FACULTAD 3

Desarrollo de un mecanismo de integración entre componentes para el marco de trabajo Sauxe

TRABAJO DE DIPLOMA PARA OPTAR POR EL TÍTULO DE

INGENIERO EN CIENCIAS INFORMÁTICAS

Autora del trabajo: Patricia Wilson Hernández.

Tutora: Ing. Lilian Álvarez Almanza.

Cotutores: Ing. Damián Pérez Alfonso.

Ing. Javier Ruiz Durán.

La Habana, Cuba

2012

Declaración de autoría

Declaro que soy la única autora de este trabajo y autorizo al Centro de Informatización de la Gestión de Entidades de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

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

Patricia Wilson Hernández	Ing. Lilian Álvarez Almanza
Autora	**Tutora**
Ing. Damián Pérez Alfonso	Ing. Javier Ruiz Durán
Co-tutor	**Co-tutor**

“La confianza en uno mismo es el primer peldaño para ascender por la escalera del éxito.

Ralph Waldo Emerson

Agradecimientos

Agradezco desmedidamente a mis tutores, ya que sin su apoyo no hubiese sido posible la terminación de este trabajo. Les agradezco por las críticas (siempre constructivas) y sobre todo por alentarme en todo momento a dar lo mejor de mí.

A mi compañero de equipo Carlitos (como cariñosamente le digo), muchísimas gracias, ya que sin su ayuda definitivamente no hubiese sido posible que me graduara. Le deseo éxitos en su vida como profesional.

A mi Yai, la mejor de las amistades que he hecho durante estos cinco años, con sus berrinches y sus malos días, con sus días buenos y sus alborotos, por sus consejos, por entenderme, apoyarme y acompañarme en los malos momentos, y también en los buenos, por estar disponible para mí siempre, aunque a veces a distancia.

A Yangzet, que aunque ya no la tengo cerquita de mí, le agradezco esos consejos que siempre me da, y esa actitud positiva que le pone a todo.

A mi familia y mi novio por su paciencia, por apoyarme en todo momento y por su espera, al ser yo en repetidas ocasiones el porqué de los cambios de planes.

Agradezco a todos aquellos que de una forma u otra hicieron posible mi llegada al final de este largo y complicado camino que ha sido esta carrera, a todos los profesores que tuve, a mis compañeras de todos los apartamentos por los que he pasado, a mis compañeros de brigada, los nuevos y los viejos, a las amistades que he hecho durante la carrera y a mis compañeros del proyecto, sin su apoyo en estos cinco años, sin su ayuda, mucha o poca, no hubiese sido posible llegar hasta aquí.

Dedicatoria

Dedico este trabajo a mi queridísima y enorme familia, en especial a los viejitos más lindos de todo el mundo, Josefa, Juana y Marco Antonio, mis adorados abuelos, y a mis padres, que son mi más grande inspiración.

Resumen

El Departamento de Tecnología, del Centro de Informatización de la Gestión de Entidades en la Universidad de las Ciencias Informáticas, centra su labor en el desarrollo de un marco de trabajo que se ha denominado Sauxe. Este marco de trabajo es una base tecnológica para el desarrollo de aplicaciones web de gestión, que provee aspectos como la seguridad, la posibilidad de uso para múltiples entidades y el soporte para varios idiomas. Dados los beneficios que Sauxe proporciona es utilizado en varios proyectos de la UCI y entidades desarrolladoras de software del país. El principal sistema que utiliza este marco de trabajo es Cedrux, solución genérica para la gestión de entidades, aplicable a todos los sectores del país.

Sauxe provee un mecanismo de integración a través del cual los módulos intercambian servicios, dado que para ejecutar con éxito el objetivo de cualquier sistema sus módulos deben comunicarse. La principal limitante de este mecanismo es su actual concepción, que solo permite establecer dos niveles para la integración entre los componentes, lo cual impide el desglose en más niveles de las aplicaciones desarrolladas sobre el marco de trabajo.

Esta tesis se centra en la propuesta de un nuevo módulo de integración para Sauxe, que permita la división en módulos de sus componentes hasta el nivel que se desee, y que elimine además las actuales deficiencias en cuanto a la configuración de los componentes, ampliando sus posibilidades de reutilización.

Palabras claves: componente, framework, integración, marco de trabajo.

Índice de contenido

Introducción 1

Capítulo I: Fundamentación teórica 5

1.1. Introducción...5

1.2. Conceptos fundamentales 5

1.2.1. Componente 5

1.2.2. Arquitectura de software 8

1.2.3. Framework 9

1.3. Integración entre componentes en diversos framework 12

1.3.1. Symfony 12

1.3.2. Zend 14

1.3.3. Spring 14

1.3.4. OSGi 16

1.3.5. Spring Dinamic Modules 18

1.4. Metodología de desarrollo 19

1.4.1. Metodologías Tradicionales o Pesadas 20

1.4.2. Metodologías Ágiles 20

1.4.3. Características del modelo 20

1.5. Herramientas...22

1.5.1. Visual Paradigm 8.0 22

1.5.2. NetBeans 7.1 23

1.5.3. Apache 2.2.9 23

1.6. Conclusiones parciales 23

Capítulo II: Características del sistema. Análisis y diseño de la solución. 24
2.1. Introducción....24
2.2. Descripción del mecanismo de integración de Sauxe....24
2.2.1. Modelo conceptual del negocio 26
2.3. Características de la solución propuesta 28
2.3.1. Modelo conceptual del sistema 30
2.4. Requisitos de la solución....32
2.4.1. Requisitos funcionales 32
2.4.2. Requisitos no funcionales 33
2.4.3. Descripción de los requisitos funcionales de la solución 33
2.5. Patrones de diseño 36
2.5.1. GRASP 37
2.5.2. GoF....37
2.6. Diagramas de clases del diseño....38
2.7. Validación del diseño propuesto....39
2.7.1. Métrica Tamaño operacional de clase (TOC) 40
2.7.2. Métrica Relación entre clases (RC)....41
2.8. Conclusiones parciales 43
Capítulo III: Implementación y pruebas....44
1.1. Introducción....44
1.2. Implementación....44
1.2.1. Resolviendo dependencias 44
1.2.2. Integrando componentes....48

1.3. Prueba 49
1.3.1. Pruebas de caja blanca 49
1.3.2. Pruebas de caja negra 50
1.3.3. Diseño de casos de prueba 51
1.3.4. Pruebas aplicadas a la solución 51
1.4. Conclusiones parciales 53
Conclusiones generales 54
Recomendaciones 55
Bibliografía 56
Glosario de términos 59
Anexos 60

Índice de imágenes

Imagen 1 Relación modularidad-coste de software. 7
Imagen 2 Ejemplo de configuración de un servicio en el fichero config.yml. 13
Imagen 3 Ejemplo de configuración de un servicio en el fichero services.yml. 13
Imagen 4 Ejemplo de invocación a un servicio. 13
Imagen 5 Ejemplo de configuración de componentes con el contenedor de Spring. 15
Imagen 6 Ejemplo de integración entre componentes en Spring. 15
Imagen 7 Capas de la arquitectura de OSGi. 16
Imagen 8 Ejemplo de configuración a través del fichero MANIFEST.MF. 18
Imagen 9 Niveles de integración que establece el mecanismo de integración de Sauxe. 24
Imagen 10 Estructura del fichero de configuración ioc.xml. 25
Imagen 11 Ejemplo de utilización de integración externa. 26
Imagen 12 Ejemplo de utilización de integración interna. 26
Imagen 13 Modelo conceptual. 26
Imagen 14 Estructura del contenido en el fichero bundle.xml. 29
Imagen 15 Modelo conceptual del sistema. 30
Imagen 16 Diagrama de clases de un componente. 38
Imagen 17 Diagrama de clases del mecanismo de integración. 39
Imagen 18 Algoritmo que separa los componentes según su estado. 45
Imagen 19 Algoritmo que resuelve las dependencias. 45
Imagen 20 Algoritmo que compara la equivalencia entre interfaces. 46

Imagen 21 Algoritmo que completa los datos de las dependencias. 47

Imagen 22 Algoritmo que entrega el resultado del servicio solicitado...48

Imagen 23 Grafo de flujo del algoritmo. 51

Índice de tablas

Tabla 1 Descripción textual del requisito Localizar componentes...33

Tabla 2 Descripción textual del requisito Persistir la configuración de los componentes. 34

Tabla 3 Descripción textual del requisito Resolver dependencias de los componentes.............................35

Tabla 4 Descripción textual del requisito Permitir instanciación de componentes desde controladores del marco de trabajo. 35

Tabla 5 Descripción textual del requisito Establecer la comunicación entre componentes. 36

Tabla 6 Criterios para evaluar la métrica TOC. 40

Tabla 7 Procedimientos por clase. 40

Tabla 8 Resultados obtenidos de la evaluación de los atributos de calidad. 41

Tabla 9 Criterios para evaluar la métrica RC...41

Tabla 10 Relaciones de uso por clase. 42

Tabla 11 Resultados obtenidos de la evaluación de los atributos de calidad. 42

Introducción

En las últimas décadas los sistemas informáticos han impuesto un nuevo orden, y esto acompañado del acelerado desarrollo de las tecnologías, ha provocado que las empresas deseen informatizar sus procesos y que se interesen más por la utilización de soluciones informáticas que permitan una interacción rápida y amigable con los usuarios.

En Cuba se lleva a cabo hace varios años un proceso de informatización de la sociedad. En este proceso la Universidad de las Ciencias Informáticas (UCI) tiene un papel protagónico, informatizando los procesos que se realizan en los distintos sectores económico-sociales del país.

En la UCI el desarrollo de software se lleva a cabo a través de proyectos organizados en varios centros, entre los que se encuentra el Centro de Informatización de la Gestión de Entidades (CEIGE). En este se ha desarrollado un marco de trabajo para el desarrollo de aplicaciones web para la gestión de entidades.

Este marco de trabajo, denominado Sauxe, brinda funcionalidades como seguridad, multi-entidad, multi-tema, multi-idioma, integración, interoperabilidad, administración de transacciones, entre otras, y es utilizado en diversos proyectos de la UCI y en entidades desarrolladoras de software del país: (Morejón, Baryolo y García, 2010)

Algunos de los proyectos de la UCI:

- Fuerza de Trabajo Calificada del MEP (Facultad 1),
- Proyecto de Informatización de los Tribunales Populares Cubanos de CEGEL (Facultad 3),
- Proyecto SEUNE (Facultad 5),

Entidades Externas:

- DTS (MININT);
- Academia de las FAR;
- DESOFT (I+D+I[1]), Habana, Camagüey y Holguín;
- Centro de desarrollo de Holguín y de Santiago de Cuba.

[1] Por las siglas de Investigación, Desarrollo e Innovación.

El principal sistema que utiliza este marco de trabajo es el sistema Cedrux, solución genérica para la gestión de entidades, aplicable a todos los sectores del país.

Las aplicaciones suelen estar divididas en módulos, los cuales son secciones de un sistema con funcionalidades específicas. (Parnas, 1972) Estos módulos necesitan interactuar, al igual que interactúan las distintas áreas de una empresa para efectuar exitosamente un proceso, es por ello que el marco de trabajo Sauxe provee un mecanismo de comunicación entre módulos, nombrado IoC (por las siglas en inglés de Inversión de Control), a través del cual consumen servicios entre ellos.

En la actualidad, dada la estructura del IoC, es complicado conocer los servicios que brinda y/o consume un componente lo cual influye en las decisiones de reutilización y portabilidad de los componentes y provoca que al producirse cambios en un componente y en sus servicios resulte complicado determinar el impacto en el sistema. (Silega, 2010)

Entre las deficiencias que posee este mecanismo de integración de Sauxe, se percibe que no se controlan las versiones de cada componente, ni sus estados, ni la solución de sus dependencias. El consumo de los servicios se realiza a partir del conocimiento del nombre del módulo que brinda el servicio en cuestión, por lo que esta referencia queda plasmada en el código, dificultando posteriores modificaciones, actualizaciones, etc. Debido a que los módulos no están descritos como componentes ni se comportan como tales, aunque existe modularidad en las aplicaciones que soporta el marco de trabajo, es complejo determinar la dependencia que existe entre los módulos, ya que estas no están explícitas en ningún fichero de configuración, siendo imprescindible para los desarrolladores tener que inspeccionar el código.

El principal desperfecto de este mecanismo es su actual concepción, en la que los servicios que brinda cada módulo están explícitos en un fichero XML (Por las siglas en inglés de Lenguaje de Marcas Extensible) que contiene el subsistema al cual pertenece (integración interna), y a su vez los servicios que brinda cada subsistema se encuentran declarados en un fichero XML global para toda la aplicación (integración externa), brindando solo dos niveles de integración entre los componentes de las aplicaciones desarrolladas en el marco de trabajo Sauxe, lo cual limita la capacidad de segmentar los componentes e imposibilita el desglose en más niveles de las aplicaciones.

A partir de la problemática expuesta anteriormente, se define como **problema a resolver** la limitante del nivel de modularidad de los componentes en las soluciones que soporta el marco de trabajo Sauxe a partir de los niveles de integración definidos.

Queda entonces definido como **objeto de estudio** la arquitectura orientada a componentes, delimitando el **campo de acción** a la integración entre componentes.

El **objetivo general** de esta investigación es desarrollar un mecanismo de integración para el marco de trabajo Sauxe que permita eliminar las limitantes en cuanto al nivel de modularidad de los componentes en las soluciones que soporta. A partir de este objetivo se plantean los siguientes **objetivos específicos**:

- Realizar un análisis sobre la arquitectura de sistemas de información para identificar las mejores prácticas que permitan resolver el problema de la investigación.
- Diseñar un mecanismo de integración entre los componentes en el marco de trabajo Sauxe para eliminar las limitantes en cuanto a la modularidad de las soluciones que soporta.
- Implementar la propuesta de solución en el marco de trabajo Sauxe para eliminar las limitantes en cuanto a la modularidad de las soluciones que soporta.
- Validar la solución propuesta mediante pruebas de caja blanca y caja negra para garantizar la calidad de la misma.

Los métodos científicos de investigación son la forma de abordar la realidad con el propósito de descubrir su esencia y sus relaciones. (Hernández y Coello, 2002)

Los **Métodos teóricos** que se emplearán durante la investigación son el **Analítico-Sintético** y el **Histórico-Lógico**, en el análisis de los principales conceptos asociados al tema de la investigación, el estudio de su evolución y comportamiento, tomando los elementos fundamentales.

Se empleará el **Método empírico Entrevista**, con el objetivo de detectar deficiencias en el mecanismo de integración que provee Sauxe, a partir de la experiencia de arquitectos y desarrolladores de los proyectos en donde es utilizado este marco de trabajo.

Resultado de esta investigación se obtendrá un mecanismo de integración para el marco de trabajo Sauxe sin limitantes en cuanto al nivel de modularidad de sus componentes.

El documento se encuentra estructurado de la siguiente forma:

En el **Capítulo 1** se presentarán conceptos asociados al tema de la investigación y se realizará un análisis de los mecanismos de integración de otros frameworks, desarrollados en diversas plataformas con el fin de obtener ideas aplicables a la solución.

En el **Capítulo 2** se identificarán y describirán los requisitos funcionales que tendrá el nuevo módulo para la integración de componentes, se describe su funcionamiento y lo que se reconocerá dentro del marco de trabajo Sauxe como un componente integrable.

En el **Capítulo 3** se validará la propuesta de solución mediante pruebas de caja blanca y caja negra.

Capítulo I: Fundamentación teórica

1.1. Introducción

Se exponen en el presente capítulo los principales conceptos asociados al tema de la investigación y se presenta un análisis realizado sobre los mecanismos de integración de distintos framework desarrollados en PHP (por las siglas en inglés de Preprocesador de Hipertexto) y en otras plataformas, estudiados con el propósito de obtener prácticas aplicables a la solución.

1.2. Conceptos fundamentales

1.2.1. Componente

Dada la diversidad en formas, tamaños y formatos que pudiera poseer, existen disímiles opiniones acerca de lo que se considera o no un componente de software.

En Software Paradigms, se define un componente de software como un segmento de código de una aplicación predefinido y encapsulado[2] que se puede combinar con otros componentes y con código adicional para producir una aplicación personalizada. (Kaisler, 2005)

Según Philippe Krutchen de Rational Rose, un componente es una parte no trivial, casi independiente y reemplazable de un sistema que cumple una función clara en el contexto de una arquitectura bien definida. (Brown y Wallnau, 1998)

Para Gartner Inc. un componente de software en tiempo de ejecución es un paquete de dinámica enlazable de uno o más programas gestionados como una unidad al cual se accede a través de interfaces documentadas que puede ser descubierto en tiempo de ejecución. (Kaisler, 2005)

[2] La encapsulación, es un enfoque metodológico para la solución de problemas que consiste en unir en la misma clase características y comportamientos (variables y métodos) que pueden considerarse pertenecientes a una misma entidad, ocultándose conscientemente la información en una parte del programa. (Budd 1994)

Clemens Szyperski, arquitecto de software afiliado a Microsoft Research, considera un componente de software como una unidad de composición con interfaces especificadas contractualmente y dependencias explícitas de contexto único, que puede ser desplegado independientemente y estar sujeto a la composición de terceros. (Szyperski, 1998)

Un aspecto fundamental de los componentes es que estos deben ser reutilizables, para lo cual deberán cumplir con las siguientes características:

- Los componentes deben tener una identificación clara y consistente que facilite su catalogación y búsqueda en repositorios.
- Deben exponer al público únicamente el conjunto de operaciones que lo caracteriza (interfaz) y ocultar sus detalles de implementación.
- Las operaciones que ofrece un componente a través de su interfaz no deben variar.
- Deben tener una documentación adecuada que facilite su evaluación, adaptación a nuevos entornos, integración con otros componentes y acceso a información de soporte. (Montilva, Arapé y Colmenares, 2003)

En el marco de esta investigación, se reconoce como un componente aquella unidad de software reutilizable, con servicios y dependencias explícitos, cuyo contrato se establece a través de interfaces.

La idea principal es segmentar las aplicaciones compactas en componentes reutilizables, que puedan ser desplegados, distribuidos y actualizados de forma independiente. Para lograr esto, se deben proporcionar los mecanismos para la interoperabilidad entre las aplicaciones y componentes. Si los componentes pueden interactuar, pueden ser combinados para crear aplicaciones de mayor tamaño en una forma flexible e incrementable.

Modularidad

El término modularidad, como la capacidad de un sistema de dividirse en módulos, está asociado a un artículo publicado en 1972 por David Parnas, profesional canadiense pionero en el campo de la ingeniería del software. En este trabajo se discutía la forma en la que la modularidad en el diseño de sistemas podía mejorar la flexibilidad y el control conceptual del sistema, acortando los tiempos de desarrollo.

La división en módulos está asociada a la descomposición significativa de un sistema de software y su agrupación en subsistemas y componentes. La tarea principal es decidir cómo empaquetar físicamente las entidades que forman la estructura lógica de una aplicación.

La capacidad de abarcar cada parte de un sistema en forma separada facilita su modificación, ya que debido a la naturaleza evolutiva del software muchas veces se debe volver hacia el trabajo previo y realizar modificaciones. Cuando es necesaria la reparación de un sistema, una apropiada división en módulos ayuda a restringir la búsqueda de la fuente de error a componentes separados. (Pressman, 2002)

Es importante, a pesar de los beneficios que ofrece la modularidad en un sistema, tener sumo cuidado de no segmentar demasiado un sistema, dado que a medida que aumenta el número de módulos, aumenta el coste o esfuerzo asociado a la integración de los mismos.

Tal como se refleja en la gráfica de la Imagen 1, existe un valor M que determina la cantidad de módulos que proporcionará un coste mínimo de desarrollo. Este valor evidentemente estará asociado a la dimensión y complejidad del software que se desee implementar. (Pressman, 2002)

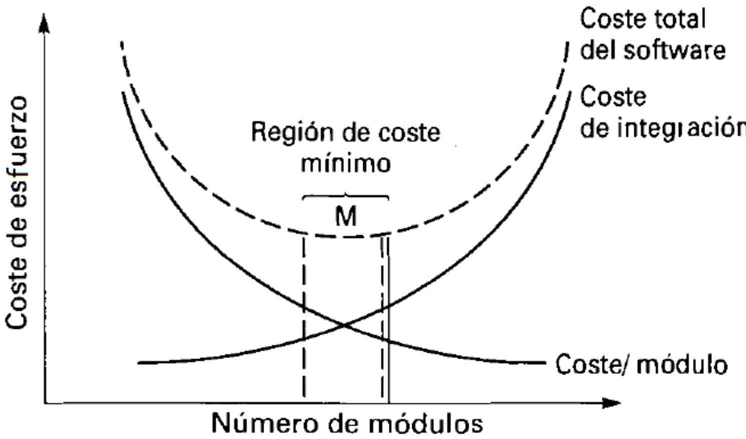


Imagen 1 Relación modularidad-coste de software.

Integración

La integración se refiere al proceso de combinar piezas en partes más complejas y, finalmente completar un producto. Este proceso incluye elementos críticos tales como las descripciones y la gestión de las interfaces, la secuencia en la que los componentes están integrados, y la comunicación entre los diferentes actores. (Larsson, 2005)

En la actualidad, el desarrollo de componentes se enfoca en construir aplicaciones de software grandes mediante la integración de componentes de software existentes a partir de que la integración permite reducir el tiempo de desarrollo del sistema. También puede reducir costos, ya que en lugar de reemplazar los sistemas existentes, la integración puede ser utilizada para enlazarlos. La complejidad de los procesos que cubren las aplicaciones de software actuales ha provocado que las funcionalidades no siempre puedan ser cubiertas por un solo componente o incluso que excedan los límites de la propia aplicación, problema resuelto a través de la integración a distintos niveles utilizando diferentes estrategias. (Díaz, 2010)

En el desarrollo basado en componentes se identifican tres niveles de integración: (Díaz, 2010)

- **Nivel de servicios**

Se presenta cuando las aplicaciones no son capaces de cubrir por sí solas todos los procesos que necesitan las entidades y deben integrarse con otras aplicaciones que las soporten. Es el nivel más complejo pues las aplicaciones a integrar no siempre comparten directrices tecnológicas.

- **Subsistema a subsistema**

Se presenta cuando los procesos abarcan funcionalidades que son responsabilidad de distintos subsistemas. Es un tanto menos complejo que el nivel anterior, a pesar de que los subsistemas involucrados pudieran estar distribuidos en diferentes entornos, por lo que se deberán evitar las dependencias fuertes.

- **Componente a componente**

Es el nivel más simple de integración, se presenta cuando se combinan componentes para dar soporte a las funcionalidades de un subsistema.

Es en este último nivel será en el que se centrará la propuesta de solución de esta investigación.

1.2.2. Arquitectura de software

Asociado de manera estrecha al concepto de componente se encuentra el de arquitectura de software, del cual existen variadas definiciones.

Los disímiles criterios expresan, a pesar de que provienen de fuentes diversas, aquellos elementos indispensables cuando de arquitectura de software se trata.

Según Roger Pressman, la arquitectura de software es la representación que capacita al ingeniero del software para: *(1)* analizar la efectividad del diseño para la consecución de los requisitos fijados, *(2)* considerar las alternativas arquitectónicas en una etapa en la cual hacer cambios en el diseño es relativamente fácil, y *(3)* reducir los riesgos asociados a la construcción del software. (Pressman, 2002)

La arquitectura representa un modelo relativamente pequeño, intelectualmente tratable, de la forma en que un sistema se estructura y sus componentes se entienden entre sí; este modelo es transferible a través de sistemas; en particular, se puede aplicar a otros sistemas que exhiben requerimientos parecidos y puede promover reutilización en gran escala. El diseño arquitectónico soporta reutilización de grandes componentes o incluso de framework en los que se pueden integrar componentes. (Reynoso, 2004)

Una de las definiciones más reconocida, aportada por el autor Paul Clements, plantea que la arquitectura de software es, a grandes rasgos, una vista general del sistema que incluye los componentes principales, la conducta de estos componentes con el resto del sistema y las formas en que deben interactuar y coordinarse para alcanzar la misión del sistema. (Clements, 1996)

Sin embargo, se ha establecido que la definición oficial sea la brindada por la IEEE, adoptada también por Microsoft:

> *"La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución."*

1.2.3. Framework

Este término originario de la lengua inglesa, como su traducción indica, se refiere a un marco de trabajo o estructura. Un framework encapsula tareas repetitivas en módulos genéricos fácilmente reutilizables, tratando de cubrir un dominio entero en lugar de problemas determinados, proporcionando una estructura de soporte predefinida en la que otros proyectos de software pueden ser organizados y desarrollados.

Muchos autores coinciden en que un framework es una arquitectura de software reutilizable que contiene tanto el diseño como el código, pero existen variados criterios acerca de lo que es un framework y las partes que lo constituyen. (Kaisler, 2005)

La corporación Taligent considera que si los objetos son abstracciones que a veces son difíciles de explicar, los framework presentan un reto más interesante, la diferencia fundamental entre un framework y una colección arbitraria de clases es que el framework no sólo describe los objetos, sino también sus interacciones. (Kaisler, 2005)

Un framework consta de clases (o estructuras), la ejecución de los componentes de las aplicaciones genéricas, así como componentes concretos que cumplen tareas especializadas. (Kaisler, 2005)

Por lo general un framework está conformado por:

Una caja de herramientas: Conjunto de elementos pre-elaborados y componentes de software rápidamente integrables. Esto disminuye la cantidad de código a generar y los riesgos de error, así como significa también un aumento de la productividad y de la capacidad de dedicar más tiempo a la implementación de funcionalidades que aporten mayor valor añadido, como la gestión de los principios rectores, los efectos secundarios, etc. (Potencier, 2012)

Una metodología: "Esquema de montaje" para las aplicaciones, que permite a los desarrolladores trabajar de manera eficiente y eficaz en los aspectos más complejos de una tarea, y el uso de mejores prácticas para garantizar la estabilidad, facilidad de mantenimiento y actualización de las aplicaciones que desarrollan. (Potencier, 2012)

Una característica importante de un framework es que los métodos definidos por el usuario para adaptarlo a menudo se invocan desde dentro del mismo, en lugar de hacerlo desde código de la aplicación del usuario. El framework a menudo desempeña el papel del programa principal en la coordinación y secuencia de las actividades de la aplicación. Esta inversión de control proporciona a los framework el poder para servir como esqueletos extensibles. Los métodos proporcionados por el usuario se adaptan a los algoritmos genéricos definidos en el framework de una aplicación en particular.

En un framework, y por tanto en la integración entre componentes, es relevante la utilización de la inversión de control. (Johnson, 1997)

Inversión de control. Inyección de dependencias

La inversión de control trata de invertir el flujo de eventos habitual de la programación. Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden ocurrir durante el ciclo de vida de un programa mediante llamadas a funciones. En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir. (Morejón, 2009)

La inversión de control es vista a menudo como una característica definitoria de un framework, ya que es una de las partes claves que lo hace diferente de una biblioteca. Cada llamada ejecuta un trabajo y devuelve el control al cliente. (Johnson, 1997)

Utilizando los métodos de programación tradicional, cada objeto es responsable de obtener las referencias a los objetos con los que colabora. Aplicando IoC, a los objetos se les proporcionan sus dependencias en el momento de la creación por una entidad externa que coordina todos los objetos en el sistema. Es decir, las dependencias se inyectan en los objetos. Por lo tanto, IoC significa una inversión de responsabilidades con respecto a la forma en que el objeto obtiene las referencias a objetos colaboradores. (Breidenbach y Walls, 2005)

En concordancia con Martin Fowler[3], se puede decir que la inyección de dependencias le permite a un framework identificar las dependencias de los componentes y gestionarlas de forma transparente para el componente.

---

[3] Autor y conferencista internacional acerca del desarrollo de software, especializado en análisis orientado a objetos y diseño, UML, patrones y metodologías ágiles de desarrollo de software, incluyendo la programación extrema. Popularizó el término Inyección de Dependencia como una forma de Inversión de Control.

1.3. Integración entre componentes en diversos framework

En aras de identificar posibles soluciones al problema de esta investigación se ha analizado cómo fluye la integración entre componentes en algunos framework desarrollados en PHP y otras plataformas, buscando aportes teóricos y/o prácticos aplicables a la solución.

1.3.1. Symfony

Symfony es un framework de desarrollo web, implementado en lenguaje PHP, que cuenta en la actualidad con dos ramas estables. La versión 1.4 (la última de la primera generación) y la versión 2, lanzada oficialmente el 28 de julio de 2011. (Mata, 2012)

En Symfony 2, con el objetivo de proporcionar ayuda al crear instancias, organizar y recuperar objetos (servicios) en una aplicación, se implementó un contenedor de servicios que permite la estandarización y centralización de la forma en que se construyen los objetos en una aplicación. Symfony 2 proporciona facilidades para el trabajo, y acentúa una arquitectura que promueve el código reutilizable y disociado. (Potencier, 2011)

La estructura de directorios de una aplicación Symfony 2 es la siguiente:

app/: Configuración de la aplicación

src/: El código PHP del proyecto

vendor/: Las dependencias de terceros

web/: El directorio raíz del servidor web

La configuración de todo el proyecto se encuentra en el directorio app\. En src se encuentra el código de la aplicación, la cual estará formada por módulos denominados *bundles*.

Un *bundle* es básicamente un directorio que contiene los archivos necesarios para un grupo de funcionalidades específicas, como por ejemplo un blog, un carrito de compras o el frontend y backend de la aplicación y representa un componente en Symfony (Ardissone, 2012)

La configuración de los *bundles* se puede realizar por dos vías, a través del archivo config.yml en la carpeta config\ del proyecto, o a través del archivo services.yml que se encuentra en la carpeta config/ dentro del *bundle*; de cualquier forma en esta configuración solo se explicitan los servicios que provee un

bundle, pudiéndose consumir dichos servicios desde cualquier parte de la aplicación. Esta configuración se registra en caché desde la primera petición.

Un servicio es cualquier objeto PHP que realiza alguna tarea "global", un objeto creado para un propósito X. Cada servicio se utiliza en toda una aplicación siempre que se necesite la funcionalidad específica que este proporciona.

Al explicitar un servicio en alguno de los archivos de configuración también se explicitan los parámetros que debe recibir este servicio así como la ubicación de la clase que implementa el servicio.

```
services:
   example1:
      class:      Desymfony\ExampleBundle\Controller\Example1
```

Imagen 2 Ejemplo de configuración de un servicio en el fichero config.yml.

```
services:
   example2:
      class:      Desymfony\ExampleBundle\Controller\Example2
```

Imagen 3 Ejemplo de configuración de un servicio en el fichero services.yml.

```
$var=$this->get('example'); //llamada al servicio de nombre example
```

Imagen 4 Ejemplo de invocación a un servicio.

Dada la concepción del mecanismo para la integración entre componentes en Symfony es posible el consumo de los servicios que estos brindan desde cualquier *bundle* de una aplicación. Entre los componentes existe un escaso nivel de dependencia pero estas no quedan explícitas en la configuración de los componentes.

En Symfony 2, los componentes consumen servicios que brindan otros componentes, a diferencia de lo que en la actualidad ocurre en el marco de trabajo Sauxe, ya que directamente se consumen métodos que se encuentran en otros componentes.

Una deficiencia notable en Symfony 2 es que sus componentes no declaran sus dependencias, siendo necesario para los desarrolladores inspeccionar del código.

1.3.2. Zend

Zend es un framework de código abierto para el desarrollo de aplicaciones y servicios web con PHP 5, implementado al 100% utilizando código orientado a objetos.

Zend 2, en su versión beta, liberada a finales de 2011, introduce un enfoque al desarrollo de aplicaciones basadas en módulos, a través de un contenedor de inyección de dependencias capaz de crear objetos al ser solicitados y gestionar la inyección de dependencias necesarias, para dichos objetos. (Zend Technologies Inc. 2011)

En el directorio *config*, que se encuentra en el directorio raíz de cada módulo, se ubican los archivos de configuración específica del módulo. Estos archivos pueden estar en cualquier formato. Es recomendable nombrar la configuración principal "*module.format*", y la configuración PHP, "*module.config.php*". Típicamente, se establece una configuración para el router así como para el inyector de dependencias.

El contenedor de inyección de dependencias de Zend 2, interviene en el proceso de integración entre los componentes, pero no cubre dicho proceso en su totalidad, dado que los componentes en su configuración expresan sus dependencias pero no sus servicios.

1.3.3. Spring

Spring es un framework creado con el objetivo de facilitar el desarrollo de aplicaciones en Java que promueve el bajo acoplamiento a través de la inversión de control. El proyecto de desarrollo de este framework se inició en el año 2003. (Breidenbach y Walls, 2005)

Spring tiene un contenedor que permite la administración del ciclo de vida y la configuración de los objetos de una aplicación. Además, este framework permite la configuración y composición de aplicaciones complejas a partir de componentes simples, pero su poder real está en cómo los componentes son inyectados en otros componentes utilizando IoC. (Breidenbach y Walls, 2005)

BeanFactory

Esta clase es el contenedor de Spring, cuya responsabilidad es crear y brindar componentes, la misma implementa el patrón de diseño Factory. (Breidenbach y Walls, 2005)

BeanFactory carga por defecto las configuraciones de todos los componentes, pero estas no serán utilizadas ni se creará ninguna instancia mientras no sea necesario. Cuando se realice una llamada al método getBean() pasándole por parámetro el nombre del componente con el que se desea trabajar, BeanFactory creará una instancia del componente configurando sus propiedades a través de la inyección de dependencias.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="foo"
      class="com.springinaction.Foo"/>
  <bean id="bar"
      class="com.springinaction.Bar"/>
</beans>
```

Imagen 5 Ejemplo de configuración de componentes con el contenedor de Spring.

En Spring la configuración de un componente o clase se lleva a cabo a través de un archivo XML (sucede así en todos los archivos de configuración de Spring). El elemento raíz es <beans>. La etiqueta <bean> se usa para indicarle al contenedor del framework sobre una clase y cómo debe ser configurada, y las etiquetas <property> y <value> para definir una propiedad y el valor que va a tomar respectivamente. En el caso que se necesite conectar un componente a otro se utilizará el subelemento <ref> como en el siguiente ejemplo, haciendo referencia al componente con el que se desea conectar:

```
<bean id="foo"
    class="com.springinaction.Foo">
  <property name="bar">
    <ref bean="bar"/>
  </property>
</bean>
<bean id="bar"
    class="com.springinaction.Bar"/>
```

Imagen 6 Ejemplo de integración entre componentes en Spring.

En Spring, los componentes no son responsables de gestionar su asociación con otros componentes, sino que se les proporcionan referencias a los componentes de los que dependen a través del contenedor. (Breidenbach y Walls, 2005)

Este framework al pertenecer a una plataforma distinta a la del marco de trabajo Sauxe no es utilizable en términos prácticos. Una de las deficiencias halladas en este framework es que en la configuración de sus componentes son identificables sus dependencias pero no los servicios que proveen.

1.3.4. OSGi

OSGi es un framework que define un sistema de módulos dinámicos para Java, que ofrece un mejor control sobre la estructura del componente, la capacidad de gestionar de forma dinámica el ciclo de vida del código, y un enfoque de acoplamiento flexible del código. (Hall y otros, 2011)

Una de las limitantes de Java es que proporciona algunos aspectos de la modularidad en la forma de orientación a objetos, pero nunca fue pensado para apoyar la programación de grandes módulos. OSGi, dado que suple la ausencia de modularidad de las aplicaciones desarrolladas en Java, proporcionando una plataforma completa y ligera para la implementación de aplicaciones basadas en componentes y orientadas a servicios dentro de una JVM (por las siglas en inglés de Máquina Virtual de Java).

Equinox es la implementación de OSGi para el motor de ejecución subyacente en el IDE Eclipse. Del mismo modo sucede con el servidor de aplicaciones GlassFish v3, cuyo motor de ejecución, Apache Felix, es la implementación del framework OSGi.

La diversidad de casos de utilización legitima el valor y la flexibilidad proporcionada por este framework a través de las tres capas conceptuales definidas en su especificación, las cuales se muestran en la Imagen 7. La tecnología OSGi define el ciclo de vida de los módulos, además de permitirles interactuar a través de servicios. (Hall y otros, 2011)

Imagen 7 Capas de la arquitectura de OSGi.

Como normalmente sucede en las arquitecturas en capas, cada capa es dependiente de la capa inmediata inferior, siendo posible el uso de las capas inferiores de OSGi sin necesidad de utilizar las superiores, pero no viceversa.

Capa de módulo: Aquí se define el concepto de *bundle*, que no es más que un archivo JAR (por las siglas en inglés de Archivo de Java) con metadatos adicionales, que contiene los archivos de clases y los recursos relacionados.

Capa del ciclo de vida: define cómo los *bundles* se instalan y se gestionan de forma dinámica en el framework OSGi.

Etapas o estados de un componente:

Instalado
Iniciado
Desinstalado
Parado
Actualizado

Capa de servicio: La capa de servicios de OSGi promueve la separación de la interfaz y la implementación. Los servicios de OSGi son interfaces de Java que constituyen el contrato conceptual entre los proveedores y los clientes de servicios.

Los *bundles* de OSGi pueden declarar explícitamente los paquetes externos de los que dependen, ya que el framework puede gestionar y verificar la compatibilidad entre *bundles* de forma automática, garantizando la coherencia entre ellos con respecto a las versiones y otras limitaciones.

Como se ha podido observar a lo largo de esta investigación, un aspecto importante para la reutilización de un componente, es su configuración, ya que permite el control de su comportamiento.

En el MANIFEST.MF, fichero que se encuentra en el directorio src dentro de la raíz del *bundle* de OSGi, se guarda además de la versión del archivo, datos tales como identidad, descripción, versión del *bundle*, y otros que el contenedor de OSGi utiliza para identificar al *bundle* y ponerlo a disposición de otros *bundles*. (Cogoluegnes, Templier y Piper, 2011)

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Spring DM Hello World
Bundle-SymbolicName: com.manning.sdmia.helloworld
Bundle-Version: 1.0.0
Export-Package: com.manning.sdmia
```

Imagen 8 Ejemplo de configuración a través del fichero MANIFEST.MF.

Este framework al pertenecer a una plataforma distinta a la del marco de trabajo Sauxe no es utilizable en términos prácticos. Entre los aspectos positivos se encuentra que la configuración del componente está ubicada en el mismo componente, el contrato de los servicios se establece a través de interfaces de java, los componentes tienen un ciclo de vida que incluye la instalación y solución de dependencias.

1.3.5. Spring Dinamic Modules

Spring DM proporciona una potente solución modular para el desarrollo de aplicaciones basadas en Spring que pueden desplegarse en un entorno de ejecución OSGi. El objetivo de la tecnología es hacer el trabajo de Spring y OSGi juntos de una manera sencilla, así como hacer frente a las limitaciones de las dos tecnologías. (Cogoluegnes, Templier y Piper, 2011)

En Spring, una gran cantidad de componentes dificulta el mantenimiento de las configuraciones en el contenedor. Además, Spring sufre falta de modularidad y no ofrece un apoyo real para mejorar esto. Por ejemplo, no hay forma de limitar la visibilidad de un componente, y cualquier otro lo puede utilizar, y puede ser visto desde el contexto de la aplicación incluso si no debe ser utilizado directamente.

Una aplicación puede tener cientos de componentes, y las ataduras entre estos forman un grafo de dependencias, otra de las limitaciones de Spring es la naturaleza estática de este grafo. Los vínculos entre los componentes se crean una sola vez, y no hay soporte integrado para ponerlos al día en tiempo de ejecución, para actualizar las dependencias entre componentes es necesario reiniciar toda la aplicación. Además, existe poco soporte para las dependencias circulares, lo que obliga a los desarrolladores a configurar sus componentes correctamente.

Por su parte, OSGi no proporciona ningún tipo de apoyo para los patrones o herramientas en el diseño e implementación de los componentes, dejando al desarrollador la libertad de elegir una arquitectura adecuada y el framework.

Utilizando Spring aumenta el poder de la plataforma, ya que permite el uso de la inyección de dependencias, AOP (por las siglas en inglés de Programación Orientada a Aspecto) y otros paradigmas modernos.

Otro desafío en el uso de OSGi se refiere a la configuración explícita y el comportamiento dinámico de los servicios. Los servicios son esenciales en la creación de cualquier aplicación OSGi porque son la única forma segura de acceder a las funciones a través de paquetes.

Spring DM utiliza un nuevo tipo de *bundle* que importa o exporta paquetes de Java, pero que no debe preocuparse por la inicialización y exportación de sus servicios o por buscar sus dependencias, ya que estas tareas serán realizadas por el framework haciendo uso de las configuraciones en XML.

Se puede apreciar que las dos tecnologías se complementan mutuamente, siendo potencialmente mayor que la suma de las partes, a partir de esto la importancia de Spring DM.

Con este framework sucede similar a los casos anteriores, no es utilizable en términos prácticos. Sin embargo posee aspectos teóricos de alta validez que pueden ser tenidos en cuenta para el diseño de la solución. Entre estos se encuentran la sencillez en términos de configuración de los componentes, la utilización de la inversión de control y la transparencia para el cliente de los servicios, respecto al mecanismo de integración.

1.4. Metodología de desarrollo

Se conoce como metodología al conjunto de procedimientos, técnicas, herramientas y al soporte documental que ayuda a los desarrolladores a realizar un software. Una metodología define quién debe hacer qué, cuándo y cómo debe hacerlo. (Jacobson, Booch y Rumbaugh, 2005)

Un modelo de procesos de desarrollo del software es una simplificación o abstracción de alto nivel de un proceso de software, que define un conjunto de actividades, acciones, tareas, fundamentos y productos de trabajo que se requieren para desarrollar software de alta calidad, constituyendo una guía para el trabajo de la ingeniería de software. (Sommerville, 2005)

1.4.1. Metodologías Tradicionales o Pesadas

Las metodologías pesadas son aquellas que están guiadas por una fuerte planificación durante todo el proceso de desarrollo, donde se realiza una intensa etapa de análisis y diseño antes de la construcción del sistema. Estas metodologías se centran en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, las herramientas y notaciones que se usarán. Dentro de estas metodologías se encuentra RUP (por las siglas en inglés de Proceso Unificado de Rational).

1.4.2. Metodologías Ágiles

Un proceso es ágil cuando el desarrollo de software es incremental, haciendo entregas pequeñas de software y con ciclos rápidos. Los clientes y desarrolladores trabajan juntos con una cercana comunicación. El método usado para el desarrollo es fácil de aprender, bien documentado y adaptable, de manera tal que permite realizar cambios de último momento. Las metodologías agiles son efectivas en proyectos con requisitos muy cambiantes y donde se exige reducir drásticamente los tiempos de desarrollo manteniendo una alta calidad de los mismos. Están especialmente orientadas para proyectos pequeños y posibilita la reducción de los costos de implantación en un equipo de desarrollo.

Dado que las metodologías de desarrollo existentes en la actualidad no son adaptables en su totalidad al software que se desarrolla en el Departamento de Tecnología del CEIGE, este utiliza un modelo para el desarrollo de software que tiene en cuenta el enfoque en espiral y el basado en componentes, Este modelo centrado en la arquitectura, iterativo e incremental tiene en cuenta además algunas de las actividades propuestas por RUP, por lo que será utilizado como guía durante la realización de este trabajo.

1.4.3. Características del modelo

- **Centrado en la arquitectura**

La arquitectura es una vista del diseño que permite a los desarrolladores ver el software, antes de comenzar su desarrollo, desde varios puntos de vista. Para obtener un producto con éxito es necesario que exista un equilibrio entre los componentes a desarrollar y la arquitectura, ya que los componentes

deben encajar en la arquitectura definida y la arquitectura debe permitir el desarrollo de los componentes. (Mariño, 2012)

- **Iterativo e incremental**

Partiendo de que el desarrollo de software es un proceso que involucra esfuerzo y tiempo, es factible dividir el trabajo en partes más pequeñas, donde cada parte es una iteración que resulta en un incremento. Las iteraciones constituyen el paso por el flujo de actividades propuesto, y los incrementos el crecimiento del producto. El control de las iteraciones reduce el costo de los riesgo al costo de un solo incremento, si se tiene que repetir la iteración, sólo se pierde el esfuerzo mal empleado de la iteración, no el valor del producto entero. (Mariño, 2012)

- **Basado en componentes**

Dado que en la actualidad los sistemas de software son cada vez más complejos y deben ser construidos en un tiempo muy corto y con una alta calidad, es conveniente el desarrollo a partir de la reutilización de componentes de software. El desarrollo basado en componente produce un gran impacto en el proceso de desarrollo de software, ya que reduce en gran medida los valores de las variables más significativas de la Ingeniería de Software: el costo, el tiempo y el esfuerzo requerido para desarrollar un producto de software; además del incremento de la calidad del software producido, aumenta la productividad de los grupos de desarrollo y la reducción del riesgo del proyecto. (Mariño, 2012)

Fases del ciclo de vida

A continuación se exponen las fases que propone el modelo de desarrollo a utilizar:

Estudio preliminar: Se define el alcance que tendrá el proyecto, se planifican las actividades de planeación de cada fase, se realizan las estimaciones de tiempo, costo y esfuerzo y se confecciona el proyecto técnico.

Modelo de negocio: Se percibe cómo funciona el negocio que se desea automatizar y se interactúa con el cliente para tener garantía de que el software desarrollado cumplirá su propósito. En caso que no exista un negocio bien definido, se identifican los conceptos del negocio que ayudan a entender el problema y se realiza el modelo conceptual con todos los conceptos del dominio identificados, facilitando la comprensión de las necesidades de los usuarios y los requisitos del software.

Requisitos: Se analiza el problema para comprender las necesidades de los interesados y expresarlas en forma de requisitos y describirlos.

Análisis y diseño: En esta fase es modelado el sistema y su forma (incluida su arquitectura) para que soporte todos los requisitos, incluyendo los requisitos no funcionales. Esto contribuye a una arquitectura sólida y estable que se convierte en un plano para la implementación. Los modelos desarrollados en esta etapa son más formales y específicos de una implementación. Se desarrollan las vistas de la arquitectura, lo diagramas de clases y modelo de dato.

Implementación: A partir de los resultados de la fase análisis y diseño se implementan, en caso que sea necesario, las funcionalidades del componente.

Integración: Se realizan actividades para lograr la integración del componente con los demás componentes del sistema.

Pruebas internas: Se le envía la planilla de solicitud de pruebas de liberación y el diseño de casos de pruebas al departamento de calidad, el cual realiza las revisiones correspondientes, ya que estas pruebas no son realizadas por el departamento.

1.5. Herramientas

En el diseño e implementación de la solución se utilizarán, en sus versiones más recientes, las herramientas definidas por el Departamento de Tecnología del CEIGE. (Bauta, 2011)

1.5.1. Visual Paradigm 8.0

Visual Paradigm es una herramienta CASE (por las siglas en inglés de Ingeniería de Software Asistida por Computadora) que utilizando UML (por las siglas en inglés de Lenguaje de Modelado Unificado) como lenguaje de modelado, permite la captura, diseño, gestión y documentación de los artefactos generados durante el proceso de desarrollo de software. (Visual Paradimg, 2011)

Esta herramienta proporciona al equipo de desarrollo de software un lenguaje estándar que facilita la comunicación entre sus integrantes y con el cliente. Además, permite realizar el proceso de ingeniería del software de forma directa (versión profesional) o inversa.

1.5.2. NetBeans 7.1

NetBeans es un IDE (por las siglas en inglés de Entorno de Desarrollo Integrado) libre y de código abierto, que con variados módulos brinda las herramientas necesarias para el desarrollo de aplicaciones profesionales de escritorio, web y aplicaciones móviles con la plataforma Java, así como con PHP y otras. (Oracle Corporation and its affiliates, 2011)

Cada módulo del NetBeans provee funciones bien definidas, tales como el soporte de Java, edición o soporte para el sistema de control de versiones.

1.5.3. Apache 2.2.9

Apache es un servidor de aplicaciones cuya adaptabilidad, robustez y estabilidad lo han hecho popular desde 1996. Proporciona un servidor seguro, eficiente y extensible que provee servicios HTTP (por las siglas en inglés de Protocolo de Transferencia de Hipertexto) en sincronía con los estándares HTTP actuales. Es una tecnología gratuita de código abierto. (The Apache Software Foundation, 2011)

1.6. Conclusiones parciales

Luego del análisis de distintos frameworks, se puede concluir que la solución más integral es Spring DM. Aunque no puede ser empleada en la solución del problema, ya que no es integrable al marco de trabajo Sauxe, su concepción aporta importantes elementos teóricos, útiles para el diseño de la solución de esta investigación.

Atendiendo a las soluciones que brindan los framework estudiados, y siguiendo la definición de componente abordada en este trabajo, se concluye que un componente debe contener una descripción con sus datos identificativos del componente. Esta descripción debe estar plasmada en un fichero de configuración que facilite su reconocimiento y modificación.

Capítulo II: Características del sistema. Análisis y diseño de la solución.

2.1. Introducción

El modelado de la fase de análisis asigna requisitos a las representaciones de datos, funciones y comportamiento. El diseño convierte el modelo de análisis en diseños de datos, arquitectónicos, de interfaz y a nivel de componentes del software. (Pressman, 2002)

En el presente capítulo se describe el funcionamiento del mecanismo para el consumo de servicios en el marco de trabajo Sauxe, haciendo énfasis en su principal deficiencia. Se presenta el modelo conceptual y son identificados y descritos los requisitos funcionales de la solución. Se describe lo que se va a reconocer como un componente integrable dentro del nuevo módulo de integración y el funcionamiento del mismo. Por último, se valida el diseño propuesto mediante la utilización de métricas.

2.2. Descripción del mecanismo de integración de Sauxe

Como se mencionó anteriormente el marco de trabajo Sauxe suministra un mecanismo para el consumo de servicios, nombrado IoC, que permite la integración entre los módulos, cuya principal deficiencia son los dos niveles que establece para la integración entre los mismos.

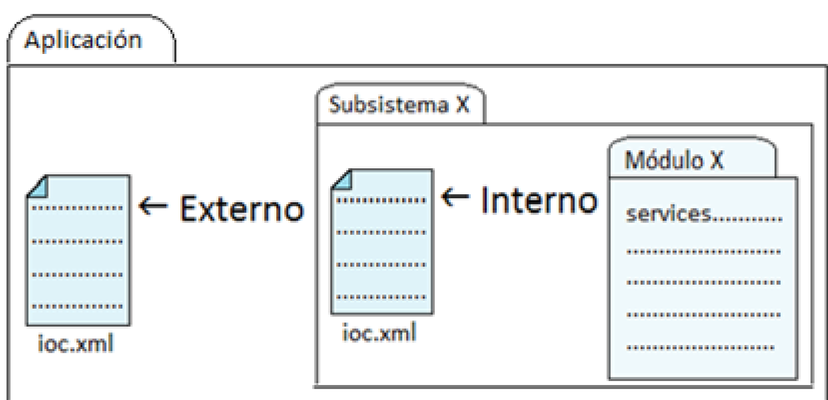


Imagen 9 Niveles de integración que establece el mecanismo de integración de Sauxe.

Este mecanismo posee dos elementos fundamentales, la descripción de los servicios y el consumo de dichos servicios, y dado que fue desarrollado para utilizarse declarativamente, si se desea integrar un módulo solo es necesario describir en el fichero de configuración correspondiente (interno o externo) los servicios que este brinda, así como los parámetros que necesita recibir cada servicio y el resultado que devuelve a la entidad que solicite el servicio. (Morejón, 2009)

```xml
<?xml version="1.0" encoding="UTF-8"?>

<ioc> <!--inicialización del xml (etiqueta principal)-->
    <portal src="portal"><!--subsistema que brinda el servicio-->
        <ObtenerSubsistema reference=""><!--nombre del servicio-->
            <!--referencia a la clase y metodo que implementa el servicio-->
            <inyector clase="PortalService" metodo="ObtenerSubsistema">
                <prototipo><!--encapsula parametros y resultado del servicio-->
                    <!--identifica nombre y tipo de un parametro-->
                    <parametro nombre="uri" tipo="stdClass" />
                    <parametro nombre="identidad" tipo="enteropos" />
                    <!--identifica el formato del resultado del servicio-->
                    <resultado clase="stdClass" />
                </prototipo><!--cierre de etiqueta-->
            </ObtenerSubsistema>
        </portal>
    </ioc>
```

Imagen 10 Estructura del fichero de configuración ioc.xml.

El consumo de servicios puede establecerse de dos formas a partir de los niveles que establece la configuración del mecanismo. Para cualquiera de los casos el fichero de configuración, se nombra ioc.xml y en caso de que la integración sea externa (entre subsistemas) se encuentra en el directorio de recursos comunes de la aplicación; para el caso en que la integración sea interna (entre módulos de un subsistema) el archivo de configuración se encuentra en el directorio de recursos comunes del subsistema.

Para el consumo de funcionalidades implementadas en diversos componentes, que es realmente lo que sucede en Sauxe, se instancia un objeto de la clase Zend_Ext_Ioc, al cual se le solicita el módulo donde se brinda el servicio y el nombre del servicio a consumir.

La sintaxis para utilizar servicios externos a un componente es la siguiente:

```
$integrator = ZendExt_IoC::getInstance();
$estructurasubsist = $integrator->subsistemas->BuscarInstancia($identidad, $idsubsistema);
```

Imagen 11 Ejemplo de utilización de integración externa.

La sintaxis para utilizar servicios internos de un componente es la siguiente:

```
$pIntegrator = ZendExt_IoC_Inter::getInstance();
$result = $pIntegrator->banco->Buscarclientes($idestructura,$limit,$start);
```

Imagen 12 Ejemplo de utilización de integración interna.

2.2.1. Modelo conceptual del negocio

El análisis orientado a objetos tiene por finalidad estipular una especificación del dominio del problema y los requerimientos desde la perspectiva de la clasificación por objetos y desde el punto de vista de entender los términos empleados en el dominio. Para descomponer el dominio del problema hay que identificar los conceptos, los atributos y las asociaciones del dominio que se juzgan importantes. El resultado puede expresarse en un modelo conceptual, el cual se muestra gráficamente en un grupo de diagramas que describen los conceptos. (Larman, 1999)

El mecanismo de integración del marco de trabajo Sauxe, a partir de la descripción de los servicios en los ficheros correspondientes, permitirá la integración externa o interna de los subsistemas o componentes respectivamente, lo cual se manifiesta gráficamente en la Imagen 13 a través de un modelo conceptual.

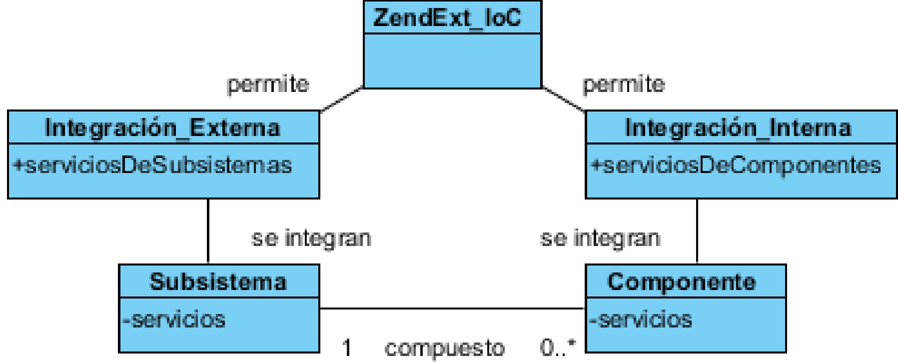


Imagen 13 Modelo conceptual.

Diccionario de datos para el Modelo conceptual del negocio

ZendExt_IoC

Descripción	Mecanismo que permite la integración entre los componentes del marco de trabajo Sauxe.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
N/A	N/A	N/A	N/A	N/A	N/A	N/A

Estados de validez

N/A

Integración_Externa

Descripción	Integración que se lleva a cabo entre subsistemas.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
serviciosDe Subsistemas	Declaración de los servicios que brinda cada subsistema en el fichero ioc.xml del directorio de recursos comunes de la aplicación.	Archivo xml.	No	Si	Declaraciones de los servicios que brindan los subsistemas.	Vacío

Estados de validez

N/A

Integración_Interna

Descripción	Integración que se lleva a cabo entre componentes de un subsistema.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
serviciosDe Componentes	Declaración de los servicios que brinda cada componente de un subsistema en el fichero ioc.xml del directorio de recursos comunes del subsistema.	Archivo xml.	No	No	Declaraciones de los servicios que brindan los componentes de un subsistema.	Vacío

Estados de validez

N/A

Subsistema

Descripción	Sección de la aplicación que brinda servicios y que puede estar integrada componentes.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
Servicios.	Implementación de funcionalidades o servicios que brinda el componente.	Archivo php.	No	No	Implementación de los servicios que brinda el subsistema.	Vacío

Estados de validez

N/A

Componente

Descripción	Sección de la aplicación que brinda servicios.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
Servicios.	Implementación de funcionalidades o servicios que brinda el componente.	Archivo php	No	No	Implementación de los servicios que brinda el componente.	Vacío

Estados de validez

N/A

2.3. Características de la solución propuesta

A partir del análisis de los framework anteriormente descritos y teniendo en cuenta las características del marco de trabajo Sauxe, se definió la siguiente solución.

Dado que un componente es un módulo que tiene claramente especificados los datos que permiten reconocerlo, así como los servicios que provee y los que consume, el marco de trabajo Sauxe reconocerá como un componente integrable aquel que en su directorio raíz contenga un fichero de configuración denominado **bundle.xml**. Esta estructura permitirá convertir cualquier sección de la aplicación en módulos, eliminando la limitante que dio origen al problema de esta investigación.

El fichero de configuración que contendrán los componentes recogerá la descripción mencionada anteriormente siguiendo la estructura que se muestra a continuación:

```
<bundle>
    <data id="" name=""  version="" state=""/>
    <services>
        <service id="">
            <interface>direccion de la interfaz</interface>
            <impl>direccion de la clase que implementa la interfaz</impl>
        </service>
    </services>
    <dependencies>
        <service id="">
            <interface>direccion de la interfaz</interface>
        </service>
    </dependencies>
</bundle>
```

Imagen 14 Estructura del contenido en el fichero bundle.xml.

La etiqueta *data* contendrá datos identificativos como el id, el nombre, la versión y el estado del componente. Este último atributo reflejará la disponibilidad del componente para ser utilizado por el mecanismo de integración en la solución de dependencias de otros componentes y para la integración.

Cada componente contendrá una lista de servicios que provee (dentro del tag <services>) y una lista de servicios de los que depende (dentro del tag <dependencies>), en el caso de que existan dependencias con otros componentes. De cada servicio en el fichero de configuración se especificará la dirección de su interfaz (dentro del tag <interface>) y la dirección de la clase que contiene su implementación (dentro del tag <impl>).

Las interfaces de los componentes contendrán comentarios en PHPNotation, para especificar los tipos de datos de los parámetros que reciben cada servicio y/o el tipo de dato del retorno, en caso de que existan. A partir de estos comentarios y los métodos definidos se establecerá la equivalencia entre interfaces.

Ya que el contrato entre los componentes se realizará a través de sus interfaces, tanto el componente proveedor como el cliente deberán poseer interfaces equivalentes, que pudieran nombrarse de igual forma. Para evitar colisiones de nombres, se propone que para versiones inferiores a PHP 5.3, se deben denominar a las clases interfaz con la siguiente estructura **NombrecomponenteNombreinterfaz.**

Al instalarse el marco de trabajo, se instalarán todos los componentes. Al iniciar la aplicación, se ejecutará la localización de todos los componentes, como resultado de esta búsqueda se registrará información de cada uno de los componentes en un fichero informativo central y en un archivo serializado, ubicados en el directorio de recursos comunes de la aplicación y en la caché. Posteriormente se ejecutará la solución de las dependencias de cada componente identificado, actualizando la información registrada referente al estado de aquellos componentes cuyas dependencias hayan sido resueltas en su totalidad.

Durante la interacción entre los componentes, se entregará al componente que solicite un servicio un objeto proxy, que representará al componente que brinda el servicio en cuestión. Este objeto proxy permitirá controlar el acceso al componente al que hace referencia y registrar trazas de integración.

Como parte de la solución se diseñará una interfaz de usuario en la aplicación para facilitar a los desarrolladores la gestión de los componentes. A través de ésta se podrá registrar un nuevo componente u otro que anteriormente quedase deshabilitado, se podrán modificar las propiedades de los componentes en explotación en la aplicación o deshabilitar un componente con el que no se desee trabajar.

2.3.1. Modelo conceptual del sistema

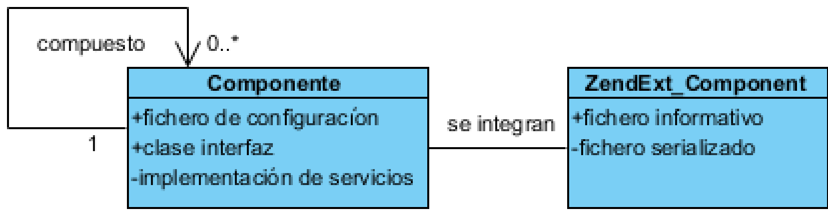


Imagen 15 Modelo conceptual del sistema.

Diccionario de datos para el Modelo conceptual del sistema

ZendExt_Component

Descripción	Mecanismo que permite la integración entre los componentes del marco de trabajo Sauxe.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
Fichero informativo.	Fichero que contiene los datos recopilados de cada componente.	Archivo xml.	Si	Si	N/A	N/A
Fichero serializado.	Fichero que contiene los datos recopilados de cada componente.	Archivo data.	Si	Si	N/A	N/A

Estados de validez

N/A

Componente

Descripción	Mecanismo que permite la integración entre los componentes del marco de trabajo Sauxe.					
Atributos					**Restricciones**	
Nombre	**Descripción**	**Tipo**	**¿Puede ser nulo?**	**¿Es único?**	**Clases válidas**	**Clases no válidas**
Fichero de configuración	Contiene datos identificativos del componente, los servicios que provee y de los que depende.	Archivo xml.	No	Si	Datos identificativos del componente.	Vacío
Clase interfaz.	Contiene la declaración de los servicios que brinda el componente.	Archivo php.	No	Si	Servicios que brinda el componente.	Vacío
Implementación de servicios.	Contiene la implementación de los servicios declarados en la interfaz.	Archivo php	No	Si	Servicios que brinda el componente.	Vacío

Estados de validez

N/A

2.4. Requisitos de la solución

El proceso de desarrollo de software comprende en sus etapas tempranas tareas orientadas a captar las necesidades o características a satisfacer en el sistema que se vaya a crear o modificar. Resultado de esta captación se obtendrán los requisitos con los que deberá cumplir la solución, que pueden variar a lo largo del ciclo de vida del sistema. Con la variabilidad de los requisitos se debe tener sumo cuidado pues proporcional a lo tardío que suceda el cambio, mayor impacto tendrá en el proyecto.

Se deberá realizar un buen proceso de captura de requisitos, el resultado serán requisitos claros, completos y consistentes; de lo contrario es probable que se construya una solución refinada que resuelva incorrectamente el problema, obteniendo pérdidas en tiempo y costes, frustración personal y clientes descontentos. (Pressman, 2002)

2.4.1. Requisitos funcionales

Los requisitos que a continuación se presentan son el resultado del estudio de sistemas similares y de entrevistas (informales) con los desarrolladores y arquitectos de los proyectos en donde es utilizado el marco de trabajo Sauxe, y por tanto en donde han impactado las deficiencias del mecanismo de integración IoC.

RF-1. Localizar componentes.

RF-2. Persistir la configuración de los componentes.

RF-3. Resolver las dependencias de los componentes.

RF-4. Permitir instanciación de componentes desde controladores del marco de trabajo.

RF-5. Establecer la comunicación entre componentes.

RF-6. Gestionar componentes.

- **RF-6.1.** Registrar componente.
- **RF-6.2.** Modificar componente.
- **RF-6.3.** Deshabilitar componente.

2.4.2. Requisitos no funcionales

Dado que el nuevo mecanismo de integración para el marco de trabajo Sauxe se utilizará internamente, no se han definido requisitos no funcionales para el mismo.

La interfaz gráfica para la gestión de los componentes deberá poseer las siguientes características o requisitos no funcionales:

- Usabilidad: La interfaz gráfica deberá ser fácil de utilizar por los usuarios.
- Interfaz: La interfaz gráfica tendrá un diseño agradable.
- Rendimiento: El sistema deberá consumir la menor cantidad de recursos.
- Legales: Las herramientas utilizadas en el desarrollo de la solución deben ser libres.

2.4.3. Descripción de los requisitos funcionales de la solución

Tabla 1 Descripción textual del requisito Localizar componentes.

Precondiciones	Se debe haber instalado el marco de trabajo Sauxe.
Flujo de eventos	
Flujo básico Localizar componentes	
1.	Realizar búsqueda introspectiva por los directorios de la aplicación.
2.	Si en un directorio se encuentra el archivo bundle.xml, identificar un componente.
3.	Del archivo bundle.xml recopilar id, nombre, estado, versión y dirección del componente.
4.	Si el directorio no contiene otros directorios, finaliza el requisito.
Pos-condiciones	
1.	Quedan localizados los componentes disponibles en la aplicación.
Flujos alternativos	
Flujo alternativo 2.a Si no se encuentra en el directorio el archivo bundle.xml.	
1.	Continuar en el paso 4 del flujo básico de eventos.
Pos-condiciones	
1.	N/A
Flujo alternativo 4.a Si el directorio contiene otros directorios.	
1.	Continuar en el paso 1 del flujo básico de eventos.
Pos-condiciones	
1.	N/A
Validaciones	
1.	N/A

Tabla 1 Descripción textual del requisito Localizar componentes (continuación).

Conceptos	Componente	**Visibles en la interfaz:** N/A **Utilizados internamente:** Archivo de configuración.
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

Tabla 2 Descripción textual del requisito Persistir la configuración de los componentes.

Precondiciones	N/A
Flujo de eventos	
Flujo básico Persistir la configuración de los componentes	
1.	Registrar en la caché de la aplicación la información de todos los componentes activos.
2.	Si no existe, crear el fichero serializado bundles.data en el directorio de recursos comunes de la aplicación.
3.	Registrar en el fichero bundles.data la información de todos los componentes activos.
4.	Si no existe, crear el fichero bundles.xml en el directorio de recursos comunes de la aplicación.
5.	Registrar en el fichero bundles.xml la información de todos los componentes activos.
6.	Finaliza el requisito.
Pos-condiciones	
1.	Queda registrada la información referente a todos los componentes disponibles en la aplicación.
Flujos alternativos	
Flujo alternativo 2.a Si existe el fichero bundles.data en el directorio de recursos comunes de la aplicación.	
1.	Sobrescribir el fichero bundles.data con la información de todos los componentes activos.
2.	Continuar en el paso 4 del flujo básico de eventos.
Pos-condiciones	
1.	N/A
Flujo alternativo 4.a Si existe el fichero bundles.xml en el directorio de recursos comunes de la aplicación.	
1.	Sobrescribir el fichero bundles.xml con la información de todos los componentes activos.
2.	Continuar en el paso 6 del flujo básico de eventos.
Pos-condiciones	
1.	N/A
Validaciones	
1.	N/A

Conceptos	Componente	**Visibles en la interfaz:** N/A **Utilizados internamente:** Archivo de configuración.
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

Tabla 3 Descripción textual del requisito Resolver dependencias de los componentes.

Precondiciones	Debe existir más de un componente activo en la aplicación.	
Flujo de eventos		
Flujo básico Resolver dependencias de los componentes		
1.	Identificar los componentes cuyas dependencias están resueltas.	
2.	Crear listado de componentes con dependencias por resolver.	
3.	Solucionar dependencias no resueltas a partir de servicios disponibles.	
4.	Si se solucionan todas las dependencias de un componente, actualizar su estado a resolved.	
5.	Si no existen más componentes con dependencias, actualizar la información de los componentes en el registro de la aplicación y en el fichero de configuración.	
6.	Finaliza el requisito.	
Pos-condiciones		
1.	Se resuelven las dependencias de los componentes según la disponibilidad de los servicios.	
Flujos alternativos		
Flujo alternativo 4.a Si no se solucionan todas las dependencias de un componente.		
1.	Continuar en el paso 5 del flujo básico de eventos.	
Pos-condiciones		
1.	N/A	
Flujo alternativo 5.a Si existen otros componentes con dependencias.		
1.	Continuar en el paso 1 del flujo alternativo 3.	
Pos-condiciones		
1.	N/A	
Validaciones		
1.	N/A	
Conceptos	Componente	Visibles en la interfaz: N/A Utilizados internamente: Archivo de configuración.
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

Tabla 4 Descripción textual del requisito

Permitir instanciación de componentes desde controladores del marco de trabajo.

Precondiciones	N/A
Flujo de eventos	
Flujo básico Permitir instanciación de componentes desde controladores del marco de trabajo	
1.	Para obtener una instancia de un componente se deberá utilizar la siguiente sintaxis: $varComp=ZendExt_Component_Component::getComponent($param1,$param2); Donde param1 será una referencia a la clase solicitante a través de $this y param2 será identificador definido en el campo id de la etiqueta service en el bundle.xml.
2.	Para acceder a los servicios del componente instanciado se deberá utilizar la siguiente sintaxis: $varX=$varComp->nombreDelServicio;
3.	Finaliza el requisito.

Tabla 4 Descripción textual del requisito

Permitir instanciación de componentes desde controladores del marco de trabajo.

Pos-condiciones		
1.	N/A	
Flujos alternativos		
1.	N/A	
Pos-condiciones		
1.	N/A	
Validaciones		
1.	N/A	
Conceptos	N/A	
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

Tabla 5 Descripción textual del requisito Establecer la comunicación entre componentes.

Precondiciones	Se debe haber instanciado algún componente.
Flujo de eventos	
Flujo básico Establecer la comunicación entre componentes	
1.	Incluir las clases correspondientes al componente proveedor del servicio.
2.	Devolver instancia del componente a integrar.
3.	Finaliza el requisito.
Pos-condiciones	
1.	N/A
Flujos alternativos	
1.	N/A
Pos-condiciones	
1.	N/A
Validaciones	
1.	N/A
Conceptos	N/A
Requisitos especiales	N/A
Asuntos pendientes	N/A

2.5. Patrones de diseño

Los patrones de diseño son soluciones a problemas repetidos en la construcción de software, y en ocasiones pueden incluir sugerencias para aplicar estas soluciones en diversos entornos. (Kaisler, 2005)

En el diseño que propone este trabajo, se utilizaron algunos patrones de diseño GoF (por las siglas en inglés de Grupo de Cuatro) y GRASP (por las siglas en inglés de Patrones Generales de Software de

Asignación de Responsabilidades), para solucionar y/o evitar diferentes problemas que pudieran aparecer durante la implementación de la solución:

2.5.1. GRASP

Experto: Propone que la clase que contenga toda la información necesaria, será la responsable de la creación de un objeto o la implementación de un método. El comportamiento se distribuye entre las que contienen la información requerida, siendo más fáciles de entender, mantener y ampliar, aumentando sus posibilidades de reutilización.

Creador: La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. Este patrón proporciona asistencia al identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases. Mediante el uso de este patrón se logra un bajo acoplamiento, facilitando la posibilidad de mantenimiento y reutilización.

Controlador: El patrón controlador funciona como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la interfaz quien recibe los datos del usuario y los envía a las distintas clases según el método invocado.

Alta cohesión: Propone que la información que almacena una clase debe de ser coherente y debe estar, en la medida de lo posible, relacionada con la clase. Al realizar un cambio en una clase con alta cohesión, todos los métodos que pueden verse afectados, estarán a la vista, en el mismo archivo. Incrementa la claridad, la reutilización y la facilidad de comprensión del diseño.

Bajo acoplamiento: Este patrón expresa que entre las clases deberán existir pocas ataduras, es decir, estas estarán lo menos relacionadas posible, de forma tal que en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de las clases, incrementando la reutilización, y disminuyendo la dependencia entre las clases.

2.5.2. GoF

Proxy: Cuando no se desea el acceso directo a un objeto sobre el que se va a aplicar determinada acción, este patrón propone la adición de un nivel que permita solamente el acceso al objeto a través de un objeto proxy sustituto, que será el responsable de controlar o mejorar el acceso al objeto real.

Facade (Fachada): Propone la definición de un único punto de conexión con un componente. Este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los componentes del subsistema.

2.6. Diagramas de clases del diseño

De manera general, para el diseño de las clases se tuvo en cuenta el patrón de diseño Experto, con el objetivo de encapsular la información de los objetos y facilitar la definición de clases más cohesivas y con menos acoplamiento, posibilitando que las dependencias existentes no impacten de manera negativa en el mantenimiento del sistema y en sus oportunidades de reutilización. Se emplearon además en este diseño los patrones Bajo Acoplamiento y Alta Cohesión con el objetivo de obtener clases más independientes, reutilizables y fáciles de mantener.

El diseño de clases correspondiente a la solución que se propone se ha dividido en dos diagramas. El diagrama de clases expuesto en la Imagen 16 corresponde a las clases a través de las cuales el marco de trabajo Sauxe manejará cada componente identificado como un objeto *bundle*, con sus datos identificativos, su lista de servicios y su lista de dependencias.

Cada componente implementará el patrón Facade ya que a cada servicio solo se accederá a través de la clase interfaz en la que se encuentra declarado.

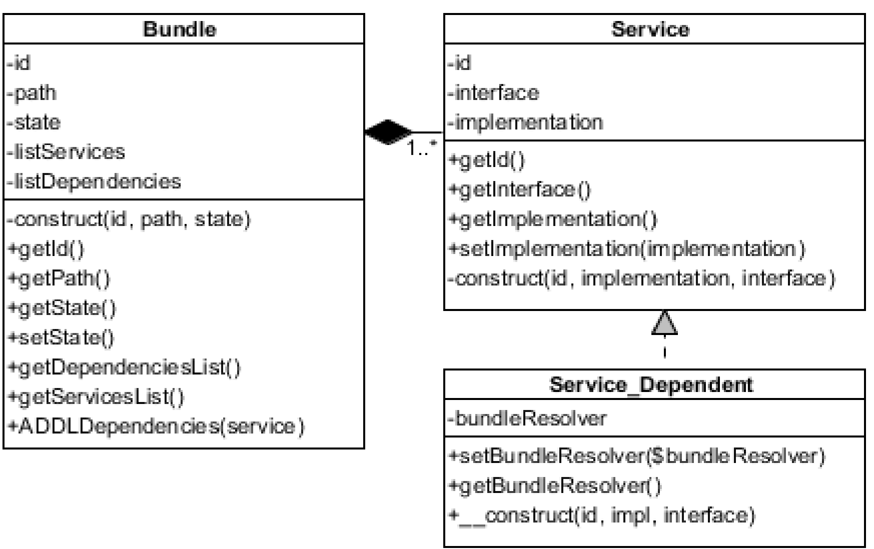


Imagen 16 Diagrama de clases de un componente.

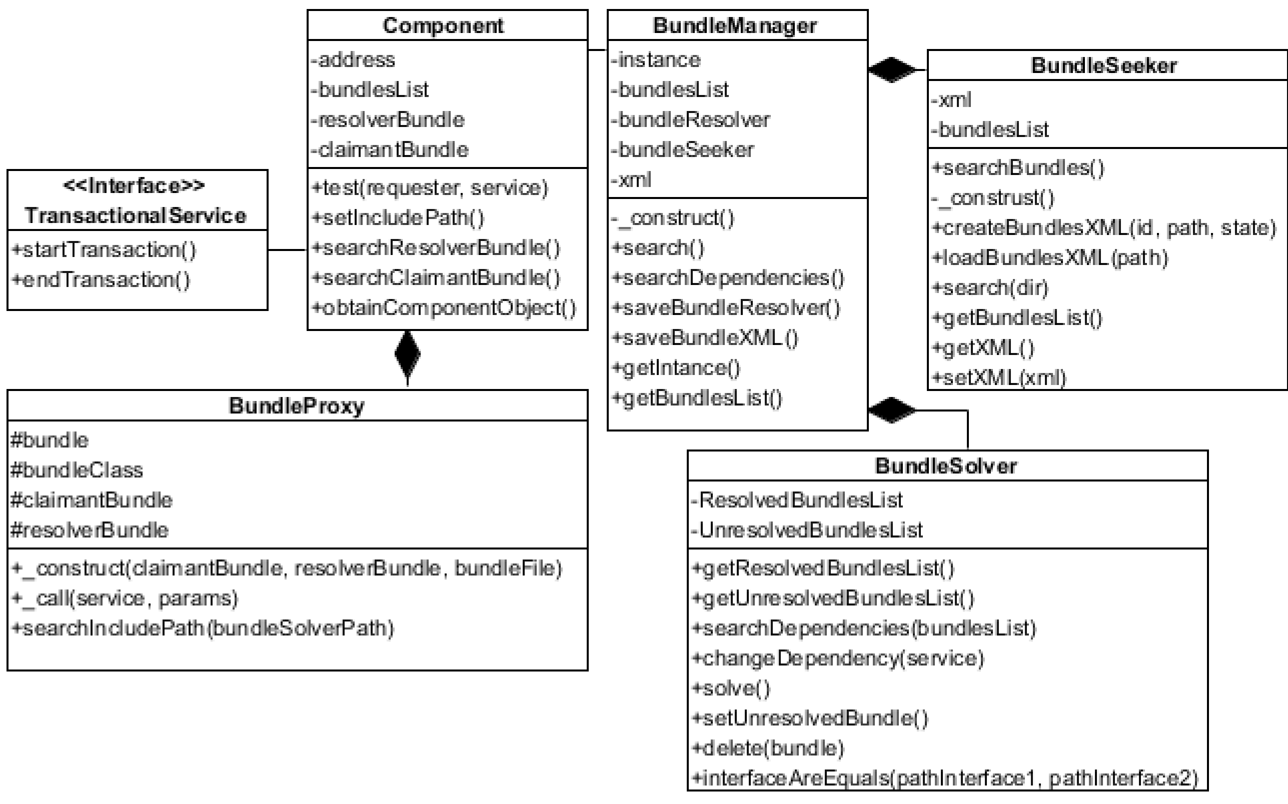


Imagen 17 Diagrama de clases del mecanismo de integración.

El diagrama de clases que se presenta en la Imagen 17 corresponde a las clases responsables de la localización y solución de dependencias de los componentes activos en el marco de trabajo (*Manager*) y el establecimiento de la comunicación entre los componentes (*Component*).

Es implementado además el patrón Proxy, ya que no se accederá directamente a los componentes, sino a través de un objeto proxy que representará al componente que brinde el servicio que se solicite.

2.7. Validación del diseño propuesto

Las métricas son una forma de obtener información cuantitativa del estado de un proyecto de desarrollo de software, partiendo de datos básicos de calidad y productividad que son analizados, comparados con promedios anteriores, y evaluados para determinar las mejoras en la calidad y productividad.

Las métricas son también utilizadas para señalar áreas con problemas de manera que se puedan desarrollar las correcciones y mejorar el proceso del software. (Pressman, 2002)

2.7.1. Métrica Tamaño operacional de clase (TOC)

Esta métrica permite evaluar los atributos de calidad responsabilidad, complejidad de implementación y reutilización a partir del valor del promedio de operaciones por clase según los criterios que se exponen a continuación:

Tabla 6 Criterios para evaluar la métrica TOC.

Atributo	Categoría	Criterio
Responsabilidad	Baja	< =Promedio.
	Media	Entre Promedio y 2*Promedio.
	Alta	> 2*Promedio.
Complejidad de implementación	Baja	< = Promedio.
	Media	Entre Promedio y 2*Promedio.
	Alta	> 2*Promedio.
Reutilización	Baja	> 2*Promedio.
	Media	Entre Promedio y 2*Promedio.
	Alta	<= Promedio.

Tabla 7 Procedimientos por clase.

Clases	Cantidad de Métodos
Bundle	15
Service	5
ServiceDependent	3
Manager	15
BundleSeeker	8
BundleSolver	10
Component	2
BundleProxy	3
TransactionalService	1
ComponentController	0

Para un total de 10 clases se obtuvo un promedio de **6** operaciones por entidad.

Tabla 8 Resultados obtenidos de la evaluación de los atributos de calidad.

Clases	Responsabilidad	Complejidad de implementación	Reutilización
Bundle	Alta	Alta	Baja
Service	Baja	Baja	Alta
ServiceDependent	Baja	Baja	Alta
Manager	Alta	Alta	Baja
BundleSeeker	Media	Media	Media
BundleSolver	Media	Media	Media
Component	Baja	Baja	Alta
BundleProxy	Baja	Baja	Alta
TransactionalService	Baja	Baja	Alta
ComponentController	Baja	Baja	Alta

Existe un 20% de las clases que intervienen en el diseño propuesto con niveles medios de responsabilidad, complejidad de implementación y reutilización, quedando solo un 20% con un alto nivel de responsabilidad y complejidad de implementación, por lo que poseen un bajo nivel de reutilización.

El 60% de las clases poseen un bajo nivel de responsabilidad, por lo que la complejidad de implementación de estas clases es baja también, lo que les proporciona un alto nivel de reutilización.

2.7.2. Métrica Relación entre clases (RC)

Esta métrica permite la evaluación de los atributos de calidad acoplamiento, complejidad mantenimiento, reutilización y cantidad de pruebas a partir del valor del promedio de relaciones de uso por clase según los criterios que se exponen a continuación:

Tabla 9 Criterios para evaluar la métrica RC.

Atributo	Categoría	Criterio
Acoplamiento	Ninguno	0
	Bajo	1
	Medio	2
	Alto	>2
Complejidad Mantenimiento	Baja	<= Promedio.
	Media	Entre Promedio y 2*Promedio.
	Alta	> 2*Promedio.

Tabla 9 Criterios para evaluar la métrica RC (continuación).

Reutilización	Baja	>2*Promedio.
	Media	Entre Promedio y 2*Promedio.
	Alta	<= Promedio.
Cantidad de Pruebas	Baja	<= Promedio.
	Media	Entre Promedio y 2*Promedio.
	Alta	> 2*Promedio.

Tabla 10 Relaciones de uso por clase.

Clases	Cantidad de Relaciones de Uso
Bundle	1
Service	2
ServiceDependent	0
Manager	1
BundleSeeker	2
BundleSolver	1
Component	0
BundleProxy	1
TransactionalService	1
Componentcontroller	0

Para un total de 10 clases se obtuvo un promedio de **1** asociación de uso por entidad.

Tabla 11 Resultados obtenidos de la evaluación de los atributos de calidad.

Clases	Cantidad de Relaciones de Uso	Acoplamiento	Complejidad de mantenimiento	Reutilización	Cantidad de Pruebas
Bundle	1	Bajo	Baja	Alta	Baja
Service	2	Medio	Media	Media	Media
ServiceDependent	0	Ninguno	Baja	Alta	Baja
Manager	1	Bajo	Baja	Alta	Baja
BundleSeeker	2	Medio	Media	Media	Media
BundleSolver	1	Bajo	Baja	Alta	Baja
Component	0	Ninguno	Baja	Alta	Baja
BundleProxy	1	Bajo	Baja	Alta	Baja
TransactionalService	1	Bajo	Baja	Alta	Baja
Componentcontroller	0	Ninguno	Baja	Alta	Baja

El 22% de las clases poseen niveles medios de acoplamiento, complejidad de mantenimiento y reutilización. Para el 78% restante de las clases existe un nivel bajo o nulo de acoplamiento, por lo que la complejidad de mantenimiento de las mismas también es baja, lo que les proporciona un alto nivel de reutilización.

2.8. Conclusiones parciales

El diseño propuesto procurando cubrir las deficiencias halladas en el actual mecanismo de integración, presenta estructura y forma de configuración nuevas para los componentes de las soluciones desarrolladas sobre el marco de trabajo Sauxe, cumpliendo además con el concepto de componente por el que se rige este trabajo.

Se puede concluir que el diseño propuesto es aceptable a partir de los valores que tomaron los atributos de calidad evaluados a través de las métricas TOC y RC.

El mecanismo de integración propuesto permite reconocer cualquier sección de la aplicación como un componente, independientemente de su cantidad y niveles de anidación, resolviendo así el problema generado a partir de los niveles de integración que establece el mecanismo de integración IoC.

Capítulo III: Implementación y pruebas

1.1. Introducción

En el presente capítulo se presentan los elementos de mayor interés de la etapa de implementación de la solución propuesta. Se presentan además los resultados obtenidos a partir de las pruebas de caja blanca y de caja negra realizadas con el fin de garantizar una alta calidad en el producto final.

1.2. Implementación

Partiendo del diseño presentado se ha implementado un nuevo mecanismo de integración para los componentes de las soluciones desarrolladas sobre el marco de trabajo Sauxe. A continuación se exponen los principales algoritmos que intervienen en los procesos de resolución de dependencias y comunicación entre los componentes.

1.2.1. Resolviendo dependencias

Previo a la resolución de las dependencias de los componentes activos en una aplicación estos deberán ser localizados, resultado de este proceso se registrará en los ficheros bundles.xml y bundles.data (fichero serializado) los datos referentes a cada uno de los componentes, sus datos identificativos (id, nombre, versión), su ubicación y su estado. Esta información será utilizada posteriormente en la resolución de las dependencias.

El algoritmo que a continuación se expone separa los componentes según su estado, para luego ejecutar el procedimiento que se muestra en la Imagen 19 y devolver un listado con todos los componentes activos en la aplicación, con su estado de solución actualizado. Es importante aclarar que cada componente que no posea dependencias será manejado como resuelto desde el inicio.

Este algoritmo solo se ejecutará cuando existan en la aplicación al menos un componente con dependencias resueltas y al menos un componente con dependencias por resolver, ya que dada la existencia únicamente de componentes no resueltos en la aplicación no tendrá sentido la ejecución del proceso de resolución de dependencias.

```
function search_Dependencies($ListaBundle) {
    foreach ($ListaBundle as $Bundle) {
        if (count($Bundle->getListaDependencias()) == 0) {
            $Bundle->setState("solved");
            $this->Lista_Bundle_Resueltosl[] = $Bundle;
        } else {
            $Bundle->setState("unresolved");
            $this->Lista_Bundle_NResueltosl[] = $Bundle;
        }
    }
    $lista1 = $this->Lista_Bundle_Resueltosl;
    $lista2 = $this->Lista_Bundle_NResueltosl;
    if (!count($lista1) == 0 && !count($lista2) == 0) {
        $this->solve();
    }
}
```

Imagen 18 Algoritmo que separa los componentes según su estado.

```
function solve() {
    foreach ($this->Lista_Bundle_Resueltosl as $Bundle_resolved) {
        foreach ($Bundle_resolved->getListaService() as $Servicio) {
            $this->change_Dependency($Servicio, $Bundle_resolved);
        }
        $this->Set_Bundle_NResolved();
    }
    $a1 = $this->Lista_Bundle_Resueltosl;
    $a2 = $this->Lista_Bundle_NResueltosl;
    return array_merge_recursive($a1,$a2);
}
```

Imagen 19 Algoritmo que resuelve las dependencias.

El algoritmo expuesto en la Imagen 19, intenta solucionar las dependencias de los componentes no resueltos a partir de los servicios que brindan los componentes ya resueltos, evaluando para cada combinación servicio-dependencia si las interfaces son similares.

Para que dos interfaces se consideren equivalentes deberán poseer en principio la misma cantidad de funcionalidades, las cuales deberán coincidir en cuanto a cantidad de parámetros de entrada, tipos de los mismos y tipos del resultado, en caso de existir. Para estas comparaciones entre las interfaces de los componentes se utilizó la API (por las siglas en inglés de Interfaz de Programación de Aplicaciones) Zend_Reflection.

```
function interfaceAreEquals($pathInterface1, $pathInterface2) {
    $pathInterface1 = str_replace("/web/../", "/", $pathInterface1);
    $pathInterface1 = str_replace("//", "/", $pathInterface1);
    $pathInterface2 = str_replace("/web/../", "/", $pathInterface2);
    $pathInterface2 = str_replace("//", "/", $pathInterface2);

    require_once $pathInterface1;
    $file = new Zend_Reflection_File($pathInterface1);
    $interface1 = $file->getClasses();

    require_once $pathInterface2;
    $file1 = new Zend_Reflection_File($pathInterface2);
    $interface2 = $file1->getClasses();

    $methods1 = $interface1[0]->getMethods();
    $methods2 = $interface2[0]->getMethods();

    if (count($methods1) == count($methods2))
        for ($i = 0; $i < count($methods2); $i++) {
            for ($j = 0; $j < count($methods2); $j++) {
                if ($methods1[$i]->getname() == $methods2[$j]->getname()) {
                    try {
                        $DocBlock1 = $methods1[$i]->getDocblock();
                        $retur1 = $DocBlock1->getTag("return");
                        $DocBlock2 = $methods2[$j]->getDocblock();
                        $retur2 = $DocBlock2->getTag("return");

                        if ($retur1 == $retur2)
                            break;
                    } catch (Exception $exc) {
                        throw new ZendExt_Exception('ECI001');
                    }
                }
            }
            if ($j == count($methods2))
                return false;
            else {
                $parameter1 = $methods1[$i]->getParameters();
                $parameter2 = $methods2[$j]->getParameters();
                if (count($parameter1) == count($parameter2))
                    for ($g = 0; $g < count($parameter1); $g++) {
```

Imagen 20 Algoritmo que compara la equivalencia entre interfaces.

```
        if ($parameter1[$g]->getClass()->name != $parameter2[$g]->getClass()->name)
            return FALSE;
                        }
                }
            }
        else
            return false;
        return true;
    }
```

Imagen 20 Algoritmo que compara la equivalencia entre interfaces (continuación).

El algoritmo que se muestra en la imagen 20 devuelve un valor booleano (verdadero o falso) según el resultado de la comparación entre interfaces. En los casos en que las interfaces comparadas resulten equivalentes, y por tanto el servicio que ofrece uno de los componentes resuelve determinada dependencia, se actualizarán los datos referentes a la dirección que implementa el servicio de una dependencia a través de la invocación del algoritmo que se muestra en la Imagen 21.

```
function change_Dependency($Servicio, $Bundle_resolved) {
    foreach ($this->Lista_Bundle_NResueltosl as $Bundle_NResolved) {
        foreach ($Bundle_NResolved->getListaDependencias() as $dependency) {
            $i1 = $Servicio->getInterface();
            $i2 = $Bundle_NResolved->getPath() . '/' . $dependency->getInterface();
            if ($this->interfaceAreEquals($i1, $i2)) {
                $dependency->setImpl($Servicio->getImpl());
                $dependency->setBundleResolver($Bundle_resolved);
            }
        }
    }
}
```

Imagen 21 Algoritmo que completa los datos de las dependencias.

1.2.2. Integrando componentes

Luego de solucionadas las dependencias posibles de los componentes activos en la aplicación solo resta establecer la comunicación entre estos al momento en que sea invocado un servicio determinado.

El algoritmo expuesto en la Imagen 22 es aquel que la clase *Component* invoca cuando se solicita un servicio, este recibirá el servicio en cuestión y sus parámetros, y devolverá al componente solicitante el resultado. Este algoritmo es el encargado además de registrar la traza de integración.

```
public function __call($service, $params) {
    $paramForBundle = '';
    foreach ($params as $param) {
        $paramForBundle.= (string) $param;
        $paramForBundle.= ',';
    }
    $paramForBundle = substr($paramForBundle, 0, strrpos($paramForBundle, ','));
    if (!method_exists($this->bundle, $service)) {
        throw new ZendExt_Exception('IOC004');
    }
    $cad_execution = "\$result = \$this->bundle->{$service}({$paramForBundle});";
    //si es un bundle con comportamiento transaccional
    if ($this->bundle instanceof ZendExt_Component_TransactionalService) {
        $this->bundle->startTransaction();
        eval($cad_execution);
        $this->bundle->endTransaction();
    } else {
        eval($cad_execution);
    }
    //registrar traza de integración
    $trace = ZendExt_Aspect_Trace::getInstance();
    $a = $this->claimantBundle->getName();
    $b = $this->resolverBundle->getName();
    $c = $this->bundleClass;
    $trace->beginTraceBundle($a, $b, $c, $service);
    return $result;
}
```

Imagen 22 Algoritmo que entrega el resultado del servicio solicitado.

Si el componente que brinda el servicio, utiliza una conexión a base de datos, u otro comportamiento transaccional, debe implementar la interfaz *TransactionalService.*

1.3. Prueba

La realización de pruebas durante el desarrollo de software es una tarea a la que se debe prestar suma importancia, dado que estas garantizarán la calidad del producto final y representan una revisión de las especificaciones, el diseño y la codificación.

Normas que pueden servir acertadamente como objetivos de las pruebas: (Myers, 1979)

1. La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Cualquier producto de ingeniería puede probarse (1) conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y, al mismo tiempo, buscando errores en cada función; (2) conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que “todas las piezas encajan”, o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. El primer enfoque de prueba se denomina prueba de caja negra y el segundo, prueba de caja blanca. (Pressman, 2002)

1.3.1. Pruebas de caja blanca

Las pruebas de caja blanca, denominadas a veces pruebas de caja de cristal es un método que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que: (1) garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo; (2) ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa; (3) ejecuten todos los bucles en sus límites y con sus límites operacionales; y (4) ejerciten las estructuras internas de datos para asegurar su validez. (Pressman, 2002)

Prueba del Camino básico

El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. (Pressman, 2002)

Es decir, si se aplica el método del camino básico, es posible conocer la cantidad de veces que, de distintas formas, pudiera ejecutarse un algoritmo, y a partir de este dato diseñar los casos de prueba necesarios para probar todas las variantes de su funcionamiento.

- Complejidad ciclomática

La complejidad ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba del camino básico, el valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez. (Pressman, 2002)

Fórmulas para el cálculo de la complejidad ciclomática:

- El número de regiones del grafo de flujo coincide con la complejidad ciclomática.
- La complejidad ciclomática V(G), de un grafo de flujo G, se define como. V(G) = A-N+2; donde A es el número de aristas del grafo y N es el número de nodos.
- La complejidad ciclomática V(G), de un grafo de flujo G, se define como V(G) = P + 1; donde P es el número de nodos predicado en el grafo de flujo G.

1.3.2. Pruebas de caja negra

Las pruebas de caja negra, también conocidas como pruebas de comportamiento, se centran en los requisitos funcionales del software, y permiten al ingeniero del software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. Esta prueba no es una alternativa a las técnicas de prueba de caja blanca, más bien se trata de un enfoque complementario que intenta descubrir diferentes tipos de errores. (Pressman, 2002)

Partición equivalente

La partición equivalente es un método de prueba de caja negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Un caso de prueba ideal descubre de forma inmediata una clase de errores que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico. (Pressman, 2002)

1.3.3. Diseño de casos de prueba

Diseñar las pruebas para validar un software puede demandar tanto esfuerzo como el propio diseño inicial del producto, esta es una tarea a la que los ingenieros del software a menudo le restan importancia, desarrollando casos de pruebas que consideran adecuados, pero que en realidad no están lo suficientemente completos.

Se deben diseñar pruebas que tengan la mayor probabilidad de encontrar el máximo de errores con la mínima cantidad de esfuerzo y tiempo posible. (Pressman, 2002)

Los casos de prueba utilizados en la realización de las pruebas de caja negra por el Departamento de Calidad se exponen en el Anexo 2.

1.3.4. Pruebas aplicadas a la solución

A continuación se detallan los resultados obtenidos en la aplicación de las pruebas de caja blanca para el caso del algoritmo **search_Dependencies()**.

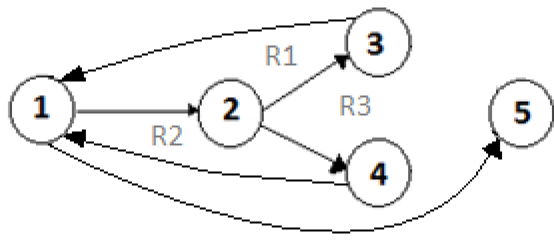


Imagen 23 Grafo de flujo del algoritmo.

El número de regiones del grafo de flujo es 3.

V(G) = 6 aristas – 5 nodos + 2 = 3.

V(G) = 2 nodos predicados + 1 = 3

La complejidad ciclomática es 3.

Conjunto básico de caminos

Camino 1: 1 – 5

Camino 2: 1 – 2 – 3 – 1 – 5

Camino 3: 1 – 2 – 4 – 1 – 5

Para verificar el correcto funcionamiento del algoritmo y confirmar la ejecución de los caminos básicos, se diseñaron casos de prueba que simularan las diversas condiciones de funcionamiento del algoritmo.

Durante la primera iteración de esta prueba, se detectó una falla en el algoritmo, ya que no se controlaba la entrada de una lista vacía o que contuviese un solo elemento, falla que se solucionó adicionando una sentencia condicional al finalizar el algoritmo, que controlará que solo se ejecute el proceso de solución de dependencias cuando en la aplicación existan al menos un componente con sus dependencias resueltas y otro con dependencias por resolver.

Luego de ejecutados, en varias iteraciones, los casos de prueba diseñados y comparados los resultados obtenidos con los esperados, se verificó la ejecución, al menos una vez, de cada una de las sentencias del algoritmo.

- **Gestión de componentes**

Para realizar las pruebas al módulo Gestión de componentes en la aplicación, se diseñaron los casos de prueba que se especifican en el Anexo 2.

Las principales No Conformidades (NC) detectadas a partir de la realización de las pruebas de caja negra, por el equipo de calidad interno del Departamento de Tecnología y por el equipo de calidad del CEIGE, se exponen en el Anexo 3.

Las NC detectadas en la aplicación fueron resueltas, realizando los correspondientes cambios en la implementación de las distintas funcionalidades asociadas a los errores encontrados.

- **Mecanismo de integración**

Se probó la pieza fundamental de la solución que se propone, el módulo de integración entre componentes en el marco de trabajo Sauxe, verificando su correcto funcionamiento utilizando la solución propuesta.

Se llevaron a cabo las siguientes tareas:

- Se tomaron en el marco de trabajo uno de los módulos principales y aquellos componentes de los cuales éste depende: Portal, Seguridad, Metadatos y Parámetros (este último perteneciente al módulo Configuración).

- Se creó para el componente seleccionado como cliente (Portal) un archivo de configuración con los datos identificativos y sus dependencias.
- Se creó para cada uno los componentes seleccionados como proveedores (Seguridad, Metadatos y Parámetros) un archivo de configuración con los datos identificativos de los componentes.
- Se creó para cada uno los componentes seleccionados como proveedores una clase interfaz con los servicios que ofrece cada componente, los parámetros que deben recibir y lo que devuelve cada servicio, en los casos correspondientes.
- Se creó para cada uno los componentes seleccionados como proveedores una clase que implementa aquellos servicios declarados en su correspondiente interfaz.
- Se comentaron las sentencias en el código asociadas a la instanciación y utilización del mecanismo de integración IoC, por las sentencias necesarias para hacer referencia al mecanismo de la propuesta de solución.

Puesto en funcionamiento el marco de trabajo se detectó el comportamiento transaccional que poseen algunos componentes, motivo por el cual se implementó la interfaz *TransactionalService.*

La realización de esta prueba permitió verificar la validez de la propuesta de solución implementada ya que el cambio de mecanismo de integración entre componentes no provocó impacto alguno en el correcto funcionamiento del marco de trabajo Sauxe, coincidiendo con los resultados esperados.

1.4. Conclusiones parciales

Los resultados obtenidos de las pruebas de caja blanca realizadas al código, las pruebas de caja negra realizadas al módulo de integración y a la aplicación, reflejados anteriormente, ratifican la validez de la implementación realizada.

El buen funcionamiento del marco de trabajo Sauxe, utilizando el módulo de integración para la nueva concepción de sus componentes, confirma el cumplimiento del objetivo principal de este trabajo.

Conclusiones generales

El estudio realizado sobre los mecanismos de integración utilizados en las recientes versiones de los framework más utilizados en las plataformas PHP y Java, permitió identificar algunas características que fueron aplicadas en la solución que se propone, en función de cumplir eficientemente el objetivo de este trabajo.

Atendiendo al problema de esta investigación y a las características tomadas de las tecnologías estudiadas, se diseñó e implementó un mecanismo de integración entre componentes en el marco de trabajo Sauxe que no limita los niveles de modularidad de las soluciones que soporta, y ofrece cobertura además a las deficiencias identificadas en el mecanismo de integración IoC y en la descripción de los componentes del marco de trabajo Sauxe.

Las pruebas de caja blanca y de caja negra realizadas a la solución que se propone revelaron errores en las funcionalidades, a partir de particularidades que no se tuvieron en cuenta durante la etapa de implementación, que fueron solucionados eficientemente, garantizando así la calidad de la propuesta.

Recomendaciones

Para futuras versiones de esta propuesta, se recomienda tener en cuenta el manejo del ciclo de vida de los componentes que propone OSGi.

Es recomendable, una vez actualizada la versión de Zend que en la actualidad utiliza el marco de trabajo Sauxe, hacer uso de las bondades que brinda el contenedor de inyección de dependencias que proporciona este framework en su versión 2.0.

Se recomienda adecuar la solución propuesta para ser utilizada sobre la versión de PHP 5.3 o superiores haciendo uso de los *namespaces* o espacios de nombre.

Bibliografía

Ardissone, Juan. 2012. «Symfony 2. El proyecto y los bundles.» *Maestros del web*. Disponible en: http://www.maestrosdelweb.com/editorial/curso-symfony2-proyecto-bundles/.

Bauta, René Rodrigo. 2011. «Arquitectura de Software. Vista Entorno de Desarrollo Tecnológico. Proyecto SAUXE.» CEIGE.

Breidenbach, Ryan y Craig Walls. 2005. *Spring in Action*. USA: Manning Publications Co.

Brown, Alan W. y Kurt C. Wallnau. 1998. *The current state of CBSE*. IEEE Sofware.

Budd, Timothy. 1994. *Introducción a la programación orientada a objetos*.

Clements, Paul C. 1996. «A survey of Architecture Description Languages». Software Engineering Institute Carnegie Mellon University.

Cogoluegnes, Arnaud, Thierry Templier y Andy Piper. 2011. *Spring Dinamic Modules In Action*. Manning Publications Co.

Díaz, Omar Antonio. 2010. «Diseño arquitectónico de una plataforma para la arquitectura distribuida en PHP basada en Servicios Web». Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas, La Habana, Cuba: Universidad de las Ciencias Informáticas.

Hall, Richard S., Karl Pauls, Stuart McCulloch y David Savage. 2011. *OSGi In Action*. Manning Publications Co.

Hernández, Rolando Alfredo y Sayda Coello. 2002. *EL PARADIGMA CUANTITATIVO DE LA INVESTIGACIÓN CIENTIFICA*. Ciudad de La Habana: Editorial Universitaria.

Jacobson, Ivar, Grady Booch y James Rumbaugh. 2005. *El Proceso Unificado de Desarrollo de Software*. Addison-Wesley.

Johnson, Ralph E. 1997. «Frameworks Home Page». *Frameworks*. Disponible en: http://st-www.cs.illinois.edu/users/johnson/frameworks.html.

Kaisler, Stephen H. 2005. *Software Paradigms*. New Jersey: John Wiley & Sons, Inc.

Larman, Craig. 1999. *UML y Patrones. Introducción al análisis y diseño orientado a objetos*. Prentice Hall, Inc.

Larsson, Stig. 2005. *IMPROVING SOFTWARE PRODUCT INTEGRATION*. Department of Computer Science and Engineering Mälardalen University.

Mariño, Diana. 2012. «Modelo de procesos de desarrollo de software para el departamento de Tecnología del CEIGE». Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas, La Habana, Cuba: Universidad de las Ciencias Informáticas.

Mata, Manel Pérez. 2012. «¿Qué es Symfony?» *TecnoRetales*. Disponible en: http://www.tecnoretales.com/linux/que-es-symfony/.

Montilva, Jonás A., Nelson Arapé y Juan Andrés Colmenares. 2003. «Desarrollo de Software Basado en Componentes.» IV Congreso de Automatización y Control.

Morejón, Yoandry. 2009. «Especificación técnica Componente IoC».

Morejón, Yoandry, Oiner Baryolo y Darien García. 2010. «ARQUITECTURA TECNOLÓGICA PARA EL DESARROLLO DE SOFTWARE».

Myers, Glenford J. 1979. *The Art of Software Testing*. Disponible en: http://www.carlosfau.com.ar/nqi/nqifiles/The%20Art%20of%20Software%20Testing%20-%20Second%20Edition.pdf.

Oracle Corporation and its affiliates. 2011. «NetBeans IDE - Features». Features. *NetBeans IDE*. Disponible en: http://netbeans.org/features/index.html.

Parnas, David L. 1972. *On the criteria to be used in descomposing systems into modules*. Comunnications of the ACM.

Potencier, Fabien. 2011. «Guía de inicio rápido. Symfony 2».

Potencier, Fabien. 2012. «Symfony at a glance». *Symfony*. Disponible en: http://symfony.com/symfony-at-a-glance.

Pressman, Roger S. 2002. *Ingeniería del software*. McGrawHill.

Real Academia Española. «Diccionario de la lengua española». Disponible en: http://buscon.rae.es/draeI/.

Reynoso, Carlos Billy. 2004. «Introducción a la Arquitectura de Software». Universidad de Buenos Aires.

Robinson, Ray Williams. 2011. «Propuesta de arquitectura de la capa de integración para sistemas de informacioón sobre tecnologías JEE». Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas, La Habana, Cuba: Universidad de las Ciencias Informáticas.

Silega, Nemury. 2010. «Guía metodológica para gestionar la integración de componentes en los proyectos del sistema CEDRUX». Trabajo de Diploma para optar por el grado de Máster en Ciencias Técnicas, La Habana, Cuba: Universidad de las Ciencias Informáticas.

Sommerville, Ivan. 2005. *Ingeniería de software Sommerville.* Madrid, España: Pearson Educación, S.A.

SpringSource, a division of VMware. «Spring Integration | SpringSource.org». *SpringSource Community*. Disponible en: http://www.springsource.org/spring-integration.

Szyperski, Clemens. 1998. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

The Apache Software Foundation. 2011. «Apache». *The Apache HTTP Server Project*. Disponible en: http://httpd.apache.org/.

Visual Paradimg. 2011. «Visual modeling tool for building enterprise applications». *Visual Paradimg website*. Disponible en: http://www.visual-paradigm.com/product/vpuml/provides/.

Zend Technologies Inc. 2011. «Programmer´s Reference Guide. Zend 2».

Glosario de términos

Artefacto Producto perceptible resultado del proceso de desarrollo de software. Puede referirse a diagramas (casos de uso, clases), documentos (plan de proyecto, descripción de requisitos) o incluso al código compilado, ya que se hace necesario realizarle pruebas al producto final.

Código abierto Término con el que se conoce al software distribuido y desarrollado libremente. El código abierto tiene un punto de vista más orientado a los beneficios prácticos de compartir el código que a las cuestiones éticas y morales las cuales destacan en el llamado software libre.

Componente Paquete de software reutilizable que ofrece un conjunto de servicios, o funcionalidades, a través de interfaces definidas, y que junto a otros componentes puede formar un sistema.

Contenedor En un marco de trabajo, es el encargado de la gestión de las instancias de los objetos.

Framework Un framework o marco de trabajo es una base tecnológica para el desarrollo de aplicaciones.

Integración Acción y efecto de integrar, en este caso componentes.

Mecanismo Conjunto de las partes de una máquina en su disposición adecuada. Estructura de un cuerpo natural o artificial, y combinación de sus partes constitutivas.

En este caso disposición de las partes involucradas para llevar a cabo la integración entre componentes del marco de trabajo Sauxe.

Módulo Pieza o conjunto unitario de piezas reutilizables en un software.

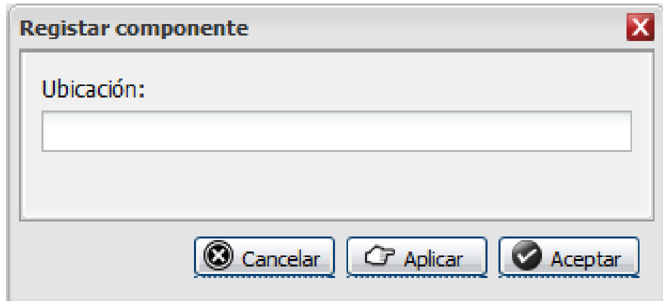
Anexos

Anexo 1 Descripción de los requisitos funcionales de la solución.

Descripción textual del requisito Registrar componente.

Precondiciones	El usuario deberá haber copiado el directorio que contiene al componente en la ubicación deseada.	
Flujo de eventos		
Flujo básico Registrar componente		
1.	Seleccionar opción **Registrar.**	
2.	Insertar ubicación del componente.	
3.	Si se presiona **Aceptar** el sistema validará la ubicación.	
4.	Si la información introducida es válida, se insertará en el archivo central *bundles.xml,* en el fichero serializado y en la caché de la aplicación los datos correspondientes al componente registrado.	
5.	Se realizará el proceso de resolución de dependencias.	
6.	Finaliza el requisito.	
Pos-condiciones		
1.	Quedarán guardados en el fichero serializado bundles.data, en el fichero bundles.xml y en la caché de la aplicación los datos del componente registrado.	
Flujos alternativos		
Flujo alternativo 3.a Si se presiona Cancelar.		
1.	Finaliza el requisito.	
Pos-condiciones		
1.	No se guardará la información.	
Flujo alternativo 4.a Si la información introducida no es válida.		
1.	Se ofrece al usuario la posibilidad de entrar información válida.	
2.	Continuar en el paso 3 del flujo básico de eventos.	
Pos-condiciones		
1.	N/A	
Validaciones		
1.	La ubicación insertada debe corresponder a un directorio real de la aplicación y que este contenga el fichero de configuración bundle.xml.	
Conceptos	**Componente**	**Visibles en la interfaz:** Ubicación. **Utilizados internamente:** Fichero de configuración.
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

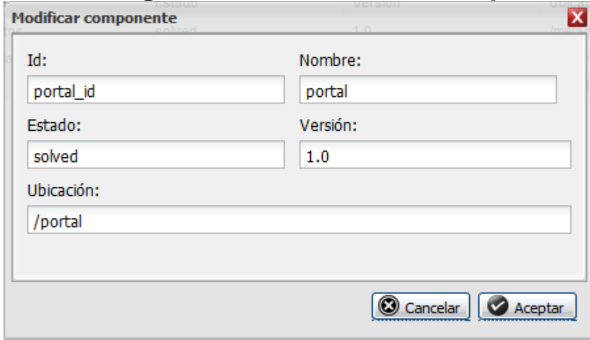
Prototipo elemental de interfaz gráfica de usuario del requisito Registrar componente.



Descripción textual del requisito Modificar componente.

Precondiciones	Se debe seleccionar el componente a modificar.	
Flujo de eventos		
Flujo básico Modificar componente		
1.	Seleccionar opción **Modificar.**	
2.	Insertar los nuevos datos del componente.	
3.	Si se presiona **Aceptar** el sistema validará la información.	
4.	Si la información introducida es válida, el sistema modificará en el archivo bundle.xml del componente, en el fichero central *bundles.xml,* en el fichero serializado y en la caché de la aplicación los datos correspondientes al componente seleccionado.	
5.	Finaliza el requisito.	
Pos-condiciones		
1.	Quedarán guardados en el fichero serializado bundles.data, en el fichero bundles.xml y en la caché de la aplicación los datos actualizados del componente seleccionado.	
Flujos alternativos		
Flujo alternativo 3.a Si se presiona Cancelar.		
1.	Finaliza el requisito.	
Pos-condiciones		
1.	No se actualizará la información del componente seleccionado.	
Flujo alternativo 4.a Si la información introducida no es válida.		
1.	Se ofrece al usuario la posibilidad de entrar información válida.	
2.	Continuar en el paso 3 del flujo básico de eventos.	
Pos-condiciones		
1.	N/A	
Validaciones		
1.	El id del componente debe ser una cadena de texto.	
2.	El nombre del componente debe ser una cadena de texto.	
3.	La versión del componente debe contener solo números.	
4.	La ubicación debe pertenecer a un directorio real.	
Conceptos	**Componente**	**Visibles en la interfaz:** Id, nombre, versión y ubicación. **Utilizados internamente:** Fichero de configuración.
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

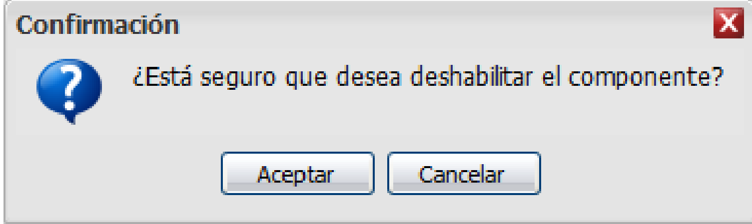
Prototipo elemental de interfaz gráfica de usuario del requisito Modificar componente



Descripción textual del requisito Deshabilitar componente.

Precondiciones	Se debe seleccionar el componente a deshabilitar.	
Flujo de eventos		
Flujo básico Deshabilitar componente		
1.	Seleccionar opción **Deshabilitar.**	
2.	Si se presiona **Aceptar** en el cuadro de confirmación, el sistema eliminará del archivo central *bundles.xml,* del fichero serializado y en la caché de la aplicación los datos correspondientes al componente seleccionado.	
3.	Se actualiza a *disable* el estado del componente en el fichero bundle.xml.	
4.	Finaliza el requisito.	
Pos-condiciones		
1.	Quedará en la aplicación el directorio que contiene al componente pero este no estará disponible para ser usado.	
Flujos alternativos		
Flujo alternativo 3.a Si se presiona Cancelar en el cuadro de confirmación.		
1.	Finaliza el requisito.	
Pos-condiciones		
1.	El componente seleccionado seguirá estando activo en la aplicación.	
Validaciones		
1.	N/A	
Conceptos	**Componente**	**Visibles en la interfaz:** N/A **Utilizados internamente:** Fichero de configuración.
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

Prototipo elemental de interfaz gráfica de usuario del requisito Deshabilitar componente.



Anexo 2 Diseños de Casos de prueba

DCP 1

Nombre del requisito	Descripción general	Escenarios de pruebas	Flujo del escenario
Registrar componente.	El sistema debe almacenar la información de un nuevo componente o un componente deshabilitado anteriormente en el fichero de configuración central, en el fichero serializado y en la caché de la aplicación.	EP 1.1: Registrar componente introduciendo una dirección válida.	– Se introduce la ubicación del componente correctamente. – Se presiona el botón Aceptar.
		EP 1.2: Registrar componente introduciendo una dirección no válida.	– Se introduce una ubicación no válida. – Se presiona el botón Aceptar.
		EP 1.3: Registrar componente dejando el campo ubicación vacío.	– Se deja el campo ubicación vacío. – Se presiona el botón Aceptar.

Descripción de variable

No	Nombre de campo	Tipo	Válido	Inválido
1	Ubicación	Campo de texto	Dirección correspondiente a cualquier directorio dentro de /apps que contenga entre sus archivos, al bundle.xml	Vacío, cualquier dirección correspondiente a un directorio que no contenga el fichero bundle.xml o cualquier dirección correspondiente a un componente deshabilitado.

Juegos de datos a probar

Id del escenario	Escenario	Ubicación	Respuesta del sistema
EP 1.1	Registrar componente introduciendo una dirección válida.	V (portal) o V (metadatos) o V (seguridad) o V (traza)	El sistema emitirá el mensaje “El componente ha sido registrado satisfactoriamente” y publicará los datos referentes al componente registrado.
EP 1.2	Registrar componente introduciendo una dirección no válida.	I (home) I (documentos)	El sistema emitirá un mensaje de error “El directorio no existe “.
		I (contabilidad) o I (instalador)	El sistema emitirá un mensaje de error “El directorio no contiene un componente”.
EP 1.3	Registrar componente dejando el campo ubicación vacío.	I (Vacío)	El sistema emitirá un mensaje de error “Por favor, verifique que no existan campos vacíos o con valores incorrectos”.

DCP 2

Nombre del requisito	Descripción general	Escenarios de pruebas	Flujo del escenario
Modificar componente.	El sistema debe actualizar la información en el fichero de configuración central, en el fichero serializado y en la caché de la aplicación de un componente seleccionado.	EP 1.1: Modificar componente introduciendo datos válidos.	– Se selecciona el componente a modificar y se presiona **Modificar.** – Se introducen todos los datos del componente correctamente. – Se presiona el botón **Aceptar**.
		EP 1.2: Modificar componente introduciendo datos inválidos.	– Se selecciona el componente a modificar y se presiona **Modificar.** – Se introducen datos inválidos del componente. – Se presiona el botón **Aceptar**.
		EP 1.3: Modificar componente dejando campos vacíos.	– Se selecciona el componente a modificar y se presiona **Modificar.** – Se introducen datos del componente dejando campos vacíos. – Se presiona el botón **Aceptar**.

Descripción de variables

No	Nombre de campo	Tipo	Válido	Inválido
1	Id	Campo de texto.	Cualquier cadena que contenga letras, números, espacios en blanco y los caracteres especiales (-) (_).	Otros caracteres especiales. Vacío.
2	Nombre	Campo de texto.	Cualquier cadena que contenga letras, números, espacios en blanco y los caracteres especiales (-) (_).	Otros caracteres especiales. Vacío.
3	Estado	Campo de texto no editable.	N/A	N/A
4	Versión	Campo de texto.	Cualquier cadena que contenga letras en minúscula y números separados por puntos. Vacío.	Cualquier cadena que contenga letras en mayúscula y caracteres especiales.
5	Ubicación	Campo de texto.	Dirección correspondiente a cualquier directorio dentro de /apps que contenga entre sus archivos, al bundle.xml	Vacío, cualquier dirección correspondiente a un directorio que no contenga el fichero bundle.xml o cualquier dirección correspondiente a un componente deshabilitado.

Juegos de datos a probar

Id del escenario	Escenario	Id	Nombre	Versión	Ubicación	Respuesta del sistema
EP 1.1	Modificar componente introduciendo datos válidos.	V(traza_id)	V(traza)	V(1.0)	V (/instalador)	El sistema emitirá el mensaje “El componente fue modificado satisfactoriamente” y publicará los datos actuales del componente seleccionado.
		V(trazaid)	V(trazas)	V(1.0v)	V (/traza)	
EP1.2	Modificar componente introduciendo datos inválidos.	V(otro_id)	V(otro)	V(1.0)	I (/home)	El sistema emitirá un mensaje de error ““El directorio no existe.
		V(otro_id)	V(otro)	V(1.0)	I (/contabilidad)	El sistema emitirá un mensaje de error “Esta ubicación no contiene un componente”.
		I (seguridad_ id)	V(traza)	V(1.0)	V (/traza)	El sistema emitirá un mensaje de error “El id se encuentra en uso por otro componente”.
		I (traza*id)	I (traz@)	I(1.0V)	V (/traza)	El sistema emitirá un mensaje de error “Por favor, verifique que no existan campos vacíos o con valores incorrectos” y mantendrá abierta la ventana para que el usuario corrija la información.
EP 1.3	Modificar componente dejando campos vacíos.	V(otro_id)	V(otro)	V(1.0)	I (Vacío)	El sistema emitirá un mensaje de error “Por favor, verifique que no existan campos vacíos o con valores incorrectos” y mantendrá abierta la ventana para que el usuario corrija la información.
		V(otro_id)	I (Vacío)	V(1.0)	V(contabilidad)	
		I (Vacío)	V(otro)	V(1.0)	V(contabilidad)	

DCP 3

Nombre del requisito	Descripción general	Escenarios de pruebas	Flujo del escenario
Deshabilitar componente.	El sistema debe eliminar la información en el fichero de configuración central, en el fichero serializado y en la caché de la aplicación de un componente seleccionado.	EP 1.1: Deshabilitar componente.	– Se selecciona el componente a deshabilitar y se presiona **Deshabilitar.** – Se presiona el botón **Aceptar** en el cuadro de confirmación. – Se muestra el mensaje “El componente fue deshabilitado satisfactoriamente”

Descripción de variable

No	Nombre de campo	Tipo	Válido	Inválido
1	N/A	N/A	N/A	N/A

Juegos de datos a probar

Id del escenario	Escenario	Respuesta del sistema
N/A	N/A	N/A

Anexo 3 No Conformidades detectadas

No	No conformidad	Aspecto correspondiente	Clasificación	Estado NC
1	Se permite insertar cualquier ubicación.	Gestionar componente/ Registrar componente.	S	Pendiente 8/6/2012 Resuelta 8/6/2012
3	No se actualizan los datos modificados a un componente.	Gestionar componente/ Modificar componente.	S	Pendiente 8/6/2012 Resuelta 8/6/2012
4	El campo de texto para la versión es editable.	Gestionar componente/ Modificar componente.	S	Pendiente 12/6/2012 Resuelta 12/6/2012
5	Cuando se dejan campos vacíos el mensaje no se corresponde con el DCP.	Gestionar componente/ Modificar componente.	S	Pendiente 15/6/2012 Resuelta 18/6/2012
6	Cambiar el mensaje de información cuando se modifica el componente.	Gestionar componente/ Modificar componente.	S	Pendiente 15/6/2012 Resuelta 18/6/2012
7	Al eliminar el último componente no se actualiza la interfaz.	Gestionar componente/ Deshabilitar componente.	S	Pendiente 15/6/2012 Resuelta 18/6/2012
8	Cuando se está deshabilitando el componente no muestra un mensaje de espera.	Gestionar componente/ Deshabilitar componente.	S	Pendiente 15/6/2012 Resuelta 18/6/2012
9	En el campo ¨Estado¨ los nombres aparecen en inglés.	Gestionar componente.	S	Pendiente 15/6/2012 Resuelta 18/6/2012