión del parámetro newComponent-Data.servicios.instanceId**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
  </componentInstance>
  <actionDone>groupfromdesktop</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>1</posx>
    <posy>1</posy>
    <numero servicios>3</numero servicios>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.Servicios.instanceId
    List Error
  </message>
</result>
```

© 2016 Vallecillos, J.

```
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-ViasPecuarias-InventarioVVPP</componentname>
         <componentalias>InventarioVVPP</componentalias>
         <mapaKML>http://www.juntadeandalucia.es/
         medioambiente/mapwms/REDIAMInventarioVVPP?
         </mapaKML>
         <capa>InventarioVVPP</capa>
      </servicios>
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-ViasPecuarias-LugaresVVPP</componentname>
         <componentalias>LugaresVVPP</componentalias>
         <instanceId>0H2Y40tUByKUPvgkVITSFT9CvMY.eq.
         </instanceId>
         <mapaKML>http://www.juntadeandalucia.es/medioambiente/
         mapwms/REDIAMInventarioVVPP?</mapaKML>
         <capa>LugaresVVPP</capa>
      </servicios>
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-RENPA-DiplomaEuropeo</componentname>
         <componentalias>Europeo13</componentalias>
         <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
         </instanceId>
         <mapaKML>http://www.juntadeandalucia.es/
         medioambiente/mapwms/REDIAMDiplomaEuropeo?
         </mapaKML>
         <capa>DiplomaEuropeo</capa>
      </servicios>
   </newComponentData>
   <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>2</posx>
      <posy>2</posy>
      <numero servicios>1</numero servicios>
       <servicios>
         <componentname>http://acg.ual.es/enia/widgets/
         RedesSociales-REDIAM-Twitter</componentname>
         <componentalias>Twitter</componentalias>
         <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
         </instanceId>
         <mapaKML/>
         <capa/>
      </servicios>
   </newComponentData>
   <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
   </interaction>
</params>
```

Descripción	Parámetro newComponentData. servicios.instanceId vacío	Resultado	ok

Parámetros de entrada

```
<params>
   <userId>1</userId>
   <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
   </componentInstance>
   <actionDone>groupfromdesktop</actionDone>
   <newComponentData>
      <instanceId></instanceId>
      <posx>1</posx>
      <posy>1</posy>
      <numero servicios>3</numero servicios>
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-ViasPecuarias-InventarioVVPP</componentname>
         <componentalias>InventarioVVPP</componentalias>
         <mapaKML>http://www.juntadeandalucia.es/
         medioambiente/mapwms/REDIAMInventarioVVPP?
         </mapaKML>
         <capa>InventarioVVPP</capa>
      </servicios>
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-ViasPecuarias-LugaresVVPP</componentname>
         <componentalias>LugaresVVPP</componentalias>
         <instanceId>0H2Y40tUByKUPvgkVITSFT9CvMY.eq.
         </instanceId>
         <mapaKML>http://www.juntadeandalucia.es/medioambiente/
         mapwms/REDIAMInventarioVVPP?</mapaKML>
         <capa>LugaresVVPP</capa>
      </servicios>
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-RENPA-DiplomaEuropeo</componentname>
         <componentalias>Europeo13</componentalias>
         <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
         </instanceId>
         <mapaKML>http://www.juntadeandalucia.es/
         medioambiente/mapwms/REDIAMDiplomaEuropeo?
         </mapaKML>
         <capa>DiplomaEuropeo</capa>
      </servicios>
   </newComponentData>
```

Valores de salida

```
<result>
   <allowed>false</allowed>
   <oldComponentData>...</oldComponentData>
      ...
   <oldComponentData>...</oldComponentData>
   <message>Not found or Empty News
      Components.Servicios.instanceId
      List Error
   </message>
</result>
```

```
    <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>2</posx>
      <posy>2</posy>
      <numero servicios>1</numero servicios>
       <servicios>
         <componentname>http://acg.ual.es/enia/widgets/
         RedesSociales-REDIAM-Twitter</componentname>
         <componentalias>Twitter</componentalias>
         <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
         </instanceId>
         <mapaKML/>
         <capa/>
      </servicios>
    </newComponentData>
    <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
    </interaction>
  </params>
```

Descripción	Omisión del parámetro newComponentData.servicios.componentName	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
  <userId>1</userId>
  <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
  </componentInstance>
  <actionDone>groupfromdesktop</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>1</posx>
    <posy>1</posy>
    <numero servicios>3</numero servicios>
    <servicios>
      <componentalias>InventarioVVPP</componentalias>
      <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/medioambiente/
      mapwms/REDIAMInventarioVVPP?</mapaKML>
      <capa>InventarioVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-LugaresVVPP</componentname>
      <componentalias>LugaresVVPP</componentalias>
      <instanceId>0H2Y40tUByKUPvgkVITSFT9CvMY.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>LugaresVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-RENPA-DiplomaEuropeo</componentname>
      <componentalias>Europeo13</componentalias>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMDiplomaEuropeo?
      </mapaKML>
      <capa>DiplomaEuropeo</capa>
    </servicios>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>2</posx>
    <posy>2</posy>
    <numero servicios>1</numero servicios>
    <servicios>
      <componentname>http://acg.ual.es/enia/widgets/
      RedesSociales-REDIAM-Twitter</componentname>
      <componentalias>Twitter</componentalias>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <mapaKML/>
      <capa/>
    </servicios>
  </newComponentData>
    <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida:

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
     ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    Components.Servicios.componentname
    List Error
  </message>
</result>
```

© 2016 Vallecillos, J.

Descripción	Parámetro newComponentData.servicios.componentName vacío	Resultado	ok
Parámetros de entrada		**Valores de salida**	

```
<params>
  <userId>1</userId>
  <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
  </componentInstance>
  <actionDone>groupfromdesktop</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>1</posx>
    <posy>1</posy>
    <numero servicios>3</numero servicios>
    <servicios>
      <componentname></componentname>
      <componentalias>InventarioVVPP</componentalias>
      <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>InventarioVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-LugaresVVPP</componentname>
      <componentalias>LugaresVVPP</componentalias>
      <instanceId>0H2Y40tUByKUPvgkVITSFT9CvMY.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/medioambiente/
      mapwms/REDIAMInventarioVVPP?</mapaKML>
      <capa>LugaresVVPP</capa>
    </servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-RENPA-DiplomaEuropeo</componentname>
      <componentalias>Europeo13</componentalias>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMDiplomaEuropeo?
      </mapaKML>
      <capa>DiplomaEuropeo</capa>
    </servicios>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>2</posx>
    <posy>2</posy>
    <numero servicios>1</numero servicios>
    <servicios>
      <componentname>http://acg.ual.es/enia/widgets/
      RedesSociales-REDIAM-Twitter</componentname>
      <componentalias>Twitter</componentalias>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <mapaKML/>
      <capa/>
    </servicios>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    newComponentsData.servicios.componentName
    List Error
  </message>
</result>
```

Descripción	Omisión del parámetro newComponentData.servicios.componentalias	Resultado	ok
Parámetros de entrada		**Valores de salida**	

```
<params>
  <userId>1</userId>
  <componentInstance>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
  </componentInstance>
  <actionDone>groupfromdesktop</actionDone>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>1</posx>
    <posy>1</posy>
    <numero servicios>3</numero servicios>
    <servicios>
      <componentname>http://acg.ual.es/wookie/widgets/
      OGC-ViasPecuarias-InventarioVVPP</componentname>
      <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <mapaKML>http://www.juntadeandalucia.es/
      medioambiente/mapwms/REDIAMInventarioVVPP?
      </mapaKML>
      <capa>InventarioVVPP</capa>
    </servicios>
```

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Not found or Empty News
    newComponentsData.servicios.componentAlias
    List Error
  </message>
</result>
```

```
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-ViasPecuarias-LugaresVVPP</componentname>
         <componentalias>LugaresVVPP</componentalias>
         <instanceId>0H2Y40tUByKUPvgkVITSFT9CvMY.eq.
         </instanceId>
         <mapaKML>http://www.juntadeandalucia.es/
         medioambiente/mapwms/REDIAMInventarioVVPP?
         </mapaKML>
         <capa>Lugares VVPP</capa>
      </servicios>
      <servicios>
         <componentname>http://acg.ual.es/wookie/widgets/
         OGC-RENPA-DiplomaEuropeo</componentname>
         <componentalias>Europeo13</componentalias>
         <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
         </instanceId>
         <mapaKML>http://www.juntadeandalucia.es/
         medioambiente/mapwms/REDIAMDiplomaEuropeo?
         </mapaKML>
         <capa>Diploma Europeo</capa>
      </servicios>
   </newComponentData>
   <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>2</posx>
      <posy>2</posy>
      <numero servicios>1</numero servicios>
      <servicios>
         <componentname>http://acg.ual.es/enia/widgets/
         RedesSociales-REDIAM-Twitter</componentname>
         <componentalias>Twitter</componentalias>
         <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
         </instanceId>
         <mapaKML/>
         <capa/>
      </servicios>
   </newComponentData>
   <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
   </interaction>
</params>
```

Descripción	**Omisión del parámetro deviceType**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
   <userId>1</userId>
   <actionDone>changeproperty </actionDone>
   <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
   </componentInstance>
   <newComponentData>
      <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <posx>4</posx>
      <posy>4</posy>
   </newComponentData>
   <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>5</posx>
      <posy>5</posy>
   </newComponentData>
   <newComponentData>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <posx>6</posx>
      <posy>6</posy>
   </newComponentData>
   <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
      </interaction>
</params>
```

Valores de salida

```
<result>
   <allowed>false</allowed>
   <oldComponentData>...</oldComponentData>
      ...
   <oldComponentData>...</oldComponentData>
   <message>Not found Interaction
      Information Error
   </message>
</result>
```

Descripción	**Fallo interno del código, se ha provocado un error en el nombre del módulo al cual se llama**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
   <userId>1</userId>
   <actionDone>changeproperty </actionDone>
   <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
   </componentInstance>
   <newComponentData>
         <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
         </instanceId>
         <posx>4</posx>
         <posy>4</posy>
      </newComponentData>
```

Valores de salida

```
<result>
   <allowed>false</allowed>
   <oldComponentData>...</oldComponentData>
      ...
   <oldComponentData>...</oldComponentData>
   <message>Internal Server Error
   </message>
</result>
```

© 2016 Vallecillos, J.

```
    <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>5</posx>
      <posy>5</posy>
    </newComponentData>
    <newComponentData>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <posx>6</posx>
      <posy>6</posy>
    </newComponentData>
    <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
    </interaction>
  </params>
```

Descripción	**Fallo en la Base de Datos de modelos de arquitecturas concretas**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Error in Architectural Models BD
  </message>
</result>
```

Descripción	**Fallo en la conexión en Wookie**	**Resultado**	**ok**

Parámetros de entrada

```
<params>
  <userId>1</userId>
  <actionDone>changeproperty </actionDone>
  <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
  </componentInstance>
  <newComponentData>
    <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
    </instanceId>
    <posx>4</posx>
    <posy>4</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
    </instanceId>
    <posx>5</posx>
    <posy>5</posy>
  </newComponentData>
  <newComponentData>
    <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
    </instanceId>
    <posx>6</posx>
    <posy>6</posy>
  </newComponentData>
  <interaction>
    <deviceType>Browser </deviceType>
    <interactionType>MouseKeyboard
    </interactionType>
    <latitude>36.827087399999996 </latitude>
    <longitude>-2.435696 </longitude>
  </interaction>
</params>
```

Valores de salida

```
<result>
  <allowed>false</allowed>
  <oldComponentData>...</oldComponentData>
    ...
  <oldComponentData>...</oldComponentData>
  <message>Error in Wookie
  </message>
</result>
```

Descripción	**Fallo en la conexión a la Base de Datos de registro de interacción**	**Resultado**	**ok**

Parámetros de entrada | **Valores de salida**

```
<params>
   <userId>1</userId>
   <actionDone>changeproperty </actionDone>
   <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
   </componentInstance>
   <newComponentData>
      <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <posx>4</posx>
      <posy>4</posy>
   </newComponentData>
   <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>5</posx>
      <posy>5</posy>
   </newComponentData>
   <newComponentData>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <posx>6</posx>
      <posy>6</posy>
   </newComponentData>
   <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
   </interaction>
</params>
```

```
<result>
   <allowed>false</allowed>
   <oldComponentData>...</oldComponentData>
      ...
   <oldComponentData>...</oldComponentData>
   <message>Error in Register Interaction
   </message>
</result>
```

Descripción	Fallo en la conexión a la Base de Datos de especificación de componentes	Resultado	ok
Parámetros de entrada		**Valores de salida**	

```
<params>
   <userId>1</userId>
   <actionDone>changeproperty </actionDone>
   <componentInstance>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
   </componentInstance>
   <newComponentData>
      <instanceId>IwdhmgBVWypAXutmmprCBkis9Jg.eq.
      </instanceId>
      <posx>4</posx>
      <posy>4</posy>
   </newComponentData>
   <newComponentData>
      <instanceId>mX01AZe07Qbr8o3S0fN5Yw.pl.gt4c.eq.
      </instanceId>
      <posx>5</posx>
      <posy>5</posy>
   </newComponentData>
   <newComponentData>
      <instanceId>3DvB8U.sl.AyUv2hspufr4x6bdsq9Q.eq.
      </instanceId>
      <posx>6</posx>
      <posy>6</posy>
   </newComponentData>
   <interaction>
      <deviceType>Browser </deviceType>
      <interactionType>MouseKeyboard
      </interactionType>
      <latitude>36.827087399999996 </latitude>
      <longitude>-2.435696 </longitude>
   </interaction>
</params>
```

```
<result>
   <allowed>false</allowed>
   <oldComponentData>...</oldComponentData>
      ...
   <oldComponentData>...</oldComponentData>
   <message>Error in component
      specification BD
   </message>
</result>
```

A.4. Pruebas de Interaction Service

Para el servicio público *Session Service*, a continuación se observa cuáles son las pruebas y errores obtenidos para cada operación.

A.4.1. Operación Register interaction

Descripción	Parámetro userId vacío	Resultado	ok
Parámetros de entrada		**Valores de salida**	

```
<params>
   <userId></userId>
   <newSession>1</newSession>
   <interaction>
      <deviceType>Browser</deviceType>
      <interactionType>MouseKeyboard
         </interactionType>
      <latitude>1</latitude>
      <longitude>1</longitude>
   </interaction>
```

```
<result>
   <registered>false</registered>
   <message>Not found or Empty
      userId Error</message>
</result>
```

© 2016 Vallecillos, J.

```
      <operationPerformed>PruebaLibre
        </operationPerformed>
      <componentId>PruebaLibre</componentId>
      <groupComponent>PruebaLibre</groupComponent>
      <ungroupComponent>PruebaLibre</ungroupComponent>
      <ungroupComponent>PruebaLibre</ungroupComponent>
      <ungroupComponent>PruebaLibre</ungroupComponent>
      <cotsget>
         <platform>Web</platform>
         <componentname>PruebaLibre </componentname>
         <componentAlias>InventarioVVPP</componentAlias>
         <instanceId>PruebaLibre </instanceId>
         <codeHTML><![CDATA[<iframe></iframe>]]>
            </codeHTML>
         <objectJava/>
         <jarJava/>
         <idHtml>iniwidget3</idHtml>
         <posx>11</posx>
         <posy>11</posy>
         <tamanox>6</tamanox>
         <tamanoy>6</tamanoy>
         <servicio maximizable>1
            </servicio maximizable>
         <servicio agrupable>1</servicio agrupable>
         <numero servicios>1</numero servicios>
         <servicios>
            <componentname>PruebaLibre
              </componentname>
            <componentalias>PruebaLibre
              </componentalias>
            <instanceId>PruebaLibre </instanceId>
            <mapaKML>PruebaLibre </mapaKML>
            <capa>PruebaLibre </capa>
         </servicios>
      </cotsget>
   </params>
```

Descripción	**Omisión del parámetro userId**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
   <newSession>1</newSession>
   <interaction>
      <deviceType>Browser</deviceType>
      <interactionType>MouseKeyboard</interactionType>
      <latitude>1</latitude>
      <longitude>1</longitude>
   </interaction>
   <operationPerformed>PruebaLibre</operationPerformed>
   <componentId>PruebaLibre</componentId>
   <groupComponent>PruebaLibre</groupComponent>
   <ungroupComponent>PruebaLibre</ungroupComponent>
   <ungroupComponent>PruebaLibre</ungroupComponent>
   <ungroupComponent>PruebaLibre</ungroupComponent>
   <cotsget>
      <platform>Web</platform>
      <componentname>PruebaLibre </componentname>
      <componentAlias>InventarioVVPP</componentAlias>
      <instanceId>PruebaLibre </instanceId>
      <codeHTML><![CDATA[<iframe></iframe>]]></codeHTML>
      <objectJava/>
      <jarJava/>
      <idHtml>iniwidget3</idHtml>
      <posx>11</posx>
      <posy>11</posy>
      <tamanox>6</tamanox>
      <tamanoy>6</tamanoy>
      <servicio maximizable>1</servicio maximizable>
      <servicio agrupable>1</servicio agrupable>
      <numero servicios>1</numero servicios>
      <servicios>
         <componentname>PruebaLibre </componentname>
         <componentalias>PruebaLibre </componentalias>
         <instanceId>PruebaLibre </instanceId>
         <mapaKML>PruebaLibre </mapaKML>
         <capa>PruebaLibre </capa>
      </servicios>
   </cotsget>
</params>
```

Valores de salida:

```
<result>
   <registered>false</registered>
   <message>Not found or Empty userId Error</message>
</result>
```

Descripción	**Omisión del parámetro operationPerformed**	**Resultado**	**ok**
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
   <userId></userId>
   <newSession>1</newSession>
   <interaction>
      <deviceType>Browser</deviceType>
      <interactionType>MouseKeyboard</interactionType>
      <latitude>1</latitude>
      <longitude>1</longitude>
   </interaction>
   <componentId>PruebaLibre</componentId>
```

Valores de salida:

```
<result>
   <registered>false</registered>
   <message>Not found or Empty userId Error</message>
</result>
```

```
    <groupComponent>PruebaLibre</groupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <cotsget>
      <platform>Web</platform>
      <componentname>PruebaLibre </componentname>
      <componentAlias>InventarioVVPP</componentAlias>
      <instanceId>PruebaLibre </instanceId>
      <codeHTML><![CDATA[<iframe></iframe>]]></codeHTML>
      <objectJava/>
      <jarJava/>
      <idHtml>iniwidget3</idHtml>
      <posx>11</posx>
      <posy>11</posy>
      <tamanox>6</tamanox>
      <tamanoy>6</tamanoy>
      <servicio maximizable>1</servicio maximizable>
      <servicio agrupable>1</servicio agrupable>
      <numero servicios>1</numero servicios>
      <servicios>
        <componentname>PruebaLibre </componentname>
        <componentalias>PruebaLibre </componentalias>
        <instanceId>PruebaLibre </instanceId>
        <mapaKML>PruebaLibre </mapaKML>
        <capa>PruebaLibre </capa>
      </servicios>
    </cotsget>
  </params>
```

Descripción	Parámetro operationPerformed vacío	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
  <userId></userId>
  <newSession>1</newSession>
  <interaction>
    <deviceType>Browser</deviceType>
    <interactionType>MouseKeyboard</interactionType>
    <latitude>1</latitude>
    <longitude>1</longitude>
  </interaction>
  <operationPerformed></operationPerformed>
  <componentId>PruebaLibre</componentId>
  <groupComponent>PruebaLibre</groupComponent>
  <ungroupComponent>PruebaLibre</ungroupComponent>
  <ungroupComponent>PruebaLibre</ungroupComponent>
  <ungroupComponent>PruebaLibre</ungroupComponent>
  <cotsget>
    <platform>Web</platform>
    <componentname>PruebaLibre </componentname>
    <componentAlias>InventarioVVPP</componentAlias>
    <instanceId>PruebaLibre </instanceId>
    <codeHTML><![CDATA[<iframe></iframe>]]></codeHTML>
    <objectJava/>
    <jarJava/>
    <idHtml>iniwidget3</idHtml>
    <posx>11</posx>
    <posy>11</posy>
    <tamanox>6</tamanox>
    <tamanoy>6</tamanoy>
    <servicio maximizable>1</servicio maximizable>
    <servicio agrupable>1</servicio agrupable>
    <numero servicios>1</numero servicios>
    <servicios>
      <componentname>PruebaLibre </componentname>
      <componentalias>PruebaLibre </componentalias>
      <instanceId>PruebaLibre </instanceId>
      <mapaKML>PruebaLibre </mapaKML>
      <capa>PruebaLibre </capa>
    </servicios>
  </cotsget>
</params>
```

Valores de salida:

```
<result>
  <registered>false</registered>
  <message>Not found or Empty
    userId Error</message>
</result>
```

Descripción	Fallo en la conexión a la Base de Datos, se ha cambiado los parámetros de conexión	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
  <userId>1</userId>
  <newSession>1</newSession>
  <interaction>
    <deviceType>Browser</deviceType>
    <interactionType>MouseKeyboard</interactionType>
    <latitude>1</latitude>
    <longitude>1</longitude>
  </interaction>
  <operationPerformed>PruebaLibre</operationPerformed>
  <componentId>PruebaLibre</componentId>
  <groupComponent>PruebaLibre</groupComponent>
```

Valores de salida:

```
<result>
  <registered>false</registered>
  <message>java.sql.SQLException:
    org.postgresql.util.PSQLException:
    FATAL: no existe la base de datos
  </message>
</result>
```

© 2016 Vallecillos, J.

```
    <componentId>PruebaLibre</componentId>
    <groupComponent>PruebaLibre</groupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <cotsget>
      <platform>Web</platform>
      <componentname>PruebaLibre </componentname>
      <componentAlias>InventarioVVPP</componentAlias>
      <instanceId>PruebaLibre </instanceId>
      <codeHTML><![CDATA[<iframe></iframe>]]></codeHTML>
      <objectJava/>
      <jarJava/>
      <idHtml>iniwidget3</idHtml>
      <posx>11</posx>
      <posy>11</posy>
      <tamanox>6</tamanox>
      <tamanoy>6</tamanoy>
      <servicio maximizable>1</servicio maximizable>
      <servicio agrupable>1</servicio agrupable>
      <numero servicios>1</numero servicios>
      <servicios>
        <componentname>PruebaLibre </componentname>
        <componentalias>PruebaLibre </componentalias>
        <instanceId>PruebaLibre </instanceId>
        <mapaKML>PruebaLibre </mapaKML>
        <capa>PruebaLibre </capa>
      </servicios>
    </cotsget>
  </params>
```

Descripción	Fallo interno de código, se ha provocado un error en las excepciones	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
  <userId>1</userId>
  <newSession>1</newSession>
  <interaction>
    <deviceType>Browser</deviceType>
    <interactionType>MouseKeyboard</interactionType>
    <latitude>1</latitude>
    <longitude>1</longitude>
  </interaction>
  <operationPerformed>PruebaLibre</operationPerformed>
  <componentId>PruebaLibre</componentId>
  <groupComponent>PruebaLibre</groupComponent>
  <ungroupComponent>PruebaLibre</ungroupComponent>
  <ungroupComponent>PruebaLibre</ungroupComponent>
  <ungroupComponent>PruebaLibre</ungroupComponent>
  <cotsget>
    <platform>Web</platform>
    <componentname>PruebaLibre </componentname>
    <componentAlias>InventarioVVPP</componentAlias>
    <instanceId>PruebaLibre </instanceId>
    <codeHTML><![CDATA[<iframe></iframe>]]></codeHTML>
    <objectJava/>
    <jarJava/>
    <idHtml>iniwidget3</idHtml>
    <posx>11</posx>
    <posy>11</posy>
    <tamanox>6</tamanox>
    <tamanoy>6</tamanoy>
    <servicio maximizable>1</servicio maximizable>
    <servicio agrupable>1</servicio agrupable>
    <numero servicios>1</numero servicios>
    <servicios>
      <componentname>PruebaLibre </componentname>
      <componentalias>PruebaLibre </componentalias>
      <instanceId>PruebaLibre </instanceId>
      <mapaKML>PruebaLibre </mapaKML>
      <capa>PruebaLibre </capa>
    </servicios>
  </cotsget>
</params>
```

Valores de salida:

```
<result>
  <registered>false</registered>
  <message>Internal Server Error
  </message>
</result>
```

Descripción	Fallo en la conexión a la Base de Datos de registro de interacción concretas	Resultado	ok
Parámetros de entrada		**Valores de salida**	

Parámetros de entrada:

```
<params>
  <userId>1</userId>
  <newSession>1</newSession>
  <interaction>
    <deviceType>Browser</deviceType>
    <interactionType>MouseKeyboard</interactionType>
    <latitude>1</latitude>
    <longitude>1</longitude>
  </interaction>
```

Valores de salida:

```
<result>
  <registered>false</registered>
  <message>Error in Register Interaction
  </message>
</result>
```

```
    <operationPerformed>PruebaLibre</operationPerformed>
    <componentId>PruebaLibre</componentId>
    <groupComponent>PruebaLibre</groupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <ungroupComponent>PruebaLibre</ungroupComponent>
    <cotsget>
        <platform>Web</platform>
        <componentname>PruebaLibre </componentname>
        <componentAlias>InventarioVVPP</componentAlias>
        <instanceId>PruebaLibre </instanceId>
        <codeHTML><![CDATA[<iframe></iframe>]]></codeHTML>
        <objectJava/>
        <jarJava/>
        <idHtml>iniwidget3</idHtml>
        <posx>11</posx>
        <posy>11</posy>
        <tamanox>6</tamanox>
        <tamanoy>6</tamanoy>
        <servicio maximizable>1</servicio maximizable>
        <servicio agrupable>1</servicio agrupable>
        <numero servicios>1</numero servicios>
        <servicios>
            <componentname>PruebaLibre </componentname>
            <componentalias>PruebaLibre </componentalias>
            <instanceId>PruebaLibre </instanceId>
            <mapaKML>PruebaLibre </mapaKML>
            <capa>PruebaLibre </capa>
        </servicios>
    </cotsget>
</params>
```


© 2016 Vallecillos, J.

Anexo B

Interfaz mashup ENIA

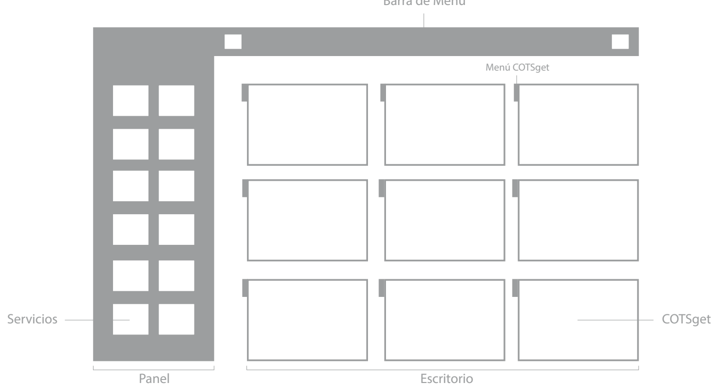
© 2016 Vallecillos, J.

Anexo B

Interfaz mashup ENIA

© 2016 Vallecillos, J.

1. Elementos de la Interfaz



En la interfaz de usuario **Enia** podemos encontrar los siguientes elementos:

COTSget: Término que hace referencia a un contenedor gráfico en el escritorio de la interfaz de usuario en el que se pueden mostrar uno o varios servicios.

Servicios: Capacidades que dispone el portal ENIA, como servicios OGC, aplicaciones independientes (apps), utilidades, redes sociales, etc.

Panel: Zona que hay en la parte izquierda de la interfaz que ofrece el catálogo de servicios.

Escritorio: Zona de visualización donde el usuario interactúa con los COTSget y/o servicios contenidos en ellos .

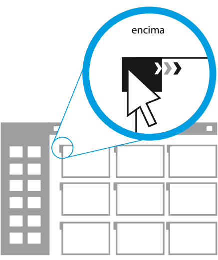
Barra de Menú: Zona superior de la interfaz de usuario, en la que podemos encontrar varios botones para realizar acciones generales.

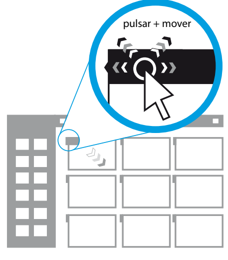
Menú COTSget: Barra de menú COSTget que aparece y donde se muestran varios botones para interactuar con el COTSget.

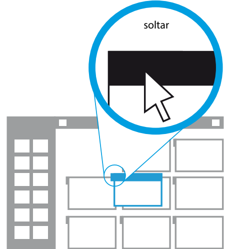
2. Acciones

Mover

Mover COTSget de un sitio a otro del Escritorio



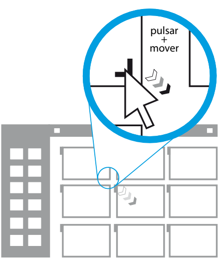


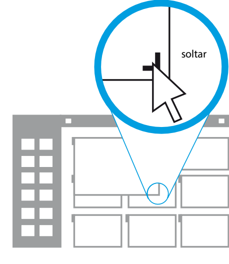


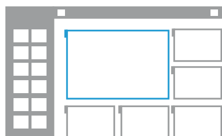
1. Situamos el puntero en el cuadro que despliega el Menú COTSget. Esperando a que se despliegue la barra completamente.
2. Hacemos click sobre cualquier parte de la barra y mantenemos pulsado mientras movemos el COTSget a la posición deseada.
3. Soltamos sobre la nueva posición y esperamos la reconfiguración del Escritorio.

Redimensionar

Cambiar el tamaño del COTSget



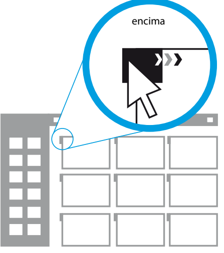


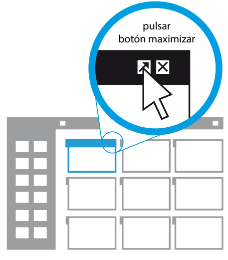


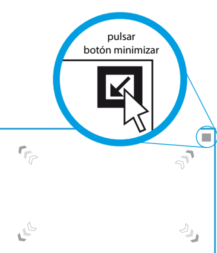
1. Situamos el puntero en el extremo inferior derecho del COTSget. Mantenemos pulsado sobre el icono de redimensión.
2. Arrastramos la esquina del COTSget hasta alcanzar el tamaño deseado.
3. Una vez soltado esperamos la reconfiguración del Escritorio.

Maximizar

Visualizar el COTSget a pantalla completa





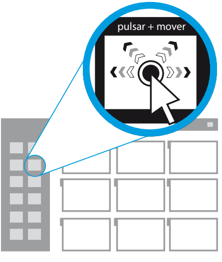


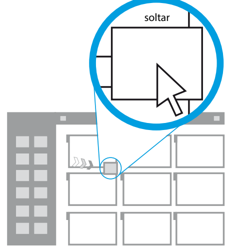
1. Situamos el puntero en el cuadro que despliega el Menú COTSget, esperando a que se despliegue la barra completamente.
2. Sin salirnos de la barra, pulsamos sobre el botón de maximizar del Menú COTSget.
3. Para volver a al vista del Escritorio, pulsamos sobre el botón minimizar.

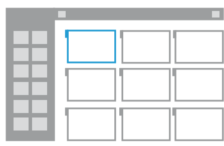
© 2016 Vallecillos, J.

Añadir

Añadir nuevos COTSget al escritorio







1. Pulsamos sobre la miniatura del servicio deseado del Panel. Sin soltar, arrastramos hasta el Escritorio.

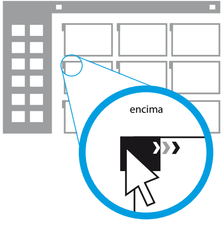
2. Soltamos sobre cualquier parte del Escritorio, excepto sobre el Menú COTSget.

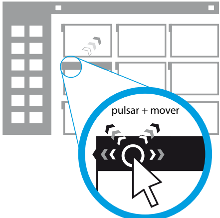
3. Se creará un COTSget nuevo que contendrá el servicio seleccionado. El escritorio se reconfigura con este cambio.

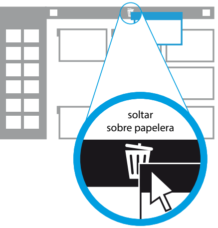
Borrar

Borrar COTSget del Escritorio. Hay dos formas de realizar esta misma acción

1. Arrastrando a la papelera de la Barra de Menú.





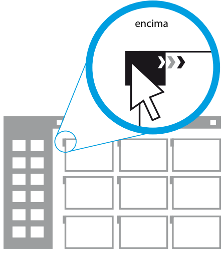


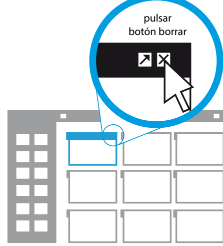
1. Situamos el puntero en el cuadro que despliega el Menú COTSget, esperando a que se despliegue la barra completamente.

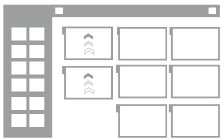
2. Hacemos click sobre cualquier parte de la barra y arrastramos el COTSget hacia el icono papelera, que ahora será visible en la Barra menú

3. Soltamos sobre el icono para borrarlo. El escritorio se reconfigura con este cambio.

2. Pulsando botón borrar en el Menú COTSget







1. Situamos el puntero en el cuadro que despliega el Menú COTSget, esperando a que se despliegue la barra completamente.

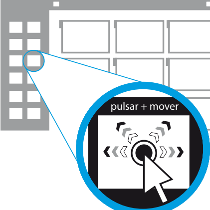
2. Sin salirnos de la barra, pulsamos sobre el botón de borrar del Menú COTSget.

3. El escritorio se reconfigura con este cambio.

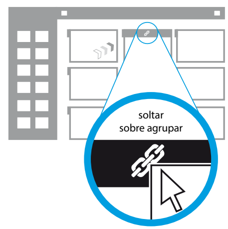
Agrupar

Combinar Servicios en un COTSgets. Se puede realizar desde dos orígenes distintos

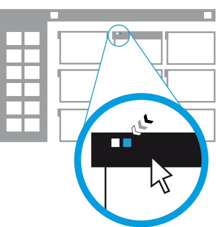
1. Arrastrando desde el Panel



1. Pulsamos sobre la miniatura del Servicio deseado del Panel. Sin soltar, arrastramos hacia el Escritorio.

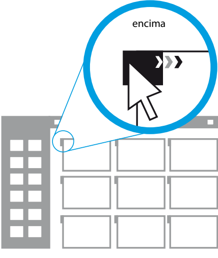


2. Soltaremos sobre el icono agrupar , que ahora será visible en el Menú COTSget, de aquellos COTSget en los que sea posible.

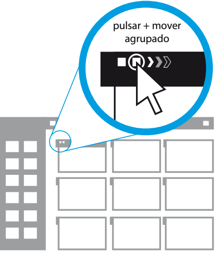


3. Se añadirá su correspondiente marcador de Servicio agrupado en el Menú COTSget.

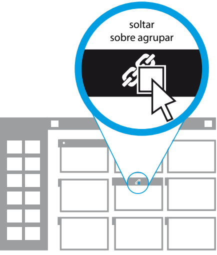
2. Desde otro COTSget



1. Situamos el puntero en el cuadro que despliega el Menú COTSget, esperando a que se despliegue la barra completamente.



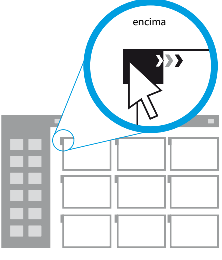
2. Pulsamos sobre el marcador de Servicio agrupado que queramos llevar a otro COTSget.



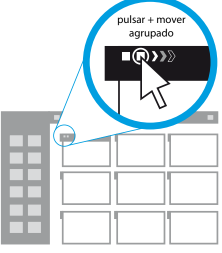
3. Soltamos sobre el icono agrupar del COTSget destino. El Servicio desaparece del COTSget original y se agrupa en el destino.

Desagrupar

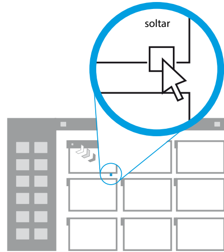
Separar Servicios combinados de un COTSget



1. Situamos el puntero en el cuadro que despliega el Menú COTSget, esperando a que se despliegue la barra completamente.



2. Pulsamos sobre el marcador de Servicio agrupado que queramos llevar a otro COTSget.

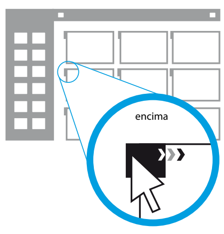


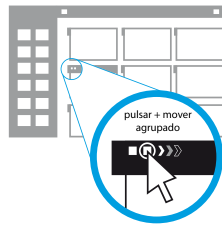
3. Soltamos sobre cualquier parte del Escritorio, excepto en los Menú COTSget. Se creará un nuevo COTSget con el Servicio correspondiente.

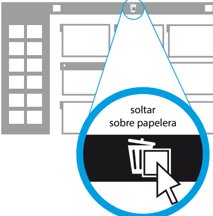
© 2016 Vallecillos, J.

Desagrupar+Borrar

Separar Servicios combinados de un COTSget y borrarlo







1. Situamos el puntero en el cuadro que despliega el Menú COTSget, esperando a que se despliegue la barra completamente.
2. Pulsamos sobre el marcador de Servicio agrupado que queramos eliminar.
3. Para borrarlo, soltamos sobre el icono papelera que ahora es visible en la Barra menú. Si es el único Servicio del COTSget, éste será eliminado del Escritorio.

Acrónimos

© 2016 Vallecillos, J.

Acrónimos

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CBSE	Component Based Software Engineering
CCM	CORBA Component Model
CIM	Computation Independent Model
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COSCORE	Cotsget based architecture operating support core
COTS	Commercial Off The Self
COTSGET	COTS Gadget
DMM	Display Management Module
DOM	Document Object Model
DSBC	Desarrollo de Software Basado en Componentes
DSL	Domain Specific Language
ECR	External Component Repositories
EDOC	Enterprise Distributed Object Computing
EJB	Enterprise JavaBeans
ENIA	ENvironmental Information Agent
IDL	Interface Description Lenguage
ISBC	Ingeniería del Software Basada en Componentes
IMM	Interaction Management Module
IU	Interfaz de Usuario

JML	Java Modeling Language
JSON	JavaScript Object Notation
MCR	Managed Component Repositories
MBE	Model Based Engineering
MDA	Model Driven Architecture
MDD	Model driven development
MDE	Model Driven Engineering
MOF	Meta Object Facility
MR	Managed Repository
OCL	Object Constraints Language
OGC	Open Geospatial Consortium
OMG	Object Management Group
ORM	Object Relational Mapping
OSGi	Open Services Gateway
PIM	Platform Independent Model
POJO	Plain Old Java Object
PSM	Platform Specific Model
QoS	Quality of Service
QoSA	Quality of Software Architecture
RDF	Resource Description Framework
REDIAM	Red de Información Medioambiental de Andalucía
REST	Representation State Transfer
ROA	Resource Oriented Architecture
RSS	Really Simple Syndication
SIG	Sistema de Información Geográfico
SOAP	Simple Object Access Protocol
SMTP	Simple Mail Transfer Protocol
TMM	Transaction Management Module
UDDI	Universal Description Discovery and Interaction
UIM	User Information Module

UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WIS	Web Based Information Systems
WSDL	Web Service Description Language
WSFL	Web Service Flow Language
XMI	XML Metadata Interchange
XML	Xtensible Markup Language

© 2016 Vallecillos, J.

Bibliografía

© 2016 Vallecillos, J.

Bibliografía

[Abiteboul et al., 2009] Abiteboul, S., Greenshpan, O., Milo, T., and Polyzotis, N. (2009). Matchup: Autocompletion for mashups. In *IEEE 25th International Conference on Data Engineering (ICDE'09)*, pages 1479–1482. IEEE.

[Aghaee et al., 2013] Aghaee, S., Pautasso, C., and De Angeli, A. (2013). Natural end-user development of web mashups. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'2013)*, pages 111–118. IEEE.

[Allsopp, 2007] Allsopp, J. (2007). *Microformats: Empowering Your Markup for Web 2.0.* friendsofED.

[Alonso et al., 2010] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2010). *Web Services: Concepts, Architectures and Applications.* Springer Publishing Company, Incorporated, 1st edition.

[Ameller, 2009] Ameller, D. (2009). Considering Non-Functional Requirements in Model-Driven Engineering. Master's thesis, Llenguatges i Sistemes Informàtics (LSI). `http://upcommons.upc.edu/pfc/handle/2099.1/7192`.

[Ardito et al., 2014] Ardito, C., Bottoni, P., Costabile, M. F., Desolda, G., Matera, M., and Picozzi, M. (2014). Creation and use of service-based distributed interactive workspaces. *Journal of Visual Languages & Computing*, 25(6):717–726.

[Barnes et al., 2014] Barnes, J. M., Garlan, D., and Schmerl, B. (2014). Evolution styles: foundations and models for software architecture evolution. *Software & Systems Modeling*, 13(2):649–678.

[Bass, 2007] Bass, L. (2007). *Software Architecture in Practice.* Pearson Education India, 2nd edition.

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice.* Addison Wesley.

[Belqasmi et al., 2011] Belqasmi, F., Glitho, R., and Fu, C. (2011). RESTful web services for service provisioning in next-generation networks: a survey. *Communications Magazine, IEEE*, 49(12):66–73.

[Benslimane et al., 2008] Benslimane, D., Dustdar, S., and Sheth, A. (2008). Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5):13–15.

[Bernstein and Haas, 2008] Bernstein, P. A. and Haas, L. M. (2008). Information Integration in the Enterprise. *Communications of the ACM*, 51(9):72–79. `http://doi.acm.org/10.1145/1378727.1378745`.

[Blair et al., 2009] Blair, G., Bencomo, N., and France, R. B. (2009). Models@run.time. *Computer*, 42(10):22–27.

[Board, 2009] Board, R. A. (2009). RSS 2.0 Specification. `http://www.rssboard.org/rss-specification`.

[Box, 1998] Box, D. (1998). *Essential COM.* Addison-Wesley Professional.

[Brown, 1999] Brown, A. (1999). Building systems from pieces with component-based software engineering. In *Constructing Superior Software*, chapter 6. Macmillan Technical Publishing, Sterling Software.

[Brunelière et al., 2010] Brunelière, H., Cabot, J., Clasen, C., Jouault, F., and Bézivin, J. (2010). Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In Kühne, T., Selic, B., Gervais, M.-P., and Terrier, F., editors, *Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science (LNCS)*, pages 32–47. Springer Berlin Heidelberg. `http://dx.doi.org/10.1007/978-3-642-13595-8_5`.

[Bruneton et al., 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The Fractal component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284.

[Buhnova et al., 2014] Buhnova, B., Vallecillo, A., Medvidovic, N., Larsson, M., López, J., and Cuellar, J. (2014). Guest editorial to the special issue on component-based software engineering and software architecture. *Science of Computer Programming*, 90:67–70.

[Buschmann et al., 2012] Buschmann, F., Ameller, D., Ayala, C. P., Cabot, J., and Franch, X. (2012). Architecture quality revisited. *Software, IEEE*, 29(4):22–24.

[Cappiello et al., 2015] Cappiello, C., Matera, M., and Picozzi, M. (2015). A ui-centric approach for the end-user development. *ACM Transactions on the Web (TWEB)*, 9(3):11.

[Cappiello et al., 2011] Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D., and Francalanci, C. (2011). DashMash: A Mashup Environment for End User Development. In Auer, S., Díaz, O., and George, editors, *Web engineering: 11th International Conference, ICWE 2011, Paphos, Cyprus, June 20-24, 2011*, volume 6757 of *Lecture Notes in Computer Science (LNCS)*, pages 152–166. Springer Berlin Heidelberg.

[Carney, 2000] Carney, D. y Long, F. (2000). What do you mean by COTS? Finally, a usefull answer. *IEEE Software*, 17(2):83–86.

[Cheng et al., 2009] Cheng, B. H., De Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., et al. (2009). Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*, pages 1–26. Springer Berlin Heidelberg.

[Chudnovskyy et al., 2013] Chudnovskyy, O., Fischer, C., Gaedke, M., and Pietschmann, S. (2013). Inter-Widget Communication by Demonstration in User Interface Mashups. In *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013*, volume 7977 of *Lecture Notes in Computer Science (LNCS)*, pages 502–505. Springer Berlin Heidelberg.

[Chudnovskyy et al., 2012] Chudnovskyy, O., Nestler, T., Gaedke, M., Daniel, F., Fernández-Villamor, J. I., Chepegin, V., Fornas, J. A., Wilson, S., Kögler, C., and Chang, H. (2012). End-user-oriented telco mashups: The omelette approach. In *Proceedings of the 21st International Conference Companion on World Wide Web*, pages 235–238. ACM.

[Cooney, 2014] Cooney, D. (2014). Introduction to Web Components. *W3C Working Group Note.*

[Criado, 2015] Criado, J. (2015). *A Trading and Model-Based Methodology for Adapting Dynamic User Interfaces.* PhD thesis, Doctoral Thesis, University of Almería.

[Crnkovic, 2001] Crnkovic, I. (2001). Component-based Software Engineering – New Challenges in Software Development. *Software Focus*, 2(4):127–133.

[Crnkovic et al., 2011] Crnkovic, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. (2011). A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615.

[Cuesta, 2002] Cuesta, C. (2002). Arquitectura de software dinámica basada en reflexión. *Universidad de Valladolid. España.*

[Curbera, 2001] Curbera, F. (2001). Web services overview. *IBM Software Group.*

[Daniel and Matera, 2014] Daniel, F. and Matera, M. (2014). *Mashups – Concepts, Models and Architectures.* Springer.

[Daniel et al., 2007] Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., and Casati, F. (2007). Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *Internet Computing, IEEE*, 11(3):59–66.

[Dean and Vigder, 1997] Dean, J. and Vigder, M. (1997). System Implementation Using Commercial off-the-shelf (COTS) Software. *NRC Publications Archive.*

© 2016 Vallecillos, J.

[DeMichiel and Keith, 2006] DeMichiel, L. and Keith, M. (2006). JSR 220: Enterprise JavaBeans 3.0. *EJB Core Contracts and Requirements, Version 3.0, Final Release.*

[Derakhshanmanesh et al., 2014] Derakhshanmanesh, M., Ebert, J., Iguchi, T., and Engels, G. (2014). Model-Integrating Software Components. In *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 3, 2014*, volume 8767 of *Lecture Notes in Computer Science (LNCS)*, pages 386–402. Springer International Publishing.

[Dhara and Leavens, 1996] Dhara, K. K. and Leavens, G. T. (1996). Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267. IEEE.

[Fan et al., 2012] Fan, K., Zhang, S., Tang, S., Wang, Y., Xu, Z., and Liu, Y. (2012). A System Architecture of Widget-Based Digital TV Interactive Platform. In *Sixth International Conference on Genetic and Evolutionary Computing (ICGEC'2012)*, pages 360–363. IEEE.

[Florian and Maristella, 2014] Florian, D. and Maristella, M. (2014). *Mashups: Concepts, Models and Architectures.* Springer Publishing Company, Incorporated.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education.

[Garlan, 2000] Garlan, D. (2000). Software architecture: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'2000)*, pages 91–101, New York, NY, USA. ACM. http://doi.acm.org/10.1145/336512.336537.

[Garlan and Schmerl, 2004] Garlan, D. and Schmerl, B. (2004). Using architectural models at runtime: Research challenges. In *Software Architecture: First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004*, volume 3047 of *Lecture Notes in Computer Science (LNCS)*, pages 200–205. Springer Berlin Heidelberg.

[Gebhardt et al., 2012] Gebhardt, H., Gaedke, M., Daniel, F., Soi, S., Casati, F., Iglesias, C. A., and Wilson, S. (2012). From mashups to Telco mashups: A survey. *IEEE Internet Computing*, 16(3):70–76.

[Gmelch and Pernul, 2012] Gmelch, O. and Pernul, G. (2012). A generic architecture for user-centric portlet integration. In *IEEE 14th International Conference on Commerce and Enterprise Computing (CEC'2012)*, pages 70–77. IEEE.

[Group, 2012] Group, W. A. W. (2012). Widgets Family of Specifications. Technical Report, W3C. *http://www.w3.org/2008/webapps/wiki/WidgetSpecs.*

[Han, 1999] Han, J. (1999). An pproach to software component specification. In *Proceedings of International Workshop on Component Based Software Engineering*, pages 97–102.

[Hassan et al., 2014] Hassan, O. A.-H., Al-Rousan, T., Taleb, A. A., and Maaita, A. (2014). An efficient and scalable ranking technique for mashups involving RSS data sources. *Journal of Network and Computer Applications*, 39:179–190.

[Heinrich et al., 2015] Heinrich, R., Schmieders, E., Jung, R., Hasselbring, W., Metzger, A., Pohl, K., and Reussner, R. (2015). Run-time architecture models for dynamic adaptation and evolution of cloud applications. *Technical Report, 1503. Department of Computer Science, Kiel, Germany.*

[Hepper, 2008] Hepper, S. (2008). Java Portlet Specification Version 2.0. *Final Release JSR 286, IBM Corp.*

[Hoyer et al., 2009] Hoyer, V., Gilles, F., Janner, T., and Stanoevska-Slabeva, K. (2009). Sap research rooftop marketplace: Putting a face on service-oriented architectures. In *World Conference on Services-I*, pages 107–114. IEEE.

[Hoyer and Stanoevska-Slabeva, 2009] Hoyer, V. and Stanoevska-Slabeva, K. (2009). Generic business model types for enterprise mashup intermediaries. In Nelson, M., Shaw, M., and Strader, T., editors, *Value Creation in E-Business Management: 15th Americas Conference on Information Systems, AMCIS 2009, SIGeBIZ track, San Francisco, CA, USA, August 6-9, 2009*, volume 36 of *Lecture Notes in Business Information Processing (LNBIP)*, pages 1–17. Springer Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-642-03132-8_1.

[Iribarne, 2003] Iribarne, L. (2003). *Un modelo de mediación para el desarrollo de software basado en componentes COTS*. Tesis Doctoral. Universidad de Almería.

[Iribarne et al., 2004] Iribarne, L., Troya, J. M., and Vallecillo, A. (2004). A trading service for cots components. *The Computer Journal*, 47(3):342–357.

[ISO/IEC, 2014] ISO/IEC (2014). ISO/IEC 19508. *Information Technology – Object Management Group – Meta Object Facility (MOF) Core.*

[Jin et al., 2008] Jin, Y., Benatallah, B., Casati, F., and Daniel, F. (2008). Understanding Mashup Development. *Internet Computing, IEEE*, 12(5):44–52.

[Kern and Kühne, 2007] Kern, H. and Kühne, S. (2007). Model interchange between aris and eclipse emf. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'2007)*, pages 1–10.

[Kilov et al., 1999] Kilov, H., Rumpe, B., and Simmonds, I. D. (1999). *Behavioral specifications of businesses and systems.* Kluwer Academic Publishers.

[Kreger et al., 2001] Kreger, H. et al. (2001). Web services conceptual architecture (wsca 1.0). *IBM Software Group*, 5:6–7.

[Meyers, 2001] Meyers, B. C. y Oberndorf, P. (2001). *Managing Software Acquisition: Open Systems and COTS Products.* The SEI Series in Software Engineering. Addison-Wesley.

© 2016 Vallecillos, J.

[Morin et al., 2009] Morin, B., Barais, O., Jezequel, J., Fleurey, F., and Solberg, A. (2009). Models@run.time to support dynamic adaptation. *Computer*, 42(10):44–51.

[Namiot and Sneps-Sneppe, 2014] Namiot, D. and Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

[Nestler et al., 2010] Nestler, T., Feldmann, M., Hübsch, G., Preußner, A., and Jugel, U. (2010). The ServFace builder-A WYSIWYG approach for building service-based applications. In Benatallah, B., Casati, F., and Rossi, G. K. G., editors, *Web Engineering: 10th International Conference, ICWE 2010, Vienna Austria, July 5-9, 2010*, volume 6189 of *Lecture Notes in Computer Science (LNCS)*, pages 498–501. Springer Berlin Heidelberg.

[OMG, 2004] OMG (2004). UML Profile For Enterprise Distributed Object Computing, Version 1.0. *OMG Document — formal/2004-02-01.*

[OMG, 2006] OMG (2006). CORBA Component Model, V4.0. *OMG Document — formal/06-04-01.*

[OpenSocial, 2011] OpenSocial (2011). OpenSocial Core Gadget Specification 2.0.1. `http://opensocial-resources.googlecode.com/svn/spec/2.0.1/Core-Gadget.xml`.

[OSGi Alliance, 2007] OSGi Alliance (2007). OSGi Service Platform Core Specification, V4.1. *Release 4, Version 4.1.*

[Papazoglou, 2008] Papazoglou, M. (2008). *Web services: principles and technology.* Pearson Education.

[Pierre et al., 2013] Pierre, D., Marc, D., and Philippe, R. (2013). Ubiquitous widgets: Designing interactions architecture for adaptive mobile applications. In *IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'2013)*, pages 331–336. IEEE.

[Richardson and Ruby, 2008] Richardson, L. and Ruby, S. (2008). *RESTful web services.* O'Reilly Media, Inc.

[Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14.

[Shirogane et al., 2008] Shirogane, J., Iwata, H., Fukaya, K., and Fukazawa, Y. (2008). Gui change method according to roles of widgets and change patterns. *IEICE Transactions on Information and Systems*, 91(4):907–920.

[Sire et al., 2009] Sire, S., Paquier, M., Vagner, A., and Bogaerts, J. (2009). A messaging API for inter-widgets communication. In *Proceedings of the 18th international conference on World wide web*, pages 1115–1116. ACM.

[Steinberg et al., 2008] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software – Beyond Object-Oriented Programming*. Pearson Education.

[Thones, 2015] Thones, J. (2015). Microservices. *Software, IEEE*, 32(1):116–116.

[Tolvanen and Kelly, 2009] Tolvanen, J.-P. and Kelly, S. (2009). Metaedit+: Defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 819–820, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1639950.1640031.

[Uckelmann et al., 2011] Uckelmann, D., Harrison, M., and Michahelles, F. (2011). An architectural approach towards the future internet of things. In *Architecting the Internet of Things*, pages 1–24. Springer.

[Vallecillo et al., 1999] Vallecillo, A., Troya, J. M., and Hernández, J. (1999). Object interoperability. In *Object-Oriented Technology ECOOP'99 Workshop Reader*, pages 1–21. Springer.

[Vallecillos et al., 2014] Vallecillos, J., Criado, J., Iribarne, L., and Padilla, N. (2014). Dynamic mashup interfaces for information systems using widgets-as-a-service. In *On the Move to Meaningful Internet Systems (OTM), Amantea, Italy, October 27-31, 2014*, volume 8842 of *Lecture Notes in Computer Science (LNCS)*, pages 438–447.

[Vallecillos et al., 2015] Vallecillos, J., Criado, J., Padilla, N., and Iribarne, L. (2015). A cloud service for cots component-based architectures. *Computer Standards & Interfaces (Elsevier)*.

[W3C, 2007] W3C (2007). Simple object access protocol (soap). *W3C Recommendation*. http://www.w3.org/TR/soap12.

[W3C, 2009] W3C (2009). Web Application Description Language. https://wadl.java.net.

[Warmer and Kleppe, 1998] Warmer, J. B. and Kleppe, A. G. (1998). *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley Professional.

[Whittle et al., 2014] Whittle, J., Hutchinson, J., and Rouncefield, M. (2014). The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85.

[Wilson et al., 2012] Wilson, S., Daniel, F., Jugel, U., and Soi, S. (2012). Orchestrated user interface mashups using W3C widgets. In Wilson, S., Daniel, F., Jugel, U., and Soi, S., editors, *Current Trends in Web Engineering: Workshops, Doctoral Symposium, and Tutorials, Held at ICWE 2011, Paphos, Cyprus, June 20-21, 2011*, volume 7059 of *Lecture Notes in Computer Science (LNCS)*, pages 49–61. Springer Berlin Heidelberg.

© 2016 Vallecillos, J.

[Yuan et al., 2014] Yuan, E., Esfahani, N., and Malek, S. (2014). Automated mining of software component interactions for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 27–36. ACM.

Este documento ha sido generado con $\LaTeX$.

Todas las *Figuras* y *Tablas* contenidas en
el presente documento son originales.

COScore: una infraestructura de servicios
para el despliegue de aplicaciones Mashup

Jesús Vallecillos Ruiz
Departamento de Informática
Grupo de Investigación de Informática Aplicada (TIC-211)
Universidad de Almería
Almería, a 16 de marzo de 2016

http://acg.ual.es

Jesús Vallecillos

COScore: una infraestructura de servicios para el despliegue de aplicaciones Mashup

En el campo de las ciencias de la computación y, en especial, en lo que respecta al desarrollo de sistemas de información, cada vez es más común la combinación de diversas tecnologías para llevar a cabo la construcción de software. La flexibilidad que aportan los sistemas software basados en componentes permite modificarlos una vez que han sido desarrollados, tanto en tareas destinadas al mantenimiento y evolución del software, como en tareas de adaptación.

El trabajo de investigación desarrollado consiste en una infraestructura de servicios para el despliegue y gestión de aplicaciones *mashup*, un tipo particular de software basado en componentes. La infraestructura da soporte a aplicaciones *mashup* de distintos dominios y que se despliegan en plataformas diferentes. El modelo de infraestructura está constituido por tres capas: cliente, dependiente e independiente de la plataforma. Las dos últimas capas constituyen el núcleo de la propuesta y juntas reciben el nombre de COScore (*COTSget-based architecture Operating Support core*).

El núcleo de la infraestructura está desarrollado como un conjunto de servicios, cada uno de los cuales agrupa una serie de operaciones relacionadas entre sí. De esta manera, el COScore ofrece toda la funcionalidad necesaria para el despliegue de aplicaciones *mashup* y, además, permite realizar todas las acciones de gestión relacionadas con el uso de dichas aplicaciones. La propuesta ha sido evaluada y validada con ENIA, un sistema de información constituido por interfaces de usuario *mashup* que se despliegan en la Web.